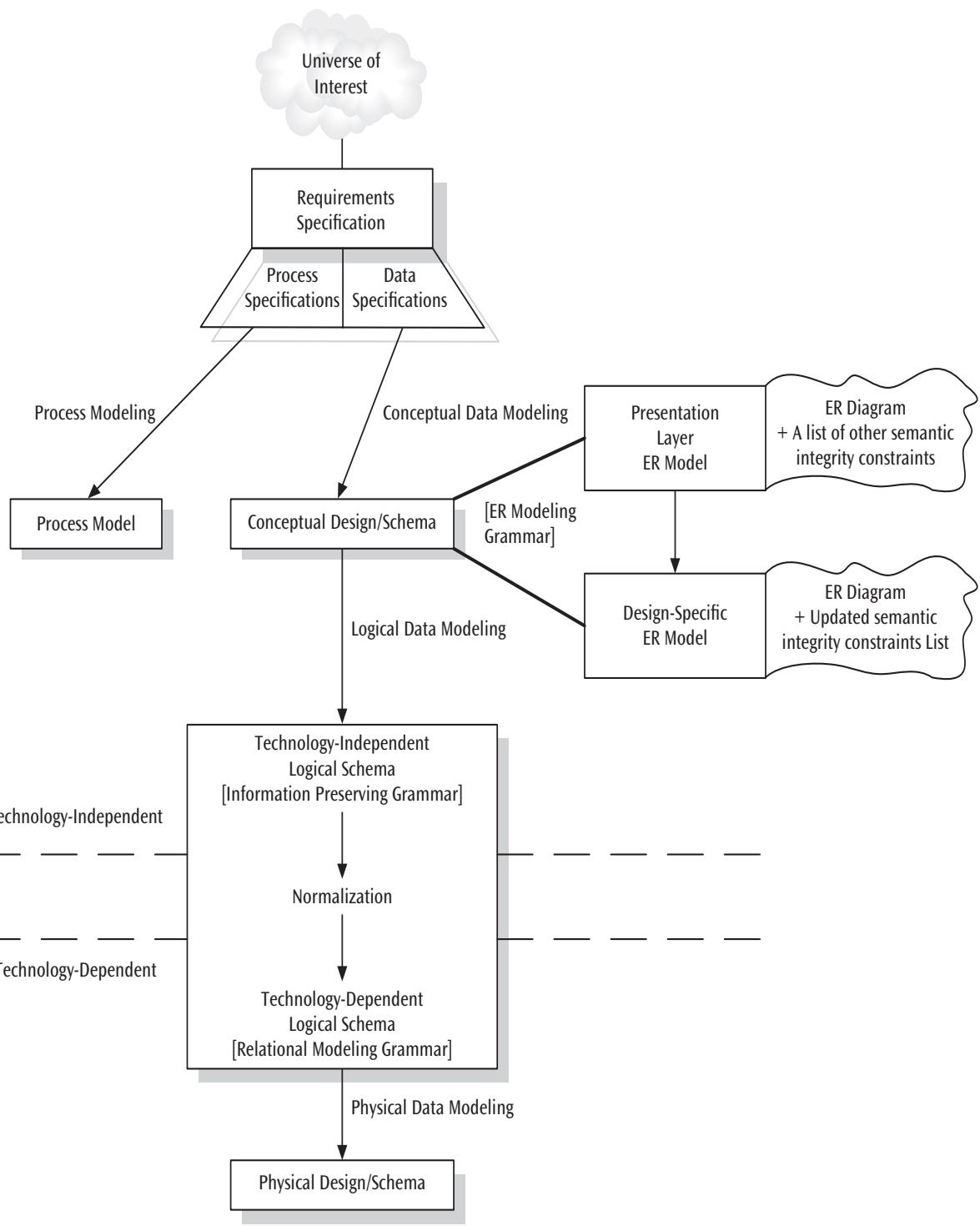


SECOND EDITION

# DATA MODELING AND DATABASE DESIGN

UMANATH | SCAMELL

# Data modeling/database design life cycle



# **DATA MODELING AND DATABASE DESIGN**



# DATA MODELING AND DATABASE DESIGN

**Second Edition**

**Narayan S. Umanath**  
*University of Cincinnati*  
**Richard W. Scamell**  
*University of Houston*



---

Australia • Brazil • Mexico • Singapore • United Kingdom • United States

This is an electronic version of the print textbook. Due to electronic rights restrictions, some third party content may be suppressed. Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. The publisher reserves the right to remove content from this title at any time if subsequent rights restrictions require it. For valuable information on pricing, previous editions, changes to current editions, and alternate formats, please visit [www.cengage.com/highered](http://www.cengage.com/highered) to search by ISBN#, author, title, or keyword for materials in your areas of interest.

**Data Modeling and Database Design,  
Second Edition**

Narayan S. Umanath and  
Richard W. Scamell

Production Director: Patty Stephan

Product Manager: Clara Goosman

Managing Developer: Jeremy Judson

Content Developer: Wendy Langeurd

Product Assistant: Brad Sullender

Senior Marketing Manager: Eric La Scola

IP Analyst: Sara Crane

Senior IP Project Manager: Kathryn Kucharek

Manufacturing Planner: Ron Montgomery

Art and Design Direction, Production

Management, and Composition:

PreMediaGlobal

Cover Image: © VikaSuh/www.Shutterstock.com

© 2015 Cengage Learning

WCN: 02-200-203

ALL RIGHTS RESERVED. No part of this work covered by the copyright herein may be reproduced, transmitted, stored, or used in any form or by any means graphic, electronic, or mechanical, including but not limited to photocopying, recording, scanning, digitizing, taping, Web distribution, information networks, or information storage and retrieval systems, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the publisher.

For product information and technology assistance, contact us at  
**Cengage Learning Customer & Sales Support, 1-800-354-9706**

For permission to use material from this text or product,  
submit all requests online at [www.cengage.com/permissions](http://www.cengage.com/permissions)

Further permissions questions can be e-mailed to  
[permissionrequest@cengage.com](mailto:permissionrequest@cengage.com)

Library of Congress Control Number: 2014934580

ISBN-13: 978-1-285-08525-8

ISBN-10: 1-285-08525-6

**Cengage Learning**

20 Channel Center Street  
Boston, MA 02210  
USA

Cengage Learning is a leading provider of customized learning solutions with office locations around the globe, including Singapore, the United Kingdom, Australia, Mexico, Brazil, and Japan. Locate your local office at [www.cengage.com/global](http://www.cengage.com/global)

Cengage Learning products are represented in Canada by  
Nelson Education, Ltd.

To learn more about Cengage Learning Solutions, visit [www.cengage.com](http://www.cengage.com)

Purchase any of our products at your local college store or at our  
preferred online store [www.cengagebrain.com](http://www.cengagebrain.com)

Printed in the United States of America  
1 2 3 4 5 6 7 18 17 16 15 14

*To Beloved Bhagwan Sri Sathya Sai Baba, the very source  
of my thoughts, words, and deeds*  
*To my Graduate Teaching Assistants and students,  
the very source of my inspiration*  
*To my dear children, Sharda and Kausik, always concerned  
about their dad overworking*  
*To my dear wife Lalitha, a pillar of courage I always lean on*

*Uma*

*There is a verse that says*  
Focus on what I'm doing right now  
And tell me that you appreciate me  
So that I learn to feel worthy  
And motivated to do more  
*Led by my family, I have always been surrounded by people  
(friends, teachers, and students) who*  
*With their kind thoughts, words, and deeds treat me in this way.*  
*This book is dedicated to these people.*

*Richard*



# BRIEF CONTENTS

Preface	xvii
<b>Chapter 1</b>	
<i>Database Systems: Architecture and Components</i>	1

## **Part I: Conceptual Data Modeling**

---

<b>Chapter 2</b>	
<i>Foundation Concepts</i>	30
<b>Chapter 3</b>	
<i>Entity-Relationship Modeling</i>	79
<b>Chapter 4</b>	
<i>Enhanced Entity-Relationship (EER) Modeling</i>	141
<b>Chapter 5</b>	
<i>Modeling Complex Relationships</i>	197

## **Part II: Logical Data Modeling**

---

<b>Chapter 6</b>	
<i>The Relational Data Model</i>	280

## **Part III: Normalization**

---

<b>Chapter 7</b>	
<i>Functional Dependencies</i>	358
<b>Chapter 8</b>	
<i>Normal Forms Based on Functional Dependencies</i>	395
<b>Chapter 9</b>	
<i>Higher Normal Forms</i>	467

**Part IV: Database Implementation Using the Relational Data Model**

---

**Chapter 10**

<i>Database Creation</i>	506
--------------------------	-----

**Chapter 11**

<i>Relational Algebra</i>	539
---------------------------	-----

**Chapter 12**

<i>Structured Query Language (SQL)</i>	567
--	-----

**Chapter 13**

<i>Advanced Data Manipulation Using SQL</i>	635
---	-----

**Appendix A**

<i>Data Modeling Architectures Based on the Inverted Tree and Network Data Structures</i>	719
---	-----

**Appendix B**

<i>Object-Oriented Data Modeling Architectures</i>	731
--	-----

**Selected Bibliography**

739
-----

**Index**

743
-----

# TABLE OF CONTENTS

<b>Preface</b>	xvii
<b>Chapter 1 Database Systems: Architecture and Components</b>	1
1.1 Data, Information, and Metadata	1
1.2 Data Management	3
1.3 Limitations of File-Processing Systems	3
1.4 The ANSI/SPARC Three-Schema Architecture	6
1.4.1 Data Independence Defined	8
1.5 Characteristics of Database Systems	10
1.5.1 What Is a Database System?	11
1.5.2 What Is a Database Management System?	12
1.5.3 Advantages of Database Systems	15
1.6 Data Models	17
1.6.1 Data Models and Database Design	17
1.6.2 Data Modeling and Database Design in a Nutshell	19
Chapter Summary	25
Exercises	25

## Part I: Conceptual Data Modeling

---

<b>Chapter 2 Foundation Concepts</b>	30
2.1 A Conceptual Modeling Framework	30
2.2 ER Modeling Primitives	30
2.3 Foundations of the ER Modeling Grammar	32
2.3.1 Entity Types and Attributes	32
2.3.2 Entity and Attribute-Level Data Integrity Constraints	35
2.3.3 Relationship Types	38
2.3.4 Structural Constraints of a Relationship Type	43
2.3.5 Base Entity Types and Weak Entity Types	52
2.3.6 Cluster Entity Type: A Brief Introduction	57
2.3.7 Specification of Deletion Constraints	58
Chapter Summary	70
Exercises	71
<b>Chapter 3 Entity-Relationship Modeling</b>	79
3.1 Bearcat Incorporated: A Case Study	79
3.2 Applying the ER Modeling Grammar to the Conceptual Modeling Process	81
3.2.1 The Presentation Layer ER Model	82
3.2.2 The Presentation Layer ER Model for Bearcat Incorporated	85

3.2.3 The Design-Specific ER Model	104
3.2.4 The Decomposed Design-Specific ER Model	111
3.3 Data Modeling Errors	119
3.3.1 Vignette 1	120
3.3.2 Vignette 2	127
Chapter Summary	134
Exercises	134
 <b>Chapter 4 Enhanced Entity-Relationship (EER) Modeling</b>	 141
4.1 Superclass/subclass Relationship	142
4.1.1 A Motivating Exemplar	142
4.1.2 Introduction to the Intra-Entity Class Relationship Type	143
4.1.3 General Properties of a Superclass/subclass Relationship	145
4.1.4 Specialization and Generalization	146
4.1.5 Specialization Hierarchy and Specialization Lattice	154
4.1.6 Categorization	157
4.1.7 Choosing the Appropriate EER Construct	160
4.1.8 Aggregation	166
4.2 Converting from the Presentation Layer to a Design-Specific EER Diagram	168
4.3 Bearcat Incorporated Data Requirements Revisited	170
4.4 ER Model for the Revised Story	171
4.5 Deletion Rules for Intra-Entity Class Relationships	182
Chapter Summary	188
Exercises	188
 <b>Chapter 5 Modeling Complex Relationships</b>	 197
5.1 The Ternary Relationship Type	198
5.1.1 Vignette 1—Madeira College	198
5.1.2 Vignette 2—Get Well Pharmacists, Inc.	203
5.2 Beyond the Ternary Relationship Type	205
5.2.1 The Case for a Cluster Entity Type	205
5.2.2 Vignette 3—More on Madeira College	206
5.2.3 Vignette 4—A More Complex Entity Clustering	212
5.2.4 Cluster Entity Type—Additional Examples	212
5.2.5 Madeira College—The Rest of the Story	216
5.2.6 Clustering a Recursive Relationship Type	221
5.3 Inter-Relationship Integrity Constraint	224
5.4 Composites of Weak Relationship Types	230
5.4.1 Inclusion Dependency in Composite Relationship Types	230
5.4.2 Exclusion Dependency in Composites of Weak Relationship Types	231
5.5 Decomposition of Complex Relationship Constructs	234
5.5.1 Decomposing Ternary and Higher-Order Relationship Types	234
5.5.2 Decomposing a Relationship Type with a Multi-Valued Attribute	235
5.5.3 Decomposing a Cluster Entity Type	240
5.5.4 Decomposing Recursive Relationship Types	241
5.5.5 Decomposing a Weak Relationship Type	244

5.6 Validation of the Conceptual Design	246
5.6.1 Fan Trap	246
5.6.2 Chasm Trap	251
5.6.3 Miscellaneous Semantic Traps	253
5.7 Cougar Medical Associates	257
5.7.1 Conceptual Model for CMA: The Genesis	259
5.7.2 Conceptual Model for CMA: The Next Generation	265
5.7.3 The Design-Specific ER Model for CMA: The Final Frontier	266
Chapter Summary	273
Exercises	273

## **Part II: Logical Data Modeling**

---

<b>Chapter 6 <i>The Relational Data Model</i></b>	280
6.1 Definition	280
6.2 Characteristics of a Relation	282
6.3 Data Integrity Constraints	283
6.3.1 The Concept of Unique Identifiers	284
6.3.2 Referential Integrity Constraint in the Relational Data Model	290
6.4 A Brief Introduction to Relational Algebra	291
6.4.1 Unary Operations: Selection ( $\sigma$ ) and Projection ( $\pi$ )	292
6.4.2 Binary Operations: Union ( $\cup$ ), Difference ( $-$ ), and Intersection ( $\cap$ )	293
6.4.3 The Natural Join (*) Operation	295
6.5 Views and Materialized Views in the Relational Data Model	296
6.6 The Issue of Information Preservation	297
6.7 Mapping an ER Model to a Logical Schema	298
6.7.1 Information-Reducing Mapping of ER Constructs	298
6.7.2 An Information-Preserving Mapping	315
6.8 Mapping Enhanced ER Model Constructs to a Logical Schema	320
6.8.1 Information-Reducing Mapping of EER Constructs	321
6.8.2 Information-Preserving Grammar for Enhanced ER Modeling Constructs	328
6.9 Mapping Complex ER Model Constructs to a Logical Schema	336
Chapter Summary	345
Exercises	347

## **Part III: Normalization**

---

<b>Chapter 7 <i>Functional Dependencies</i></b>	358
7.1 A Motivating Exemplar	359
7.2 Functional Dependencies	365
7.2.1 Definition of Functional Dependency	365
7.2.2 Inference Rules for Functional Dependencies	366
7.2.3 Minimal Cover for a Set of Functional Dependencies	367
7.2.4 Closure of a Set of Attributes	372
7.2.5 When Do FDs Arise?	374

<b>7.3 Candidate Keys Revisited</b>	374
7.3.1 Deriving Candidate Key(s) by Synthesis	375
7.3.2 Deriving Candidate Keys by Decomposition	379
7.3.3 Deriving a Candidate Key—Another Example	382
7.3.4 Prime and Non-prime Attributes	386
Chapter Summary	390
Exercises	390
 <b>Chapter 8 Normal Forms Based on Functional Dependencies</b>	 395
8.1 Normalization	395
8.1.1 First Normal Form (1NF)	396
8.1.2 Second Normal Form (2NF)	398
8.1.3 Third Normal Form (3NF)	401
8.1.4 Boyce-Codd Normal Form (BCNF)	404
8.1.5 Side Effects of Normalization	407
8.1.6 Summary Notes on Normal Forms	418
8.2 The Motivating Exemplar Revisited	420
8.3 A Comprehensive Approach to Normalization	424
8.3.1 Case 1	424
8.3.2 Case 2	431
8.3.3 A Fast-Track Algorithm for a Non-Loss, Dependency-Preserving Solution	436
8.4 Denormalization	442
8.5 Role of Reverse Engineering in Data Modeling	443
8.5.1 Reverse Engineering the Normalized Solution of Case 1	445
8.5.2 Reverse Engineering the Normalized Solution of URS2 (Case 3)	451
8.5.3 Reverse Engineering the Normalized Solution of URS3 (Case 2)	453
Chapter Summary	457
Exercises	458
 <b>Chapter 9 Higher Normal Forms</b>	 467
9.1 Multi-Valued Dependency	467
9.1.1 A Motivating Exemplar for Multi-Valued Dependency	467
9.1.2 Multi-Valued Dependency Defined	469
9.1.3 Inference Rules for Multi-Valued Dependencies	470
9.2 Fourth Normal Form (4NF)	472
9.3 Resolution of a 4NF Violation—A Comprehensive Example	476
9.4 Generality of Multi-Valued Dependencies and 4NF	478
9.5 Join-Dependencies and Fifth Normal Form (5NF)	480
9.6 A Thought-Provoking Exemplar	490
9.7 A Note on Domain Key Normal Form (DK/NF)	497
Chapter Summary	498
Exercises	498

## Part IV: Database Implementation Using the Relational Data Model

---

<b>Chapter 10</b>	<i>Database Creation</i>	506
10.1	Data Definition Using SQL	507
10.1.1	Base Table Specification in SQL/DDL	507
10.2	Data Population Using SQL	524
10.2.1	The INSERT Statement	525
10.2.2	The DELETE Statement	528
10.2.3	The UPDATE Statement	530
	Chapter Summary	532
	Exercises	532
<b>Chapter 11</b>	<i>Relational Algebra</i>	539
11.1	Unary Operators	542
11.1.1	The Select Operator	542
11.1.2	The Project Operator	544
11.2	Binary Operators	546
11.2.1	The Cartesian Product Operator	546
11.2.2	Set Theoretic Operators	549
11.2.3	Join Operators	551
11.2.4	The Divide Operator	557
11.2.5	Additional Relational Operators	560
	Chapter Summary	563
	Exercises	563
<b>Chapter 12</b>	<i>Structured Query Language (SQL)</i>	567
12.1	SQL Queries Based on a Single Table	569
12.1.1	Examples of the Selection Operation	569
12.1.2	Use of Comparison and Logical Operators	572
12.1.3	Examples of the Projection Operation	578
12.1.4	Grouping and Summarizing	580
12.1.5	Handling Null Values	583
12.1.6	Pattern Matching in SQL	593
12.2	SQL Queries Based on Binary Operators	597
12.2.1	The Cartesian Product Operation	597
12.2.2	SQL Queries Involving Set Theoretic Operations	599
12.2.3	Join Operations	602
12.2.4	Outer Join Operations	608
12.2.5	SQL and the Semi-Join and Semi-Minus Operations	612
12.3	Subqueries	613
12.3.1	Multiple-Row Uncorrelated Subqueries	613
12.3.2	Multiple-Row Correlated Subqueries	625
12.3.3	Aggregate Functions and Grouping	628
	Chapter Summary	631
	Exercises	631

<b>Chapter 13 Advanced Data Manipulation Using SQL</b>	635
13.1 Selected SQL:2003 Built-In Functions	635
13.1.1 The SUBSTRING Function	636
13.1.2 The CHAR_LENGTH (char) Function	639
13.1.3 The TRIM Function	640
13.1.4 The TRANSLATE Function	643
13.1.5 The POSITION Function	644
13.1.6 Combining the INSTR and SUBSTR Functions	645
13.1.7 The DECODE Function and the CASE Expression	646
13.1.8 A Query to Simulate the Division Operation	649
13.2 Some Brief Comments on Handling Dates and Times	651
13.3 Hierarchical Queries	656
13.3.1 Using the CONNECT BY and START WITH Clauses with the PRIOR Operator	658
13.3.2 Using the LEVEL Pseudo-Column	660
13.3.3 Formatting the Results from a Hierarchical Query	661
13.3.4 Using a Subquery in a START WITH Clause	661
13.3.5 The SYS_CONNECT_BY_PATH Function	663
13.3.6 Joins in Hierarchical Queries	664
13.3.7 Incorporating a Hierarchical Structure into a Table	665
13.4 Extended GROUP BY Clauses	668
13.4.1 The ROLLUP Operator	668
13.4.2 Passing Multiple Columns to ROLLUP	669
13.4.3 Changing the Position of Columns Passed to ROLLUP	671
13.4.4 Using the CUBE Operator	672
13.4.5 The GROUPING () Function	674
13.4.6 The GROUPING SETS Extension to the GROUP BY Clause	676
13.4.7 The GROUPING_ID ()	677
13.4.8 Using a Column Multiple Times in a GROUP BY Clause	679
13.5 Using the Analytical Functions	681
13.5.1 Analytical Function Types	682
13.5.2 The RANK () and DENSE_RANK () Functions	684
13.5.3 Using ROLLUP, CUBE, and GROUPING SETS Operators with Analytical Functions	687
13.5.4 Using the Window Functions	688
13.6 A Quick Look at the MODEL Clause	692
13.6.1 MODEL Clause Concepts	693
13.6.2 Basic Syntax of the MODEL Clause	693
13.6.3 An Example of the MODEL Clause	694
13.7 A Potpourri of Other SQL Queries	700
13.7.1 Concluding Example 1	700
13.7.2 Concluding Example 2	702
13.7.3 Concluding Example 3	704
13.7.4 Concluding Example 4	704
13.7.5 Concluding Example 5	705
<b>Chapter Summary</b>	706
<b>Exercises</b>	707
<b>SQL Project</b>	711

<b>Appendix A   <i>Data Modeling Architectures Based on the Inverted Tree and Network Data Structures</i></b>	<b>719</b>
A.1 Logical Data Structures	719
A.1.1 Inverted Tree Structure	719
A.1.2 Network Data Structure	721
A.2 Logical Data Model Architectures	722
A.2.1 Hierarchical Data Model	722
A.2.2 CODASYL Data Model	726
Summary	729
Selected Bibliography	729
<b>Appendix B   <i>Object-Oriented Data Modeling Architectures</i></b>	<b>731</b>
B.1 The Object-Oriented Data Model	731
B.1.1 Overview of OO Concepts	732
B.1.2 A Note on UML	735
B.2 The Object-Relational Data Model	737
Summary	738
Selected Bibliography	738
<b>Selected Bibliography</b>	<b>739</b>
<b>Index</b>	<b>743</b>



# PREFACE

## QUOTE

*Everything should be made as simple as possible—but no simpler.*

—Albert Einstein

Popular business database books typically provide broad coverage of a wide variety of topics, including data modeling, database design and implementation, database administration, the client/server database environment, the Internet database environment, distributed databases, and object-oriented database development. This is invariably at the expense of deeper treatment of critical topics, such as principles of data modeling and database design. Using current business database books in our courses, we found that in order to properly cover data modeling and database design, we had to augment the texts with significant supplemental material (1) to achieve precision and detail and (2) to impart the depth necessary for the students to gain a robust understanding of data modeling and database design. In addition, we ended up skipping several chapters as topics to be covered in a different course. We also know other instructors who share this experience. Broad coverage of many database topics in a single book is appropriate for some audiences, but that is not the aim of this book.

The goal of *Data Modeling and Database Design, Second Edition* is to provide core competency in the areas that every Information Systems (IS), Computer Science (CS), and Computer Information Systems (CIS) student and professional should acquire: **data modeling and database design**. It is our experience that this set of topics is the most essential for database professionals, and that, covered in sufficient depth, these topics alone require a full semester of study. It is our intention to address these topics at a level of technical depth achieved in CS textbooks, yet make palatable to the business student/IS professional with little sacrifice in precision. We deliberately refrain from the mathematics and algorithmic solutions usually found in CS textbooks, yet we attempt to capture the precision therein via heuristic expressions.

*Data Modeling and Database Design, Second Edition* provides not just hands-on instruction in current data modeling and database design practices, it gives readers a thorough conceptual background for these practices. We do not subscribe to the idea that a textbook should limit itself to describing what is actually being practiced. Teaching only what is being practiced is bound to lead to knowledge stagnation. Where do practitioners learn what they know? Did they invent the relational data model? Did they invent the ER model? We believe that it is our responsibility to present not only industry “best practices” but also to provide students (future practitioners) with concepts and techniques that are not necessarily used in industry today

but can enliven their practice and help it evolve without knowledge stagnation. One of the coauthors of this book has worked in the software development industry for over 15 years, with a significant focus on database development. His experience indicates that having a richness of advanced data modeling constructs available enhances the robustness of database design and that practitioners readily adopt these techniques in their design practices.

In a nutshell, our goal is to take an IS/CS/CIS student/professional through an intense educational experience, starting at conceptual modeling and culminating in a fully implemented database design—**nothing more and nothing less**. This educational journey is briefly articulated in the following paragraphs.

## STRUCTURE

---

We have tried very hard to make the book “fluff-free.” It is our hope that every sentence in the book, including this preface, adds value to a reader’s learning (and *footnotes* are no exception to this statement).

The book begins with an introduction to rudimentary concepts of data, metadata, and information, followed by an overview of data management. Pointing out the limitations of file-processing systems, **Chapter 1** introduces database systems as a solution to overcome these limitations. The architecture and components of a database system that makes this possible are discussed. The chapter concludes with the presentation of a framework for the database system design life cycle. Following the introductory chapter on database systems architecture and components, the book contains four parts.

### Part I: Conceptual Data Modeling

Part I addresses the topic of conceptual data modeling—that is, modeling at the highest level of abstraction, independent of the limitations of the technology employed to deploy the database system. Four chapters (Chapters 2–5) are used in order to provide an extensive discussion of conceptual data modeling. Chapter 2 lays the groundwork using the *Entity-Relationship (ER) modeling grammar* as the principal means to model a database application domain. Chapter 3 elaborates on the use of the ER modeling grammar in progressive layers and exemplifies the modeling technique with a comprehensive case called Bearcat Incorporated. This is followed by a presentation in Chapter 4 of richer data modeling constructs that overlap with object-oriented modeling constructs. The Bearcat Incorporated story is further enriched to demonstrate the value of Enhanced ER (EER) modeling constructs. Chapter 5 provides exclusive coverage of modeling complex relationships that have meaningful real-world significance. At the end of Part I, the reader ought to be able to fully appreciate the value of conceptual data modeling in the database system design life cycle.

This second edition of *Data Modeling and Database Design* includes the following major enhancements:

- The material in Chapters 2 and 3 has been reorganized and better streamlined so that the reader not only learns the ER modeling grammar but is able to develop very simple applications of ER modeling. In Chapter 3, the modeling method steps have been reconfigured across the Presentation Layer and

Design-Specific layer of the ER model. Also, the unique learning technique via error detection exclusively developed by us is presented at the end of Chapter 3.

- The intra-entity class relationships are introduced with a new simpler example at the beginning of Chapter 4.
- The already extensive coverage of complex relationships in Chapter 5 is augmented by a few newer modeling ideas. Additional examples clarifying decomposition of complex relationships in preparation for logical model mapping have also been added to this chapter.

## Part II: Logical Data Modeling

Part II of the book is dedicated to the discussion of migration of a conceptual data model to its logical counterpart. Since the relational data model architecture forms the basis for the logical data modeling discussed in this textbook, Chapter 6 focuses on its characteristics. Other logical data modeling architectures prevalent in some legacy systems, the hierarchical data model, and the CODASYL data model appear in Appendix A. An introduction to object-oriented data modeling concepts is presented in Appendix B. The rest of Chapter 6 describes techniques to map a conceptual data model to its logical counterpart. An *information-preserving logical data modeling grammar* is introduced and contrasted with existing popular mapping techniques that are information reducing. A comprehensive set of examples is used to clarify the use and value of the information-preserving grammar.

An important addition to the current edition of the book is a section on mapping complex relationship types to the logical tier.

## Part III: Normalization

Part III addresses the critical question of the “goodness” of a database design that results from a conceptual and logical data modeling processes. *Normalization* is introduced as the “scientific” way to verify and improve the quality of a logical schema that is available at this stage in the database design. Three chapters are employed to cover the topic of normalization. In Chapter 7, we take a look at data redundancy in a relation schema and see how it manifests as a problem. We then trace the problem to its source—namely, undesirable functional dependencies. To that end, we first learn about functional dependencies axiomatically and how inference rules (Armstrong’s axioms) can be used to derive candidate keys of a relation schema. In Chapter 8, the solution offered by the normalization process to data redundancy problems triggered by undesirable functional dependencies is presented. After discussing first, second, third and Boyce-Codd normal forms individually, we examine the side effects of normalization—namely, dependency preservation and non-loss decomposition and their consequences. Next, we present real-world scenarios of deriving full-fledged relational schemas (sets of relation schemas), given sets of functional dependencies using several examples. The useful topic of denormalization is covered next. Reverse engineering a normalized relational schema to the conceptual tier often forges insightful understanding of the database design and enables a database designer to become a better data modeler. Despite its practical utility, this

topic is rarely covered in database textbooks. Chapter 9 completes the discussion of normalization by examining multi-valued dependency (MVD) and join-dependency (JD) and their impact on a relation schema in terms of fourth normal form (4NF) and Project/Join normal form, viz., PJNF (also known as fifth normal form—5NF) respectively.

An interesting enhancement in Chapter 8 is the introduction of a fast-track algorithm to achieve a non-loss, dependency-preserving 3NF design. Two distinct examples demonstrating the use of the algorithm are presented. The discussion of MVD and 4NF, of JD and 5NF, and their respective expressiveness of ternary and n-ray relationships is presented in Chapter 9. Additional examples offer unique insights into apparently conflicting alternative solutions.

## **Part IV: Database Implementation Using the Relational Database Model**

Part IV pertains to database implementation using the relational data model. Spread over four chapters, this part of the book covers relational algebra and the ANSI/ISO standard Structured Query Language (SQL). Chapter 10 focuses on the data definition language (DDL) aspect of SQL. Included in the discussion are the SQL schema evolution statements for adding, altering, or dropping table structures, attributes, constraints, and supporting structures. This is followed by the development of SQL/DDL script for a comprehensive case about a college registration system. The chapter also includes the use of INSERT, UPDATE, and DELETE statements in populating a database and performing database maintenance.

Chapters 11, 12, and 13 focus on relational algebra and the use of SQL for data manipulation. **Chapter 11** concentrates on E. F. Codd's eight original relational algebra operations as a means to specify the logic for data retrieval from a relational database. SQL, the most common way that relational algebra is implemented for data retrieval operations, is the subject of Chapter 12. Chapter 13 covers a number of built-in functions used by SQL to work with strings, dates, and times, and it illustrates how SQL can be used to do retrievals against hierarchically structured data. This chapter also provides an introduction to some of the features of SQL that facilitate the summarization and analysis of data. The chapter ends with an SQL database project that provides students with a real-life scenario to test and apply the skills and concepts presented in Part IV.

## **FEATURES OF EACH CHAPTER**

---

Since our objective is a crisp and clear presentation of rather intricate subject matter, each chapter begins with a simple introduction, followed by the treatment of the subject matter, and concludes with a chapter summary and a set of exercises based on the subject matter.

## **WHAT MAKES THIS BOOK DIFFERENT?**

---

Every book has strengths and weaknesses. If lack of breadth in the coverage of database topics is considered a weakness, we have deliberately chosen to be weak in that dimension. We have not planned this book to be another general book on

database systems. We have chosen to limit the scope of this book exclusively to data modeling and database design since we firmly believe that this set of topics is the core of database systems and must be learned in depth by every IS/CS/CIS student and practitioner. Any system designed robustly has the potential to best serve the needs of the users. More importantly, a poor design is a virus that can ruin an enterprise.

In this light, we believe these are the unique strengths of this book:

- It presents conceptual modeling using the entity-relationship modeling grammar including extensive discussion of the enhanced entity-relationship (ER) model.

*We believe that a conceptual model should capture all possible constraints conveyed by the business rules implicit in users' requirement specifications. To that end, we posit that an ER diagram is not an ER model unless accompanied by a comprehensive specification of characteristics of and constraints pertaining to attributes. We accomplish this via a list of semantic integrity constraints (sort of a conceptual data dictionary) that will accompany an ER diagram, a unique feature that we have not seen in other database textbooks. We also seek to demonstrate the systematic development of a multi-layer conceptual data model via a comprehensive illustration at the beginning of each Part. We consider the multi-layer modeling strategy and the heuristics for systematic development as unique features of this book.*

- It includes substantial coverage of higher-degree relationships and other complex relationships in the entity-relationship diagram.

*Most business database books seem to provide only a cursory treatment of complex relationships in an ER model. We not only cover relationships beyond binary relationships (e.g., ternary and higher-degree relationships), we also clarify the nuances pertaining to the necessity and efficacy of higher-degree relationships and the various conditions under which even recursive and binary relationships are aggregated in interesting ways to form cluster entity types.*

- It discusses the information-preserving issue in data model mapping and introduces a new information-preserving grammar for logical data modeling. *Many computer scientists have noted that the major difficulty of logical database design (i.e., transforming an ER schema into a schema in the language of some logical model) is the information preservation issue. Indeed, assuring a complete mapping of all modeling constructs and constraints that are inherent, implicit or explicit, in the source schema (e.g., ER/EER model) is problematic since constraints of the source model often cannot be represented directly in terms of structures and constraints of the target model (e.g., relational schema). In such a case, they must be realized through application programs; alternatively, an information-reducing transformation must be accepted (Fahrner and Vossen, 1995). In their research, initially presented at the Workshop on Information Technologies (WITS) in the ICIS (International Conference on Information Systems) in Brisbane,*

*Australia, Umanath and Chiang (2000) describe a logical modeling grammar that generates an information preserving transformation. Umanath further revised this modeling grammar based on the feedback received at WITS. We have included this logical modeling grammar as a unique component of this textbook.*

- It includes unique features under the topic of normalization rarely covered in business database books:
  - Inference rules for functional dependencies (*Armstrong's axioms and derivations of candidate keys from a set of functional dependencies*)
  - Derivation of canonical covers for a set of semantically obvious functional dependencies
  - Rich examples to clarify *the basic normal forms (first, second, third, and Boyce-Codd)*
  - Derivation of a complete logical schema *from a large set of functional dependencies considering lossless (non-additive) join properties and dependency preservation*
  - Reverse engineering a logical schema *to an entity-relationship diagram*
  - Advanced coverage of fourth and fifth normal form (*project-join normal form, abbreviated "PJNF"*) *using a variety of examples*
- It supports in-depth coverage of relational algebra with a significant number of examples of their operationalization in ANSI/ISO SQL.

## **A NOTE TO THE INSTRUCTOR**

---

The content of this book is designed for a rigorous one-semester course in database design and development and may be used at both undergraduate and graduate levels. Technical emphasis can be tempered by minimizing or eliminating the coverage of some of the following topics from the course syllabus: Enhanced Entity-Relationship (EER) Modeling (Chapter 4) and the related data model mapping topics in Chapter 6 (Section 6.8) on Mapping Enhanced ER Modeling Constructs to a Logical Schema; Modeling Complex Relationships (Chapter 5); and higher normal forms (Chapter 9). The suggested exclusions will not impair the continuity of the subject matter in the rest of the book.

## **SUPPORTING TECHNOLOGIES**

---

Any business database book can be effective only when supporting technologies are made available for student use. Yet, we don't think that the type of book we are writing should be married to any commercial product. The specific technologies that will render this book highly effective include a drawing tool (such as Microsoft Visio), a software engineering tool (such as ERWIN, ORACLE/Designer, or Visible Analyst), and a relational database management system (RDBMS) product (such as ORACLE, SQL Server, or DB2).

## SUPPLEMENTAL MATERIALS

---

The following supplemental materials are available to instructors when this book is used in a classroom setting. Some of these materials may also be found on the Cengage Learning Web site at [www.cengage.com](http://www.cengage.com).

- **Electronic Instructor's Manual:** The Instructor's Manual assists in class preparation by providing suggestions and strategies for teaching the text, and solutions to the end-of-chapter questions/problems.
- **Sample Syllabi and Course Outline:** The sample syllabi and course outlines are provided as a foundation to begin planning and organizing your course.
- **Cognero Test Bank:** Cognero allows instructors to create and administer printed, computer (LAN-based), and Internet exams. The Test Bank includes an array of questions that correspond to the topics covered in this text, enabling students to generate detailed study guides that include page references for further review. The computer-based and Internet testing components allow students to generate detailed study guides that include page references for further review. The computer-based and Internet testing components allow students to take exams at their computers, and also save the instructor time by automatically grading each exam. The Test Bank is also available in Blackboard and WebCT versions posted online at [www.course.com](http://www.course.com).
- **PowerPoint Presentations:** Microsoft PowerPoint slides for each chapter are included as a teaching aid for classroom presentation, to make available to students on the network for chapter review, or to be printed for classroom distribution. Instructors can add their own slides for additional topics they introduce to the class.
- **Figure Files:** Figure files from each chapter are provided for the instructor's use in the classroom.
- **Data Files:** Data files containing scripts to populate the database tables used as examples in Chapters 11 and 12 are provided on the Cengage Learning Web site at [www.cengage.com](http://www.cengage.com).

## ACKNOWLEDGMENTS

---

We have never written a textbook before. We have been using books written by our academic colleagues, always supplemented with handouts that we developed ourselves. Over the years, we accumulated a lot of supplemental material. In the beginning, we took the positive feedback from the students about the supplemental material rather lightly until we started to see comments like "I don't know why I bought the book; the instructor's handouts were so good and much clearer than the book" in the student evaluation forms. Our impetus to write a textbook thus originated from the consistent positive feedback from our students.

We also realized that, contrary to popular belief, business students are certainly capable of assimilating intricate technical concepts; the trick is to frame the concepts in meaningful business scenarios. The unsolicited testimonials from our alumni about

the usefulness of the technical depth offered in our database course in solving real-world design problems reinforced our faith in developing a book focused exclusively on data modeling and database design that was technically rigorous but permeated with business relevance.

Since we both teach database courses regularly, we have had the opportunity to field-test the manuscript of this book for close to 10 years at both undergraduate-level and graduate-level information systems courses in the Carl Lindner College of Business at the University of Cincinnati and in the C. T. Bauer College of Business at the University of Houston. Hundreds of students—mostly business students—have used earlier drafts of this textbook so far. Interestingly, even the computer science and engineering students taking our courses have expressed their appreciation of the content. This is a long preamble to acknowledge one of the most important and formative elements in the creation of this book: our students.

The students' continued feedback (comments, complaints, suggestions, and criticisms) have significantly contributed to the improvement of the content. As we were cycling through revisions of the manuscript, the graduate teaching assistants of Dr. Umanath were a constant source of inspiration. Their meaningful questions and suggestions added significant value to the content of this book. Dr. Seamell was ably assisted by his graduate assistants as well.

We would also like to thank the following reviewers whose critiques, comments, and suggestions helped shape every chapter of this book's first edition:

Akhilesh Bajaj, *University of Tulsa*

Iris Junlgas, *Florida State University*

Margaret Porciello, *State University of New York/Farmingdale*

Sandeep Purao, *Pennsylvania State University*

Jaymeen Shah, *Texas State University*

Last, but by no means the least, we gratefully acknowledge the significant contribution of Deb Kaufmann and Kent Williams, the development editors of our first and second editions, respectively. We cannot thank them enough for their thorough and also prompt and supportive efforts.

Enjoy!

N. S. Umanath

R. W. Seamell

# CHAPTER 1

# DATABASE SYSTEMS: ARCHITECTURE AND COMPONENTS

Data modeling and database design involve elements of both art and engineering. Understanding user requirements and modeling them in the form of an effective logical database design is an artistic process. Transforming the design into a physical database with functionally complete and efficient applications is an engineering process.

To better comprehend what drives the design of databases, it is important to understand the distinction between data and information. Data consists of raw facts—that is, facts that have not yet been processed to reveal their meaning. Processing these facts provides information on which decisions can be based.

Timely and useful information requires that data be accurate and stored in a manner that is easy to access and process. And, like any basic resource, data must be managed carefully. Data management is a discipline that focuses on the proper acquisition, storage, maintenance, and retrieval of data. Typically, the use of a database enables efficient and effective management of data.

This chapter introduces the rudimentary concepts of data and how information emerges from data when viewed through the lens of metadata. Next, the discussion addresses data management, contrasting file-processing systems with database systems. This is followed by brief examples of desktop, workgroup, and enterprise databases. The chapter then presents a framework for database design that describes the multiple tiers of data modeling and how these tiers function in database design. This framework serves as a roadmap to guide the reader through the remainder of the book.

## **1.1 DATA, INFORMATION, AND METADATA**

---

Although the terms are often used interchangeably, information is different from data. **Data** can be viewed as raw material consisting of unorganized facts about things, events, activities, and transactions. While data may have implicit meaning, the lack of organization renders it valueless. In other words, **information** is data in context—that is, data that has been organized into a specific context such that it has value to its recipient.

As an example, consider the digits 2357111317. What does this string of digits represent? One response is that they are simply 10 meaningless digits. Another might be

## Chapter 1

the number 31 (obtained by summing the 10 digits). A mathematician may see a set of prime numbers, viz., 2, 3, 5, 7, 11, 13, 17. Another might see a person's phone number with the first three digits constituting the area code and the remaining seven digits the local phone number. On the other hand, if the first digit is used to represent a person's gender (1 for male and 2 for female) and the remaining nine digits the person's Social Security number, the 10 digits would mean something else. Numerous other interpretations are possible, but without a context it is impossible to say what the digits represent. However, when framed in a specific context (such as being told that the first digit represents a person's gender and the remaining digits the Social Security number), the data is transformed into information. It is important to note that "information" is not necessarily the "Truth" since the same data yields different information based on the context; information is an inference.

**Metadata**, in a database environment, is data that describes the properties of data. It contains a complete definition or description of database structure (i.e., the file structure, data type, and storage format of each data item), and other constraints on the stored data. For example, when the structure of the 10 digits 2357111317 is revealed, the 10 digits become information, such as a phone number. Metadata defines this structure. In other words, through the lens of metadata, data takes on specific meaning and yields information.<sup>1</sup> Metadata may be characterized as follows:

- The lens to view data and infer information
- A precise definition of the context for framing the data

Table 1.1 contains metadata for the data associated with a manufacturing plant. Later in this chapter, we will see that in a database environment, metadata is recorded in what is called a data dictionary.

Record Type	Data Element	Data Type	Size	Source	Role	Domain
PLANT	Pl_name	Alphabetic	30	Stored	Non-key	
PLANT	Pl_number	Numeric	2	Stored	Key	Integer values from 10 to 20
PLANT	Budget	Numeric	7	Stored	Non-key	
PLANT	Building	Alphabetic	20	Stored	Non-key	
PLANT	No_of_employees	Numeric	4	Derived	Non-key	

**TABLE 1.1** Some metadata for a manufacturing plant

As reflected in Table 1.1, the smallest unit of data is called a **data element**. A group of related data elements treated as a unit (such as Pl\_name, Pl\_number, Budget, Building,

---

<sup>1</sup>With the advent of the data warehouse, the term "metadata" assumes a more comprehensive meaning to include business and technical metadata, which is outside the scope of the current discussion.

and No\_of\_employees) is called a **record type**. A set of values for the data elements constituting a record type is called a record instance or simply a **record**. A **file** is a collection of records. A file is sometimes referred to as a **data set**. A company with 10 plants would have a PLANT file or a PLANT data set that contains 10 records.

## 1.2 DATA MANAGEMENT

---

This book focuses strictly on management of data, as opposed to the management of human resources. Data management involves four actions: (a) data creation, (b) data retrieval, (c) data modification or updating, and (d) data deletion. Two data management functions support these four actions: Data must be accessed and, for ease of access, data must be organized.

Despite today's sophisticated information technologies, there are still only two primary approaches for accessing data. One is **sequential access**, where in order to get to the *n<sup>th</sup>* record in a data set it is necessary to pass through the previous *n-1* records in the data set. The second approach is **direct access**, where it is possible to get to the *n<sup>th</sup>* record without having to pass through the previous *n-1* records. While direct access is useful for *ad hoc* querying of information, sequential access remains essential for transaction processing applications such as generating payroll, grade reports, and utility bills.

In order to access data, the data must be organized. For sequential access, this means that all records in a file must be stored (organized) through some order using a unique identifier, such as employee number, inventory number, flight number, account number, or stock symbol. This is called sequential organization. A serial (unordered) collection of records, also known as a "heap file," cannot provide sequential access. For direct access, the records in a file can be stored serially and organized either randomly or by using an external index. A randomly organized file is one in which the value of a unique identifier is processed by some sort of transformation routine (often called a "hashing algorithm") that computes the location of records within the file (relative record numbers). An indexed file makes use of an index external to the data set similar in nature to the one found at the back of this book to identify the location where a record is physically stored.

As discussed in Section 1.5, a database takes advantage of software called a database management system (DBMS) that sits on top of a set of files physically organized as sequential files and/or as some form of direct access files. A DBMS facilitates data access in a database without burdening a user with the details of how the data is physically organized.

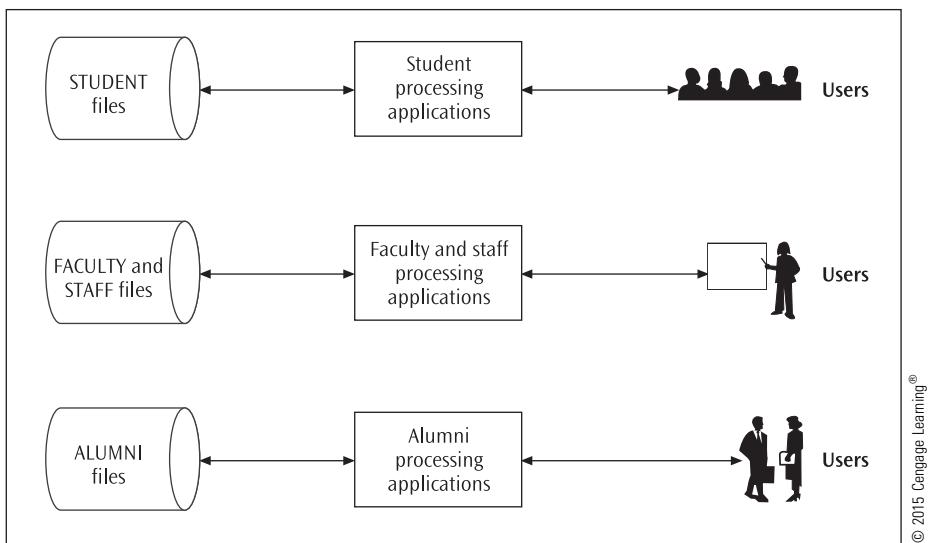
## 1.3 LIMITATIONS OF FILE-PROCESSING SYSTEMS

---

Computer applications in the 1960s and 1970s focused primarily on automating clerical tasks. These applications made use of records stored in separate files and thus were called file-processing systems. Although file-processing systems for information systems applications have been useful for many years, database technology has rendered them obsolete except for their use in a few legacy systems such as some payroll and customer

billing systems. Nonetheless, understanding their limitations provides insight into the development of and justification for database systems.

Figure 1.1 shows three file-processing systems for a hypothetical university. One processes data for students, another processes data for faculty and staff, and a third processes data for alumni. In such an environment, each file-processing system has its own collection of private files and programs that access these files.



**FIGURE 1.1** An example of a file-processing environment

While an improvement over the manual systems that preceded them, file-processing systems suffer from a number of limitations:

- **Lack of data integrity**—**Data integrity** ensures that data values are correct, consistent, complete, and current. Duplication of data in isolated file-processing systems leads to the possibility of inconsistent data. Then it is difficult to identify which of these duplicate data is correct, complete, and/or current. This creates data integrity problems. For example, if an employee who is also a student and an alumnus changes his or her mailing address, files that contain the mailing address in three different file-processing systems require updating to ensure consistency of information across the board. Data redundancy across the three file-processing systems not only creates maintenance inefficiencies, it also leads to the problem of not knowing which is the current, correct, and /or complete address of the person.
- **Lack of standards**—Organizations with file-processing systems often lack or find it difficult to enforce standards for naming data items as well as for accessing, updating, and protecting data. The absence of such standards can

lead to unauthorized access and accidental or intentional damage to or destruction of data. In essence, security and confidentiality of information may be compromised.

- *Lack of flexibility/maintainability*—Information systems make it possible for end users to develop information requirements that they had never envisioned previously. This inevitably leads to a substantial increase in requests for new queries and reports. However, file-processing systems are dependent upon a programmer who has to either write or modify program code to meet these information requirements from isolated data. This can bring about information requests that are not satisfied or programs that are inefficiently written, poorly documented, and difficult to maintain.

These limitations are actually symptoms resulting from two fundamental problems: lack of integration of related data and lack of program-data independence.

- *Lack of data integration*—Data is separated and isolated, and ownership of data is compartmentalized, resulting in limited data sharing. For example, to produce a list of employees who are students and alumni at the same time, data from multiple files must be accessed. This process can be quite complex and time consuming since a program has to access and perform logical comparisons across independent files containing employee, student, and alumni data. In short, lack of integration of data contributes to all of the problems listed previously as symptoms.
- *Lack of program-data independence*—In a file-processing environment, the structural layout of each file is embedded in the application programs. That is, the metadata of a file is fully coded in each application program that uses the particular file. Perhaps the most often-cited example of the program-data dependence problem occurred during the file-processing era, when it was common for an organization to expand the zip code field from five digits to nine digits. In order to implement this change, every program in the employee, student, and alumni file-processing systems containing the zip code field had to be identified (often a time-consuming process itself) and then modified to conform to the new file structure. This not only required modification of each program and its documentation but also recompiling and retesting of the program. Likewise, if a decision was made to change the organization of a file from indexed to random, since the structure of the file was mapped into every program using the file, every program using the file had to be modified. Identifying all the affected programs for corrective action was not a simple task, either. Thus, because of lack of program-data independence, file-processing systems lack flexibility since they are not amenable to structural changes in data. Program-data dependence also exacerbates data security and confidentiality problems.

It is only through attacking the problems of lack of program-data independence and lack of integration of related data that the limitations of file-processing systems can be

eliminated. If a way is found to deal with these problems so as to establish centralized control of data, then unnecessary redundancy can be reduced, data can be shared, standards can be enforced, security restrictions can be applied, and integrity can be maintained. One of the objectives of database systems is to integrate data without programmer intervention in a way that eliminates data redundancy. The other objective of database systems is to establish program-data independence, so that programs that access the data are immune to changes in storage structure (how the data is physically organized) and access technique.

The Time Life company experienced many of these problems in its early days. Time Life was established in 1961 as a book-marketing division. It took its name from *Time* and *Life* magazines, which at the time were two of the most popular weeklies on the market. Time Life gained fame as a seller of book series that were mailed to households in monthly installments, operating as book sales clubs. Most of the series were more or less encyclopedic in nature (e.g., *The LIFE History of the United States*, *The Time-Life Encyclopedia of Gardening*, *The Great Cities*, *The American Wilderness*, etc.), providing the basics of the subjects in the way it might be done in a series of lectures aimed at the general public. Over the years, more than 50 series were published.

During the 1970s and first half of the 1980s, Time Life exhibited all of the characteristics of a file-processing system. A separate collection of files was maintained for each book series. Thus, when the company sought to promote a new series to its existing customer base, a customer who had purchased or was currently subscribing to several book series already would receive multiple copies of the same glossy brochure promoting the new series. In addition, it was not uncommon for a customer to receive the same brochure at multiple addresses if that customer had used different mailing addresses when subscribing to different publications. In the mid-1980s, the company replaced its separate file-processing systems with an integrated database system that eliminated much of the data duplication and lack of data integrity that characterized the previous file-processing environment in which it had been operating.

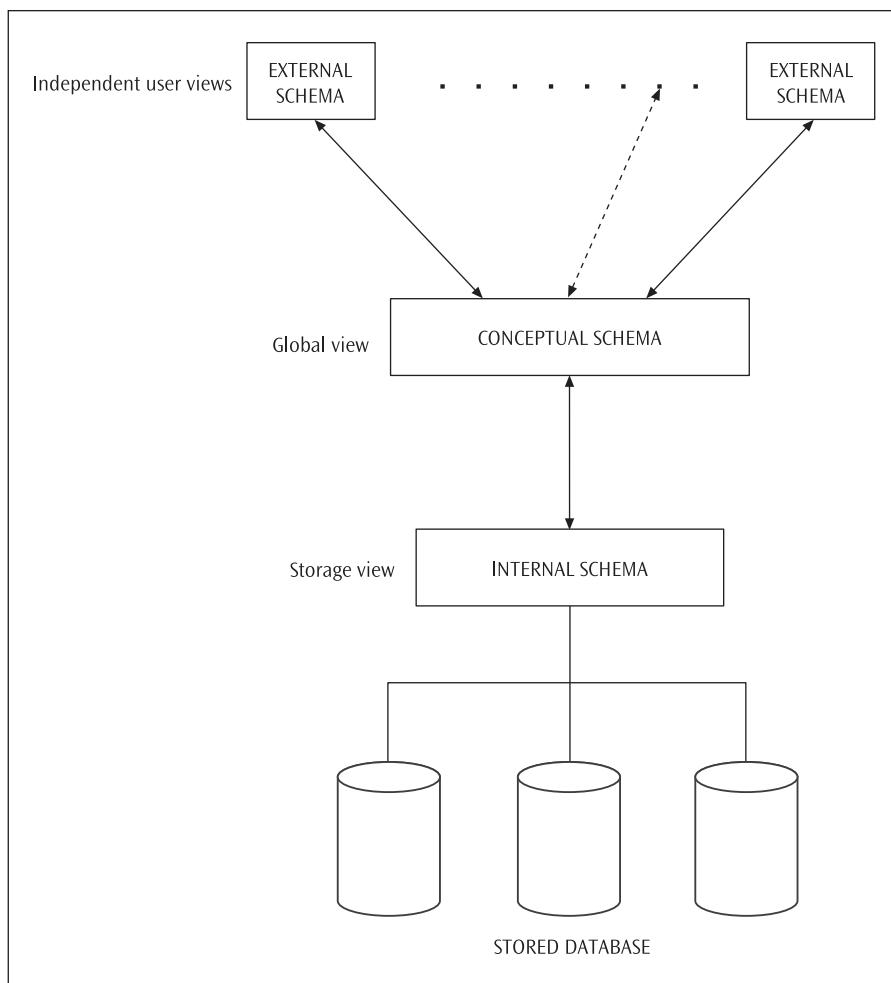
## 1.4 THE ANSI/SPARC THREE-SCHEMA ARCHITECTURE

---

In the 1970s, the Standards Planning and Requirements Committee (SPARC) of the American National Standards Institute (ANSI) offered a solution to these problems by proposing what came to be known as the **ANSI/SPARC three-schema architecture**.<sup>2</sup> The ANSI/SPARC three-schema architecture, as illustrated in Figure 1.2, consists of three perspectives of metadata in a database. The conceptual schema is the nucleus of the three-schema architecture. Located between the external schema and internal schema, the **conceptual schema** represents the global conceptual view of the structure of the entire database for a community of users. By insulating applications/programs from changes in physical storage structure and data access strategy, the conceptual schema achieves program-data independence in a database environment.

---

<sup>2</sup>In a database context, the word “schema” stands for “description of metadata.”



**FIGURE 1.2** The ANSI/SPARC three-schema architecture

The **external schema**<sup>3</sup> consists of a number of different user **views**<sup>4</sup> or subschemas, each describing portions of the database of interest to a particular user or group of users. The conceptual schema represents the global view of the structure of the entire database for a community of users. The conceptual schema is the consolidation of user views. The data specification (metadata) for the entire database is captured by the conceptual

<sup>3</sup>While an external schema is technically a collection of external subschemas or views, the term “external schema” is used here in the context of either an individual user view or a collection of different user views.

<sup>4</sup>Informally, a “view” is a term that describes the information of interest to a user or a group of users, where a user can be either an end user or a programmer. See Chapter 6 (Section 6.4) for a more precise definition of a “view.”

schema. The **internal schema** describes the physical structure of the stored data (how the data is actually laid out on storage devices) and the mechanism used to implement the access strategies (indexes, hashed addresses, and so on). The internal schema is concerned with efficiency of data storage and access mechanisms in the database. Thus, the internal schema is technology dependent, while the conceptual schema and external schemas are technology independent. In principle, user views are generated on demand through logical reference to data items in the conceptual schema independent of the logical or physical structure of the data.

### 1.4.1 Data Independence Defined

Data independence is the central concept driving a database system, and the very purpose of a three-schema architecture is to enable data independence. The theme underlying the concept of data independence is that *when a schema at a lower level is changed, the higher-level schemas themselves are unaffected by such changes*. In other words, when a change is made to storage structure or access strategy in the internal schema, there will be no need to make any changes in the conceptual or external schemas; only the mapping information—i.e., transforming requests and results between levels of schema—between a schema and higher-level schemas need to be changed. Only then can it be said that data independence is fully supported.

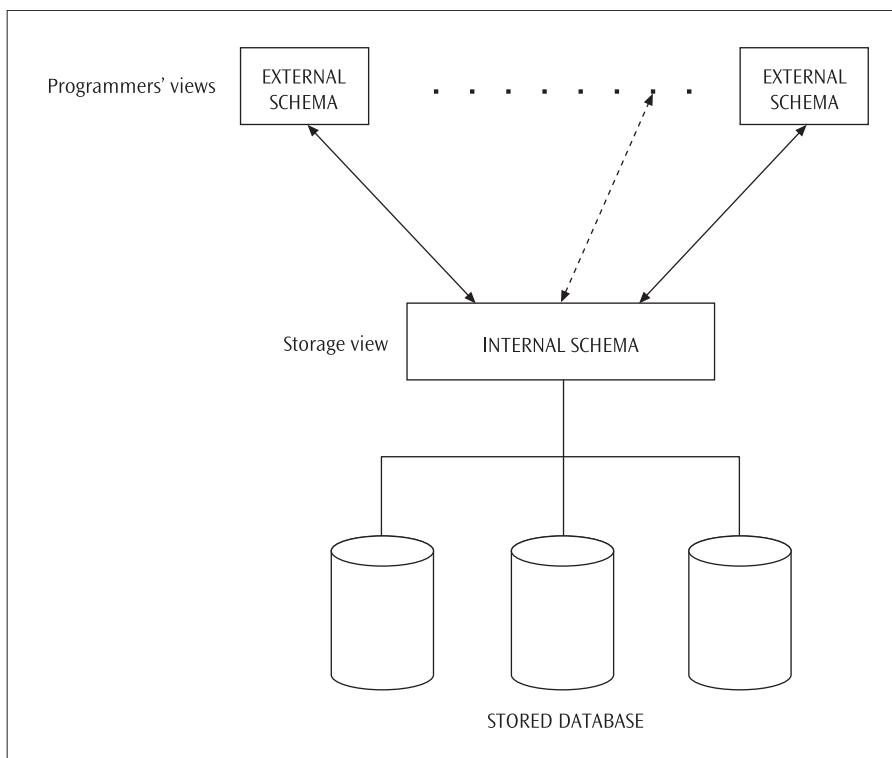
For instance, suppose direct access to data ordered by zip code is required. This may be recorded as “direct access” in the conceptual schema, and a certain type of indexing technique may be employed in the internal schema. This fact will be available as the mapping information so that if/when the indexing technique in the internal schema is changed, only the mapping information gets changed, and the conceptual schema is unaffected. Incidentally, the external views are completely shielded from even the knowledge of this change in the internal schema. That is, the specification and implementation of a change in the indexing mechanism on zip code does not require any modification and testing of the application programs that use the external views containing zip code.

This capacity to change the internal schema without having to change the conceptual or external schema is sometimes referred to as **physical data independence**. The internal schema may be changed when certain file structures are reorganized or new indexes are created to improve database performance. The physical data independence enables implementation of such changes without requiring any corresponding changes in the conceptual or external schemas.

Likewise, enhancements to the conceptual schema in the form of growth or restructuring will have no impact on any of the external views (subschemas) since all external views are spawned from the conceptual schema only by logical reference to elements in the conceptual schema. For instance, redefinition of logical structures of a data model (such as adding or restructuring tables in a relational database) may sometimes be in order. Since the external views (subschemas) are generated exclusively by logical references, the user views are immune to such logical design changes in the conceptual schema. This property is often called **logical data independence**. Logical data independence also enables a user (external) view to be immune to changes in the other user views.

A file-processing system, in contrast, may be viewed as a two-schema architecture consisting of the internal schema and the programmer’s view (external schema), as shown

in Figure 1.3. Here, the programmer's view corresponds to the physical structure of the data, meaning that the physical structure of data (internal schema) is fully mapped (incorporated) into the application program. The file-processing system lacks program-data independence because any modification to the storage structure or access strategy in the internal schema necessitates changes to application programs and subsequent recompilation and testing. In the absence of a conceptual schema, the internal schema structures are necessarily mapped directly to external views (or subschemas). Consequently, changes in the internal schema require appropriate changes in the external schema; therefore, data independence is lost. Because changes to the internal schema, such as incorporating new user requirements and accommodating technological enhancements, are expected in a typical application environment, absence of a conceptual schema essentially sacrifices data independence. In short, file-processing systems lack data independence because they employ what amounts to a two-schema architecture.

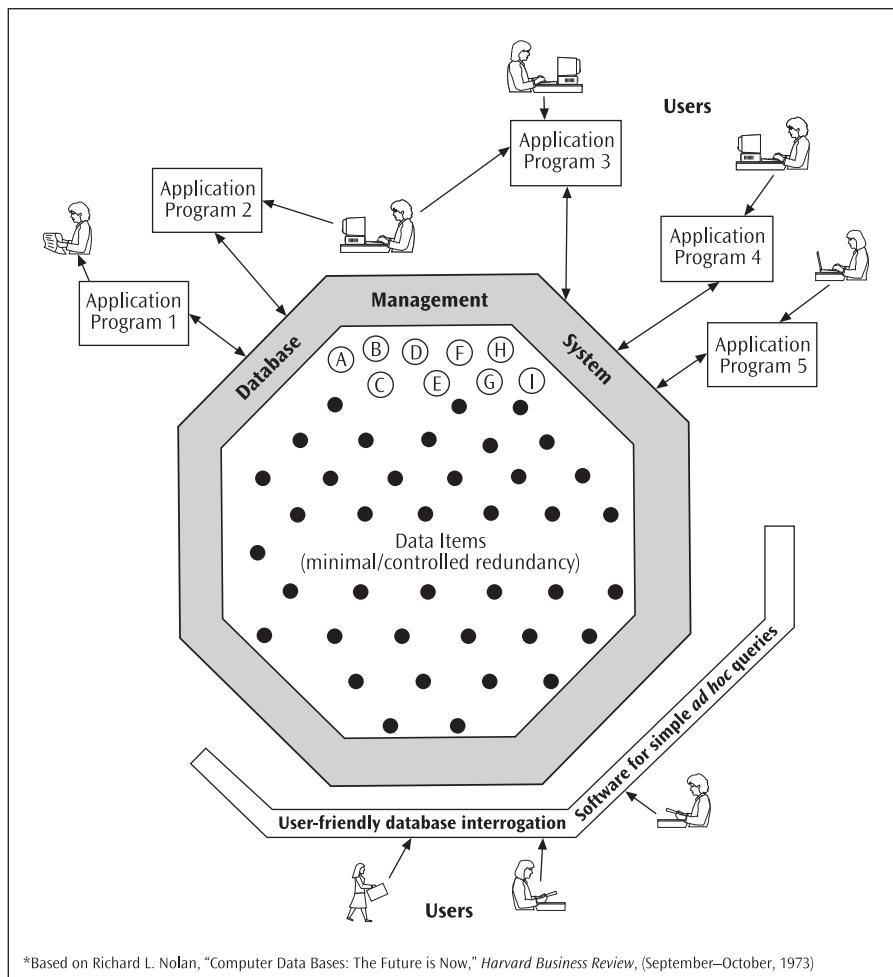


**FIGURE 1.3** The file-processing system: a two-schema architecture

The three-schema architecture described in this section is required to achieve data independence. It is worthwhile to remind the reader at this point that the conceptual schema, external schema, and internal schema are essentially expressions of metadata in a hierarchical transition from technology-independent state to technology-dependent state. The complete data modeling and design process is about modeling metadata.

## 1.5 CHARACTERISTICS OF DATABASE SYSTEMS

Database systems seek to overcome the two root causes of the limitations that plague file-processing systems by creating a single integrated set of files that can be accessed by all users. This integrated set of files is known as a **database**. A database management system (typically referred to as a DBMS) is a collection of general-purpose software that facilitates the processes of defining, constructing, and manipulating a database for various applications. Figure 1.4 provides a layman's view of the difference between a database and a database management system. This illustration shows how neither a user nor a programmer is able to access data in the database without going through the database management system software. Whether a program is written in Java, C, COBOL, or some other language, the program must "ask" the DBMS for the data, and the DBMS will fetch the data. SQL (Structured Query Language) has been established as the language for accessing data in a database by the International Organization for Standardization (ISO) and the American National Standards Institute (ANSI). Accordingly, any application program that seeks to access a database must do so via embedded SQL statements.



**FIGURE 1.4** An early view of a database system

An important purpose of this book is to discuss how to organize the data items conceptualized in Figure 1.4. In reality, data items do not exist in one big pool surrounded by the database management system. Several different architectures exist for organizing this data. One is a hierarchical organization, another is a network organization, and a third is relational; in this book, the relational approach is emphasized.<sup>5</sup> While the data items that collectively comprise the database at the physical level are stored as sequential, indexed, and random files, the DBMS is a layer on top of these files that frees the user and application programs from the burden of knowing the structures of the physical files (unlike a file-processing system).

Next, let us look more closely at what constitutes a database, a database management system, and finally a database system.

### 1.5.1 What Is a Database System?

A system is generally defined as a set of interrelated components working together for some purpose. A **database system** is a self-describing collection of interrelated data. A database system includes data and metadata. Here are the properties of a database system:

- Data consists of recorded facts that have implicit meaning.
- Viewed through the lens of metadata, the meaning of recorded data becomes explicit.
- A database is self-describing in that the metadata is recorded within the database, not in application programs.
- A database is a collection of files whose records are logically related to one another. In contrast with that of a file-processing system, integration of data as needed is the responsibility of the DBMS software instead of the programmer.
- Embedded pointers and various forms of indexes exist in the database system to facilitate access to the data.

A database system may be classified as single-user or multi-user. A single-user database system supports only one user at a time. In other words, if user A is using the database, users B and C must wait until user A has completed his or her database work. When a single-user database runs on a personal computer, it is also called a desktop database system. In contrast, a multi-user database system supports multiple users concurrently. If the multi-user database supports a relatively small number of users (usually fewer than 50) or a specific workgroup within an organization, it is called a workgroup database system. If the database is used by the entire organization and supports many users (more than 50, usually hundreds) across many locations, the database is known as an enterprise database system.

The term “enterprise database system” is somewhat misleading. In the early days of database processing, the goal was to have a single database for the entire organization.

---

<sup>5</sup>Two relatively new data modeling architectures (the object-oriented data model and the object-relational model) also exist. Appendix B briefly discusses each of these architectures. Appendix A reviews architectures based on the hierarchical and network organizations.

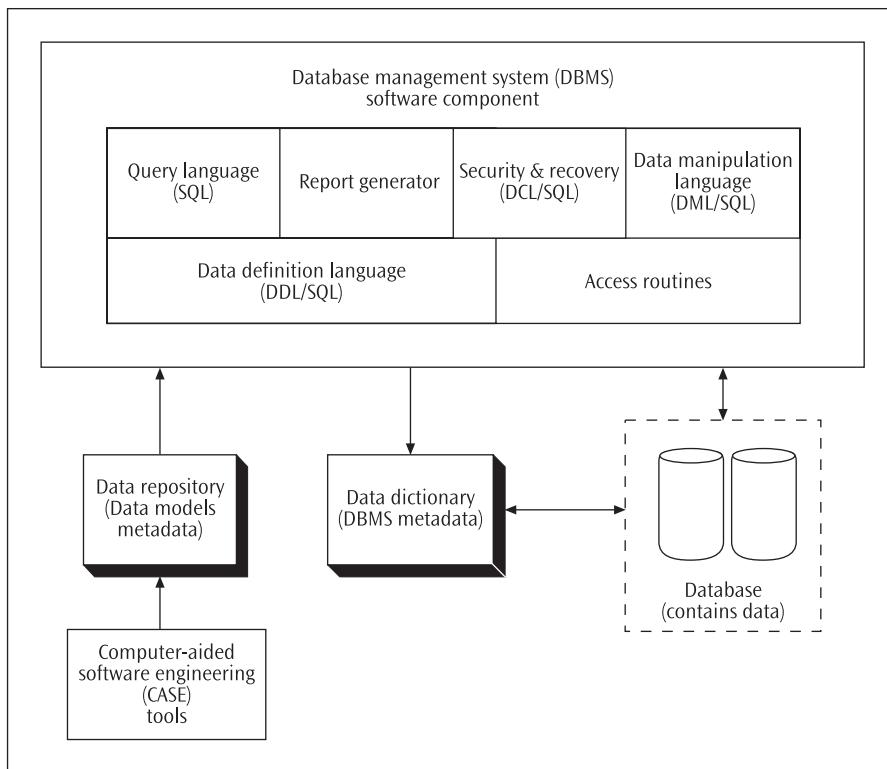
While this type of database is possible in a small organization, in large organizations multiple databases exist that are indeed used enterprise-wide. For example, large oil companies have databases organized by function: one database for exploration, one database for refining, another for marketing, a fourth for royalty payments, and so on. On the other hand, a consumer-product company might have several product databases. Each one of these is an enterprise database system since its use extends enterprise-wide.

A natural extension to the enterprise database system is the concept of distributed database systems. With the tremendous strides made in network and data communication technology in the last two decades, distribution of databases over a wide geographic area has become highly feasible. A **distributed database (DDB)** is a collection of multiple logically interrelated databases that may be geographically dispersed over a computer network. A **distributed database management system (DDBMS)** essentially manages a distributed database while rendering the geographical distribution of data transparent to the user community. The advent of DDBs facilitated replacement of the large, centralized, monolithic databases of the 1980s with decentralized autonomous database systems inter-related via a computer network.

Another important trend that emerged in the 1990s is the development of data warehouses. The distinguishing characteristic of a **data warehouse** is that it is mainly intended for decision-support applications used by knowledge workers. As a consequence, data warehouses are optimized for information retrieval rather than transaction processing. By definition, a data warehouse is subject-oriented, integrated, nonvolatile, and time-variant. Since its relatively recent inception, data warehousing has evolved rapidly in large corporations to support business intelligence, data mining, decision analytics, and customer relations management (CRM).

### 1.5.2 What Is a Database Management System?

Figure 1.5 illustrates the components of a database system, consisting of the DBMS, database, data dictionary, and data repository. A **database management system (DBMS)** is a collection of general-purpose software that facilitates the processes of defining, constructing, and manipulating a database. The major components of a DBMS include: one or more query languages; tools for generating reports; facilities for providing security, integrity, backup, and recovery; a data manipulation language for accessing the database; and a data definition language used to define the structure of data. As shown in Figure 1.5, Structured Query Language (SQL) plays an integral role in each of these components. SQL is used in the **data definition language (DDL)** for creating the structure of database objects such as tables, views, and synonyms. SQL statements are also generated by programming languages used to build reports in order to access data from the database. In addition, people involved in the data administration function use **data control languages (DCLs)** that make use of SQL statements to (a) control the resource locking required in a multi-user environment, (b) facilitate backup and recovery from failures, and (c) provide the security required to ensure that users access only the data that they are authorized to use.



**FIGURE 1.5** Components of a database system

**Data manipulation languages (DMLs)** facilitate the retrieval, insertion, deletion, and modification of data in a database. SQL is the most well-known nonprocedural<sup>6</sup> DML and can be used to specify many complex database operations in a concise manner. Most DBMS products also include procedural language extensions to supplement the capabilities of SQL, such as Oracle PL/SQL. Other examples of procedural language extensions are languages such as C, Java, Visual Basic, and COBOL, in which pre-compilers extract data manipulation commands written in SQL from a program and send them to a DML compiler for compilation into object code for subsequent database access by the run-time subsystem.<sup>7</sup> Finally, the access routines handle database access at run time by passing requests to the file manager of the operating system to retrieve data from the physical files of the database.

Much as a dictionary is a reference book that provides information about the form, origin, function, meaning, and syntax of words, a **data dictionary** in a DBMS environment

<sup>6</sup>SQL is known to be a nonprocedural language since it only specifies what data to retrieve as opposed to specifying how actually to retrieve it. A procedural language specifies how to retrieve data in addition to what data to retrieve.

<sup>7</sup>The run-time subsystem of a database management system processes applications created by the various design tools at run time.

stores metadata that provides such information as the definitions of the data items and their relationships, authorizations, and usage statistics. The DBMS makes use of the data dictionary to look up the required data component structures and relationships, thus relieving application developers (end users and programmers) from having to incorporate data structures and relationships in their applications. In addition, any changes made to the physical structure of the database are automatically recorded in the data dictionary. This removes the need to modify application programs that access the modified structure.

As a simple illustration of what constitutes a data dictionary, consider a database for a university with four tables (i.e., data sets) of user data: a STUDENT table, an ADVISOR table, a COURSE table, and an ENROLLMENT table. Assume that the STUDENT table contains one row for each of the 40,000 students, one row for each of the 500 academic advisors in the ADVISOR table, one row for each of the 2,000 courses listed in the COURSE table, and one row each for the enrollment of a student in a course in the ENROLLMENT table. As shown in the following table, the DB\_TABLES data dictionary table<sup>8</sup> would include one row for each table containing user data. While the DB\_TABLES data dictionary table shown here (see Table 1.2) contains only the table name, number of columns, number of rows, and primary key for each table of user data, a data dictionary table comparable to DB\_TABLES in a commercial database management system product might have several dozen columns of data about the STUDENT table.

Table Name	Number of Columns	Number of Rows	Primary Key
STUDENT	4	40000	Student_Number
ADVISOR	3	500	Advisor_Name
COURSE	3	2000	Course_Number
ENROLLMENT	3	160000	(Student_Number, Course_Number)

**TABLE 1.2** The data dictionary table, DB\_TABLES

The DB\_TABLES data dictionary table indicates that the STUDENT table contains four columns but, other than indicating that the Student\_Number column is the primary key, does not contain any information about any of the other column names. However, this data is contained in the following data dictionary table (see Table 1.3) as DB\_TABLES\_COLUMNS. DB\_TABLES\_COLUMNS contains one row for each column in each of the four tables STUDENT, ADVISOR, COURSE, and ENROLLMENT. As was the case for DB\_TABLES, a data dictionary table comparable to DB\_TABLES\_COLUMNS in a commercial database management systems product might have several dozen columns of data about each column in each table of user data.

---

<sup>8</sup>Data dictionary tables are typically accessed via built-in views.

Column Name	Table Name	Data Type	Length
Student_Number	STUDENT	Integer	4
First_Name	STUDENT	Text	20
Last_Name	STUDENT	Text	30
Major	STUDENT	Text	10
Advisor_Name	ADVISOR	Text	25
Phone	ADVISOR	Text	12
Department	ADVISOR	Text	15
Course_Number	COURSE	Integer	4
Title	COURSE	Text	10
Number_Hours	COURSE	Decimal	4
Student_Number	ENROLLMENT	Integer	4
Course_Number	ENROLLMENT	Integer	4
Grade	ENROLLMENT	Text	2

**TABLE 1.3** The data dictionary table, DB\_TABLES\_COLUMNS

It is important to note that the content of these data dictionary tables is updated whenever a change is made to the database. For example, if a PROFESSOR table that consists of five columns is added to the database, then one row would be inserted into the DB\_TABLES data dictionary table and five rows would be inserted into the DB\_TABLES\_COLUMNS data dictionary table. Each time a row is inserted into the PROFESSOR table, the Number of Rows column in the DB\_TABLES data dictionary table would be incremented by one.

While not a component of the DBMS per se, the data repository has become an integral part of the data management suite of tools. The **data repository** is a collection of metadata about data models and application program interfaces. CASE (computer-aided software engineering) tools such as Oracle Designer and ERWIN that are used for developing a conceptual/logical schema<sup>9</sup> interact with the data repository and are independent of the database and the DBMS.

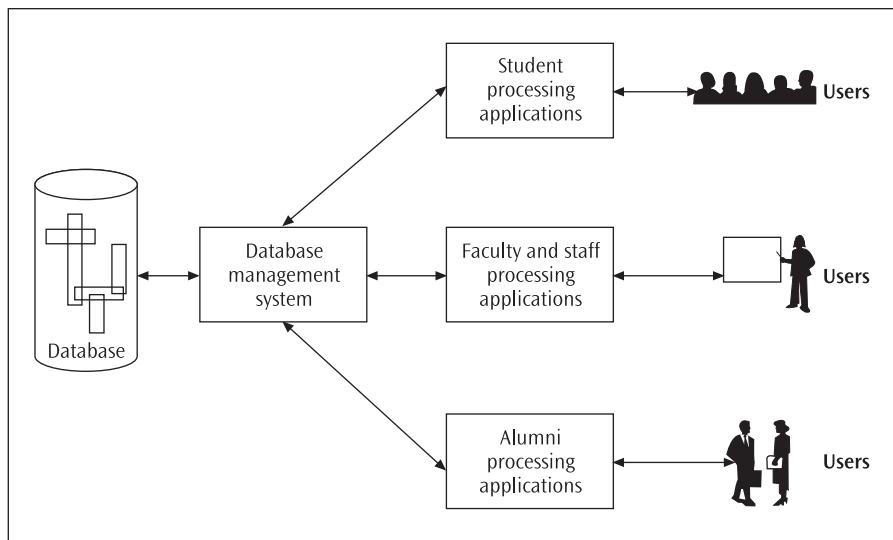
### 1.5.3 Advantages of Database Systems

A database system is comprised of a database, general-purpose software called the database management system that creates and manipulates the database, a data dictionary, and appropriate hardware. While an organization could write its own special-purpose

---

<sup>9</sup>The data modeling activity includes the development of the conceptual and logical schemas. The role of the conceptual schema in database design is discussed in Section 1.6.

database management system software to create and manipulate the database, a database management system is usually purchased from a software vendor such as IBM, Microsoft, or Oracle. Figure 1.6 illustrates how a database system for the hypothetical university introduced earlier might be structured.



**FIGURE 1.6** An example of a database system

As mentioned previously, in file-processing systems, access to information is achieved through programs written in conventional programming languages by application programmers. In contrast, database systems contain a variety of tools such as query languages and report writers that make it possible for end users to access and make use of much of the information needed for analysis and decision-making.

Data integrity ensures that data is consistent, accurate, and reliable. A properly designed database system allows data integrity to be achieved through controlling redundancy. As we will see throughout this book, while database systems do not eliminate redundancy (that is, storing the same piece of data in more than one place), they can—in fact they must, through adherence to business rules that take the form of constraints—control unnecessary and harmful redundancies that often accompany file-processing systems.

Recall that file-processing systems are plagued by lack of integration of data and lack of program-data independence. The reason for program-data dependence is that data structures used to physically store data must be referred to directly by the application program. Database systems do not suffer from this limitation as data structures are stored, along with the data, in the database as opposed to being recorded in the application program. The DBMS accesses the database and supplies data appropriate to the needs of individual application programs. The DBMS accomplishes this using the data dictionary where the metadata is recorded. Thus the data dictionary in a database environment insulates application programs from changes to the structure of data.

The notion of data relatability involves the creation of logical relationships between different types of records, usually in different files. In a file-processing environment, information often cannot be generated without a programmer writing or at least modifying an application program to consolidate the files. With the advent of database systems, all that is necessary is for one to specify the data to be combined; the DBMS will perform the necessary operations to accomplish the task.

## 1.6 DATA MODELS

---

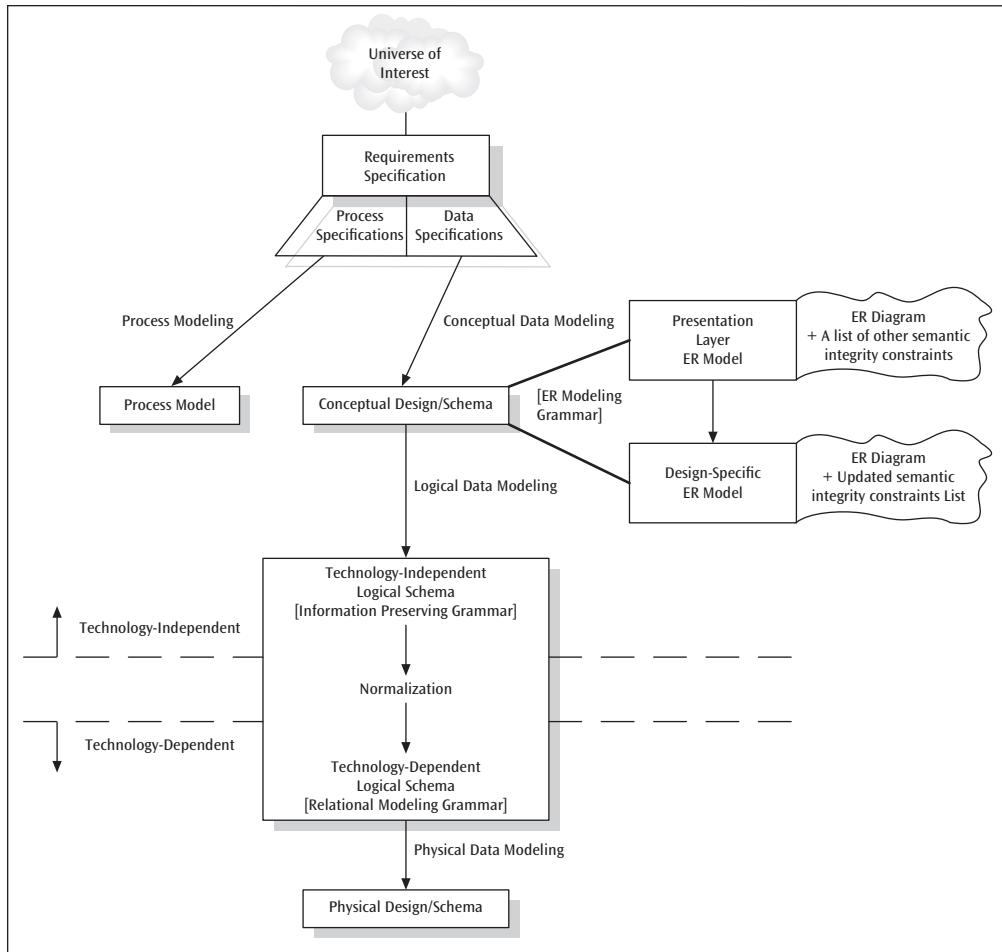
A model is a simplified expression of observed or unobservable reality used to perceive relationships in the real world. Models are used in many disciplines and can take a variety of forms. Some are small-scale physical representations of reality, like model aircraft in a wind tunnel. Others take the form of mathematical models that use a set of mathematical symbols and logic to describe the behavior of a system—such as econometric models, optimization models, and statistical models. In the mathematical computing field, directed graphs (digraphs) are used to model relationships among variables and essentially represent an analog model. Likewise, in the database design arena, a **data model** is used to represent real-world phenomena. Data modeling is considered the most important part of the database design process.

A drawing of a chair is essentially a model of the real-world object “chair.” A chair constructed using Legos is also a model of the real-world object “chair.” However, modeling an intangible real-world object that cannot be physically grasped, such as a department, requires a different approach—one that makes use of descriptors. For example, a department can be described by specifying that it has a name, can have multiple locations, has a number, has a budget, has a certain number of employees, and the like. This is an example of a data model that forms a conceptual expression of the intangible real-world object “department” using descriptors. Objects, tangible as well as intangible, such as employees, inventory, sales, and projects, can be modeled in a similar manner.

As simplified abstractions of reality, data models enable better understanding of data specifications, such as data types, relationships, and constraints gathered from user requirements. Data models serve as the blueprints for designing databases. Just as one would not build a house from incomplete blueprints with known errors, it is imperative that a data model be a complete and accurate reflection of the data requirements of a database system.

### 1.6.1 Data Models and Database Design

Figure 1.7 illustrates the role of data models in the phases of database design. A database represents some aspect of the real world called the *universe of interest*. For a small organization, the universe of interest may be all functionalities of the company (marketing, finance, accounting, production, human resources, and so on). A large oil company may have islands of interest centered around functions such as exploration, refining, and marketing, and a separate database designed for each function. A large consumer-product company might be oriented around product groups and have separate databases for different product groups.



**FIGURE 1.7** Data modeling/database design life cycle

The initial step in the design process is the **requirements specification**. During this step, systems analysts review existing documents and systems and interview prospective users in an effort to identify the objectives to be supported by the database system. The output of the requirements specification activity is a set of data and process specifications. This is essentially an organized conglomeration of user-specified restrictions on the organization's activities (business processes) that must be reflected in the database and/or database applications. Such restrictions are commonly referred to as **business rules**.

In order to define the data requirements, one needs to know the process requirements—that is, what is going to be done with the data. For example, suppose a company is going to sell a product. What processes are involved? When a company sells a product, it bills the customers who purchase the product. Then, shipping has to be notified to dispatch the product to the customer. Shipping also has to check the inventory and make sure that inventory levels are adjusted as a result of sales. The inventory system must make sure that inventory levels are optimal and, accordingly, replenish inventory periodically.

Data is required in order to accomplish processes such as those just mentioned. Customers' names, addresses, and telephone numbers are needed for billing purposes. For shipping, a shipping address is required. In the inventory system one needs to know in which warehouse and in which bin a particular product is located. One also needs to know quantity on hand, quantity on order, and the lead time required by the supplier to fill an order. In short, data and process requirements go hand-in-hand. This book assumes that the requirements specification has been done, and separates out the data requirements from the process requirements.<sup>10</sup> The data requirements will be narrated in the form of stories and vignettes. For example, Chapters 3, 4, and 5 contain stories that provide the data requirements for two hypothetical organizations: Bearcat Incorporated and Cougar Medical Associates.

## 1.6.2 Data Modeling and Database Design in a Nutshell

Data modeling and database design follow a life cycle that includes three tiers: conceptual data modeling, logical data modeling, and physical data modeling. As shown in Figure 1.7, this book is organized to follow this progression. Let us use a vignette to demonstrate in a nutshell the complete data modeling and database design life cycle this book is all about:

*Clients use law firms to file and/or defend lawsuits in which they participate. Law firms employ lawyers; however, some lawyers simply engage in private practice. Clients are represented by lawyers who handle lawsuits for these clients.*

Now that we have set up the scenario, let us proceed with the data modeling process.

### 1.6.2.1 Conceptual Data Modeling

The conceptual data model describes the structure of the data to be stored in the database without specifying how and where it will be physically stored or the methods used to retrieve it. The conceptual design activity is technology independent. During the conceptual design, the focus should be on capturing the user-specified business rules in all their richness, unconstrained by the boundaries of the anticipated technology or DBMS product that will be used for implementation. The product of the conceptual design activity is the conceptual schema.

Several conceptual data modeling methods exist (for example, ER modeling and NIAM modeling<sup>11</sup>), each with its own specific grammar. This book uses the ER modeling technique due to its significant popularity in the database design sphere. The conceptual model is portrayed in two progressive layers: the Presentation Layer ER model and the Design-Specific ER model. The Presentation Layer ER model is intended for user-analyst communication and entails the database analyst working with the user community and

---

<sup>10</sup>The early technique for expressing process requirements was “data flow diagrams.” The current methods include UML (Unified Modeling Language) and BPMN (Business Process Modeling Notation). An examination of approaches for identifying process requirements is usually part of systems analysis and design and thus is not discussed here.

<sup>11</sup>Entity relationship (ER) modeling is a “design by analysis” modeling approach and is top-down in nature, while NIAM (Nijsen Information Analysis Methodology) modeling is a “design by synthesis” approach and is bottom-up in nature.

developing a script (a Presentation Layer ER diagram) accompanied by a set of semantic integrity constraints.<sup>12</sup> Next, the Presentation Layer ER model (both the ER diagram and semantic integrity constraints) is mapped (translated) to the Design-Specific ER model. The Design-Specific ER model incorporates technical details needed for the final database design but not quite necessary for interactions with the user community; this is not the right tool for user communication—it is intended for interaction with the database designer. Chapters 2, 3, 4, and 5 discuss these ER models in detail.

A conceptual data model is supposed to capture all business rules that are implicitly or explicitly present in the requirements specification. Simply going through a detailed process of representing the requirements in a conceptual model does not assure that the resulting conceptual design captures all business rules. Therefore, it is imperative that a validation step is included as the concluding activity in the conceptual modeling phase to systematically verify that the conceptual design developed indeed answers all the questions stated implicitly or explicitly in the requirements specification. Where the validation fails, the model should be refined accordingly. Since the ER modeling grammar is used for the conceptual design activity in this book, validation of an ER model must be addressed. Validating an ER model is discussed at the end of Chapter 5.

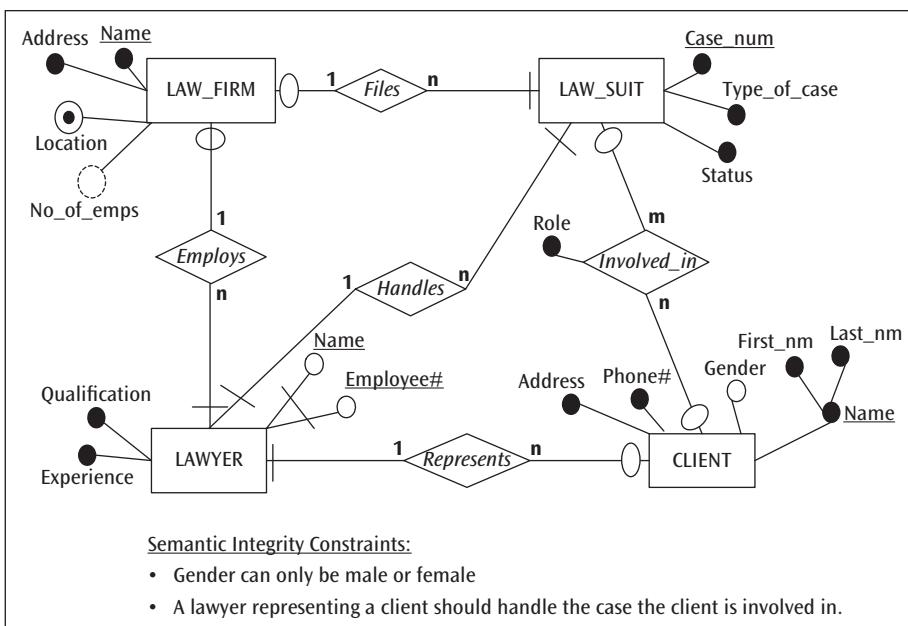
Let us now further develop the vignette introduced at the beginning of this section and use the ER modeling grammar to capture the scenario described in the vignette as a conceptual data model.

*Law firms are perhaps identified by their respective names and may contain other descriptors like main address, locations, specialty, number of employees, etc. In this modeling example, let us just use a few descriptors for all the participating entities. Clients typically have a name, gender, address, phone number, etc. and can be identified by their respective names. Likewise, lawyers are perhaps identified by their respective names, even though the ones working for a law firm may also have unique employee numbers of some sort. Some of the other attributes of a lawyer may include: qualification, experience, etc. Lawsuits may each have a unique case number assigned by the court along with attributes like type of case, current status, and numerous other details about the case. Other simple as well as complex business rules pertaining to the scenario will often be present in the requirements specification; the conceptual modeling grammar (here, ER modeling grammar) may not have constructs to capture some of these simple/complex business rules. “Gender can only be male or female” is a simple business rule that cannot be captured in an ER diagram. “A lawyer representing a client should handle the case the client is involved in” is an example of a complex business rule that an ER diagram cannot capture.*

The conceptual data model for this scenario expressed using the ER modeling grammar is displayed in Figure 1.8a. First, it must be noted that the ER model has (a) a diagram and (b) a list of Semantic Integrity Constraints (SIC list). The diagram follows the ER modeling grammar, and the SIC list includes business rules that the ER diagram (ERD) is incapable of recording.

---

<sup>12</sup>It is not possible to capture all business rules contained in a requirements specification in the ER diagram. Semantic integrity constraints are business rules that are not captured in the ER diagram.



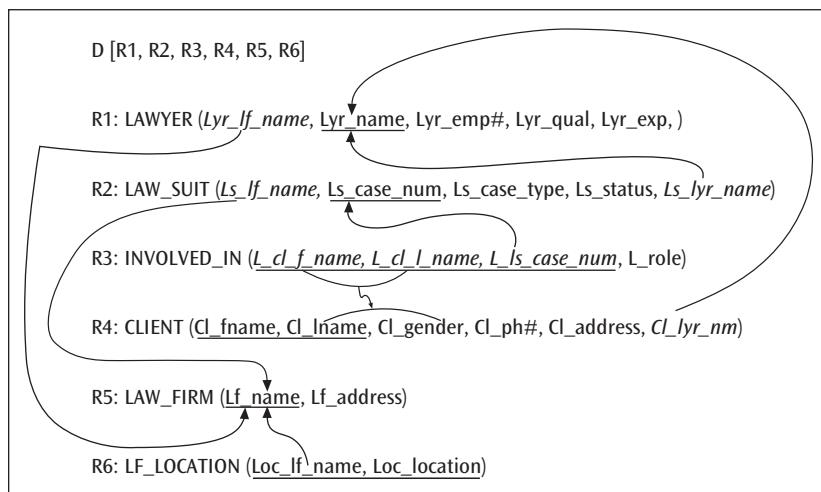
**FIGURE 1.8a** ER model for a simple legal system

The ERD is a pictorial representation of the business rules reflected in the requirements specification. As stated earlier in this section, ER modeling is covered in Chapters 2, 3, 4, and 5. So, here we can use a simple intuitive interpretation of the ERD. LAW\_FIRM, LAWYER, CLIENT, and LAW\_SUIT appear as entity units, with specific attributes describing each of them (see Figure 1.8a). The diamonds represent specific associations among these entity units. One can see the user-friendliness of this diagram and appreciate how it can be used by systems analysts as a tool to communicate with the user community about how the user requirements are being modeled for ultimate implementation as a database system.

### 1.6.2.2 Logical Data Modeling

As a consequence of the technology-independent orientation of the conceptual design activity, it is possible that the conceptual schema may contain constructs not directly compatible with the technology intended for implementation. It is also possible that some of the design may require refinement to eliminate data redundancy problems. Transforming a conceptual schema to a schema more compatible with the implementation technology of choice becomes necessary. Thus, the second tier of the data modeling activity is called logical design. The product of the logical design activity is the logical schema, which evolves from a technology-independent representation to a technology-dependent representation. The latter is typically modeled using the hierarchical, network, or relational architecture. The relational model forms the basis for most of the material in this textbook. Discussion of characteristics of the other models is confined to the appendices. Logical design and the role of normalization are discussed in detail in Chapters 6 through 9.

The relational schema for the vignette mapped from the ER model in Figure 1.8a is displayed in Figure 1.8b. It should be obvious to the reader that the relational schema is certainly a user-friendly rendition of the metadata specification and cannot be used for communication with the user community. The relational schema here is the database design using the relational architecture and is intended for communication with the database designer/architect/administrator. From an intuitive stance, one can see that, for starters, each entity unit in the ER is mapped to a relation schema. All associations except one among these entity units are marked by an expression of connectivity between attributes in the respective entity units using simple arrow marks. The relationship *Involved\_in*, however, is expressed as a relation schema, due to the nature of the association between CUSTOMER and LAW\_SUIT (m:n). Please note that the other three diamonds convey an association of (1:n or n:1). Likewise, the descriptor Location of LAW\_FIRM, indicating multiple locations (see Figure 1.8a), is mapped as a separate relation schema in the logical model.



**FIGURE 1.8b** Relational schema for the ER model in Figure 1.8a

### 1.6.2.3 Physical Data Modeling

Once a logical schema has been developed, the process moves on to the physical design tier. Here, the major task is to determine the internal storage structure and access strategies for the database. While innovations in storage technology abound in the marketplace, the robust conceptual strategies for storage structures/architectures remain the ideas of parallelizing disk access employing what is popularly known as RAID (Redundant Array of Independent Disks) access. Briefly, the concept is centered on using a large array of independent disk storage devices to act as a single high-performance “logical disk.” Parallelism across the array of disks is achieved using a technique called “data striping.” Since the focus of this book is on in-depth coverage of data modeling and database design, physical database design is outside its scope. Nonetheless, it provides an extensive coverage of physical design-compatible specification for the relational database architecture; this material appears in Chapters 10 through 13.

The transition from a logical schema to a physical design entails an intermediate step of transforming the logical schema to a database language. In the relational database architecture, this language is called SQL. For a quick preview of SQL, Figure 1.8c presents the relational schema displayed in Figure 1.8b in SQL—more precisely, using the Data Definition Language (DDL) subset of SQL.

```

CREATE TABLE lawyer
(Lyr_lf_name    char (45),
Lyr_name        char (30) CONSTRAINT pk_lawyer PRIMARY KEY,
Lyr_emp#        char (11) CONSTRAINT unq_emp# UNIQUE,
Lyr_qual        char (17) CONSTRAINT nn_qual NOT NULL,
Lyr_exp         smallint CONSTRAINT nn_exp NOT NULL
);

CREATE TABLE law_firm
(Lf_name         char (45) CONSTRAINT pk_lawyer PRIMARY KEY,
Lf_address      char (30) CONSTRAINT nn_qual NOT NULL,
);
;

CREATE TABLE law_suit
(Ls_lf_name     char (50) CONSTRAINT fk_law_firm REFERENCES law_firm
(Lf_name)
ON DELETE SET NULL ON UPDATE CASCADE,
Ls_case_num    char (17) CONSTRAINT pk_lawsuit PRIMARY KEY,
Ls_case_type   char (11) CONSTRAINT nn_case_type NOT NULL,
Ls_status       char (7)  CONSTRAINT nn_status NOT NULL,
Ls_lyr_name    char (30) CONSTRAINT nn_lyr_name NOT NULL
CONSTRAINT fk_lawyer REFERENCES lawyer (Lyr_name)
ON DELETE SET DEFAULT ON UPDATE CASCADE,
);
;

CREATE TABLE client
(Cl_fname       char (15) CONSTRAINT nn_fname NOT NULL,
Cl_lname        char (23) CONSTRAINT nn_lname NOT NULL,
Cl_gender       char (1)  CONSTRAINT nn_gender NOT NULL
CONSTRAINT chk_gender CHECK (cl_gender IN ('M', 'F')),
Cl_ph#          number (10) CONSTRAINT nn_ph# NOT NULL,
Cl_address      char (30) CONSTRAINT nn_address NOT NULL,
Cl_lyr_nm       char (30) CONSTRAINT nn_lyr_nm NOT NULL
CONSTRAINT fk_cl_lawyer REFERENCES lawyer (Lyr_name)
ON DELETE SET DEFAULT ON UPDATE CASCADE,
CONSTRAINT pk_client PRIMARY KEY (Cl_fname, Cl_lname)
);
;
```

**FIGURE 1.8c** DDL/SQL for the relational schema in Figure 1.8b

```

CREATE TABLE lf_location
(Loc_lf_name      char (45),
Loc_location      char (30),
CONSTRAINT pk_location PRIMARY KEY (Loc_lf_name, Loc_location)
);

CREATE TABLE involved
(L_cl_f_name      char (15),
L_cl_l_name      char (23),
L_role           char (5) CONSTRAINT nn_role NOT NULL,
L_ls_case_num    char (17) CONSTRAINT nn_ls_case_num not null
CONSTRAINT fk_case_num REFERENCES law_suit (Ls_case_num)
ON DELETE CASCADE ON UPDATE RESTRICT,
CONSTRAINT fk_client FOREIGN KEY (L_cl_f_name, L_cl_l_name)
REFERENCES client (cl_f_name, cl_l_name) ON DELETE CASCADE ON UPDATE CASCADE
);

```

**FIGURE 1.8c** DDL/SQL for the relational schema in Figure 1.8b (continued)

As shown in Figure 1.7, the physical design activity is fully technology dependent. Physical design involves using the tools of a particular DBMS product to create the database and to design and develop applications that address the high-level requirements of the universe of interest. The objective here is twofold: (a) developing an appropriate structure for the database and (b) keeping focus on performance while determining the physical structure for the database. A good physical database design is impossible without the database designer understanding the “job mix” for the particular application environment—that is, the mix of transactions, queries, applications, etc.

The ANSI/SPARC three-schema architecture in Figure 1.3 forms the basis for the data modeling/database design life cycle shown in Figure 1.7. Starting with the conceptual design activity and progressing through the logical design and physical design activities mirrors the nucleus of interest of the three-schema architecture—that is, the conceptual schema. This figure is replicated at the beginning of Parts I, II, III, and IV of this book to serve as a roadmap through the topics covered in the chapters of each part.

## Chapter Summary

---

Data consists of raw facts—that is, facts that have not yet been organized or processed to reveal their meaning. Information is data in context—that is, data that has been organized into a specific context that has meaning and value. Metadata describes the properties of data. It is through the lens of metadata that data becomes information.

Data management involves four actions: creating, retrieving, updating, and deleting data. Two data management functions support these actions: organizing data and accessing data. Two primary forms of access are sequential access and direct access.

Database systems have been successful because they overcome the problems associated with the lack of integration of data and program-data dependence that plague their predecessors, file-processing systems. Database management system (DBMS) software has been the vehicle that has allowed many organizations to move from a file-processing environment to a database system environment. Among the components of a DBMS are tools for (a) retrieving and analyzing data in a database, (b) creating reports, (c) creating the structure of database objects, (d) protecting the database from unauthorized use, and (e) facilitating recovery from various types of failures. In a DBMS environment, the data dictionary (metadata that describes characteristics of data) functions as the lens through which data in the database is viewed.

The ANSI/SPARC three-schema architecture divides a database system into three levels or tiers. The external level is closest to the users and is concerned with the way data is used or viewed by individual users. The conceptual level is technology independent and represents the global or community view of the entire database. The internal level is the one closest to physical storage and is concerned with the way the data is physically stored. As such, the internal level is technology dependent. The data as perceived at each level are described by a schema (or subschemas, in the case of the external level). A file-processing system is essentially a two-tier architecture with only external and internal levels. Without the conceptual schema, the internal schema must be mapped directly into external views. Thus, changes in the internal schema require appropriate changes in the external subschemas; this is how data independence is lost.

Data models play a crucial role in database design. Data models describe the database structure. The approach described in this book begins with the creation of a conceptual schema that describes the structure of the data to be stored in the database without specifying how and where it will be stored or the methods used to retrieve it. The conceptual schema takes the form of Presentation Layer and Design-Specific ER models and, once appropriately validated, serves as input to the logical design activity. During logical design, the technology-independent conceptual schema evolves into a technology-dependent logical schema. This technology-dependent logical schema is subsequently used during the physical design activity. The transition from logical design to physical design is initiated by the execution of the DDL/SQL script.

## Exercises

---

1. What is the difference between data, metadata, and information?
2. Demonstrate your understanding of data, metadata, and information using an example.
3. Describe the four actions involved in data management.

## Chapter 1

4. Distinguish between sequential access and direct access. Give an example of a type of application for which each is particularly appropriate.
5. Identify a common task in a payroll system for which sequential access is more appropriate than direct access and explain why this is so.
6. What is the difference between a serial collection of data and a sequential collection of data? Which can be used for direct access?
7. What is the purpose of an external index?
8. What is data integrity, and what is the significance of a lack of data integrity?
9. Describe the limitations of file-processing systems. How do database systems make it possible to overcome these limitations?
10. Using the Internet, trace the history of ANSI and ISO and their relevance to the information systems discipline. Write a summary of your findings.
11. Describe the structure of the ANSI/SPARC three-schema architecture. Compare this structure with that of the two-schema architecture inherent in a file-processing system.
12. Explain why a file-processing system may be referred to as belonging to a two-schema architecture.
13. Define data independence.
14. What is the difference between logical and physical data independence? Why is the distinction between the two important?
15. What is the difference between a database and a database management system?
16. Since ANSI and ISO have adopted SQL as the standard language for database access, explore via the Internet the history and features of SQL and its appropriateness for database access. Write a summary of your findings.
17. Write a short essay (one or two pages) about distributed databases using information available from Internet resources.
18. Write a short essay (one or two pages) about data warehousing using information available from Internet resources.
19. Oil companies have functional databases, and the consumer-product industry tends to have product databases. How do financial institutions and the airline industry classify their enterprise database systems? Use Internet resources to find the answer, and record your findings.
20. Find out and describe briefly what a CASE tool is, using Internet sources.
21. Distinguish between a model and a data model.
22. What is the role of data models in database design?
23. Write a short essay (one or two pages) summarizing the content of the *Harvard Business Review* article on databases cited in Figure 1.4.

# PART

## CONCEPTUAL DATA MODELING

### INTRODUCTION

---

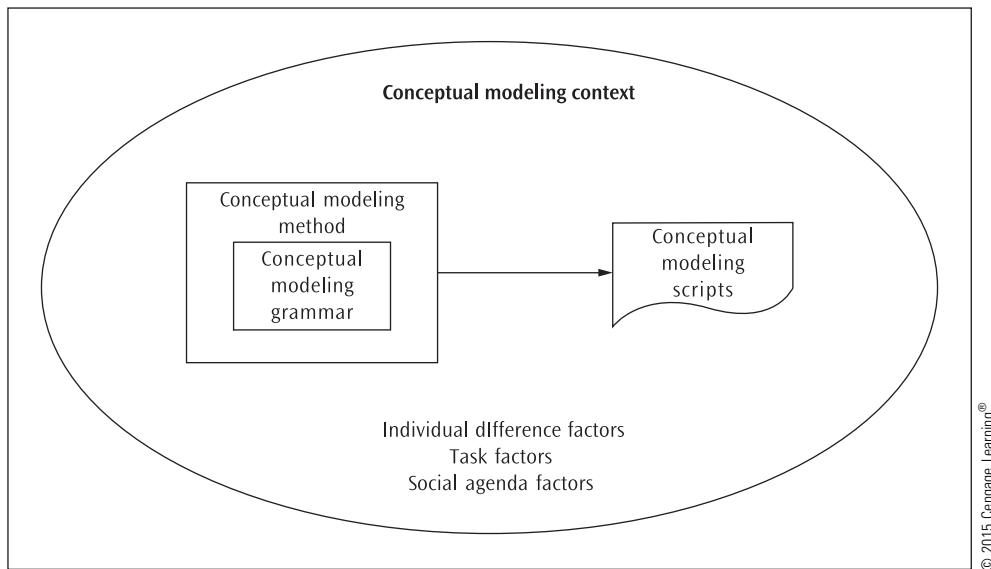
Database systems typically have a very limited understanding of what the data in the database actually mean. Therefore, **semantic modeling** (the overall activity of attempting to represent meaning) can be a valuable precursor to the database design process to capture at least some of the meaning conveyed by the users' business rules. The term "conceptual modeling" is often used as a synonym for semantic modeling. In this book, **conceptual modeling** refers to only data specifications (not process specifications) of the user requirements, at a high level of abstraction. Batini, Ceri, and Navathe (1992) have argued that conceptual modeling in database design is justified because it encourages user participation in the design process, allows the model to be more DBMS independent, facilitates understanding of how the database fits into the organization as a whole, and eases maintenance of schemas and applications in the long run. Sophisticated interpretations of the meaning of data are still left to the user (Date, 2004).

## Part I

Conceptual modeling entails constructing a representation of selected phenomena in a certain application domain—for example, an airline management system, a patient care system, or a university enrollment system. A conceptual model portrays data specifications at the highest level of abstraction in the data modeling hierarchy. The input source for this modeling phase is the business rules culled from the requirements specification supplied by the user community. A conceptual model includes a context, a grammar, and a method:

- A context is the setting in which the phenomenon being modeled transpires.
- A grammar defines a set of constructs and rules for modeling the phenomenon.
- A method describes how to use the grammar.

The product of conceptual modeling is a conceptual modeling “script,” such as an ER model, NIAM model, or semantic object model for the phenomenon being modeled. This framework for conceptual modeling is based on Wand and Weber (2002), and is illustrated in Figure I.1.

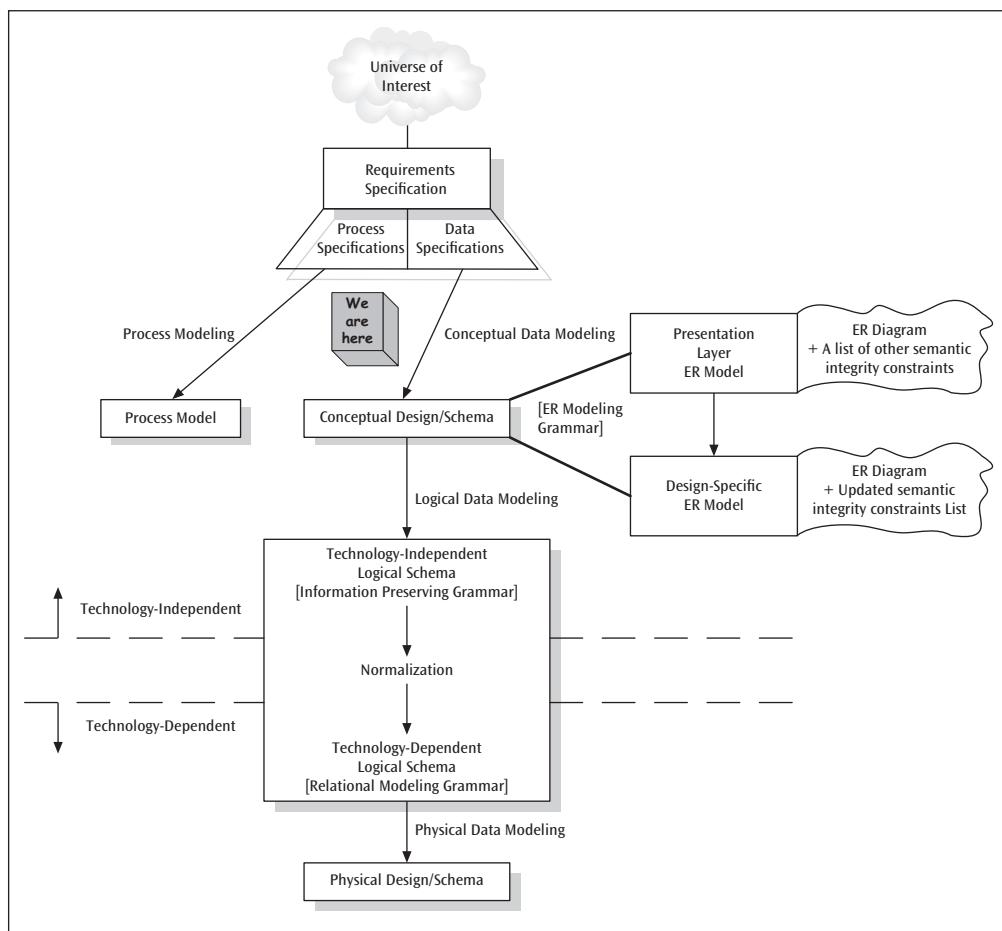


**FIGURE I.1** Wand and Weber's framework for research in conceptual modeling

The chapters in Part I introduce and elaborate upon conceptual data modeling using the Wand and Weber framework. Chapter 2 introduces the fundamental constructs and rules for the entity-relationship (ER) modeling grammar, which is used in this book as the tool for conceptual data modeling. The constructs pertain to inter-entity class binary relationships. Chapter 3 employs a comprehensive case to illustrate the method component of the Wand and Weber framework. In this chapter, the method to use the ER modeling grammar is explicated in progressive steps leading to the emergence of a specific script (i.e., an ER diagram and a semantic integrity constraints list) for the case in question.

Chapter 4 introduces newer constructs that enhance the ER (EER) modeling grammar with means to model intra-entity class relationships. An extension to the case from Chapter 3 incorporating a story line that requires application of EER constructs is used here to demonstrate the method pertaining to the use of the EER constructs. Chapter 5 presents higher-order relationships, namely relationships of degree 3 and beyond, innovative use of the grammar, and a few additional ER constructs (e.g., cluster entity type, interrelationship dependency). A second comprehensive case is used to illustrate the method component of the modeling framework.

Figure I.2, which replicates Figure 1.7, is a “road map” that serves as an overview of the process of data modeling and database design, indicating how the topics in Part I fit into the overall picture.



**FIGURE I.2** Road map for data modeling and database design

# CHAPTER 2

# FOUNDATION CONCEPTS

This chapter introduces the fundamental constructs and rules for conceptual data modeling using the entity-relationship (ER) modeling grammar as the modeling tool. The basic units of the ER model—that is, entity type, entity class, attribute, unique identifier, and relationship type—are treated in detail. The chapter also includes a brief introduction to cluster entity type and a comprehensive treatment of the incorporation of deletion rules in the ER modeling grammar.

## **2.1 A CONCEPTUAL MODELING FRAMEWORK**

---

There are many modeling schemes that attempt to represent the data semantics of a database application, and in fact many of these schemes bear a strong family resemblance to one another and often use graphical representations. The entity-relationship (ER) modeling grammar originally proposed by Peter Chen in 1976 and further refined by others over the years is probably the most widely accepted data modeling tool for conceptual design; it was chosen by ANSI in 1988 as the standard model for Information Resource Directory Systems (IRDSs). It aspires to capture the overall data semantics of an application in a concise manner in terms appropriate for subsequent mapping to specific database models. Though criticized by some for its insufficiency for the completion of a database design (Date, 2004), the ER modeling grammar is an effective tool for communicating technical information in the development of large database applications. Thus, the ER model is used in this book to illustrate conceptual data modeling.

## **2.2 ER MODELING PRIMITIVES**

---

Table 2.1 contains a series of terms (referred to here as **primitives**) that correlate the real world to the conceptual world and serve as a foundation for the entity-relationship (ER) modeling grammar. The real world consists of (a) tangible objects of an **object type** (such as a specific student, a specific piece of furniture, or a specific building), or (b) intangible objects of an object type (a specific department, a particular project, or a certain course).

Real World Primitive	Conceptual Primitive
Object {type}	Entity(ies) {type}
Object (occurrence)	Entity(ies) {instance}
Property	Attribute
Fact	Value
Property value set	Domain
Association	Relationship
Object class	Entity class

**TABLE 2.1** Equivalence between real world primitives and conceptual primitives

In the conceptual world, an object type is referred to as an **entity type**. Objects belonging to an object type are considered to be **entities** or **entity instances** of the corresponding entity type. The concept of an entity is the most fundamental one of the ER modeling grammar and serves as the foundation for other concepts. Instead of the actual person or object, Anna Li, the entity representing Anna Li, takes the form of a record in a STUDENT data set. Thus, actual students are student objects and are referred to in terms of a STUDENT object type, whereas a representation of the STUDENT object type is called the STUDENT entity type.

In the real world there can be many occurrences of a particular object type. For example, there can be 35,000 students enrolled and taking courses in a university during a semester—thus, 35,000 student objects. In this case, there would be 35,000 student entities represented by 35,000 records in a STUDENT data set. The collection of these 35,000 student entities is referred to as an **entity set**.

An object type can have many properties. For example, a STUDENT object type has properties such as student number, date of birth, gender, and so on. Correspondingly, an entity type is said to have **attributes**.

An entity or entity instance is created when a value is supplied for some attribute(s). Thus, a STUDENT data set with 35,000 records would contain values that represent the facts associated with the 35,000 student objects. In addition, in the real world, a fact is drawn from a **property value set**. In the conceptual world, the value of an attribute comes from a **domain** of possible values. For example, the domain for the attribute **Gender** can be (Male, Female), whereas the domain associated with the set of two-character U.S. postal codes is (AK, AL, AR, . . . , WY). A domain can be either explicit or implicit. The domain for **Gender** is an example of an explicit domain consisting of a set of only two possible values. The domain for the attribute **Salary** is an example of an implicit domain because it is not practical to explicitly list the set of all possible salaries between, say, \$10,000 and \$2,000,000.

In the real world, there are **associations** between objects of different object types. For example, students enroll in courses, suppliers supply parts, and salespersons process orders. In the conceptual world, these associations are referred to as **relationships**.

Two other terms need to be introduced as part of our foundation concepts: object class and entity class. In the real world, an **object class** is a generalization of related object types that have shared properties; in the conceptual world, an **entity class** is a generalization of different related entity types that have shared attributes. For example, the entity

type CHAIR and the entity type TABLE would both refer to the entity class FURNITURE. Likewise, STUDENT, FACULTY, and CUSTOMER entity types can be said to belong to an entity class called HUMAN\_BEING.

## 2.3 FOUNDATIONS OF THE ER MODELING GRAMMAR

The basic building blocks of ER modeling include entity types, attributes, and relationship types. A set of attributes makes up or gives structure to an entity type. A rule in the grammar specifies that entity types can only be associated via a relationship type and that a relationship type can only show association between entity types.

### 2.3.1 Entity Types and Attributes

As described in Section 2.2, an entity type is a set of related attributes that comprise a conceptual representation of an object type. Typically, an entity type takes the form of a singular noun (a person, thing, place, or concept). Attributes shape an entity type. An entity type can participate in one or more relationships with other entity types. Table 2.2 lists a number of characteristics possessed by an attribute. First, each attribute has a name that generally conforms to a standardized naming convention. For example, Company A may use the attribute **Ename** to represent the name of an employee in the EMPLOYEE entity type but may require a different attribute name (e.g., **Empname**) to be used for the name of the employee who manages a particular department in the DEPARTMENT entity type. Company B, on the other hand, may require that each employee name attribute in the database make use of the same attribute name, **Ename**. Then, the employee name in the EMPLOYEE entity type is referred to as **EMPLOYEE.Ename**, whereas the employee name in the DEPARTMENT entity type would be referred to as **DEPARTMENT.Ename**.

Attribute	Characteristics
Name	Standardized naming convention
Type	Numeric, alphabetic, alphanumeric, logical, date/time, etc.
Classification	Atomic or composite/molecular
Category	Single-valued or multi-valued
Source	Stored (real) or derived (virtual)
Domain*	Property value set—implicit or explicit
Value	Conceptual representation of a fact about a property
Optionality	Optional value or mandatory value
Role	Key (unique identifier) or non-key

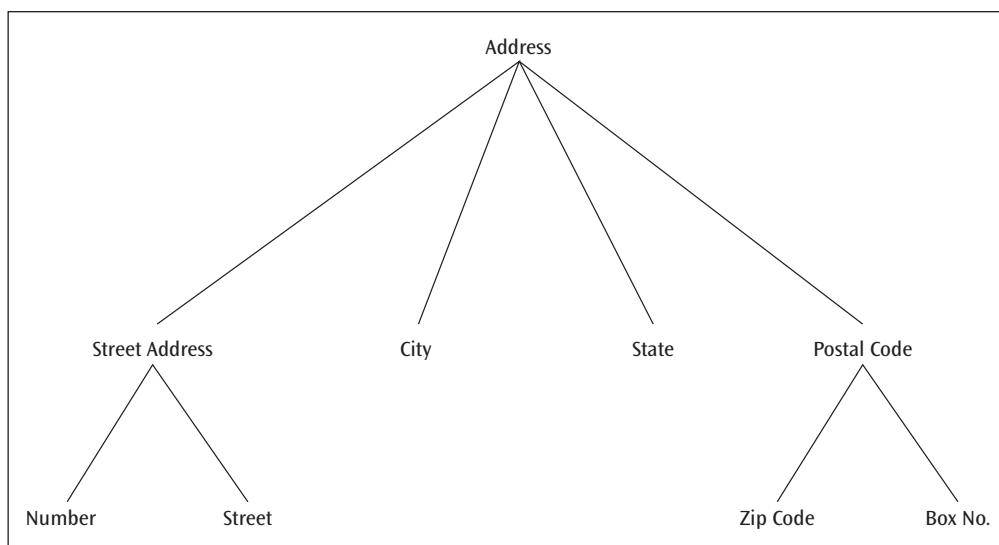
\*In a stricter sense, other characteristics of an attribute (e.g., type, value) are also viewed as part of the domain of an attribute.

**TABLE 2.2** Characteristics of attributes

A variety of **data types** can be associated with attributes. A “numeric” data type is used when an attribute’s value can consist of positive and negative numbers; it is often used in arithmetic operations. Numeric attributes can further be constrained so as to allow

only integer values, decimal values, and so on. The “alphabetic” data type permits an attribute to consist of only letters and spaces, whereas an “alphanumeric” data type allows the value of an attribute to consist of text, numbers (telephone numbers, postal codes, account numbers, and so on), and certain special characters. Alphabetic and alphanumeric data types should not be used for attributes involved in arithmetic operations. Likewise, an attribute not involved in an arithmetic operation should not be defined as a numeric data type even if it contains only numbers (telephone number, Social Security number) to enable textual manipulations. A “logical” data type is associated with an attribute whose value can be either true or false. Attributes with a “date” data type occur frequently in database applications—for example, date of birth, date hired, or flight date.

A particularly important characteristic of an attribute is its classification—that is, whether it is an atomic attribute or a composite attribute. An attribute that has a discrete factual value and cannot be meaningfully subdivided is called an **atomic** or **simple attribute**. On the other hand, a **composite** or **molecular attribute** can be meaningfully subdivided into smaller subparts (i.e., atomic attributes) with independent meaning. **Salary** is an atomic attribute because it cannot be meaningfully divided further. Depending on the user’s specification, **Name** can be an atomic attribute or a composite attribute made up of **First name**, **Middle initial**, and **Last name**. Figure 2.1 illustrates how an address might be modeled in the form of a hierarchy of composite attributes.

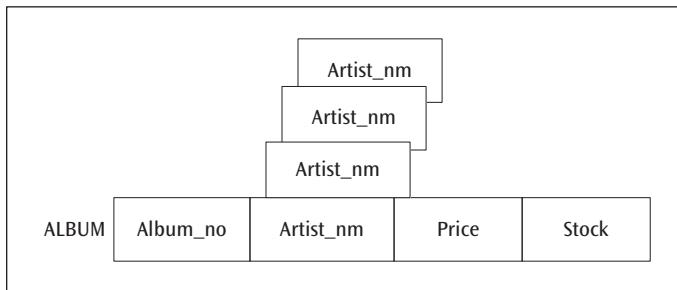


**FIGURE 2.1** Example of a composite attribute hierarchy

An attribute can be either a **stored attribute** or a **derived attribute**. In some cases, two or more attributes are related in the sense that the value of one can be calculated or derived from the values of the other(s). For example, if **Flight time** is calculated as the difference between the arrival time at the destination and departure time at the point of origin, then **Flight time** in this case can be a derived attribute since it need not be stored.

Most attributes have a single value for a particular entity and are referred to as **single-valued attributes**. For example, **Date of birth** is a single-valued attribute of an employee. There are attributes, however, that can have more than one value. For example, a programmer may be skilled in several programming languages, thus making the attribute **Skill** a **multi-valued**

**attribute.** Figure 2.2 contains another example of a multi-valued attribute; here, whereas **Album\_no**, **Price**, and **Stock** are single-valued attributes, **Artist\_nm** is a multi-valued attribute.



**FIGURE 2.2** Example of a multi-valued attribute

For each entity of an entity type, some attributes *must* be assigned a value. Such attributes are referred to as **mandatory attributes**. On the other hand, attributes that need not be assigned a value for each entity are referred to as **optional attributes**. For example, it is possible that the attribute **Salary** might be classified as an optional attribute if the salary of each employee need not necessarily be provided or is unknown. Another attribute, **Commission**, might be classified as an optional attribute because a commission is not necessarily meaningful for job types other than, perhaps, “salesperson.” Attributes classified as optional are assigned a special value called “null” when their value is not available or is unknown.<sup>1</sup>

Composite and/or multi-valued attributes that are nested as a meaningful cluster are called **complex attributes**.<sup>2</sup> As an example, let’s consider the medical profile for a patient as shown here:

**Medical\_profile (Blood (Type, Cholesterol (HDL, LDL, Triglyceride), Sugar), Height, Weight, {Allergy (Code, Name, Intensity)})**

Observe that composite attributes are enclosed in parentheses ( ), whereas multi-valued attributes are enclosed in braces { }. The medical profiles for two patients appear in Figure 2.3.

		SUL Sulfa Severe									
		PCN Penicillin Mild									
Patient 1		B+	53	111	151	133	71	151	POL	Pollen	Mild
Medical_profile		Blood_type	HDL	LDL	Triglyceride	Sugar	Height	Weight	Allergy_code	Allergy_name	Intensity
		A- 41 111 197 93 67 119 ML Mold Moderate									
Patient 2		Blood_type	HDL	LDL	Triglyceride	Sugar	Height	Weight	Allergy_code	Allergy_name	Intensity

**FIGURE 2.3** Medical profiles of two patients

<sup>1</sup>“Value unknown” and “value exists but is not available” constitute what is typically called “missing data.”

<sup>2</sup>A complex attribute is most likely an entity type instead of an attribute. Nonetheless, the purpose here is to explain the existence of an attribute called a complex attribute.

### 2.3.2 Entity and Attribute-Level Data Integrity Constraints

Data integrity constraints, generally referred to as just **integrity constraints**, are rules that govern the behavior of data at all times in a database. Integrity constraints stem from the business rules that emerge from the user requirements specification for a database application system. In other words, integrity constraints are technical expressions of business rules. To that extent, the integrity constraints prevail and thus must be preserved across all three tiers of data modeling—conceptual, logical, and physical.

A **business rule**, in a data modeling context, is a short statement, expressed in a precise, unambiguous manner, of a specific condition or procedure relevant to the universe of interest (application domain) being modeled. Business rules often are implicitly interleaved in the requirements specification and are culled as explicit expressions. Although this process of developing business rules from the requirements specification is not quite scientific, it certainly can be systematic. A step-by-step analysis of the requirements specification will enable an analyst/modeler to extract, as business rules, specific conditions and procedures inherent in the universe of interest. Furthermore, such a systematic analysis will facilitate identification of ambiguities that, when clarified by the user community, will yield additional business rules and facilitate correction of other business rules. Business rules not only facilitate development of data models, they aid in validating them.

Business rules are typically grouped into three categories:

- Explicit business rules
- Inferred business rules
- Ambiguities, which require clarification

Consider the following example, in which the universe of interest for modeling is a university academic program. Here, to illustrate the development of business rules, is a short extract from a comprehensive requirements specification for this application domain:

*There are several colleges in the university. A college offers many courses, and a college term is divided into four quarters—Fall, Winter, Spring, and Summer—during which one or more of these courses may be offered. Every quarter, at least 23 courses are offered. The college has several instructors. Instructors teach; that is why they are called instructors. Often, not all instructors teach during all quarters, however. Furthermore, instructors are capable of teaching a variety of courses that the college offers.*

At the outset, four entity types—COLLEGE, COURSE, QUARTER, and INSTRUCTOR—may emerge from this narrative. The business rules that can be extracted from the narrative include:

- There are four quarters—specifically, Fall, Winter, Spring, and Summer.
- A college offers many courses.
- At least 23 courses are offered during every quarter.
- A college has several instructors.
- An instructor need not teach in a given quarter.
- An instructor is capable of teaching a variety of courses offered by the college.
- *An instructor must teach in some quarter.*
- *An instructor must be capable of teaching at least one course that the college offers.*

Observe that the first six business rules are obvious from the narrative, whereas the last two (shown in italics) are inferred and precisely specified from the requirements specification as a whole.

A systematic study of the narrative reveals ambiguities requiring clarification by the user community, who can be asked such questions as:

- Is a particular course offered in more than one quarter?
- Are there courses that are still “in the books” but no longer offered?
- Can an instructor teach for more than one college in the university?

Answers to such questions will sharpen the requirements specification by way of additional business rules. In short, business rules are indispensable in the process of translating a requirements specification to integrity constraints.

In general, integrity constraints are considered part of the schema (the description of the metadata) in that they are declared along with the structural design of the data model (conceptual, logical, and physical) and hold for all valid states of a database that correctly model an application. Although it is possible to specify all the integrity constraints as a part of the modeling process, some cannot be expressed explicitly or implicitly in the schema of the data model. For instance, an ER diagram (conceptual schema) is not capable of expressing domain constraints of attributes. Likewise, there are other constraints that a logical schema (relational schema) is not capable of expressing (this topic is covered in more detail in Chapter 6). As a consequence, such constraints are carried forward through the data modeling tiers in textual form and are often referred to as **semantic integrity constraints**.

At the conceptual tier of data modeling, two types of data integrity constraints pertaining to entity types and attributes are specified:

- *Domain constraint*—This is imposed on an attribute to ensure that its observed value is not outside the defined domain.
- *Uniqueness constraint*—This requires entities of an entity type to be uniquely identifiable. It is sometimes referred to as the “key constraint.”

### 2.3.2.1 Domain Constraint

The concept of domain was briefly introduced in Section 2.2. The characteristics of an attribute discussed in Section 2.3.1 (name, type, class, and domain) are in themselves data integrity constraints on the attribute. For example, defining the attribute **Name** as 30 positions alphanumeric would be a constraint on the attribute **Name**. Similarly, defining the attribute **Salary** as eight positions numeric would be a constraint on the attribute **Salary**. However, the requirement that the attribute **Salary** must be in the range \$60,000 to \$80,000 is an example of an implicit domain constraint.<sup>3</sup> Although constraints are intended to make sure that the integrity of data is maintained, a domain constraint cannot ensure total data integrity. For example, if an employee’s Salary is \$65,000 but is erroneously entered as \$56,000, the domain constraint will result in the identification of the error. On the other hand, if the erroneous entry is \$67,000—that is, within the domain—the

---

<sup>3</sup>Strictly speaking, the data type and size of an attribute can also be construed as domain constraints on the attribute.

error will not be identified. Here is an example of an explicit specification of a domain constraint: A **Student\_type** takes a value from the set {FR, SO, JR, SR, GR}. Here is another example: **Music\_skill** takes a value from the set {Rock, Jazz, Classical}.

### 2.3.2.2 Unique Identifier of an Entity Type

It is important to be able to uniquely identify entities in the entity set of the entity type. An atomic or composite attribute whose values are distinct for each entity in the entity set is the **unique identifier**<sup>4</sup> of the entity type. This attribute, then, can be used to distinguish an entity from other entities in the entity set. Note that specifying a unique identifier for the entity type signifies the imposition of the uniqueness constraint on the entity type. It is also important to recognize that, instead of being limited to finding one unique identifier for the entity type, a uniqueness constraint identifies all irreducible unique identifiers of the entity type.<sup>5</sup>

Every attribute plays only one of three roles in an entity type: It is a key attribute, a non-key attribute, or a unique identifier. Any attribute that is a constituent part of a unique identifier is a **key attribute**. An attribute that is not a constituent part of a unique identifier is a **non-key attribute**.<sup>6</sup>

A graphical representation of an entity type along with its attributes is portrayed in Figure 2.4. Here, the entity type PATIENT is indicated by a rectangle and attributes are indicated by circles. Various attribute characteristics are shown in the figure as follows:

- **Date\_of\_birth** and **Address** are optional atomic attributes (shown as empty circles).
- **Phone** is an optional multi-valued attribute (shown as an empty double circle).
- **Age** is an optional atomic attribute derived by subtracting date of birth from the current date (shown as a dotted empty circle).
- **SSN** is a mandatory atomic attribute (shown as a dark circle) that serves as a unique identifier. Note that attribute names of unique identifiers are underlined.
- **Pat\_Name** is a mandatory composite attribute composed of the atomic attributes **F\_Name**, **M\_int**, and **L\_Name**. Although **F\_Name** and **L\_Name** are mandatory atomic attributes, **M\_int** is an optional atomic attribute.
- **Pat\_ID** is a mandatory composite attribute and serves as a second unique identifier.
- **Medical\_profile** is a complex attribute named as such to reflect the complexity of the attribute hierarchy.

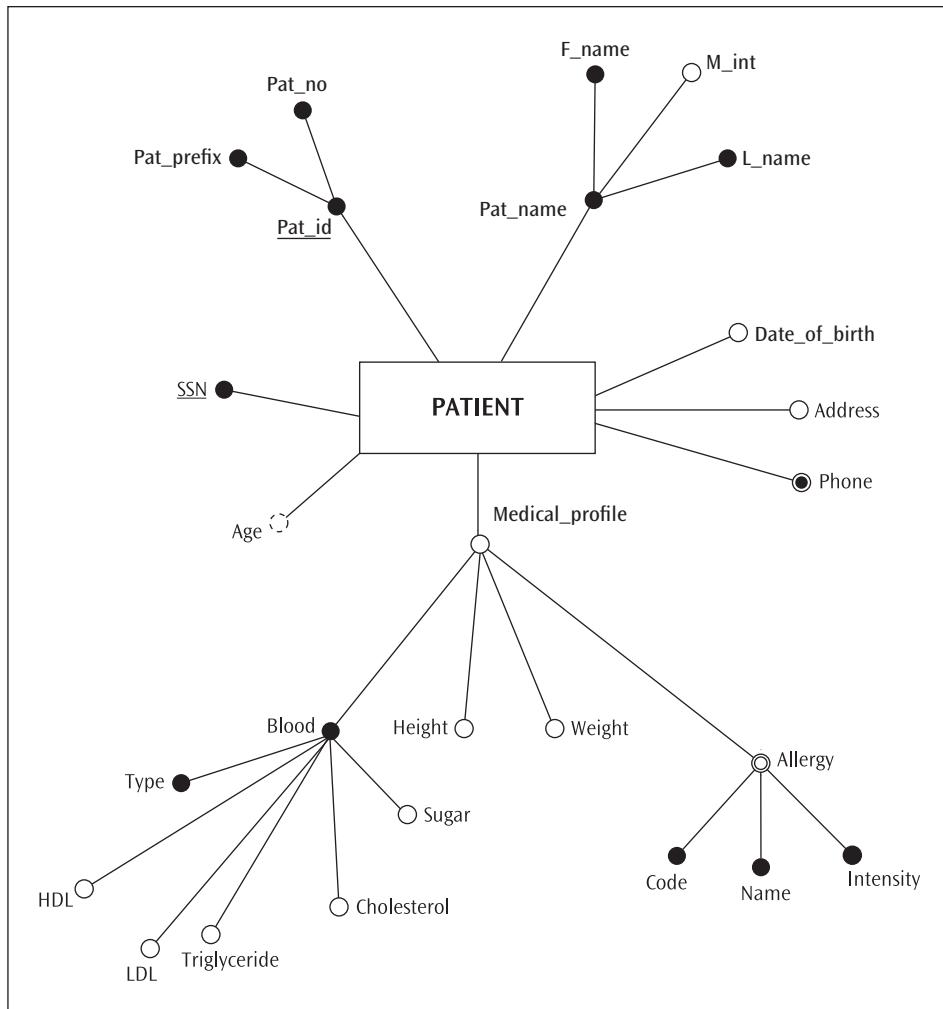
#### NOTE

When a composite or complex attribute is optional and one or more of its proper subsets (atomic or composite) is/are mandatory (e.g., **Medical Profile** in Figure 2.4), it simply means that when the composite or complex is present, at least the mandatory subset(s) must be present.

<sup>4</sup>Another popular term, “primary key,” is deliberately avoided here because a strict definition of the term is possible only in the context of a relation schema discussed in Chapter 6.

<sup>5</sup>A unique identifier is irreducible when none of its proper subsets is a unique identifier. The concept of irreducibility is discussed in detail in Chapter 6.

<sup>6</sup>In formal terms, a key attribute is a proper subset of a unique identifier and a non-key attribute is any attribute that is not a subset of a unique identifier.



**FIGURE 2.4** Graphical representation of the entity type **PATIENT**

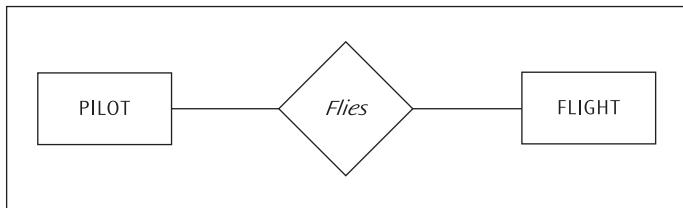
Observe that in ER diagrams, a rectangular box is used to represent an entity type whereas a circle symbolizes an attribute. The atomic attributes that make up a composite attribute are attached to the circle that represents the composite attribute. Dark circles represent mandatory attributes, whereas empty circles signify optional attributes. The name of the attribute appears adjacent to the circle. A multi-valued attribute (e.g., **Phone** and **Allergy**) is shown by a double circle, and a derived attribute (e.g., **Age**) is shown by a dotted circle.

### 2.3.3 Relationship Types

A relationship type is a meaningful association among entity types. The degree of a relationship type is defined as the number of entity types participating in that relationship type. A relationship type is said to be binary (or of degree two) when two entity types are involved. Relationship types that involve three entity types (of degree three) are defined as ternary relationships. Although a relationship of degree four can be referred to as a quaternary

relationship, a more generalized term is “n-ary relationship,” in which the degree of the relationship type is “n.” An entity type related to itself is termed a “recursive relationship type.”

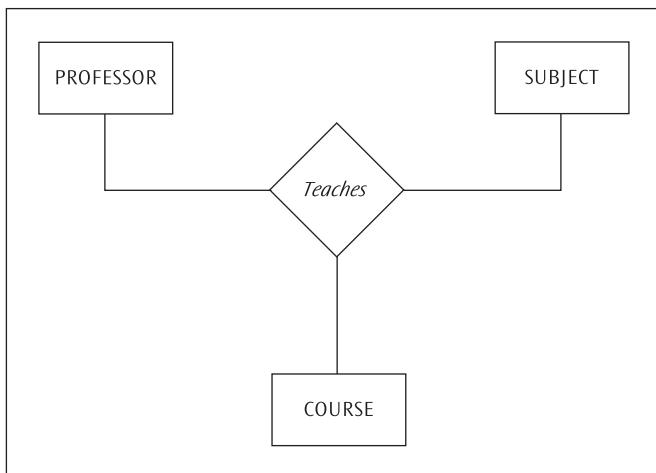
Figures 2.5 through 2.9 illustrate binary, ternary, quaternary, and recursive relationship types, respectively; in the figures, a diamond signifies a relationship type. Figure 2.5 illustrates a binary relationship type called *Flies* between PILOT and FLIGHT. A particular pilot flying a specific flight is an instance of the relationship type *Flies*. This relationship instance is often referred to as a “relationship.” The set of all relationship instances involving pilots and flights is called a **relationship set**.



**FIGURE 2.5** A binary relationship

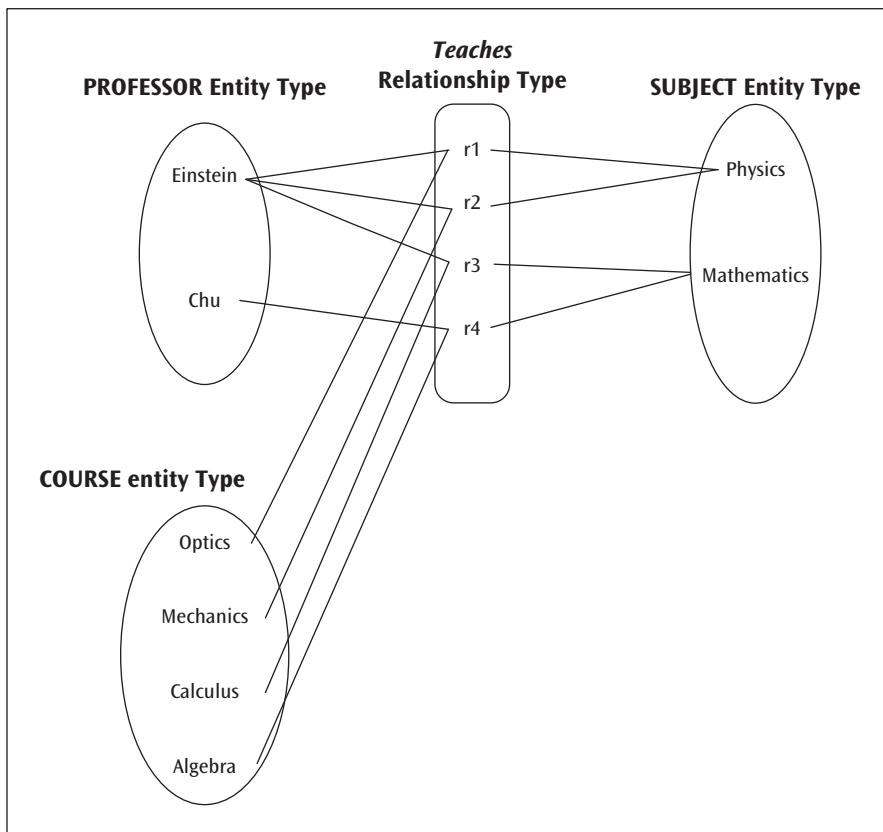
Figure 2.6 illustrates the ternary relationship type *Teaches* among PROFESSOR, SUBJECT, and COURSE. An instance of the relationship type *Teaches* involves a particular Professor teaching a certain Course in a specific Subject. Another relationship instance of *Teaches* could involve the same Professor teaching another Course in the same Subject area. A third instance of the relationship type *Teaches* could involve the same Professor teaching a Course in a different Subject area. A fourth relationship instance of *Teaches* could involve another Professor teaching a different Course in the previous Subject area. Examples of the four relationship instances just described are as follows:

- Example 1: Professor Einstein teaches the course Optics in Physics.
- Example 2: Professor Einstein teaches the course Mechanics in Physics.
- Example 3: Professor Einstein teaches the course Calculus in Mathematics.
- Example 4: Professor Chu teaches the course Algebra in Mathematics.



**FIGURE 2.6** A ternary relationship

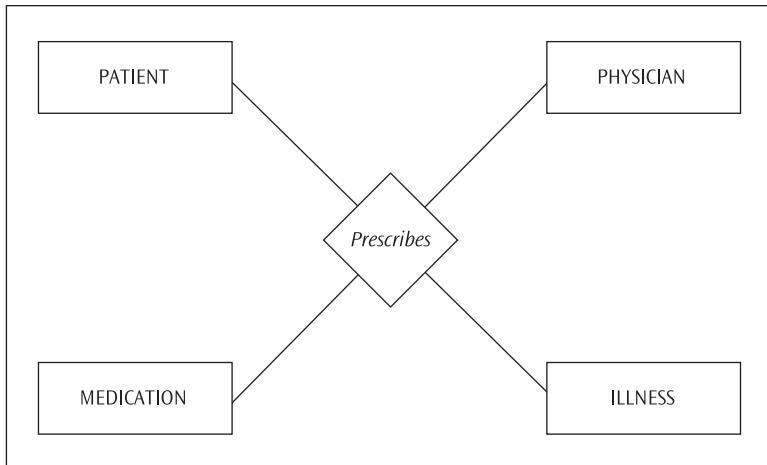
The diagrammatic representation of the relationship among entity types in terms of relationship instances among the instances of the participant entity types is called an **instance diagram**. Figure 2.7 uses an instance diagram in the form of a vertical ellipse to illustrate these relationship instances. Ternary relationship types are discussed in greater detail in Chapter 5.



**FIGURE 2.7** Four instances of the *Teaches* relationship type

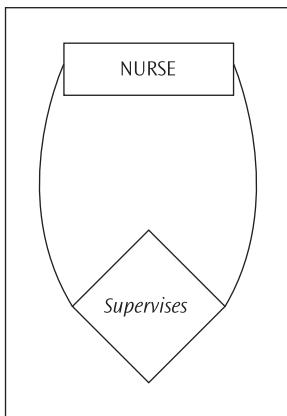
Figure 2.8 illustrates the quaternary relationship type *Prescribes* among PHYSICIAN, PATIENT, MEDICATION, and ILLNESS. An instance of the relationship type *Prescribes* involves a particular Physician prescribing a specific Medication to treat a particular Patient for a certain Illness. A second instance of the relationship type *Prescribes* could involve the same Physician prescribing the same Medication to treat a different Patient for the same Illness. Examples of these two relationship instances are as follows:

- Example 1: Dr. Fields prescribes Advil to treat Sharon Moore for a headache.
- Example 2: Dr. Fields prescribes Advil to treat Michelle Li for a headache.



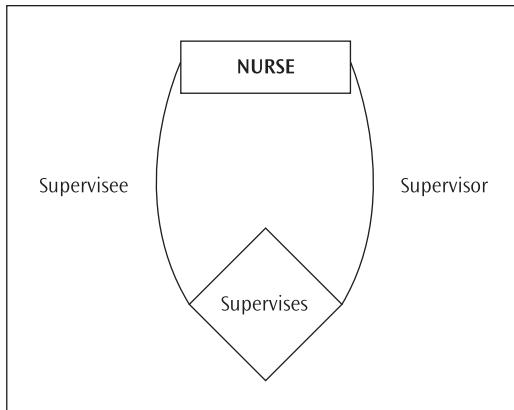
**FIGURE 2.8** A quaternary relationship

Quaternary relationship types are discussed in greater detail in Chapter 5. Figure 2.9 illustrates a recursive relationship type in which a NURSE acts as a supervisor of other nurses. Two instances of this relationship type might involve the same Nurse (e.g., Florence Nightingale) supervising two different Nurses (e.g., Jean Warren and Michael Evans).

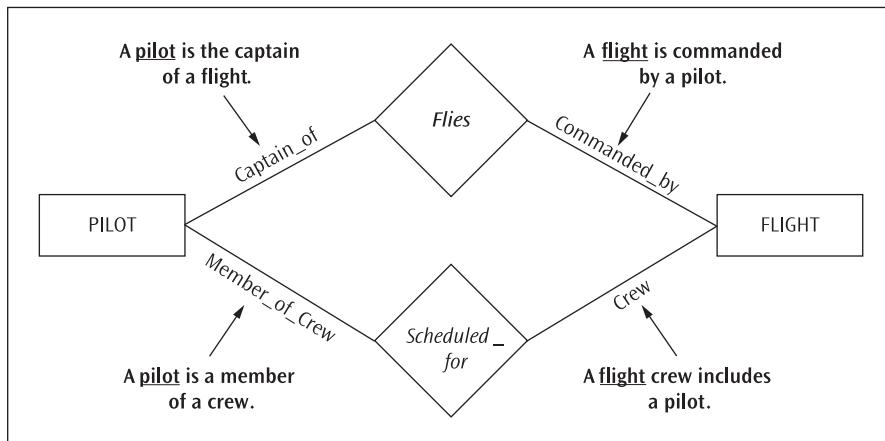


**FIGURE 2.9** A recursive relationship

The participation of an entity type in a relationship type can be indicated by its **role name**. When used in recursive relationship types, role names describe the function of each participation. The use of role names to describe the *Supervises* relationship type that appears in Figure 2.9 is given in Figure 2.10. One participation of NURSE in the *Supervises* relationship is given the role name of *Supervisor* to reflect the possibility that a nurse may supervise other nurses, and the second participation is given the role name of *Supervisee* to indicate the possibility that a nurse may be supervised by another nurse.

**FIGURE 2.10** Role names in a recursive relationship

Role names may also be used when two entity types are associated through more than one relationship type. For example, consider the possibility that the PILOT and FLIGHT entity types are associated through the two relationship types *Flies* and *Scheduled\_for*. *Flies* exists in order to indicate the specific pilot in charge of the flight, whereas *Scheduled\_for* indicates all pilots and co-pilots who are members of the flight crew. As shown in Figure 2.11, the use of role names can clarify the purpose of each relationship.

**FIGURE 2.11** Role names in binary relationships

Except for the previous two kinds of relationships, role names are often unnecessary when the relationship specification is unambiguous.<sup>7</sup>

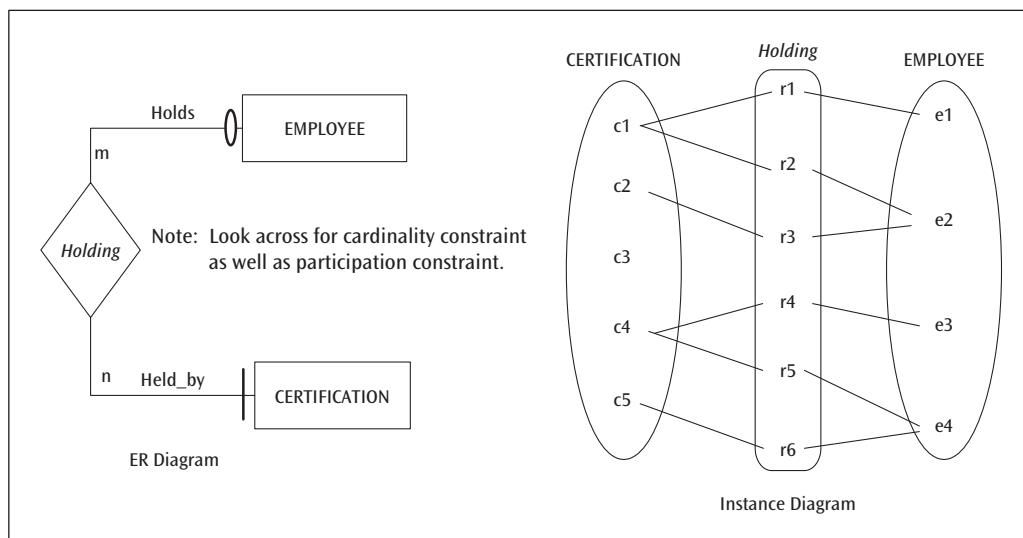
---

<sup>7</sup>In commercial software engineering tools, role names often replace the relationship symbol in the specification of a relationship type.

### 2.3.4 Structural Constraints of a Relationship Type

The data integrity constraints pertaining to relationship types specified in an ER diagram are referred to as the **structural constraints of a relationship type**.<sup>8</sup> Together, two independent structural constraints define a relationship type: cardinality constraint (also known as “mapping cardinality” or “connectivity”) and participation constraint. A relationship type is not fully specified until both structural constraints are explicitly imposed in the model. In what follows, binary and recursive relationship types are used to introduce these structural constraints.

The **cardinality constraint** for a binary relationship type is a constraint that specifies the maximum number of entities to which another entity can be associated through a specific relationship. For example, a binary relationship between the two entity types EMPLOYEE and CERTIFICATION may possess an m:n cardinality constraint, meaning that (a) each Employee entity can be related to many Certification entities (up to n), and (b) each Certification entity can be related to many (up to m) Employee entities (see Figure 2.12).



**FIGURE 2.12** Cardinality ratio and participation constraint for an m:n relationship

The relationship type *Holding*, between CERTIFICATION and EMPLOYEE, is reflected by the “m” on the edge connecting EMPLOYEE to *Holding* and the “n” on the edge connecting CERTIFICATION to *Holding*, indicating that one Employee holds many Certifications (no more than n) and one Certification is held by many Employees (no more than m). Observe that the cardinality constraint specification uses a look-across notation (one Employee holds no more than n Certifications, and one Certification is held by no more than m Employees, where “m” and “n” almost always indicate different values for “many”).

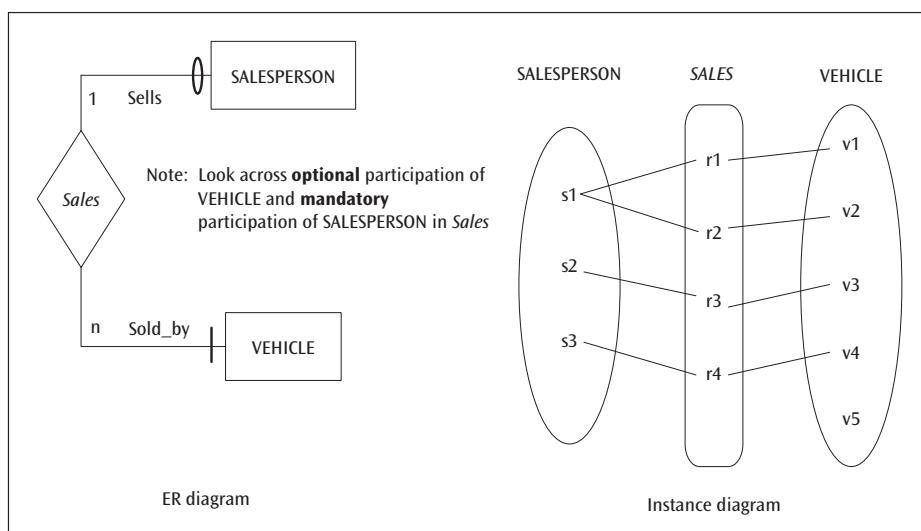
For a binary relationship between entity types A and B, four mapping cardinalities are possible:

- m:n—An entity in entity set A is associated with no more than m entities in entity set B; likewise, an entity in entity set B is associated with no more

<sup>8</sup>Sometimes (for example, in UML—Unified Modeling Language), the term “multiplicity” is used instead of “structural constraints.”

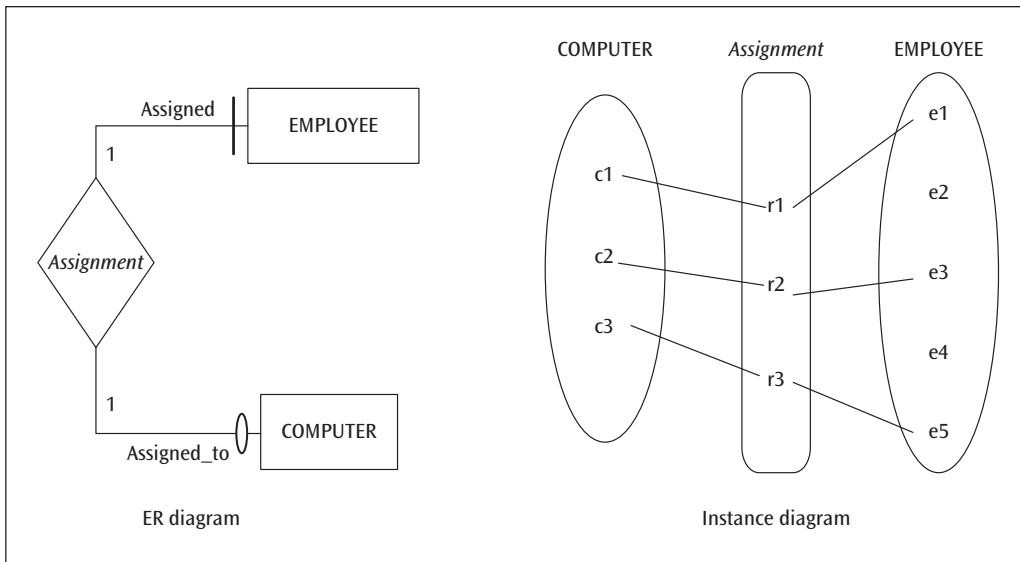
than n entities in entity set A. This is the general form of a cardinality constraint in a binary relationship. An example of an m:n cardinality constraint would involve the two entity types EMPLOYEE and CERTIFICATION, where each Employee can hold many different Certifications and each Certification can be held by many different Employees (see Figure 2.12).

- 1:n—An entity in entity set A is associated with no more than n entities in entity set B; however, an entity in entity set B is associated with no more than one entity in entity set A. When m takes a value of 1, the general form m:n becomes 1:n. The *Sales* relationship type in Figure 2.13 is an example of a 1:n cardinality constraint. A relationship of this type is sometimes referred to as a **parent-child relationship (PCR)**, in which the 1 side (SALESPERSON) is the parent and the n side (VEHICLE) is the child.<sup>9</sup>
- n:1—An entity in entity set A is associated with no more than one entity of entity set B; however, an entity in entity set B is associated with as many as n entities in entity set A. This is just a reverse expression of the cardinality constraint, 1:n. The *Sales* relationship type presented earlier also serves as an example of n:1 cardinality constraint (see Figure 2.13).
- 1:1—An entity in entity set A is associated with no more than one entity of entity set B, and an entity in entity set B is associated with no more than one entity in entity set A. When both m and n take a value of 1, the general form m:n becomes 1:1. A 1:1 cardinality constraint would exist between the two entity types EMPLOYEE and COMPUTER if each Employee was assigned no more than a single Computer and each Computer was assigned to no more than a single Employee (see Figure 2.14).



**FIGURE 2.13** Cardinality ratio of 1:n and partial participation of VEHICLE and total participation of SALESPERSON in Sales

<sup>9</sup>The original source of this acronym is said to be the practitioners' world. Bachmann used it in the data structure diagram that predates the ER model. The term was also prevalent in hierarchical and network data models before the advent of relational data model.



**FIGURE 2.14** Cardinality ratio of 1:1 and partial participation of *EMPLOYEE* and total participation of *COMPUTER* in *Assignment*

When the cardinality ratio in a binary relationship is 1:n, one of the entity types in the relationship (PCR) is unambiguously the parent and the other is the child. On the other hand, in a 1:1 binary relationship type it is not possible to unequivocally assign the parent or child role to either of the participating entity types. In fact, from a modeling perspective, both entity types will have to be evaluated in both roles. In an m:n binary relationship type, both entity types hold the role of parent because both a 1:n relationship and a 1:m relationship underlie an m:n relationship, where the relationship itself serves as a pseudo-entity type taking on the role of the child in both of the underlying relationships—that is, a child with two parents.

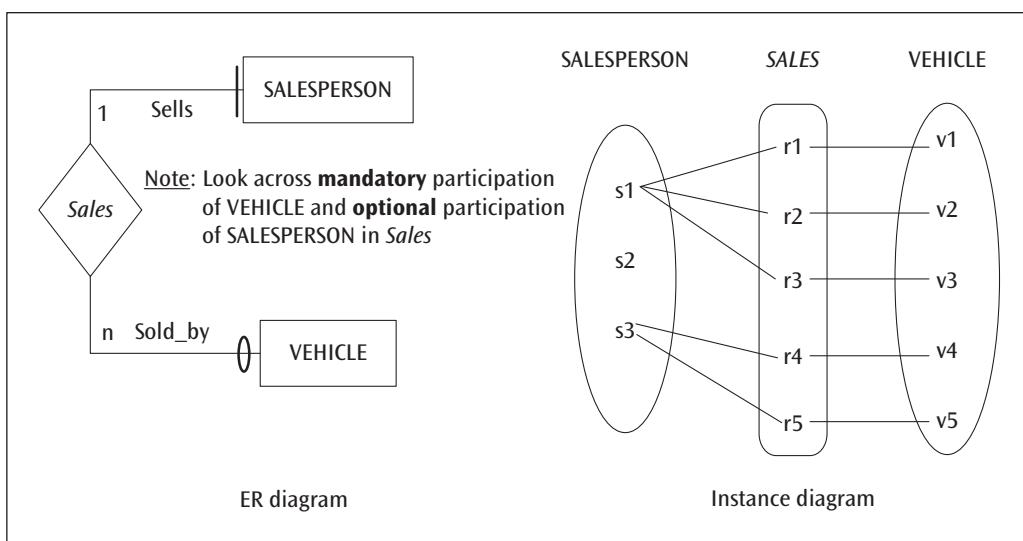
The cardinality constraint reflects the **maximum cardinality** of the entity types participating in the binary relationship type. The maximum cardinality indicates the maximum number of relationship instances in which an entity participates. For example, in the *Sales* relationship type shown in Figure 2.13, a *Salesperson* entity is connected to a maximum of n *Sales* relationship instances, whereas a *Vehicle* entity is connected to a maximum of one *Sales* relationship instance.

The **participation constraint** for an entity type in a relationship type is based on whether, in order to exist, an entity of that entity type needs to participate in that relationship. In a binary relationship type, the entity will be related to an entity of the other entity type through this relationship type. Participation can be total or partial. If, in order to exist, every entity of an entity type must participate in the relationship, then participation of the entity type in that relationship type is termed **total participation**. On the other hand, if an entity in an entity set can exist without participating in the relationship, then participation of the entity type in that relationship type is called **partial participation**. Total and partial participation are also commonly referred to as **mandatory** and **optional** participation, respectively. For example, if every *Salesperson* must have sold at least one *Vehicle*, then there is total participation of *SALESPERSON* in the *Sales* relationship type. Similarly, if a *Salesperson* need not have sold any *Vehicle*, then there is partial

## Chapter 2

participation of SALESPERSON in the *Sales* relationship type. Because the participation constraint specifies the minimum number of relationship instances in which each entity can participate, it reflects the **minimum cardinality** of an entity type's participation in a relationship type.

For instance, in Figure 2.15, the oval (or bubble) next to the VEHICLE entity type indicates that a Salesperson may or may not sell a vehicle. In other words, the participation of the entity type SALESPERSON in the *Sales* relationship type is conveyed by looking across the *Sales* relationship to the VEHICLE side. What about the participation of VEHICLE in the *Sales* relationship? In Figure 2.15, a Vehicle *must* be sold, obviously, by a Salesperson.

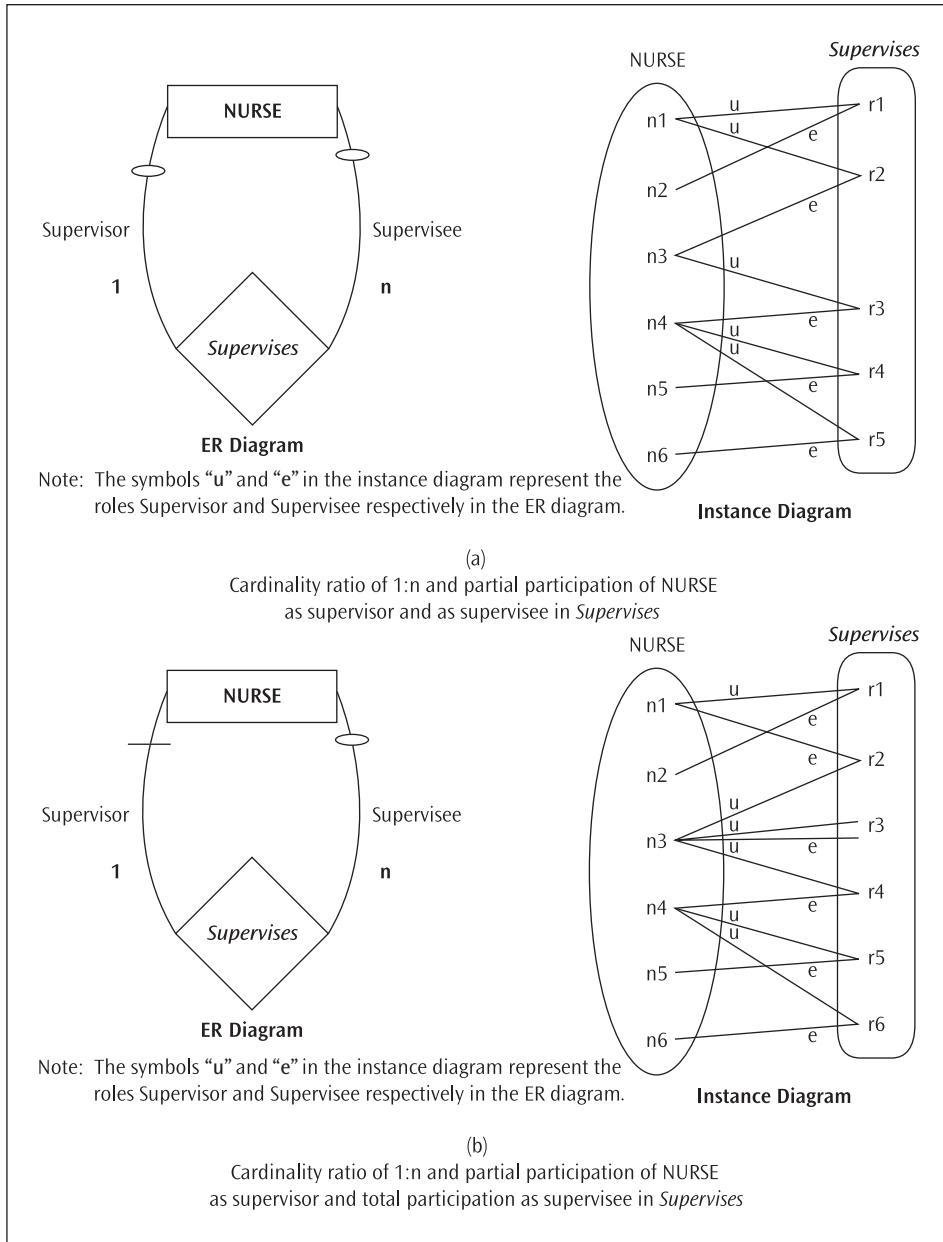


**FIGURE 2.15** Cardinality ratio of 1:n and total participation of VEHICLE and partial participation of SALESPERSON in *Sales*

Total participation of an entity type in a relationship type is also called **existence dependency**. In Figure 2.15, therefore, VEHICLE has existence dependency on the *Sales* relationship type, whereas SALESPERSON does not have existence dependency on the *Sales* relationship type.

The structural constraints for the recursive relationship involving the NURSE entity type appear in Figures 2.16a and 2.16b. These examples reflect a cardinality ratio of 1:n, meaning that a Nurse supervises many other Nurses—at most, n. Based on the directional nature of the cardinality ratio, this automatically implies that a Nurse is supervised by no more than one Nurse. The oval and the hash (|) introduced in Figure 2.15 are used to describe the participation constraints. In Figure 2.16a, the small oval on the right-hand side of the ER diagram indicates that a Nurse may or may not supervise other Nurses, whereas the oval on the left-hand side indicates that a Nurse need not be supervised (i.e., there are Nurses who are not supervised by another Nurse). The instance diagram in Figure 2.16a

illustrates the partial participation of NURSE as Supervisor and as Supervisee in the *Supervises* relationship type. For example, observe how Nurse n1 is the supervisor of Nurses n2 and n3 but is not supervised by another Nurse. On the other hand, Nurse n2 is a supervisee of Nurse n1 (i.e., is supervised by Nurse n1) but is not a supervisor of any Nurses.



**FIGURE 2.16** Structural constraints for recursive relationships: cardinality ratio of 1:n

## Chapter 2

In Figure 2.16b, the small oval on the right-hand side of the ER diagram indicates that a Nurse may or may not supervise other Nurses. However, the hash on the left-hand side indicates that every Nurse must be supervised by another Nurse. Observe that in the instance diagram, Nurse n3 is the supervisor of Nurses n1, n3, and n4 (i.e., Nurse n3 acts as his/her own supervisor). In addition, note how each of the six relationship instances pertains to a different one of the six nurses, indicating that each nurse is supervised.

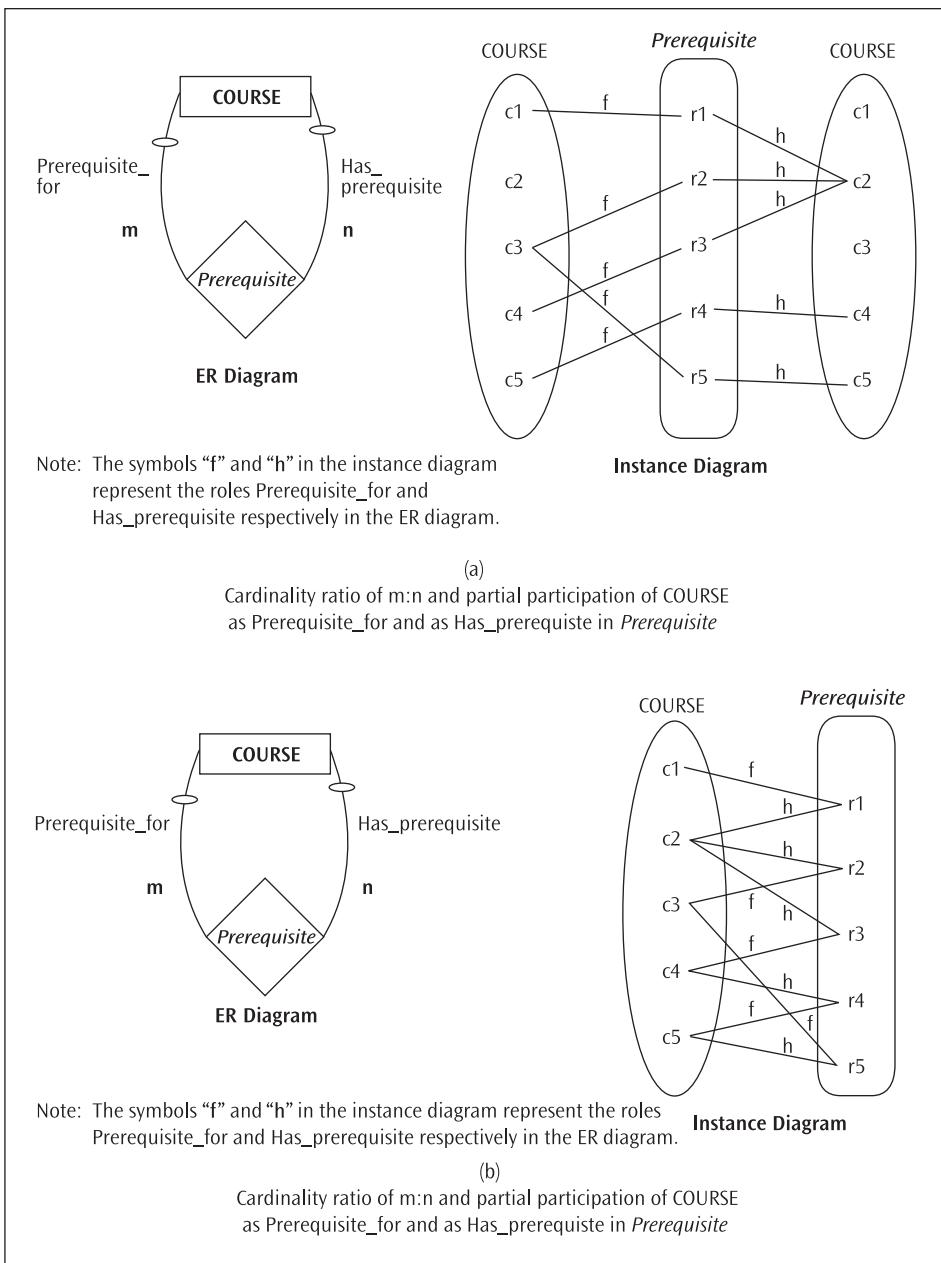
A recursive relationship type with an m:n cardinality ratio appears in Figure 2.17. In this relationship, a Course may not only serve as a prerequisite for many other courses but may also have many other courses as its prerequisites. The instance diagram in Figure 2.17a illustrates this *Prerequisite* relationship type by showing a duplicate copy of the COURSE entity type. Observe how Course c2 has Courses c1, c3, and c4 as its prerequisites. In addition, note that Course c3 is a prerequisite of Course c2 as well as of Course c5, whereas Course c2 is not a prerequisite of any other courses. Moreover, note that Courses c1 and c3 have no prerequisites. The instance diagram in Figure 2.17b illustrates the same relationships among courses in a truly recursive sense through the use of only one COURSE entity type.

The attributes described in Section 2.3.1 can also be assigned to relationship types. For example, consider the 1:n relationship type in Figure 2.18. Here, Figure 2.18a depicts the condition in which **Rent** is a mandatory attribute of DORMITORY, conveying the semantics of one fixed rent per Dormitory. If, on the other hand, **Rent** is shown as an attribute of the relationship type (Figure 2.18b), the semantics of the diagram changes to the following: The mandatory rent changes for a given dormitory based on the occupancy—that is, each student may pay a different rent even in the same dormitory. In a 1:n relationship type, attributes of the relationship can alternatively be shown as attributes of the child entity type in the relationship without altering the semantics of the relationship.

Consider the example shown in Figure 2.18c. Here, **Rent** is shown as an attribute of the entity type STUDENT, meaning that the **Rent** can be different for different Students. Thus, the semantics expressed in Figures 2.18b and 2.18c are the same. When **Rent** is included as an attribute of DORMITORY (Figure 2.18a), even if it is shown as a multi-valued attribute, it is impossible to identify the rent paid by each individual student living in the dormitory. In short, when rent varies by occupant, rent can be an attribute of STUDENT instead of Occupies without affecting the semantics. That is, an attribute of a relationship type can always be stored in the child entity type in the relationship without changing the semantics conveyed.

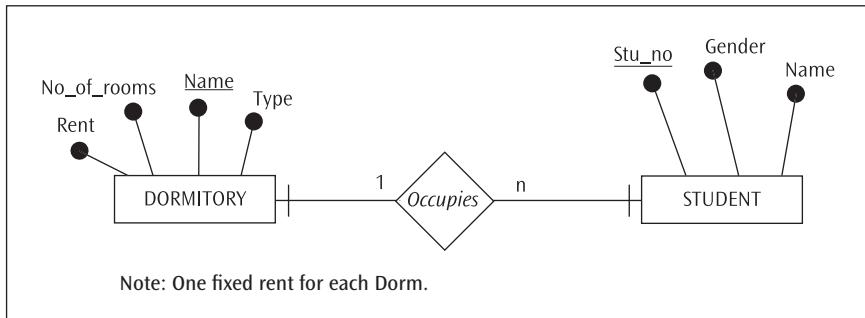
Next, consider the 1:1 relationship type *Heads* between PROFESSOR and DEPARTMENT, which is shown in Figure 2.19. In a semantic sense, the attributes **Start\_dt** and **End\_dt** belong to the relationship type *Heads* because the value of this attribute is determined based on when a particular Professor assumed the duties of the Head of a particular Department and when it was relinquished.

Here, the attributes **Start\_dt** and **End\_dt** can be included either in the entity type PROFESSOR or in the entity type DEPARTMENT because the cardinality ratio of the relationship is 1:1, implying that either entity type can be the child in the relationship.

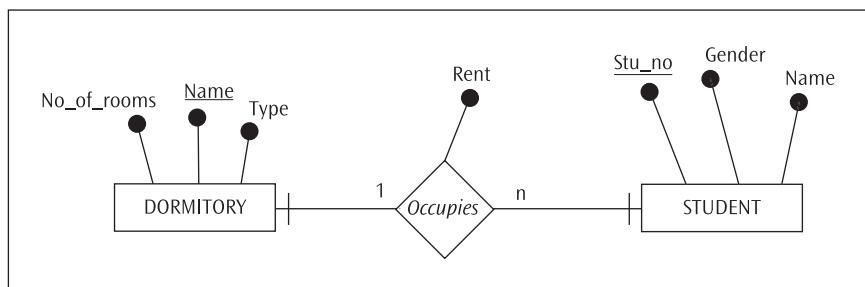


**FIGURE 2.17** Structural constraints for recursive relationships: cardinality ratio of m:n

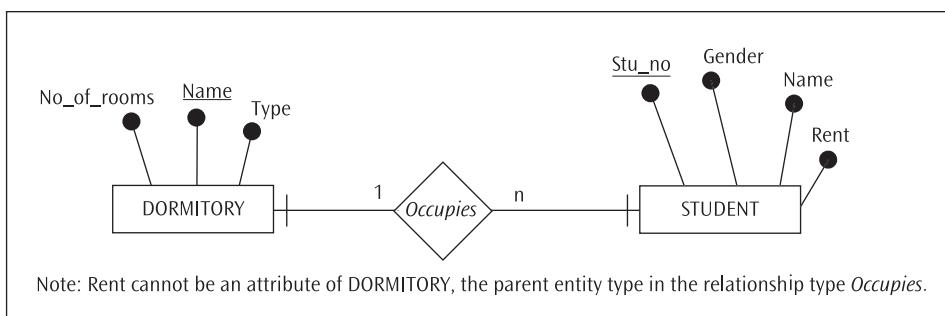
## Chapter 2



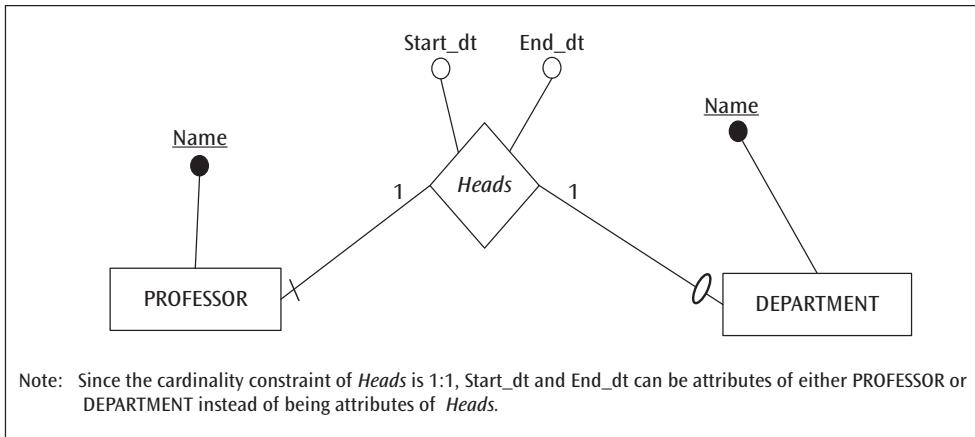
**FIGURE 2.18a** Rent as an attribute of DORMITORY, the parent in the 1:n relationship type



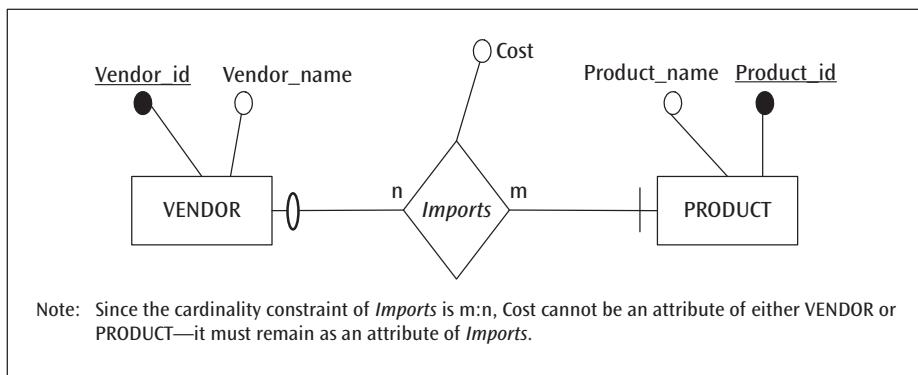
**FIGURE 2.18b** Rent as an attribute of a relationship in a 1:n relationship type



**FIGURE 2.18c** Rent as an attribute of the entity type STUDENT instead of an attribute of the relationship type *Occupies*

**FIGURE 2.19** Attributes of a relationship in a 1:1 relationship type

Attributes of m:n relationship types cannot be shown anywhere other than as attributes of the relationship type itself. For example, consider the attribute **Cost** in the *Imports* relationship type shown in Figure 2.20. Because the cost incurred by a vendor to import a product is determined by a vendor-product combination, the cost can only be specified as an attribute of *Imports* and not as an attribute of either VENDOR or PRODUCT.



(a) ER diagram

Vendor		Imports			Product	
Vendor_id	Vendor_name	Vendor_id	Product_id	Cost	Product_name	Product_id
V3	Buffet Inc.	V3	P11	7	Soup	P11
V5	Gates Inc.	V5	P11	7	Noodles	P13
V7	Jobs Inc.	V5	P17	17	Chocolate	P17
		V7	P17	19	Nuts	P19
		V7	P23	13	Coffee	P23

(b) Sample dataset

**FIGURE 2.20** An attribute of a relationship in an m:n relationship type

### 2.3.5 Base Entity Types and Weak Entity Types

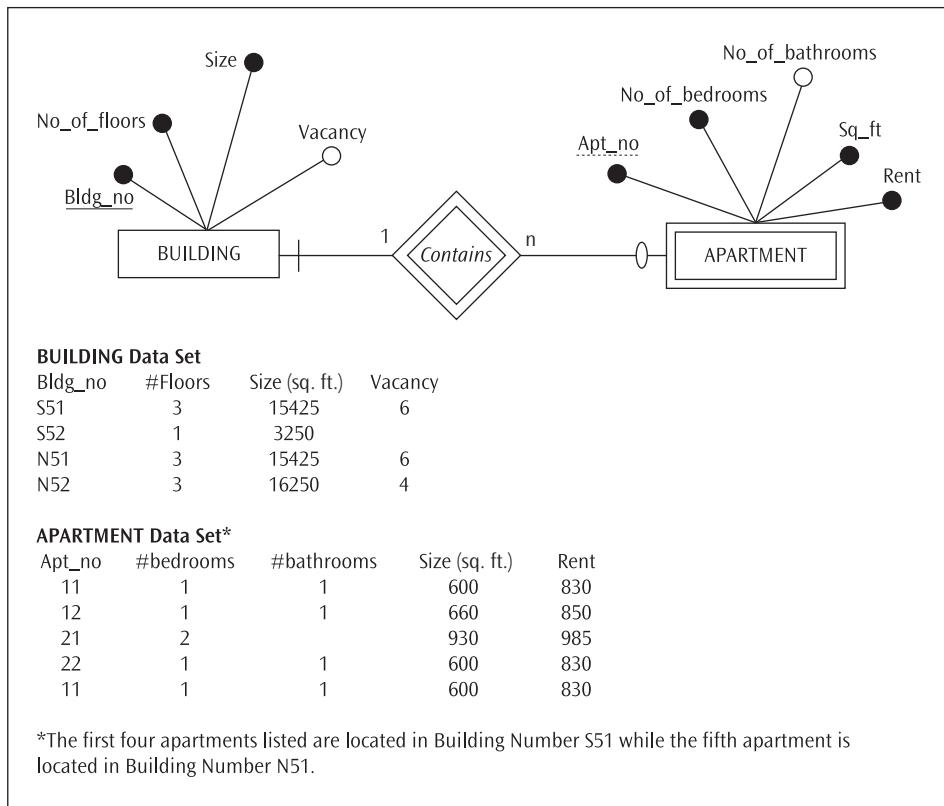
As a brief review of the relationship between entities and attributes, recall that a set of attributes gives structure to an entity type. When each attribute is given a value (at least one attribute should have a non-null value), an entity of this entity type is created. A second set of values for each of the attributes that constitute the entity type results in a second entity of this entity type. A collection of all entities of this entity type becomes an entity set of this entity type.

An entity type in which the entities have independent existence (that is, each entity is unique) is referred to as a **base** (or **strong**) **entity type**. For example, the set of attributes **Part#**, **Part\_name**, **Color**, **Cost**, **Price**, and so on could constitute the *base* entity type INVENTORY if there are no duplicate entities in that entity set. Independent existence of each entity of a base entity type is accomplished via the uniqueness of value for an attribute, atomic or composite, in the entity set. INVENTORY, for example, is a base entity type because **Part#** is different for each entity. No two entities in the INVENTORY entity set have the same **Part#** even though any two or more entities may have the same part name and/or color, and/or cost, and/or price. In other words, **Part#** is a unique identifier of INVENTORY. In another base entity type, the identification of the independent existence of entities in the entity set may require a composite attribute. For example, **Survey#**, **Lot#**, **Block#**, and **Plot#** together may be required for establishing the independent existence of entities in the base entity type PROPERTY. In short, the property of independent existence in a base entity type implies the presence of a unique identifier in the entity type. Incidentally, some base entity types may have more than one unique identifier. For example, both the composite attribute **[State, License\_plate#]** and the atomic attribute **Vehicle\_id#** are unique identifiers of an automobile.

ER modeling grammar allows for the conceptualization of an entity type that does not have independent existence—that is, an entity type that does not have its own unique identifier. Such an entity type is called a **weak entity type**. Presence of duplicate entity instances of this entity type in the entity set is legal. Independent existence of a weak entity can be achieved only by borrowing part or all of its unique identifier from another entity (or other entities) through one or more of a special type of relationship called an “identifying relationship.” Such a dependence of a weak entity on other entities is called “identification dependency.” For example, several apartments in an apartment complex may have exactly the same properties (e.g., size, number of bedrooms, number of bathrooms, etc.), including the same apartment number. That is, the same apartment number may appear in different buildings of the complex. Therefore, the unique identification of an apartment is impossible without associating it with a building, assuming that each building has an independent existence. Semantically, this means that an apartment cannot exist apart from the building in which it is located.

Figure 2.21 illustrates the relationship between a BUILDING entity type and an APARTMENT entity type and shows their respective data sets. BUILDING is a base entity type because it has a unique identifier, **Bldg\_no**. APARTMENT, however, is a weak

entity type because it does not have a unique identifier of its own. To distinguish a base entity type from a weak entity type, as shown in Figure 2.21, the weak entity type is depicted as a double rectangular box. To signify the identification dependency of APARTMENT on BUILDING, a double diamond is used to portray the identifying relationship type, *Contains*. Recall that a single diamond is used to represent a “regular” relationship between entity types.



**FIGURE 2.21** Example of a weak entity type

An atomic or composite attribute in a weak entity type (which, in conjunction with a unique identifier of the parent entity type in the identifying relationship type, uniquely identifies weak entities) is called the **partial key** of the weak entity type and is denoted by a dotted underline. The partial key of a weak entity type is sometimes referred to as a **discriminator**. The sample data shown in the BUILDING and APARTMENT data sets in Figure 2.21 illustrate how **Apt\_no** is unique for the apartments within Building S51 but is not unique within the APARTMENT data set. In order to identify each apartment in the apartment complex, the unique identifier of the BUILDING in which the apartment is located (**Bldg\_no**) must be concatenated with the partial key of APARTMENT (**Apt\_no**).

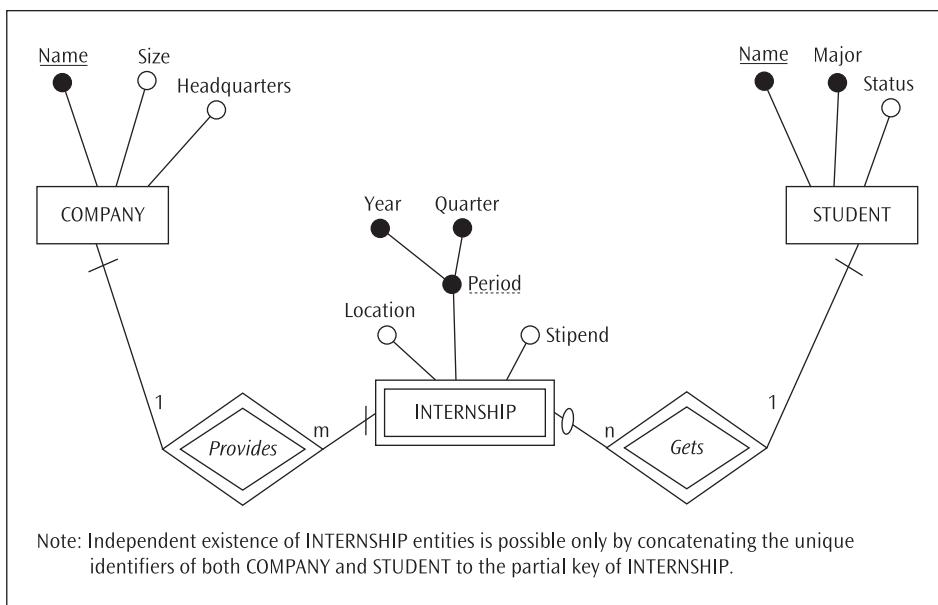
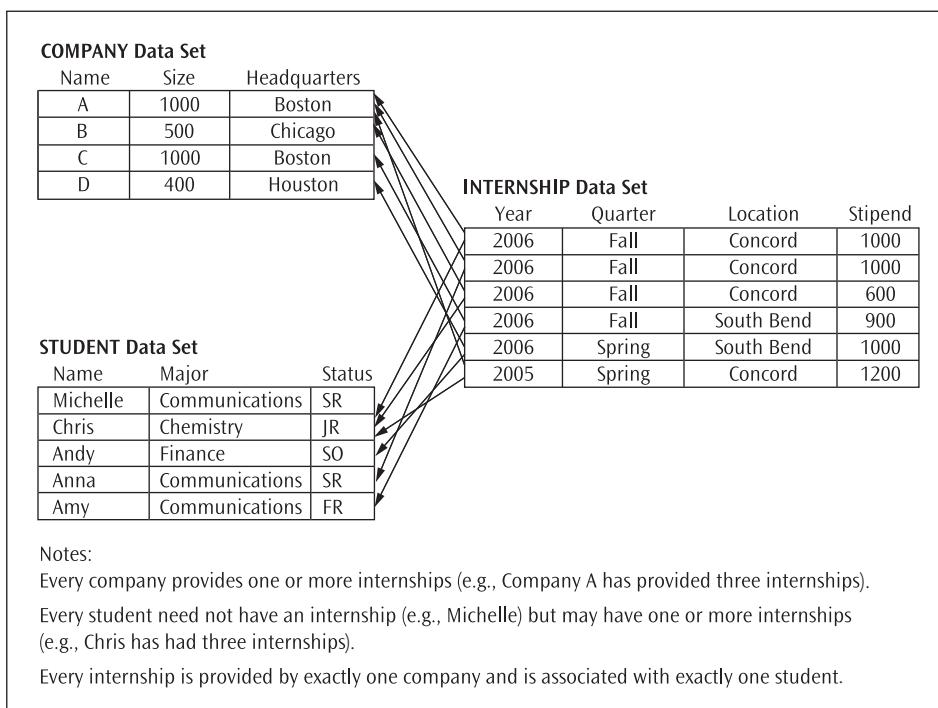
Recall that existence dependency of an entity type in a relationship type implies mandatory participation of all entities in the entity set of said entity type in that relationship (see Figures 2.12, 2.13, 2.14, and 2.15 for examples). Note that any entity type (base or weak), in either possible role (parent or child) in a relationship type, is existent-dependent on the relationship type if the participation constraint on the entity type in this relationship type is total. By virtue of being identification-dependent on the identifying parent entity type, a weak entity type is always also existent-dependent on the corresponding identifying relationship type. In the previous example (Figure 2.21), APARTMENT has existence dependency on *Contains* because APARTMENT has identification dependency on BUILDING through the identifying relationship type, *Contains*.

Table 2.3 compares and contrasts existence dependency and identification dependency.

Existence Dependency	Identification Dependency
When the <i>participation of an entity type in a relationship type</i> is total ( $\min = 1$ ), the entity type is said to have existence dependency on the relationship type irrespective of the other entity type(s) present in the relationship.	When an <i>entity type is dependent on (an)other entity type(s)</i> for its unique identification, the dependent entity type is said to have identification (ID) dependency on the identifying entity type(s) in a relationship.
<i>Existence dependency is on a relationship</i> irrespective of the type of entity type (base, weak, cluster), number of entity types in the relationship, or the role of the existent-dependent entity type (parent, child) in the relationship.	<i>ID dependency is on entity type(s)</i> in which a weak/gerund entity type is dependent for its unique identification through as many identifying relationships as there are identifying parent entity types. <i>The ID-dependent weak/gerund entity type is always the child in the identifying relationship type(s).</i>
Existential dependency can be on a regular or identifying relationship type and is <i>not limited by the role of the entity type (parent, child)</i> in the relationship.	ID dependency always involves only identifying relationship type(s) and <i>pertains only to the child (weak/gerund) entity type</i> in the identifying relationship.
It is possible for existence dependency to occur without the presence of ID dependency—that is, <i>existence dependency is independent of ID dependency</i> .	<i>It is impossible for ID dependency to occur unaccompanied by existence dependency</i> of the ID-dependent child in the identifying relationship(s). ID-dependency on an entity type implies existence dependency on the relationship with that entity type.
An entity type existent-dependent on a relationship type can never have partial participation in that relationship.	An entity type ID-dependent <i>through a relationship type</i> can never have partial participation in that relationship.

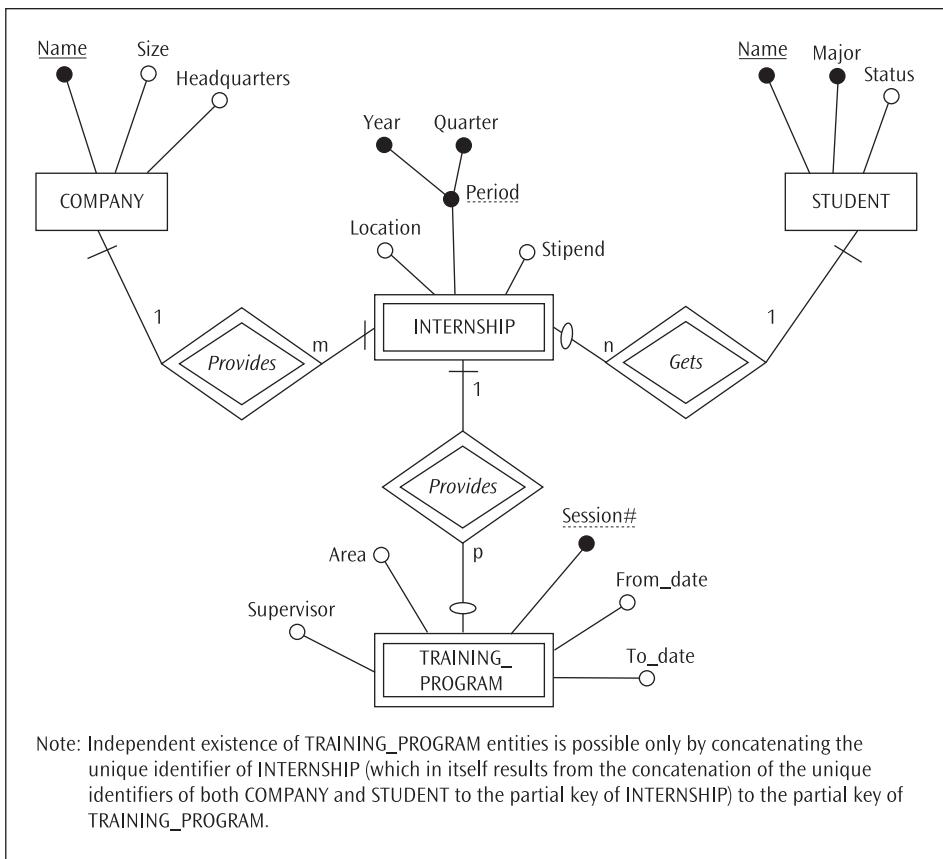
**TABLE 2.3** Existence dependency and identification dependency compared

Additional examples of a weak entity type appear in Figures 2.22 through 2.26. Note in Figure 2.22 that the weak entity type INTERNSHIP has a collective identification dependency on both COMPANY and STUDENT via two independent identifying relationships. Sample data for Figure 2.22 is shown in Figure 2.23.

**FIGURE 2.22** Example of a weak entity type with multiple identifying parents**FIGURE 2.23** Sample data for the COMPANY, STUDENT, and INTERNSHIP entity types in Figure 2.22

## Chapter 2

As a second example, Figure 2.24 portrays a weak entity type, TRAINING\_PROGRAM, that is identification-dependent on another weak entity type, INTERNSHIP. Sample data sets are shown in Figure 2.25. Note that in both cases, the identification-dependent weak entity type is also existent-dependent on the identifying relationship type.

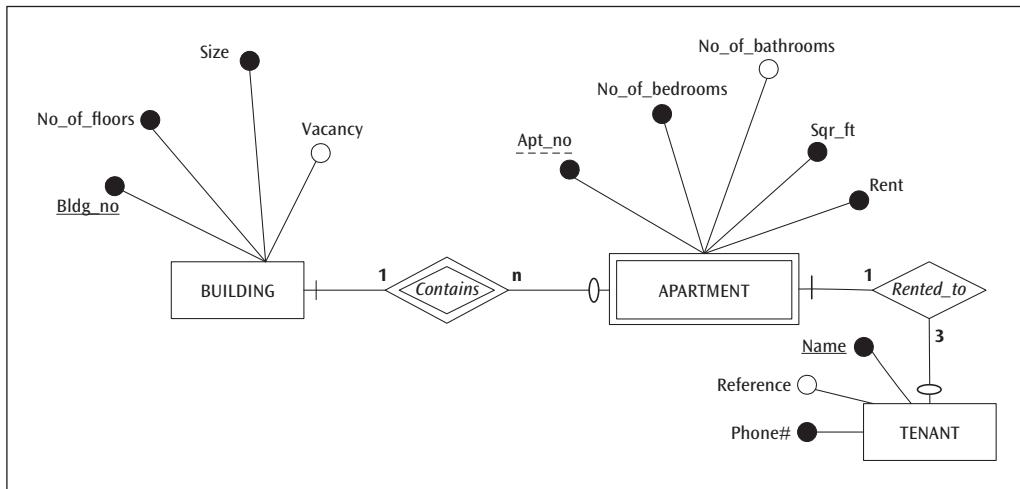


**FIGURE 2.24** A weak entity type, TRAINING\_PROGRAM, that is identity-dependent on another weak entity type, INTERNSHIP

INTERNSHIP Data Set				TRAINING_PROGRAM Data Set				
Year	Quarter	Location	Stipend	Session#	Supervisor	Area	From_date	To_date
2006	Fall	Concord	1000					
2006	Fall	Concord	1000	3	Ken	Microsoft Office	12/2/2006	12/7/2006
2006	Fall	Concord	600	3	Ken	Open Office	12/10/2006	12/13/2006
2006	Fall	South Bend	900	2	Ken	Microsoft Office	12/2/2006	12/7/2006
2006	Spring	South Bend	1000	3	John	Microsoft Project	12/10/2006	12/12/2006
2005	Spring	Concord	1200					

**FIGURE 2.25** Sample data for the INTERNSHIP and TRAINING\_PROGRAM entity types of Figure 2.24

The final example in this section demonstrates how a weak entity type, though identification-dependent on some base entity type(s), may also participate in other “regular” (non-identifying) relationship(s). The example shown in Figure 2.26 is a minor extension to the example discussed earlier using Figure 2.21.



**FIGURE 2.26** A weak entity type participating in a “regular” relationship

Apartments are rented to tenants. The structural constraints of this regular relationship indicate that an apartment need not be rented (stay vacant) or can be rented to no more than three tenants. A tenant, on the other hand, must rent an apartment. (Perhaps that is why the entity type is named TENANT!) Also, a tenant cannot rent more than one apartment. Observe that in this regular 1:n relationship, not only is the weak entity type APARTMENT not identification-dependent on the base entity type TENANT, it is not even existent-dependent on the relationship type *Rented\_to*. Also, the weak entity type APARTMENT is the parent in the *Rented\_to* relationship and the base entity type TENANT, and the child in the *Rented\_to* relationship is existent-dependent on *Rented\_to*. Finally, since the value of n (maximum cardinality) in this 1:n relationship happened to be known as 3, the specification indicates the cardinality constraint as 1:3.

### 2.3.6 Cluster Entity Type: A Brief Introduction

A **cluster entity type** is a virtual entity type emerging from a grouping operation on a collection of entity types and the relationship(s) among them. Sometimes referred to as an “aggregate entity type” or a “composite/molecular object type,” a cluster entity type does not have a “real” existence, unlike a base or weak entity type.

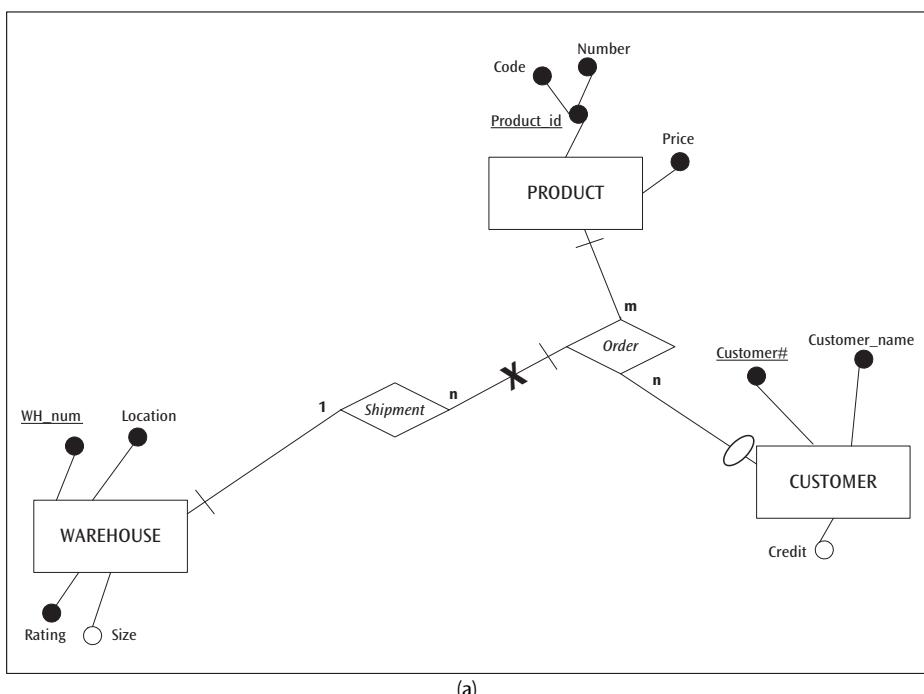
Imagine a scenario in which a company ships Products to Customers from its Warehouses. All customers ought to receive some Product(s) from the warehouse. (Perhaps that is why they are called Customers!) Some not-so-popular Products may just sit on the shelves of the Warehouses, unordered. Every Warehouse ought to ship Products to Customers. Clearly, a Customer should receive a specific Product shipped from only one Warehouse—that is, there is no duplicate shipping.

Perhaps the initial inclination is to model this scenario as a ternary relationship among the three entity types: PRODUCT, WAREHOUSE, and CUSTOMER. But a ternary relationship would enable an uncontrolled opportunity for any Warehouse to ship any

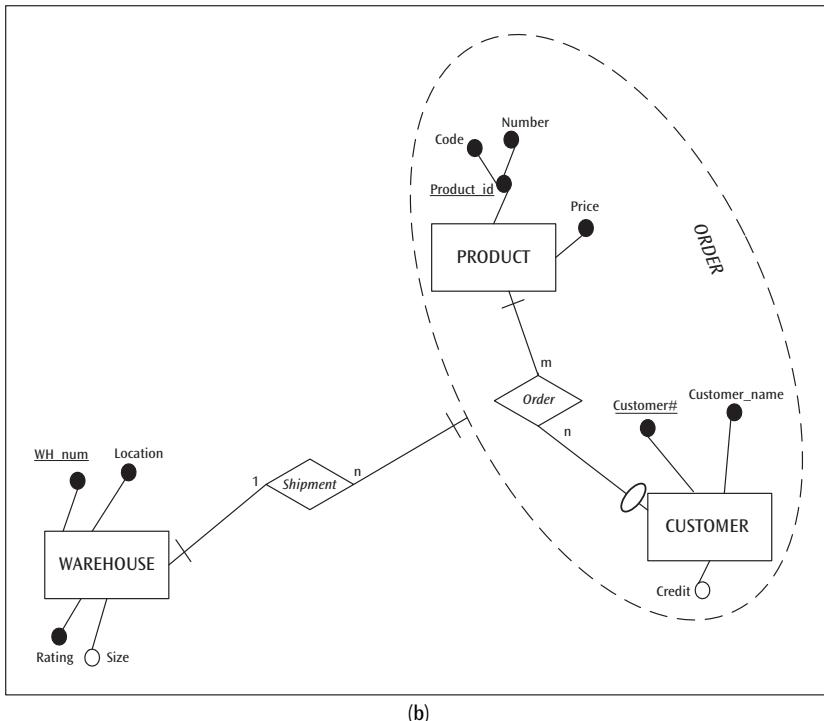
Product to any Customer. In fact, in the extreme case, the model would permit all ordered Products to be shipped to all Customers by all the Warehouses. Rich modeling constructs like a ternary relationship type can result in unmanageable relationship patterns; therefore, they require careful evaluation before being incorporated. Figure 2.27a captures the semantics conveyed by a business rule that prohibits more than one Warehouse from shipping the same Product to the same Customer. However, the ER modeling grammar does not permit a relationship type linked to another relationship type—a syntactic error in the ER modeling grammar. In order to resolve this issue, a new construct labeled “cluster entity type” is introduced in Figure 2.27b. The virtual entity type labeled ORDER denoted by a dotted line encircling the composite object PRODUCT-Order-CUSTOMER is referred to as the cluster entity type. Observe that in Figure 2.27b the relationship *Shipment* connects the base entity type WAREHOUSE to the cluster entity type ORDER, thus averting the syntactic error found in Figure 2.27a. The structural constraints of the relationship type *Shipment* now ensure that a Customer can get an ordered Product shipped from only one Warehouse. The use and value of cluster entity types is covered in great detail in Chapter 5.

### 2.3.7 Specification of Deletion Constraints

In a relationship between entity types, deletion of an entity from an entity set has implications. A valuable construct in the ER modeling grammar that handles this condition is the **deletion constraint**. As has been the case so far in this chapter, only binary relationship types are being considered at this point. Higher-degree relationship types are discussed in Chapter 5. In the context of every relationship type in an ER model, the deletion of an entity from the entity set of the parent entity type in the relationship requires specific action either in the parent entity set or in the child entity set in order to maintain consistency of the relationships



**FIGURE 2.27** Customer order for a product shipped from only one warehouse



**FIGURE 2.27** Customer order for a product shipped from only one warehouse  
(continued)

in the database. Note that no action is needed when an entity is deleted from the entity set of the child entity type in the relationship except when the cardinality constraint of the relationship is 1:1. When the cardinality constraint of a relationship type is 1:1, both the entity types in the relationship can be a parent or a child and should be evaluated accordingly.

Four rules apply to deletion constraints: the restrict rule, the cascade rule, the set null rule, and the set default rule.<sup>10</sup> Here are descriptions of each:

- When an attempt is made to delete a parent entity in a relationship, if the deletion should be disallowed when child entities related to this parent in this relationship exist, the **restrict rule** is specified on the parent entity type in the relationship.
- When an attempt is made to delete a parent entity in a relationship, if all child entities related to this parent in this relationship should be deleted along with the parent entity, the **cascade rule** is specified on the child entity type in the relationship.
- When an attempt is made to delete a parent entity in a relationship, if all child entities related to this parent in this relationship should be retained but no

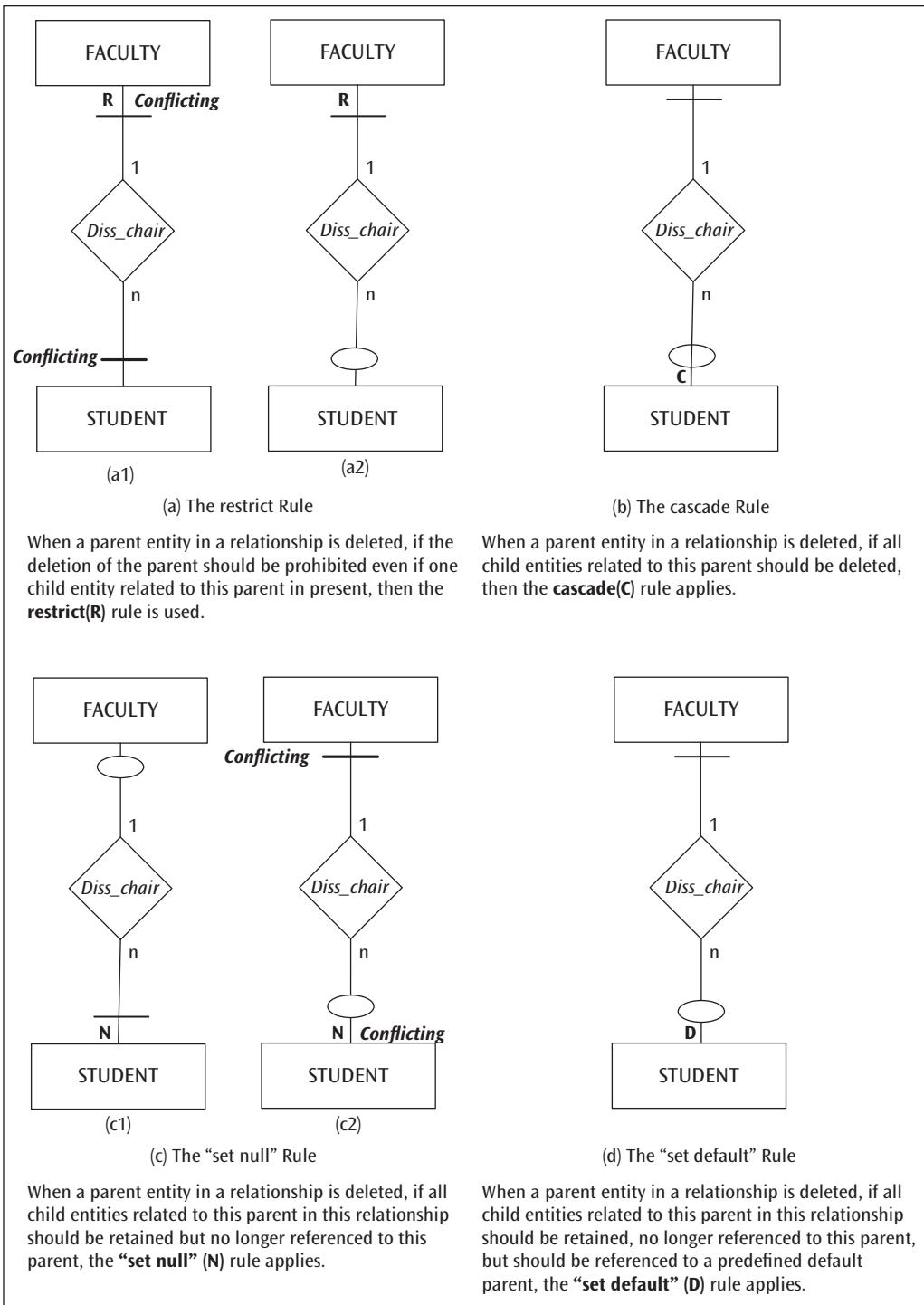
<sup>10</sup>Some argue that the deletion constraints belong in physical database design. It is our view that the semantics for the deletion constraints also emerges from user-specified business rules and ought to be captured, modeled, and passed through the data modeling tiers. A similar constraint is necessary on those occasions when the value of the unique identifier of the parent entity is changed. This type of constraint is called an update constraint. This is discussed in Section 10.1.1.1 of Chapter 10.

- longer referenced to this parent while the deletion of the parent entity is allowed, the **set null rule** is specified on the child entity type in the relationship.
- When an attempt is made to delete a parent entity in a relationship, if all child entities related to this parent in this relationship should be retained despite the deletion of the parent entity type by shifting the parent reference to a predefined default parent, the **set default rule** is specified on the child entity type in the relationship.

Figure 2.28 illustrates these deletion rules in the context of a relationship between the entity type FACULTY and the entity type STUDENT. In Figure 2.28a, the restrict rule (R) prohibits the deletion of a faculty member serving as the dissertation chair of one or more Ph.D. students. Observe the conflict between the restrict rule and the total participation constraint on FACULTY in this relationship. Since every faculty member must participate in the *Dis\_chair* relationship, meaning must be the dissertation chair of some Phd\_students(s), the restrict rule here (Figure 2.28a1) prevents the deletion of any faculty member from this entity set ever. In other words, though syntactically correct, this condition is semantically almost always incorrect since restricting any entity set from deletion forever is highly improbable, if not impossible, in a typical application domain. Figure 2.28a2 remedies this semantic error through the specification of partial participation of FACULTY in the *Dis\_chair* relationship when the deletion rule imposed is “restrict.” On the other hand, in Figure 2.28b, the cascade rule (C) implies that the deletion of a faculty member leads to the deletion of all Ph.D. students for whom the faculty member serves as dissertation chair. The set null rule (N) allows a Ph.D. student to exist without a dissertation chair by simply nullifying the relationship of the Ph.D. student with the faculty member, should the faculty member be removed from the FACULTY entity set (see Figure 2.28c). Here, observe how total participation in Figure 2.28e is incompatible with the set null rule and must therefore be corrected to permit partial participation of STUDENT in the *Diss\_chair* relationship. Alternatively, when the total participation of the Ph.D. student in the relationship is a necessary condition, the set null rule must be replaced by a compatible deletion rule. Finally, the set default rule (D) portrayed in Figure 2.28d is somewhat similar to the set null rule. Here, instead of nullifying the relationship, the Ph.D. student is linked to a predetermined (default) dissertation chair, should the student’s current dissertation chair be deleted. Conventionally, when a deletion constraint is not specified, the restrict rule is implied.

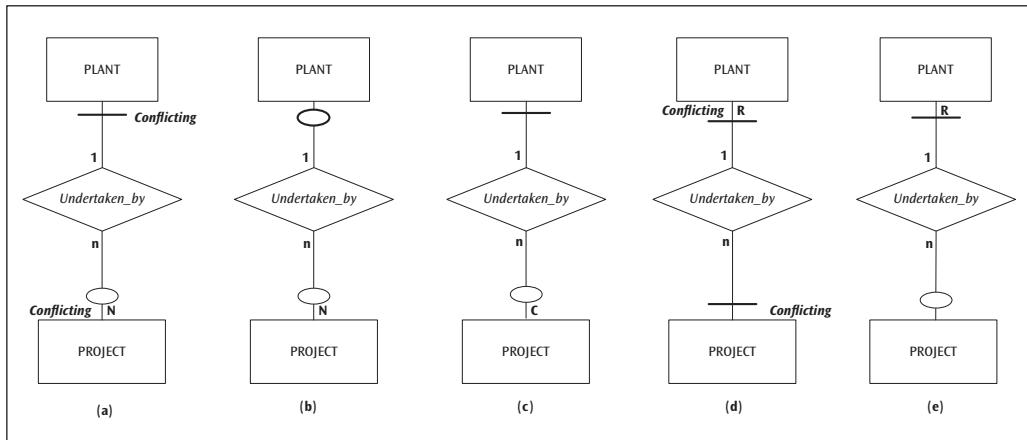
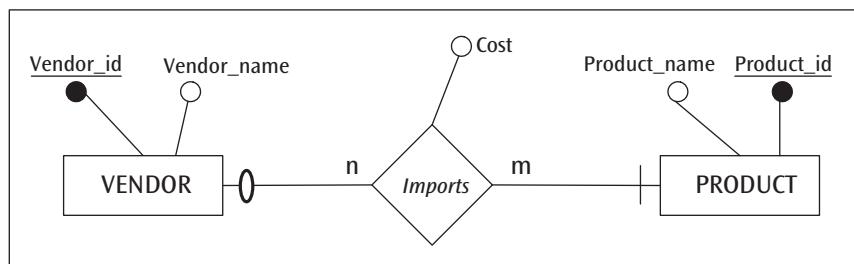
A second example for the application of deletion rules for a 1:n relationship type is shown in Figure 2.29. Observe the conflict between participation constraint and deletion rule in Figures 2.29a and 2.29d.

In an m:n binary relationship type, both entity types hold the role of a parent since both a 1:n relationship and a 1:m relationship underlie an m:n relationship where the relationship serves as a pseudo-entity type taking on the role of the child in both the underlying relationships. For this reason, sometimes the relationship type in an m:n relationship is referred to as a “relationship entity type” or more often an “associative entity type.” Figure 2.30 portrays an extension of the m:n relationship exhibited in Figure 2.20, highlighting the expression of the relationship type as an associative entity type (see Figure 2.30c). The sample dataset (Figure 2.30b) can facilitate understanding of the structure of the associative entity type as one where the entities are uniquely identified by the concatenation of the unique identifiers of the two parent entity types in the relationship. An important inference ensues from this: The relationship type in an m:n relationship (associative entity type) will have total participation (existence dependency) in the relationships with both the parent entity types.



**FIGURE 2.28** Deletion rules applied to a 1:n relationship type

## Chapter 2

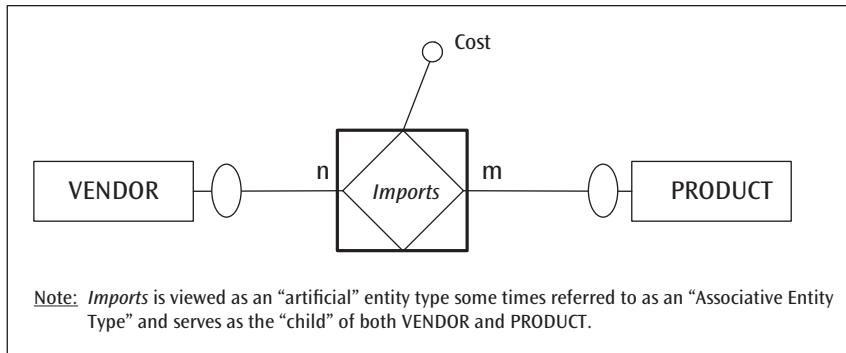
**FIGURE 2.29** Application of deletion rules in a 1:n relationship type: Another example

(a) An m:n relationship type

Vendor		Imports			Product		
Vendor_id	Vendor_name	Vendor_id	Product_id	Cost	Product_name	Product_id	
V3	Buffet Inc	V3	P11	7	Soup	P11	
V5	Gates Inc.	V5	P11	7	Noodles	P13	
V7	Jobs Inc.	V5	P17	17	Chocolate	P17	
		V7	P17	19	Nuts	P19	
		V7	P23	13	Coffee	P23	

(b) Sample dataset for the ER diagram in 2.30a

**FIGURE 2.30** An m:n relationship type depicted as an associative entity type

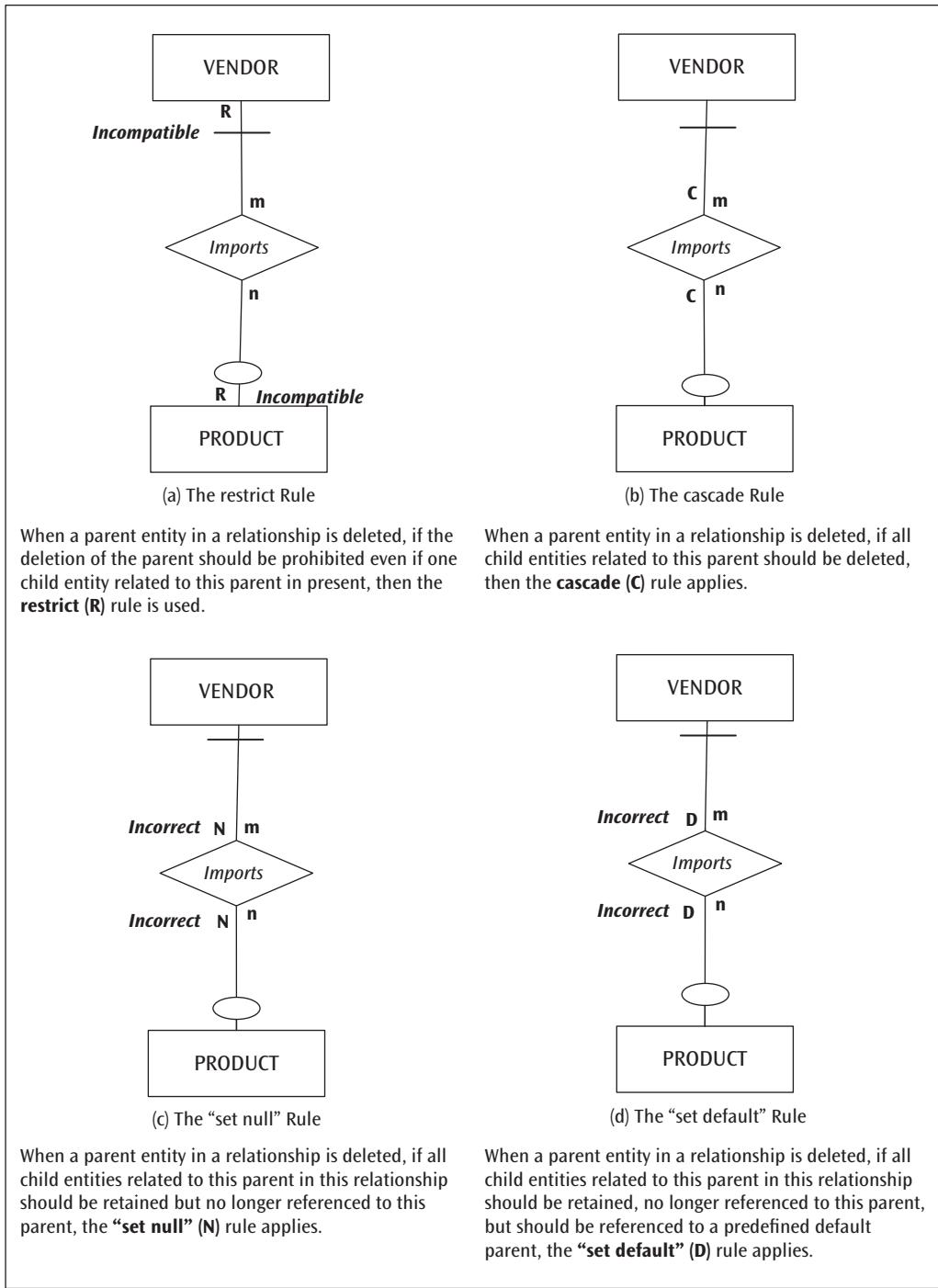


(c) An associative entity type

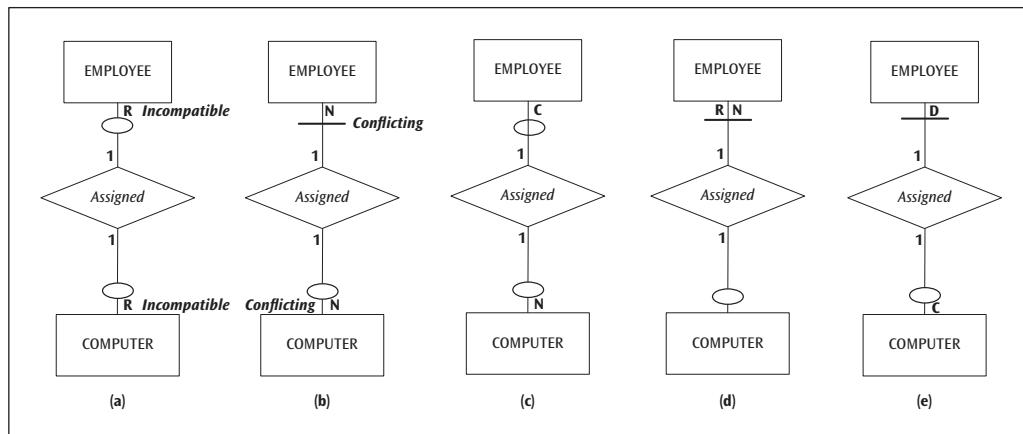
**FIGURE 2.30** An m:n relationship type depicted as an associative entity type (continued)

This portrayal further clarifies that an m:n relationship entails two deletion rules since there are two parents for one child in this relationship. Figure 2.31 explicates the role of deletion constraints in an m:n relationship type. Observe that there are two deletion constraints in the m:n relationship type. In Figure 2.31a, the “R” next to VENDOR informs that when an entity from the entity set of the VENDOR entity type is deleted, if that entity participates in the *Imports* relationship type (meaning that it is related to one or more Product entities), then the deletion of the vendor entity is disallowed. Since the participation of VENDOR in the *Imports* relationship is optional, the restrict rule on VENDOR is compatible with the [partial] participation constraint on VENDOR. On the contrary, the “R” next to PRODUCT restricting the deletion of a Product entity if it participates in the *Imports* relationship, though correct in syntax, is incompatible with the [total] participation constraint imposed on PRODUCT in this relationship. Please note how the deletion constraints C, N, and D prevailing on the child entity type appear next to the relationship type *Imports* in Figures 2.31(b), (c), and (d). The set null constraint (N), however, cannot ever be imposed on a relationship type (associative entity type) of an m:n relationship since nullifying the relationship here entails disabling the unique identification of the associative entity type. The same logic applies to the set default (D) constraint. It is critical to understand that the cascade constraint pertaining to the deletion of an entity type—say, VENDOR in an m:n relationship—does not imply cascading deletion of related entities from the PRODUCT entity set. The cascading deletion (C) applies to the related instances in the relationship—that is, entities of the associative entity type. The same logic applies to the set default (D) constraint.

## Chapter 2

**FIGURE 2.31** Deletion rules applied to an m:n relationship type

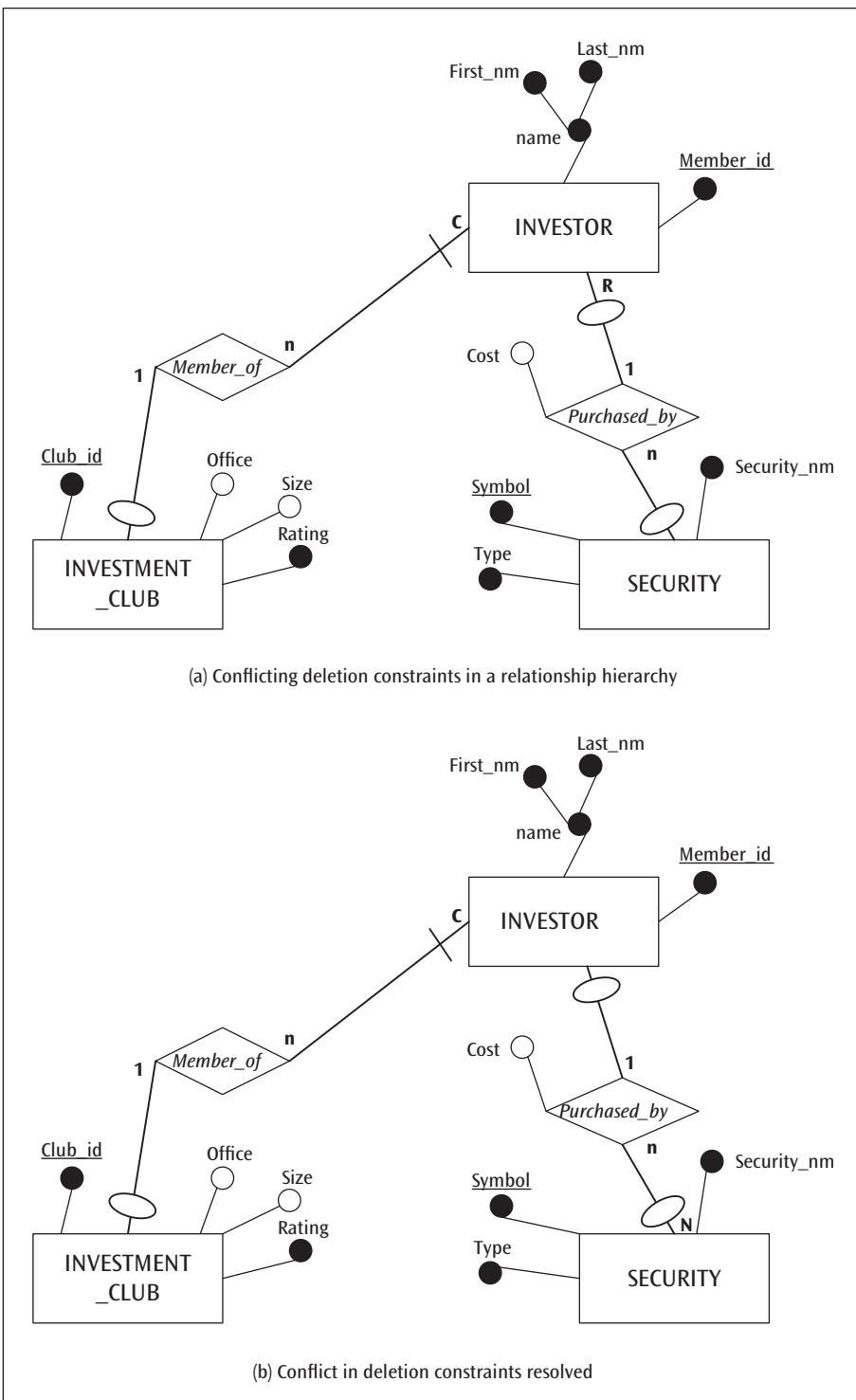
Application of deletion rules on 1:1 relationships also requires careful consideration. Figure 2.32 presents a few scenarios of a 1:1 relationship and the specification of deletion constraints in the relationship. We have already seen that the deletion constraint restrict (R) on the parent entity type that has total participation (existence dependency) in a relationship type, though correct in syntax, is incompatible with the semantics of the relationship type. Restricting any entity set from deletion forever is highly improbable, if not impossible, in a typical application domain (Figure 2.32a). Likewise, in Figure 2.32b, total participation of the COMPUTER entity type in the *Assigned* relationship conflicts with the set null option of the deletion constraint since nullifying the participation of an entity in the entity set of the COMPUTER entity type implies partial participation of COMPUTER in the *Assigned* relationship type. In fact, in Figure 2.32c this conflict is resolved—the COMPUTER entity type in the role of a child in the *Assigned* relationship type is no longer existent-dependent on the *Assigned* relationship type, thus allowing the set null deletion constraint to prevail on COMPUTER. On the other hand, when COMPUTER is considered in the parent role, deletion of a computer entity is set to trigger a cascading deletion of the employee entity if one is participating in the *Assigned* relationship. The cascade deletion constraint on EMPLOYEE is equally applicable even if EMPLOYEE is existent-dependent (total participation) on the *Assigned* relationship type.



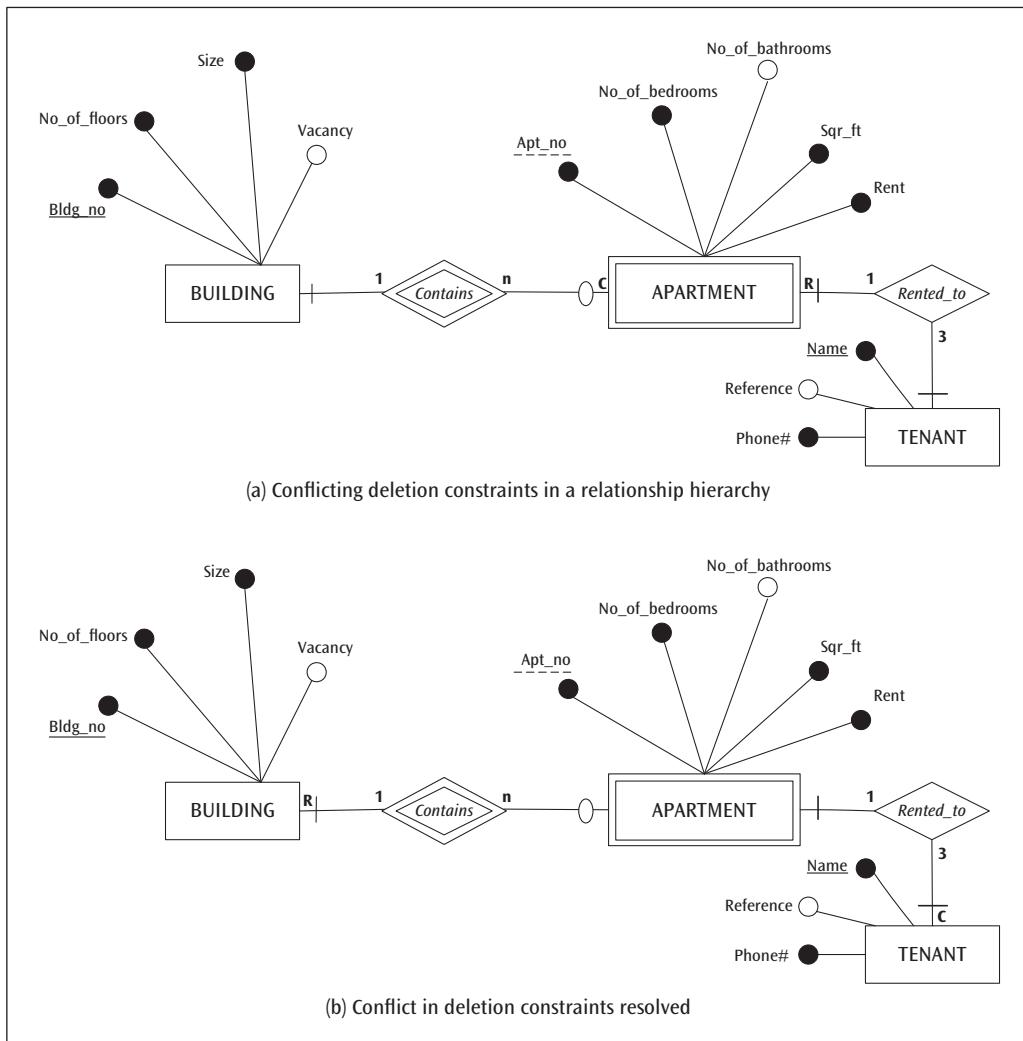
**FIGURE 2.32** Deletion rules applied to a 1:1 relationship type

Observe that in a 1:1 binary relationship type, since both participating entity types can be the parent and child of each other, two deletion rules will have to be imposed. Figure 2.32d represents the condition where the deletion of the parent entity type EMPLOYEE is prohibited if an employee entity participates in the *Assigned* relationship type. But then, the partial participation of EMPLOYEE in the *Assigned* relationship type is in synch with the restrict deletion constraint imposed on EMPLOYEE. The set null deletion constraint also imposed on EMPLOYEE pertains to the role of EMPLOYEE as the child in the *Assigned* relationship type. The set default deletion rule applicable only on the child entity type (similar to set null and cascade) is demonstrated in Figure 2.32e.

Deletion rules, if not carefully specified, may also cause conflict in a hierarchy of relationships. An example is presented in Figure 2.33. If an Investment Club closes down, the cascade deletion constraint in the *Member\_of* relationship type specifies that all the Investors related to this specific Investment Club be removed (Figure 2.33a). However, the restrict deletion constraint imposed on the INVESTOR entity type in the *Purchased\_by* relationship type requires that an Investor entity not be removed if it participates in the *Purchased\_by* relationship type—that is, if that Investor is related to one or more Security entities. Thus, the INVESTOR entity type is subject to conflicting deletion constraints through the two relationship types, *Purchased\_by* and *Member\_of*, in which it participates. A way to resolve this conflict by changing just one of the deletion constraints is demonstrated in Figure 2.33b—that is, by replacing the restrict deletion constraint on the parent entity type INVESTOR in the *Purchased\_by* relationship type by the set null deletion constraint on the child entity type SECURITY. A second illustration stems from an earlier example (Figure 2.26) reproduced in Figure 2.34. As in the previous example, the cascade deletion constraint on APARTMENT in the *Contains* relationship type and the restrict deletion constraint on it in the *Rented\_to* relationship type are mutually conflicting (Figure 2.34a). Observe that the resolution of this conflict is accomplished here to demonstrate an alternative solution. The cascade deletion constraint on APARTMENT in the *Contains* relationship type is replaced by a restrict deletion constraint on the parent entity type of the *Contains* relationship type—that is, BUILDING. However, the solution is not quite complete; notwithstanding the resolution of the conflict just identified, the restrict deletion constraint on APARTMENT in the *Rented\_to* relationship type is incompatible with the total participation of APARTMENT in this relationship type. Replacing the restrict deletion constraint on APARTMENT in the *Rented\_to* relationship type with a cascade deletion constraint on the child entity type, TENANT, of *Rented\_to* solves this problem, as shown in Figure 2.34b.

**FIGURE 2.33** Deletion constraints in a relationship hierarchy

## Chapter 2

**FIGURE 2.34** Deletion rules of a relationship hierarchy

In summary, deletion rules are also business rules and, therefore, an inherent part of the requirements specification. Deletion constraints express the deletion rules in terms of modeling specifications. In a PCR, the deletion constraint restrict (R) always entails an action on the parent entity type, whereas the other three constraints (N, C, D) always result in actions on the child. Deletion rules cannot be arbitrarily applied in a relationship type. Here are a few guidelines in this regard:

- Specification of the set null constraint on an entity type existent-dependent on the relationship type is invalid.
- Specification of the set null constraint on a weak entity type in the identifying relationship(s) type is invalid since a weak entity type is always existent-dependent on its identifying relationship(s).

- Specification of the set null constraint is invalid in an m:n binary relationship type since the relationship type serving the role of child entity type for two parent entity types is always existent-dependent on its relationships with both the parents.
- Specification of the restrict constraint (either explicitly or by default) on an entity type existent-dependent on the relationship type is invalid.
- In the case of a 1:1 binary relationship type, simultaneous specification of the restrict constraint, either explicitly or by default, on both the participating entity types is incorrect since mutually referencing entities of the two entity types will result in a deadlock—that is, deletion of either entity may get restricted by the other, consequently neither can ever be deleted.
- Similar conflicts may arise in a hierarchy of relationships where the cascade constraint specified at one level may contradict the restrict constraint specified at the next lower level in the hierarchy.

## Chapter Summary

The fundamental constructs and rules for conceptual data modeling can be understood using the Wand and Weber (2002) framework for research in conceptual modeling. The entity-relationship (ER) modeling grammar is a popular tool for conceptual data modeling.

The entity-relationship (ER) model was developed by Peter Chen in 1976. Using this model, object types are conceptualized as entity types. Objects belonging to an object type are considered to be entities of the corresponding entity type. Properties of an object type are represented as attributes of an entity type. An entity is created when a value is supplied for each attribute. Some, but not all, attribute values can be null. Associations exist among objects of different object types. In conceptual modeling, these associations are referred to as relationships among entity types.

An attribute possesses a number of characteristics. These include a name, a data type, and a class (atomic or composite). Furthermore, an attribute can be stored or derived, single or multi-valued, and mandatory or optional. An atomic attribute or collection of atomic attributes (i.e., a composite attribute) can serve as a unique identifier of an entity type. Every attribute plays only one of three roles in an entity type. It is a key attribute, a non-key attribute, or a unique identifier. Any attribute that is a constituent part of a unique identifier is a key attribute. An attribute that is not a constituent part of a unique identifier is a non-key attribute.

Business rules supplied by the users expressed in terms of constraints allow data integrity to be achieved. At the conceptual tier of data modeling, two types of data integrity constraints must be specified: (a) the domain constraint imposed on an attribute to ensure that its observed value is not outside the defined domain, and (b) the key (or uniqueness) constraint, which requires entities of an entity type to be uniquely identifiable.

A relationship type is a meaningful association among entity types. The degree of a relationship is defined as the number of entity types participating in a relationship type. A relationship type is said to be binary or “of degree two” when two entity types are involved. Relationship types that involve three entity types (of degree three) are referred to as “ternary relationships,” whereas relationships that involve four or more entity types are referred to as “n-ary relationships.” An entity type related to itself is termed a “recursive relationship type.” A relationship type is not fully specified until two structural constraints are explicitly imposed. These are the cardinality constraint and the participation constraint. Role names are used to indicate the participation of entity types in relationship types. In addition, a relationship type can have attributes.

An entity type where the entities have independent existences (that is, where each entity is unique) is referred to as a base or strong entity type. An entity type that does not have independent existence (where some entities in the entity set may be identical) is known as a weak entity type. An attribute, atomic or composite, in a weak entity type, which in conjunction with a unique identifier of the parent entity type in the identifying relationship type uniquely identifies weak entities, is called the partial key or the discriminator of the weak entity type.

A cluster entity type is a virtual entity type emerging from a grouping operation on a collection of entity types and the relationship(s) among them. It does not have a “real” existence, unlike a base or weak entity type. Nonetheless, as an ER modeling grammar construct, it enriches conceptual modeling. A brief introduction of the cluster entity type I presented in this chapter. More sophisticated use for this construct appears in chapter 5.

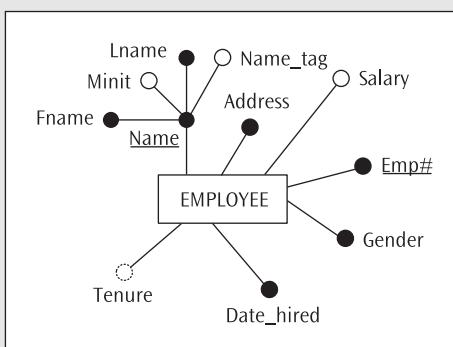
In the context of every relationship type in an ER model, the deletion of an entity from the entity set of the parent entity type in the relationship requires specific action either in the parent entity set or in the child entity set in order to maintain consistency of the relationships in the database. It must be noted that the semantics for the deletion constraints emerge from user-specified business

rules and should therefore be part of data modeling right from the start. Four distinct rules that apply to deletion constraints are discussed as the last topic in this chapter.

## Exercises

---

1. What is the difference between the conceptual world and the real world? Is it possible for a conceptual model to represent reality in total? Why or why not?
2. Use examples to distinguish between the following:
  - a. an object type and an entity type
  - b. an object and an entity
  - c. a property and an attribute
  - d. an entity and an entity instance
  - e. an association and a relationship
  - f. an object class and an entity class
3. Describe various data types associated with attributes.
4. What is the difference between a stored attribute and a derived attribute?
5. What would be the domain of the attribute **County\_name** in the state of Texas?
6. Distinguish among a simple attribute, a single-valued attribute, a composite attribute, a multi-valued attribute, and a complex attribute. Develop an example similar to Figure 2.3 that illustrates the differences among these attributes.
7. What is a unique identifier of an entity type? Is it possible for there to be more than one unique identifier for an entity type?
8. What is the difference between a key attribute and a non-key attribute?
9. Consider the EMPLOYEE entity type shown here.
  - a. List all key and non-key attributes.
  - b. What is (are) the unique identifier(s)?
  - c. Which attribute(s) is (are) derived attributes?
  - d. Using the following figure as a guide, develop sample data for four employees that illustrate the nature of the various mandatory and optional attributes in the EMPLOYEE entity type. Be sure to illustrate the various ways the **Name** attribute might appear.



10. Discuss how to distinguish between an entity type and an attribute.

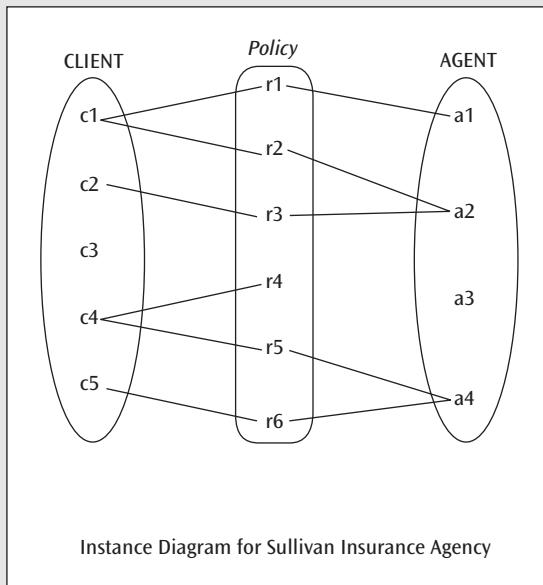
## Chapter 2

11. Give an example of three entity types and accompanying attributes that might be associated with a database for a car rental agency.
12. What is a relationship type? How does a relationship type differ from a relationship instance?
13. What is meant by the “degree” of a relationship?
14. What is the value of using role names to describe the participation of an entity type in a relationship type?
15. What is the difference between a binary relationship that exhibits a 1:1 cardinality constraint and a binary relationship that exhibits a 1:n cardinality constraint?
16. Describe how *Married\_to* can be modeled as a recursive relationship.
17. Create an example of a recursive relationship with an m:n cardinality constraint.
18. Distinguish between a participation constraint and minimum cardinality.
19. Why can total participation of an entity type in a relationship type also be referred to as existence dependency of that entity type in that relationship type?
20. How do cardinality constraints and participation constraints relate to the notions of total and partial participation?
21. Discuss the difference between existence dependency and identification dependency.
22. Give an example of a relationship type between two entity types where an attribute can be assigned to the relationship type instead of to one of the two entity types.
23. What is the difference between a base entity type and a weak entity type? When is a weak entity type used in data modeling?
24. Define the term “partial key.”
25. A small university is comprised of several colleges. Each college has a name, location, and size. A college offers many courses over four college terms or quarters—Fall, Winter, Spring, and Summer—during which one or more of these courses are offered. Course#, name, and credit hours describe a course. No two courses in any college have the same course#; likewise, no two courses have the same name. Terms are identified by year and quarter, and they contain numbers. Courses are offered during every term. The college also has several instructors. Instructors teach; that is why they are called instructors. Often, not all instructors are scheduled to teach during all terms, but every term has some instructors teaching. Also, the same course is never taught by more than one instructor in a specific term. Further, instructors are capable of teaching a variety of courses offered by the college. Instructors have a unique employee ID; their name, qualifications, and experience are also recorded.

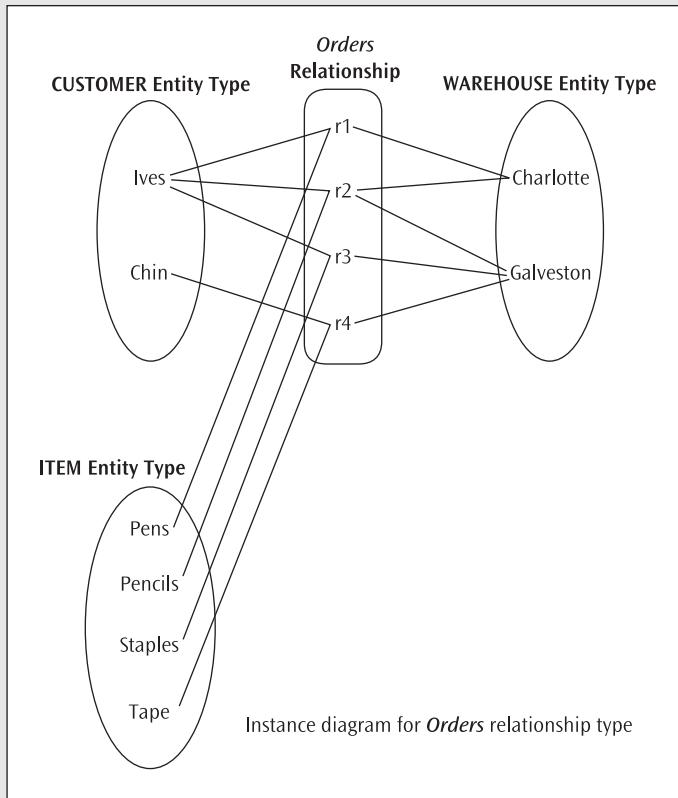
For this narrative, perform the following tasks:

- a. List the business rules explicitly stated and implicitly indicated in the narrative.
- b. Study the narrative carefully and identify the missing information required for developing a semantically complete conceptual data model.

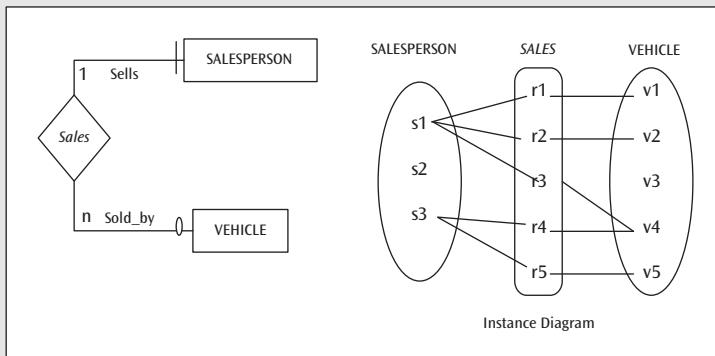
26. The instance diagram shown here illustrates the relationship between Sullivan Insurance Agency's agents and clients. Using this instance diagram, write the narrative that describes the relationship between agents and clients. Your narrative should include a description of both the cardinality ratio and participation constraints implied in the instance diagram. In addition, draw the ER diagram that fully describes the relationship between the company's agents and clients.

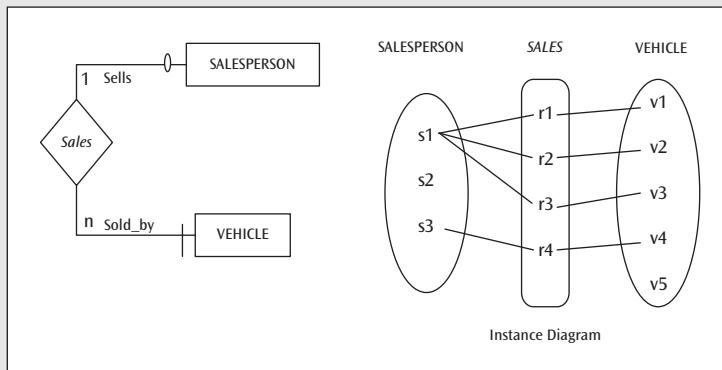


27. Revise the ER diagram you drew in the previous exercise to include the following mandatory attributes: CLIENT—ID number, name, address (city, state, zip), phone number(s), birthdate; AGENT—agent number, name, phone number, area; and commission received by an agent for selling a *Policy* to a client.
28. Using the instance diagram depicting the ternary relationship *Orders* shown on the next page, answer the following questions:
- Identify an error seeded in the diagram and correct the error.
  - Which customers order pens from the Galveston warehouse?
  - Which items are ordered by customers from both warehouses?
  - Which warehouse fills one or more orders of items from both customers?
  - Describe orders filled from both warehouses.
  - What changes must be made to the instance diagram for order r1 to involve both pencils and pens?

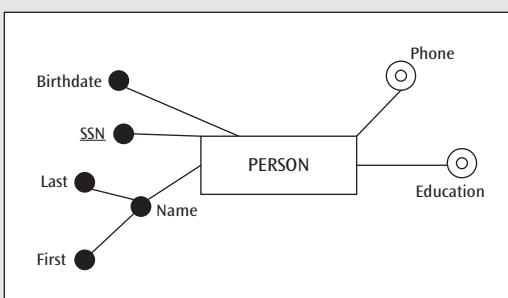
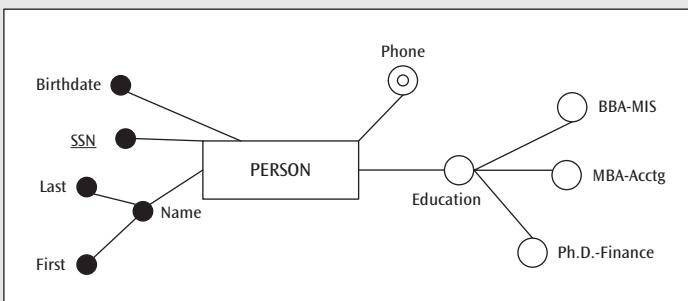


29. The following two ER diagrams contain both a cardinality ratio constraint and a participation constraint.
- In the first ER diagram, is the instance diagram on the right consistent with the ER diagram on the left? Why or why not?
  - In the second ER diagram, is the instance diagram on the right consistent with the ER diagram on the left? Why or why not?





30. Suppose you want to show that a person can have multiple degrees. Would each of the following two ER diagrams get the job done? Why or why not? What is the difference?

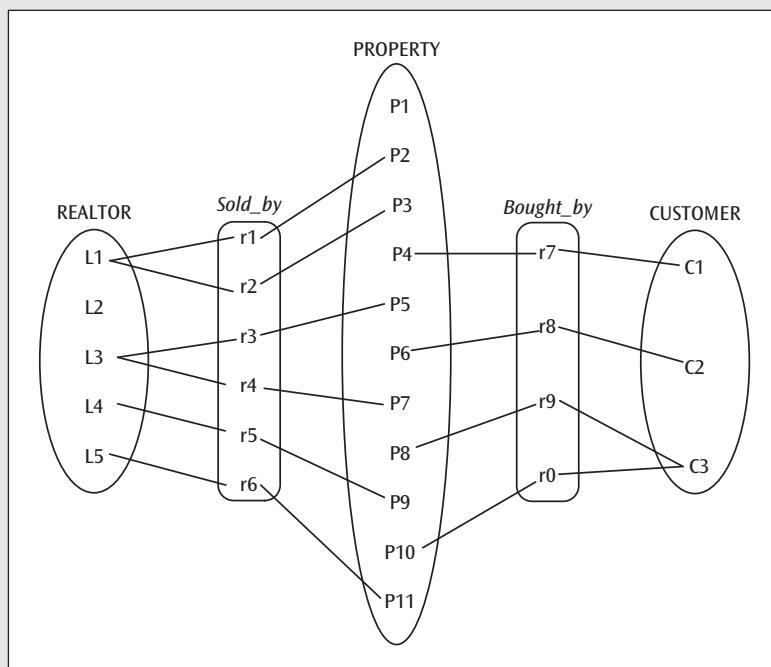
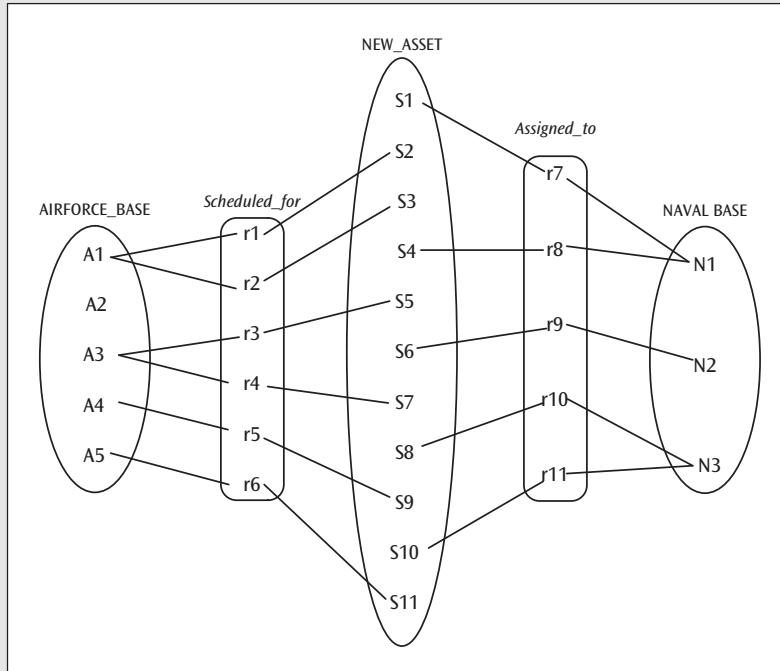


31. Adams, Ives, and Scott Incorporated is an agency that specializes in representing clients in the fields of sports and entertainment. Given the nature of the business, some employees are given a company car to drive, each company car being assigned to a particular employee. Each employee has a unique employee number, plus an address and set of certifications. Not all employees have earned one or more certifications. Company cars are identified by their respective vehicle IDs and also have a license plate number, make, model, and year. Employees represent clients. Not all employees represent clients, whereas some employees represent many clients. Each client is represented by one and only one employee. Sometimes, clients refer one another to use Adams, Ives, and Scott to represent them. A given client can refer one or more other clients. A client may or may not have been referred to Adams, Ives, and Scott by another client, but a client may be referred by only one other client. Each client is assigned a unique client number. Additional attributes recorded for each client are: name, address, and date of birth.

## Chapter 2

Draw an ER diagram that shows the entity types and relationship types for Adams, Ives, and Scott. You must name each relationship type and define its structural constraints; however, it is not necessary that you provide role names.

32. Draw the ER diagram for the two instance diagrams depicted here.



33. This vignette is a small excerpt from a comprehensive case about a clinic. Various physicians and surgeons working for a clinic are on an annual salary [o]. These doctors are identified by their respective employee numbers. The other descriptors of a doctor are: name, gender, address, and phone. Each physician's specialty and rank [o] are captured; each surgeon's, specialty and skill are also captured; a surgeon may have one or more skills.

Every physician serves as a primary care physician for at least seven patients; however, no more than 20 patients are allotted to a physician. Every patient is assigned one physician for primary care. Some patients need surgeries; others don't. Surgeons perform surgeries for the patients in the clinic. Some do a lot of surgeries; others do just a few. The date and operation theater [o] for each surgery needs to be recorded, too. *Removal of a surgeon from the clinic database is prohibited if that surgeon is scheduled to perform any surgery. However, if a patient chooses to pull out of the surgery schedule, all surgeries scheduled for that patient are cancelled.*

Data for patients include: patient number (the unique identifier of a patient), name, gender, date of birth, blood type, cholesterol (consisting of HDL, LDL, and triglyceride), blood sugar, and the code and name of allergies, if any.

Physicians may prescribe medications to patients; thus, it is necessary to capture which physician(s) prescribe(s) what medication(s) to which patient(s) along with dosage and frequency. *In addition, no two physicians can prescribe the same medication to the same patient. If a physician leaves the clinic, all prescriptions written by that physician should also be removed because this information is retained in the archives.*

A patient may be taking several medications, and a particular medication may be taken by several patients. Despite its list price, a medication's cost varies from patient to patient, perhaps because of the difference in insurance coverage. The cost of a medication for a patient needs to be captured. A medication may interact with several other medications. *When a medication is removed from the system, its interaction with other medications, if any, should be voided. When a patient leaves the clinic, all the medication records for that patient are removed from the system.*

Medications are identified by either their unique medication codes or by their unique names. Other attributes of a medication are its classification, list price, and manufacturer [o]. For every medication, either the medication code or the medication name must be present—not necessarily both.

Note: [o] indicates optionality of value for the attribute. Develop an ER model for this scenario.



# CHAPTER 3

## ENTITY-RELATIONSHIP MODELING

This chapter discusses the application of ER modeling grammar to conceptual data modeling. It expounds upon the “Method” component of the Wand and Weber framework for conceptual modeling research that was discussed at the beginning of Part I (see Figure I.1). The discussion is framed in the context of Bearcat Incorporated, a manufacturing company with several plants located in the northeastern part of the United States. This example database application is also used in subsequent chapters.

Section 3.1 provides the narrative data requirements for Bearcat Incorporated. Section 3.2 applies the ER modeling grammar developed in Chapter 2 to a full-scale conceptual modeling process of the Bearcat Incorporated requirements. The two basic layers of the ER model are introduced, progressing from an end-user communication tool (the Presentation Layer ER model) to a database design-oriented product (the Design-Specific ER model). The Design-Specific ER model is ready for direct mapping to the logical tier.

Research in conceptual modeling indicates that modeling errors are not uncommon, especially among novice data modelers. Section 3.3 presents a few common data modeling errors using a couple of vignettes. Detecting and correcting modeling errors is used in this section as a powerful technique to learn ER modeling.

### **3.1 BEARCAT INCORPORATED: A CASE STUDY**

The case narrated in this section may initially appear overwhelming. However, it is not unusual to find a multitude of data requirements or business rules like these in the real world of database design. The intent of this chapter is to present an array of requirements, then to show how the ER modeling grammar can be applied to systematically dissect the case content into manageable scenarios. We will see how ER diagrams are developed for each scenario, and how these scenarios are synthesized into a full-fledged ER model.

What follows are the user requirements or business rules for Bearcat Incorporated, which will serve as a starting point for our discussion.

Bearcat Incorporated is a manufacturing company with several plants in the northeastern United States. These plants are responsible for leading different projects that the company might undertake, depending on a plant’s function. A certain plant might even be associated with several projects, but a project is always under the control of just one plant. Some plants do not undertake any projects at all. If a plant is closed down, the

projects undertaken by that plant cannot be canceled. The project assignments from a closed plant must be temporarily removed in order to allow the project to be transferred to another plant.

Employees work in these plants, and each employee works in only one plant. A plant may employ many employees but must have at least 100 employees in order to exist. A plant with employees cannot be closed down. Every plant is managed by an employee who works in the same plant; but every employee is not a plant manager, nor can an employee manage more than one plant. Company policy dictates that every plant must have a manager. Therefore, an employee currently managing a plant cannot be deleted from the database. If a plant is closed down, the employee no longer manages the plant but becomes an employee of another plant.

Some employees are assigned to work on projects and in some cases might even be assigned to work on several projects simultaneously. For a project to exist, it must have at least one employee assigned to it. A project might need several employees, depending on its size and scope. As long as an employee is assigned to a project, his or her record cannot be removed from the database. However, once a project ends, the employee records are removed from the database and all assignments of employees to that project must be removed.

Some employees also supervise other employees, but all employees need not be supervised; the employees that are supervised are supervised by just one employee. An employee may be a supervisor of several employees but of no more than 20. The Human Resources Department uses a designated default employee number to replace a supervisor who leaves the company. It is not possible for an employee to be his or her own supervisor.

Some employees may have several dependents. Bearcat Incorporated does not allow both husband and wife to be an employee of the company. Also, a dependent can only be a dependent of one employee at any time.

Bearcat Incorporated offers credit union facilities as a service to its employees and to their dependents. An employee is not required to become a member of Bearcat Credit Union (BCU). However, most employees and some of their dependents have accounts with BCU. Some BCU accounts are individual accounts, and others are joint accounts held by an employee and his or her dependent(s). Every BCU account must belong to at least one employee or dependent. Each joint account must involve no more than one employee and no more than one of his or her dependents. If an employee leaves the company, all dependents and BCU accounts of that employee must be removed. In addition, as long as a dependent has a BCU account, deletion of the dependent is not permitted.

To nurture the hobbies of employees' dependents, Bearcat Incorporated sponsors recreational opportunities. Dependents need not have a hobby, but some dependents may have several hobbies. Because some hobbies are not as popular as others, every hobby need not have participants. If a dependent is no longer in the database, no records of that dependent's participation in hobbies should exist in the database. Finally, as long as at least one dependent participates in a hobby, that hobby should continue to exist.

All plants of Bearcat Incorporated have a plant name, number, budget, and building. A plant has three or more buildings. Each plant can be identified by either its name or number. Bearcat Incorporated operates seven plants, and the plant numbers can be any in the range 10 through 20. However, either the plant number or plant name must always be recorded.

The name of an employee of Bearcat Incorporated consists of the first name, middle initial, last name, and a nametag. Employee numbers are used to identify employees in the company. However, names can also be used as identifiers. Both employee number and employee name must be recorded. Where two or more employees have the same name, a one-position numeric nametag is used so that up to 10 otherwise duplicate names can be distinguished from one another. Sometimes, an employee's middle initial may not be available.

Although the address, gender (male or female), and hiring date of each employee must be recorded, salary information is optional. Salaries at Bearcat Incorporated range from \$35,000 to \$90,000. Also, the salary of an employee cannot exceed the salary of the employee's supervisor.

The date on which an employee starts working as a manager of a plant should be gathered. In addition, the number of employees working in each plant should be gathered; this can be computed. Information about the dependents related to each employee, such as the dependent's name, relationship to the employee, birth date, and gender, should also be captured. The dependent's name as well as how the dependent is related to the employee is mandatory. A mother or daughter must be a female, a father or son must be a male, and a spouse can be either male or female. Since a dependent cannot exist independently of an employee, the dependent's name and relationship to the employee, in conjunction with either the employee name or the employee number, is used to identify the dependents of an employee. The number of dependents of each employee must also be captured, but this (like the number of employees working in each plant) can be computed.

Projects have unique names and numbers, and their locations must be specified. Every project must have a project number but sometimes may not have a project name; project numbers range from 1 to 40. Bearcat Incorporated's projects are located in the cities of Stafford, Bellaire, Sugarland, Blue Ash, and Mason. The amount of time an employee has been assigned to a particular project should be recorded for accounting purposes.

BCU accounts are identified by a unique account ID, composed of an account number and an account type (C: checking account; S: savings account; I: investment account). For each account, the account balance is recorded. Only the account ID is required for every account. Account numbers contain a maximum of six alphanumeric characters.

Hobbies are identified by a unique hobby name and include a code that indicates whether the hobby is an indoor or outdoor activity and another code that indicates if the hobby is a group or individual activity. The time spent by a dependent per week on each hobby and the associated annual cost are also captured.

## 3.2 APPLYING THE ER MODELING GRAMMAR TO THE CONCEPTUAL MODELING PROCESS

---

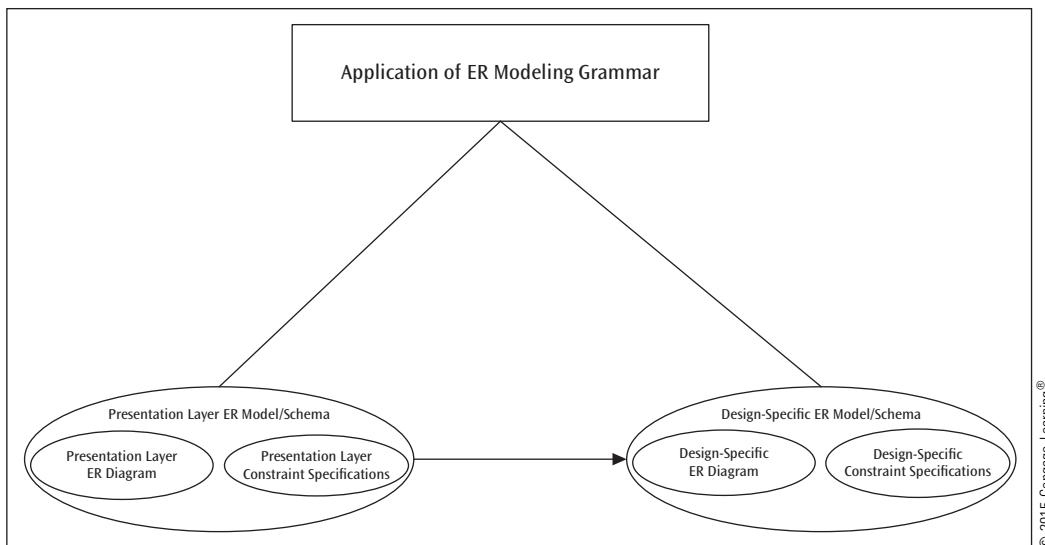
The ER modeling grammar for conceptual modeling serves two major purposes:

- As a communication/presentation device used by an analyst to interact with the end-user community (the **Presentation Layer ER model/schema**)

- As a design tool at the highest level of abstraction to convey a deeper-level understanding to the database designer (the **Design-Specific ER model/schema**)

An ER model includes (1) an **ER diagram (ERD)** portraying entity types, attributes for each entity type, and relationships among entity types, and (2) **semantic integrity constraints** that reflect the business rules about data not captured in the ERD.

Figure 3.1 illustrates how the Presentation Layer ER model and the Design-Specific ER model fit into the conceptual modeling activity. The development of the Presentation Layer ER model entails the analyst working with the user community and culminates in the generation of a script—that is, a Presentation Layer ERD, accompanied by a list of semantic integrity constraints. Next, the Presentation Layer ER model (including the ERD and semantic integrity constraints) is mapped or translated to the Design-Specific ER model. Section 3.2.1 describes the method for the development of the Presentation Layer ER model for Bearcat Incorporated, whereas Section 3.2.2 focuses on the methods for the development of the Design-Specific ER model.



© 2015 Cengage Learning®

**FIGURE 3.1** Conceptual modeling method using the ER modeling grammar

### 3.2.1 The Presentation Layer ER Model

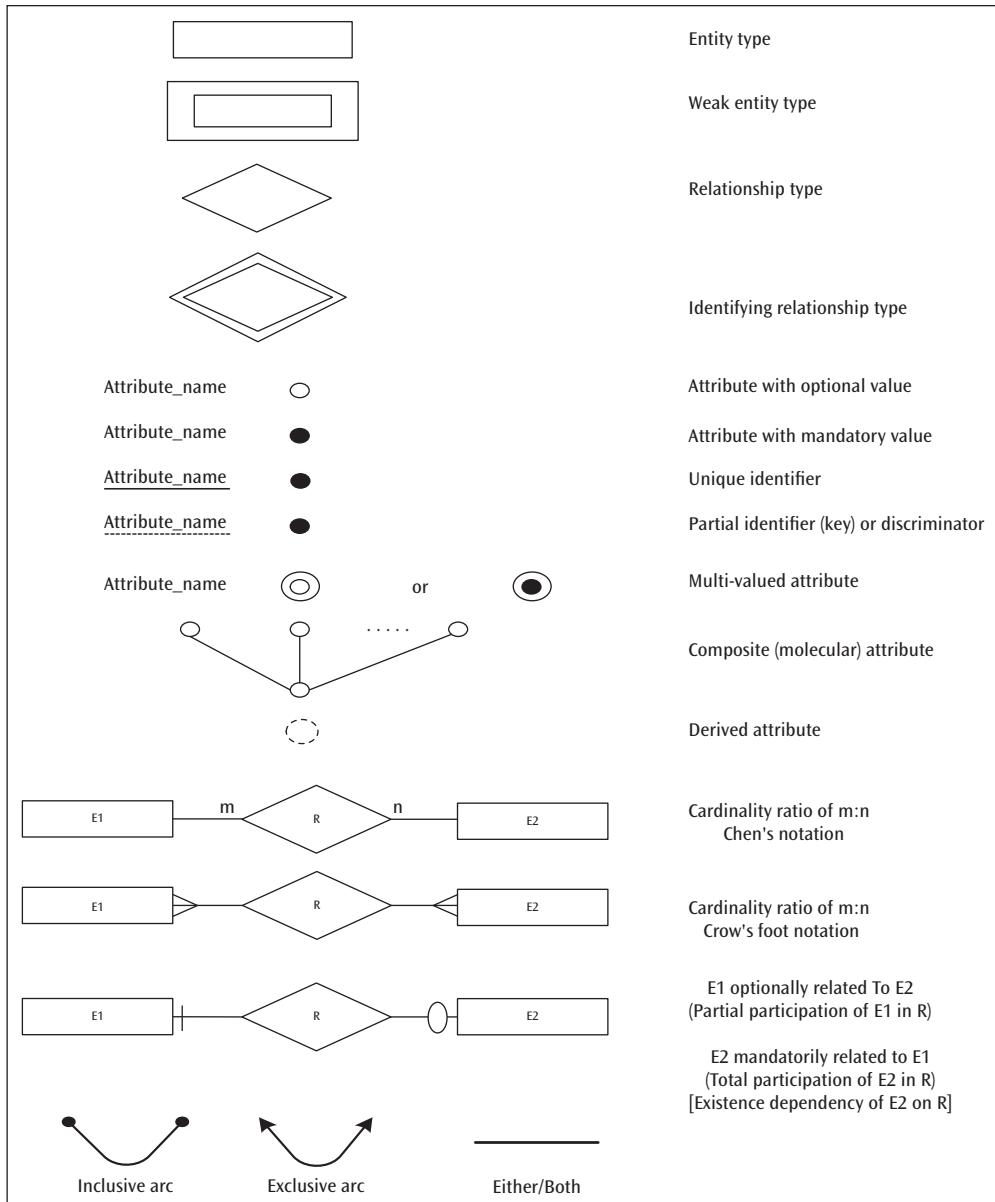
As the name implies, this layer of the ER model serves the principal purpose of communicating with the end-user community. The Presentation Layer ERD is a surface-level expression of the application domain, and the semantic integrity constraints are a reiteration of the business rules that are not captured in the Presentation Layer ERD.

While a variety of notational variations for entity-relationship diagrams appear in the conceptual data modeling literature, only a handful are relatively popular and are also used in commercial computer-aided software engineering (CASE) tools. Often, the

notational variations occur in the expression of the *properties* of a relationship type. In all these notations, an entity type is expressed by a rectangular box, whereas a weak entity type appears as a double rectangular box. In order to contrast it from a weak entity type, the entity type is often referred to as a base (or strong) entity type. A relationship type is shown as a diamond, whereas an identifying relationship type (a relationship type that connects a weak entity type to its identifying parent entity type) is shown as a double diamond. The CASE tools tend to avoid the use of the diamond for a relationship type, instead labeling the edges connecting the entity types to capture the semantics of the relationship. There are just a couple of different ways to express attributes graphically. This text uses the convention in which the optional/mandatory property can be explicitly expressed. Thus, an attribute is shown by a circle with the name of the attribute written adjacent to the circle. A dark circle represents an attribute with a mandatory value (also known as a mandatory attribute), whereas an empty circle indicates an attribute with an optional value (also known as an optional attribute). Component attributes constituting a composite attribute are attached to the circle that represents the composite attribute. A multi-valued attribute is shown by a double circle, whereas a derived attribute is shown by a dotted circle. An attribute that serves as a unique identifier of a base entity type is underlined with a solid line, whereas the partial key of a weak entity type (also known as a discriminator) is underlined with a dotted line. Note that an entity type can and often does have multiple unique identifiers and that each identifier can be an atomic or composite attribute.

Figure 3.2 summarizes the notational scheme used for the Presentation Layer ERDs in this book. Although it is based on Peter Chen's original notation (1976), it also incorporates a few desirable features of other commonly used notational schemes. A relationship is shown through the use of edges (lines) that connect the relationship type to the participating entity types. Both the cardinality ratio and the participation constraint are expressed via a "look across" approach. The oval adjacent to E2 in Figure 3.2 indicates that E1 is optionally related to E2—that is, there can be entities ( $e_{11}, e_{12}, \dots, e_{1x}, \dots, e_{1n}$ ) of E1 not related to any entity of E2. As stated in Section 2.3.4, this is known as partial participation of E1 in R. The bar (|) adjacent to E2 signifies that E2 is mandatorily related to E1.<sup>1</sup> As discussed in Section 2.3.4, this is known as total participation of E2 in R. This also implies that E2 has existence dependency on R—that is, in order for an entity  $e_{21}$  of E2 to exist, it must participate in a relationship  $r_1$  with an entity  $e_{1x}$  of E1. The Crow's Foot notation to specify the relationship properties is also popular and essentially replaces the **m** and **n** in the Chen scheme with fork-like symbols. The Crow's Foot notation, originally introduced by Everest (1986) for the KnowledgeWare software, follows the same "look across" strategy to specify both cardinality ratios and participation constraints, as is done in the Chen scheme. The meaning of the exclusive, inclusive arcs and the either/both (hash) notation is discussed later in this chapter.

<sup>1</sup>Mandatory participation in Chen's original notational scheme is implicitly indicated by the absence of the oval (symbol for partial participation). We, however, employ an explicit indicator—that is, the bar (|).

**FIGURE 3.2** Summary of Presentation Layer ER diagram notation

As a high-level diagrammatic portrayal of the application domain, an ERD is not capable of capturing some of the finer business rules that are part of the data requirements. The specification of constraints is the mechanism to record the business rules not captured in the ERD. Together, the ERD and the semantic integrity constraints must preserve all the information conveyed in the data requirements of the application. The ERD coupled with the semantic integrity constraints represents the conceptual schema for the

application, and because the ER modeling grammar in this case expresses the conceptual schema, the resulting script is referred to as the ER model/schema.

### 3.2.2 The Presentation Layer ER Model for Bearcat Incorporated

The development of an ERD often begins by examining the data requirements specification for the application domain in an effort to extract possible entity types, attributes, and relationships among entities. Conceptual modeling is a heuristic process, as opposed to a scientific process, and is therefore intuitive and iterative in nature. For example, at first glance, it may appear that COMPANY should be an entity type in the ERD. However, a careful reading of the data requirements in Section 3.1 indicates that if we specify COMPANY as an entity type, it will have exactly one entity, Bearcat Incorporated. In other words, the scope of the application domain here is just one company, not a number of companies. Thus, an entity type called COMPANY is not necessary. In fact, notice that, in the narrative, no attributes have been specified for Company; therefore, an entity type COMPANY simply cannot exist unless we make up some attributes. Recall that attributes give shape to an entity type and therefore an entity type cannot exist without at least one attribute.

Alternatively, one could be futuristic and speculate on the possibility of Bearcat Incorporated acquiring other companies or simply deciding to organize itself into several autonomous business units (divisions). Then, the COMPANY entity type in the data model would add value. The sole purpose of single-instance entity types is to provide an avenue for future expansion of the database. However, a designer could speculate numerous scenarios, which could lead to inefficient design based on unsubstantiated expansion of the scope of the desired database application. Any expansion of the design's scope, based on anticipated future needs, should be done in consultation with the user community. For instance, in order to choose this approach, the designer should elicit a list of attributes from the user for a COMPANY entity type.

One way to arrive at the first cut of an ERD is to begin by listing all discernible data elements from the narrative, treating them all as attributes, and then attempting to group these attributes based on apparent commonalities among them. This should lead to the identification of different clusters of attributes, each of which may now be designated as an entity type and labeled with an appropriate name culled from the narrative. A review of the leftover data elements in the list should facilitate recognition of some of them as possible links among the entity types previously identified. Thus, relationship types can be generated. Note once again that an entity type cannot exist without having one or more attributes. In addition, while a stand-alone entity type is not technically prohibited in an ERD, an entity type, in general, does not exist in an ERD without being related to at least one other entity type. The short list of remaining data elements, if there are any, can be reconciled as either nonexistent in the data requirements or as part of an entity type or relationship type created thus far or as other new entity types. The first cut of the ERD at this point is ready for recursive refinement. This approach is usually labeled the **synthesis approach**.

Alternatively, one could consider an **analysis approach** for discovering entity types from the narrative. This approach begins with a search for things that can be labeled by singular nouns (a person, place, thing, or concept), which are modeled as entity types of

the ERD. This is followed by gathering properties that appear to belong to individual entity types. These properties, also typically nouns, are labeled as attributes of that entity type. As was the case in the synthesis approach, throughout this process the identification of relationships among various entity types must also be recognized.

Under both the synthesis and the analysis approach, caution should be exercised to ensure that elements outside the scope of the narrative are not brought into the modeling process based on an individual's whims.

The excerpts that follow, taken from the Bearcat Incorporated narrative presented in Section 3.1, illustrate the application of the analysis approach to identifying possible entity types and their attributes. Capitalized nouns constitute the entity types, whereas nouns in *bold monofont* are the attributes of the respective entity types.

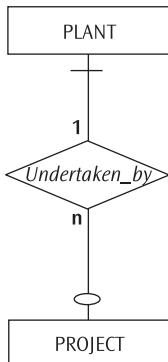
- Bearcat Incorporated is a manufacturing company with several PLANTS.... All plants of Bearcat Incorporated have a **Plant name**, **Number**, **Budget**, and **Building**....
- These plants are responsible for leading different PROJECTs.... Projects have ... **Names** and **Numbers**, and their **Location** must be specified....
- EMPLOYEEs work in these plants.... The **Name** of an employee ... consists of the **First name**, **Middle initial**, **Last name**, and a **Nametag**. While the **Address**, **Gender**, ... and **Date hired** ... must be recorded, **Salary** information is optional.... The **Start date** of an employee as a manager of a plant should also be gathered. There is also the requirement that the **Number of employees** working in each plant be available.... Company policy dictates that every plant must have a **MANAGER**....
- Some employees may have several DEPENDENTs.... Information about the dependents related to each employee, such as the dependent's **Name**, **Relationship to the employee**, **Birth date**, and **Gender**, should also be captured.... There is also the requirement that the **Number of dependents** of each employee be captured....
- Bearcat Incorporated offers CREDIT UNION facilities as a service to its employees and to their dependents.... BCU accounts are identified by a unique **Account ID** composed of an **Account number** and an **Account type** (C: checking account; S: savings account; I: investment account). For each account, the **Account balance** is recorded.
- To nurture the HOBBY(ies) of employees' dependents, Bearcat Incorporated sponsors recreational opportunities. Hobbies are identified by a unique **Hobby name** and include one code that indicates whether the hobby is an indoor or outdoor activity and another code that indicates if the hobby is a group or individual activity. The **Time spent** by a dependent per week on each hobby and the associated **Annual cost** are also captured.

### 3.2.2.1 Development of the ER Diagram

A series of eight boxes is used to illustrate how the relationships among the entity types for Bearcat Incorporated can be detected. Some of these boxes contain an excerpt from the Bearcat Incorporated data requirements described in Section 3.1 plus the ERD for the italicized sentences that appear in the excerpt. The use of information in sentences not

italicized in a particular box will be described later in this chapter. Note that in order to enhance the clarity of the presentation, attributes are not shown in these ERDs.

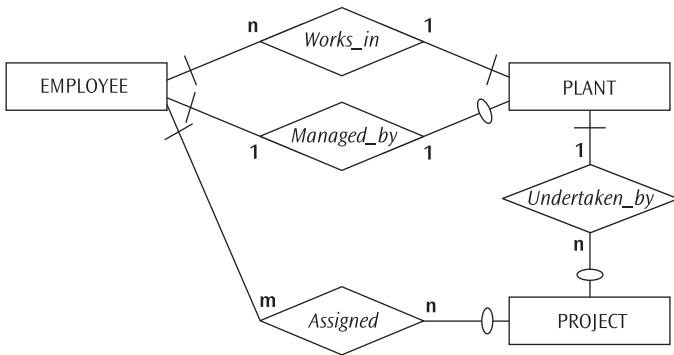
Bearcat Incorporated is a manufacturing company that has several plants in the northeastern part of the United States. *These plants are responsible for leading different projects that a company might undertake, depending on a plant's function. A certain plant might even be associated with several projects but a project is always under the control of just one plant. Some plants do not undertake any projects at all.* If a plant is closed down, the projects undertaken by that plant cannot be canceled. The project assignments from a closed plant must be temporarily removed in order to allow the project to be transferred to another plant.



#### BOX 1

In Box 1, observe that the cardinality ratio of the relationship type (*Undertaken\_by*) is shown as 1:n by looking across, from PLANT to PROJECT. This is because a plant can be associated with several projects but a project is always under the control of a single plant. Also, since the data requirements explicitly specify that “*Some plants do not undertake any projects at all,*” one can infer that a plant may or may not undertake/control a project (i.e., a plant optionally controls a project). This is indicated in the diagram in Box 1 by looking across, from the PLANT entity type to the PROJECT entity type, and placing the oval optionality marker just above the PROJECT entity type. Accordingly, the participation constraint of PLANT in this relationship is said to have the value “partial.” Likewise, since every project must be controlled by a plant, a look across, from the PROJECT entity type to the PLANT entity type, signifies the mandatory participation of the PROJECT in the relationship through the bar (|) just below the PLANT.

*Employees work in these plants and each employee works in only one plant. A plant could employ many employees but must have at least 100 employees. A plant with employees cannot be closed down. Every plant is managed by an employee who works in the same plant; but every employee is not a plant manager nor can an employee manage more than one plant. Company policy dictates that every plant must have a manager. Therefore, an employee currently managing a plant cannot be deleted from the database. If a plant is closed down, the employee no longer manages the plant but becomes an employee of another plant. Some employees are assigned to work on projects and in some cases might even be assigned to work on several projects simultaneously. For a project to exist, it must have at least one employee assigned to it. Projects might need several employees depending on their size and scope. As long as an employee is assigned to a project, his or her record cannot be removed from the database. However, once a project ends it is removed from the system and all assignments of employees to that project must be removed.*

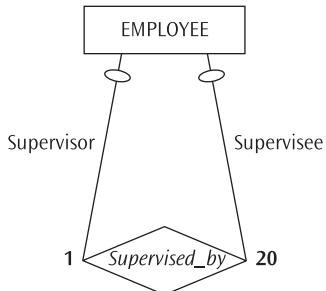


## BOX 2

In Box 2, the *Works\_in* relationship type between the PLANT entity type and the EMPLOYEE entity type indicates that every employee must work in a plant and only in one plant (look across, from EMPLOYEE to PLANT) and that a plant contains many employees. However, note that the requirement that a plant must have at least 100 employees in order to exist is not reflected in the ERD. Instead, the ERD in Box 2 only indicates that a plant must have at least one employee and may have more than one employee. A second relationship type, *Managed\_by*, also exists between the PLANT and EMPLOYEE entity types in order to show that (a) each plant must be managed by one and no more than one employee, and (b) an employee may manage one plant but all

employees are not managers. Observe that, in Section 3.2.2, MANAGER was identified as a possible base entity type. The basis for the design decision portrayed in Box 2 is discussed at the end of this section. The relationship type (*Assigned*) between the EMPLOYEE and PROJECT entity types exhibits an m:n cardinality ratio. The oval next to PROJECT indicates that every employee need not be assigned to a project (looking across, from EMPLOYEE), and the bar next to EMPLOYEE indicates that every project has at least one employee assigned to it (looking across, from PROJECT).

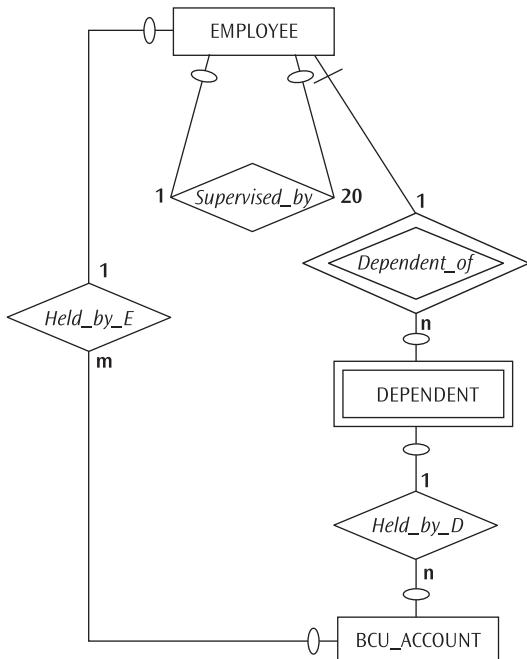
*Some employees also supervise other employees but all employees need not be supervised—the employees that are supervised are supervised by just one employee. An employee may be a supervisor of several employees, but no more than 20. The Human Resources Department uses a designated default employee number to replace a supervisor who leaves the company. It is not possible for an employee to be his or her own supervisor.*



### BOX 3

The *Supervised\_by* relationship type in Box 3 is a recursive relationship, indicating that a given employee *may* be supervised but that there can be only one supervisor per employee. At the same time, an employee may supervise many other employees—no more than 20, though. Finally, an employee need not be supervised; and an employee need not be a supervisor either.

Some employees may have several dependents. Bearcat Incorporated does not allow both husband and wife to be an employee of the company. Also, a dependent can only be a dependent of one employee at any time. Bearcat Incorporated offers credit union facilities as a service to its employees and to their dependents. An employee is not required to become a member of Bearcat Credit Union (BCU). However, most employees and some of their dependents have accounts in BCU. Some BCU accounts are individual accounts and others are joint accounts between an employee and his or her dependent(s). Every BCU account must belong to at least an employee or a dependent. Each joint account must involve no more than one employee and no more than one of his or her dependents. If an employee leaves the company, all dependents and BCU accounts of the employee must be removed. In addition, as long as a dependent has a BCU account, deletion of the dependent is not permitted.

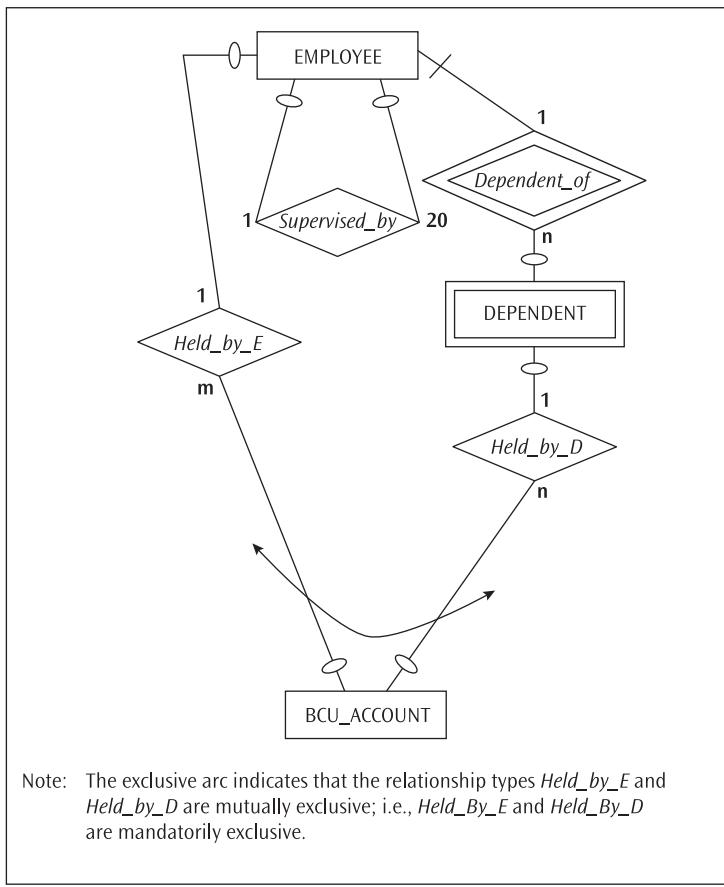


BOX 4

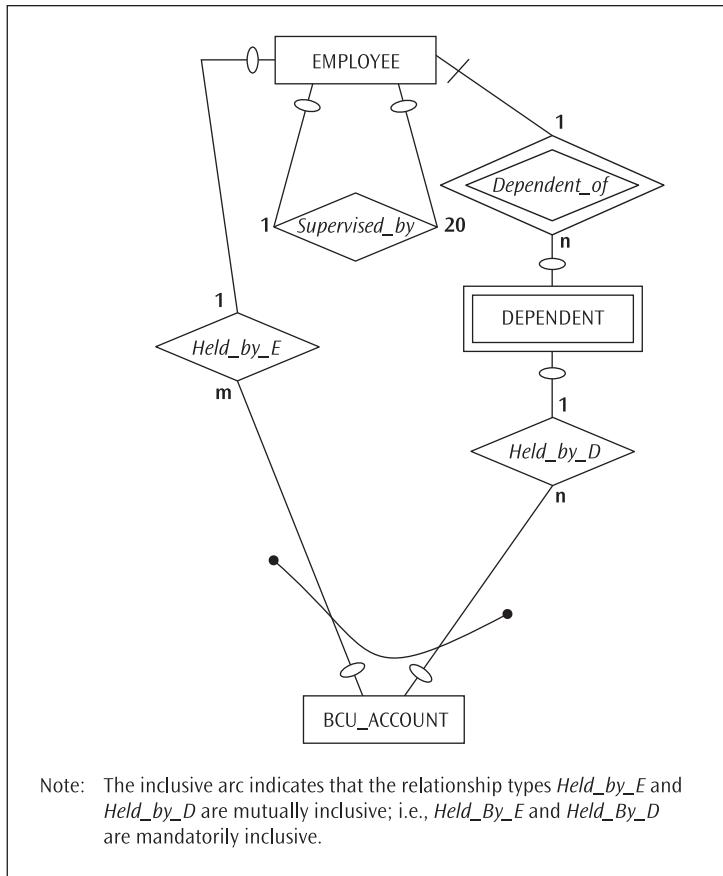
The DEPENDENT entity type in Box 4 is shown as a weak entity type, and the *Dependent\_of* relationship type is shown as an identifying relationship type. How do we know that DEPENDENT is a weak entity type? The fact is, we do not know from what is stated in Box 4. The statement “*a dependent can only be a dependent of one employee at any time*” only indicates total participation of DEPENDENT in the relationship. Any entity type, base or weak, has existence dependency in a relationship type if its participation in the relationship is total. The reason we know that DEPENDENT is a weak entity type participating as a child in an identifying relationship type with EMPLOYEE is that, later in the narrative, the following is stated: “Since a dependent cannot exist independently of an

employee, the dependent's name and relationship to the employee, in conjunction with either the employee name or the employee number, is used to identify the dependents of an employee." Observe that while not all employees have dependents, each dependent is related to one and only one employee.

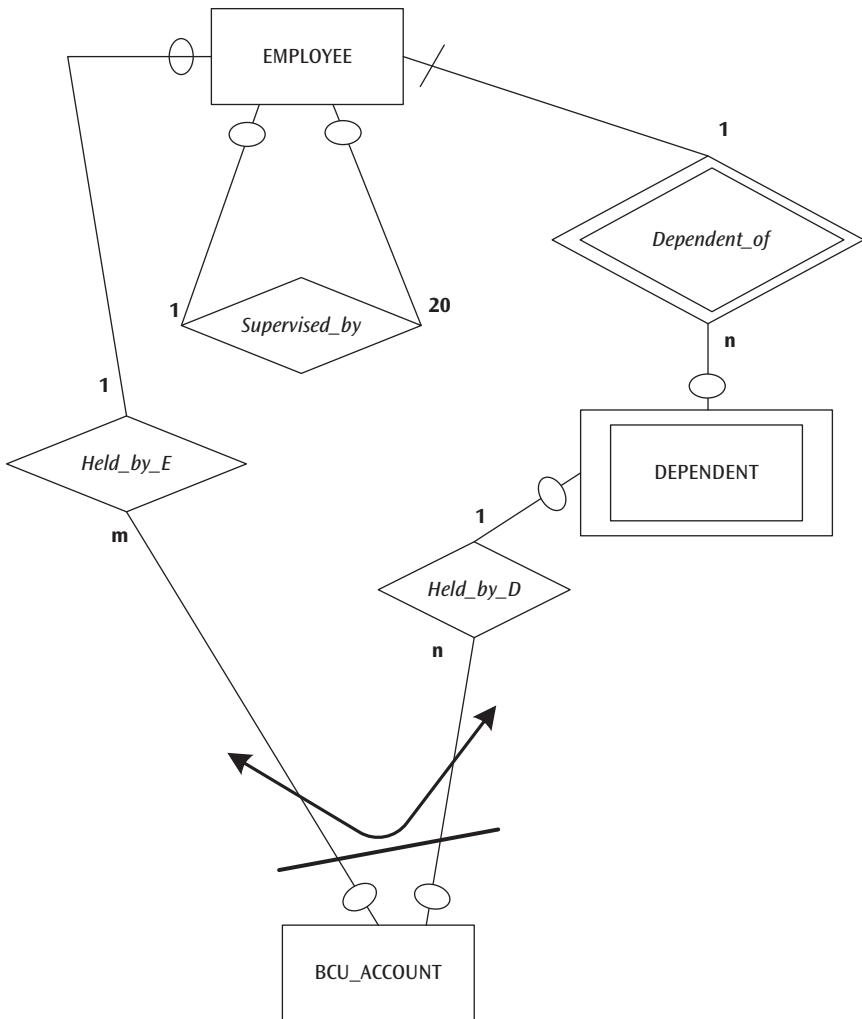
The BCU\_ACCOUNT entity type participates in two relationships: *Held\_by\_E* and *Held\_by\_D*. The *Held\_by\_E* relationship type reflects the fact that an employee may have a BCU account but that not all BCU accounts are held by employees. In addition, no more than one employee can be associated with a BCU account. The *Held\_by\_D* relationship type reflects a similar relationship between the BCU\_ACCOUNT entity type and the DEPENDENT entity type. Here, while a dependent may have several BCU accounts, he or she need not have a BCU account. Likewise, a BCU account need not be associated with a dependent, but if it is, then there can be no more than one dependent per BCU account. The business rules that every BCU account must belong to at least one employee or a dependent and, if held jointly, must belong to an employee and his or her dependent—not an employee or the dependent of *any* other employee—are not reflected in the ERD in Box 4. Business rules that cannot be captured in the ERD must be included in the list of semantic integrity constraints that accompanies the ERD as part of the ER model.



#### BOX 5

**BOX 6**

What if joint accounts between employees and dependents are not permitted? This means that the same BCU account entity cannot be related to an employee entity as well as a dependent entity. That is, the relationship types *Held\_by\_E* and *Held\_by\_D* are mutually exclusive. An **exclusive arc**, as shown in Box 5, represents this constraint. Likewise, if, for any reason, the user requirement specifies that all BCU accounts must be jointly held between an employee and a dependent, then an **inclusive arc** is used to express such a business rule as a constraint in the ERD, as shown in Box 6.



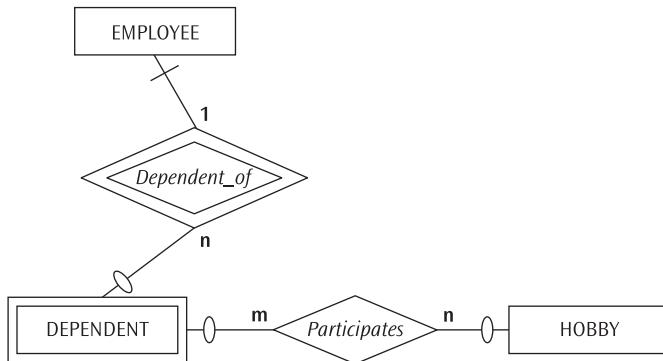
Note: The exclusive arc indicates that the relationship types *Held\_by\_E* and *Held\_by\_D* are mutually exclusive; the either/both hash indicates that every entity in the entity set of **BCU\_ACCOUNT** must participate in one or the other of *Held\_by\_E* and *Held\_by\_D*.

#### BOX 7

Observe that the business rule “*Every BCU account must belong to at least one employee or dependent*” cannot be captured in the ERD using either the inclusive arc or the exclusive arc. Use of an inclusive arc here would indicate that every BCU account *must* always belong jointly to an employee and a dependent. An exclusive arc, on the other hand, would mean that a BCU account *must never* belong jointly to an employee and a dependent. The business rule, however, indicates that every BCU account *must*

belong to *either* an employee or a dependent, or sometimes an employee and a dependent together. The either/both (hash) notation in Box 7 is designed to capture this semantic. Accordingly, some BCU accounts are held independently by employees and dependents, whereas others are held jointly between employees and dependents. However, a BCU account cannot exist without participating in at least one of the two relationships: *Held\_by\_E* and *Held\_by\_D*.

*Bearcat Incorporated sponsors recreational opportunities for the dependents of employees in order to nurture the hobbies of the dependents. Dependents need not have a hobby, but it is possible that some dependents may have several hobbies. Because some hobbies are not as popular as others, every hobby need not have participants. If a dependent is no longer in the database, all records of the participation of that dependent in hobbies should not exist in the database either. Finally, as long as at least one dependent participates in a hobby, that hobby should continue to exist.*

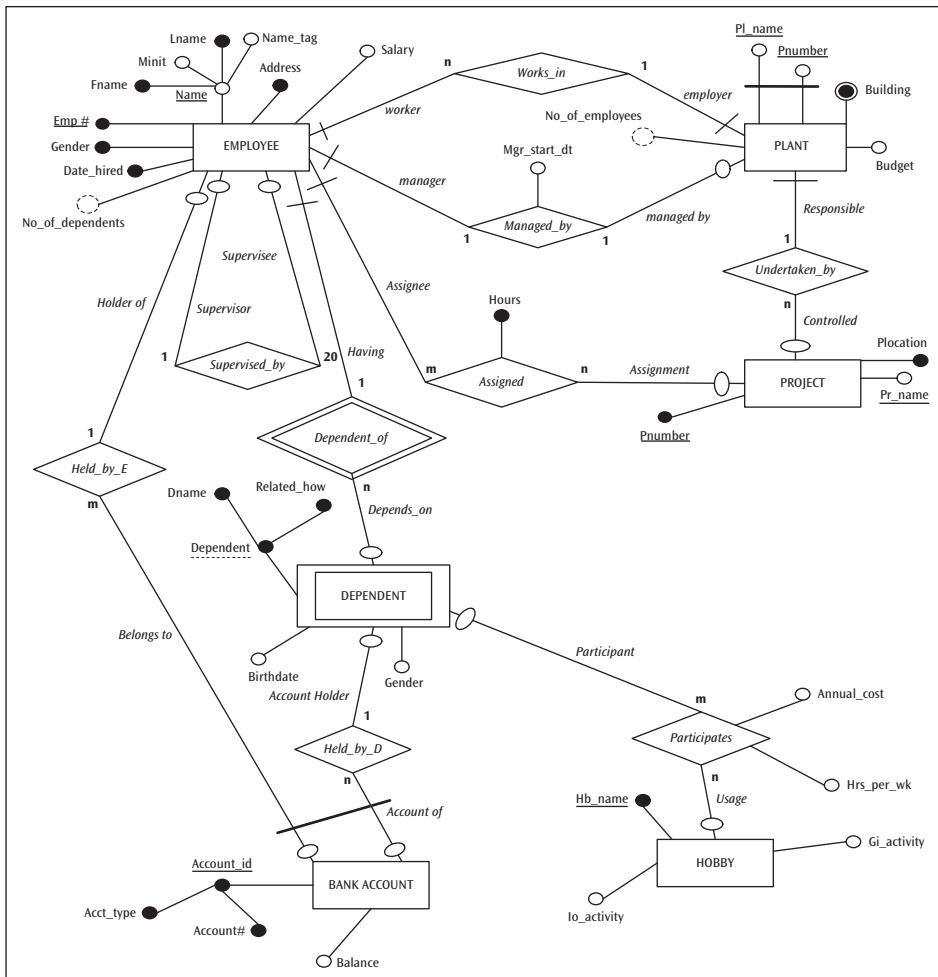


#### BOX 8

The relationship type (*Participates*) between the DEPENDENT entity type and the HOBBY entity type shown in Box 8 reflects an m:n cardinality ratio. Observe that a dependent may optionally have many hobbies (up to n) and that up to m dependents may participate in a hobby. Also, a hobby need not have any participants. Observe that while DEPENDENT is a weak entity type, its relationship with HOBBY is not an identifying relationship type.

A review of the data requirements for Bearcat Incorporated in Section 3.1 also makes it possible to identify the attributes of each of the entity types. Figure 3.3<sup>2</sup> is a collective representation of Boxes 1, 2, 3, 7, and 8 and includes the attributes. Although the attribute names in the ERD are arbitrary, as much as possible, meaningful abbreviated attribute names have been used to identify the attributes. The following comments elaborate on selected attributes in the ERD.

<sup>2</sup>All ER diagrams that appear in this book were created using Microsoft Visio.



**FIGURE 3.3** Presentation Layer ER diagram for Bearcat Incorporated—Stage I

- **Mandatory attributes**—The data requirements in Section 3.1 indicate that a number of attributes are mandatory—that is, they cannot have missing values. These attributes are shown as dark circles. Attributes shown as empty circles are *optional attributes* (can have missing values).
- **Multi-valued attributes**—Since a plant must have three or more buildings, in addition to a plant name, number, and budget, **Building** is shown as a mandatory multi-valued attribute.
- **Derived attributes**—A derived attribute is shown as a dotted circle. Since the data requirements indicate that the number of employees working in each plant and the number of dependents of each employee can be calculated, the attributes **No\_of\_employees** and **No\_of\_dependents** appear as derived attributes. Recall that derived attributes are not stored in the database.

- *Composite attributes*—The definitions of both the employee name and the account ID in the data requirements indicate that both can be subdivided into smaller subparts composed of atomic attributes. In the ERD, the composite attribute **Name** of the EMPLOYEE entity type is shown as consisting of the atomic attributes **Fname**, **Minit**, **Lname**, and **Name\_tag** since “*...the name of an employee of Bearcat Incorporated consists of the first name, middle initial, last name and a name tag.*” In addition, “*... BCU accounts are identified by a unique account ID, composed of an account number and an account type...*” which gives rise to the definition of **Account\_id** as a composite attribute consisting of the atomic attributes **Account#** and **Acct\_type**.
- *Attributes of relationship types*—The data requirements suggest that certain relationship types can also have attributes. For example, the optional attribute **Mgr\_start\_dt** of the *Managed\_by* relationship type is used to record the date on which an employee begins managing a plant. Since **Mgr\_start\_dt** is associated with the 1:1 relationship type *Managed\_by*, this attribute could have been assigned to either the EMPLOYEE or PLANT entity type. A second attribute of a relationship type is the mandatory attribute **Hours** of the *Assigned* relationship type reflecting the number of hours an employee works on a project. Note that the placement of this attribute in either the EMPLOYEE or PROJECT entity types will not convey the intended meaning. The same notion applies to the two optional attributes, **Annual\_cost** and **Hrs\_per\_wk**, of the *Participates* relationship type, which indicate the annual cost and hours per week that a dependent spends on a particular hobby.
- *Unique identifiers*—Embedded within the data requirements is the definition of certain attributes as unique identifiers. The names of these attributes (for example, **Emp#** of EMPLOYEE, **Pnumber** and **Pl\_name** of PLANT<sup>3</sup>, and **Account\_id** of BCU\_ACCOUNT) appear underlined in the ERD.
- *Partial keys*—Since the combination of the dependent name and the dependent’s relationship to the employee is unique for a given employee, the composite attribute **Dependent**, a composite attribute, consisting of the atomic attributes **Dname** and **Related\_how**, appears as a partial key and is underlined with a dotted line in the ERD.<sup>4</sup>

<sup>3</sup>Note that **Pnumber** and **Pl\_name** are shown as optional attributes in Figure 3.3. Both are also designated as unique identifiers (underlined). This may initially appear contradictory because we expect unique identifiers to be mandatory attributes. However, at the conceptual level, the task is to simply identify all possible unique identifiers of an entity type. The implication here is that in any entity of this entity type (PLANT) it is enough if either **Pnumber** or **Pl\_name** has a value. This requirement is captured by the either/both (hash) notation. Later, at the logical level, only one of these unique identifiers (**Pnumber** or **Pl\_name**) will be chosen to serve as the primary unique identifier. At that point, the primary unique identifier is made mandatory.

<sup>4</sup>It is conceivable that a DEPENDENT entity type can have a naturally occurring unique identifier (e.g., Social Security number), in which case it should be modeled as a base entity type. Since the requirements specification of Bearcat Incorporated does not include such an attribute, DEPENDENT is modeled here as a weak entity type identification-dependent on the identifying parent EMPLOYEE.

### 3.2.2.2 Mapping Deletion Rules to the ER Diagram

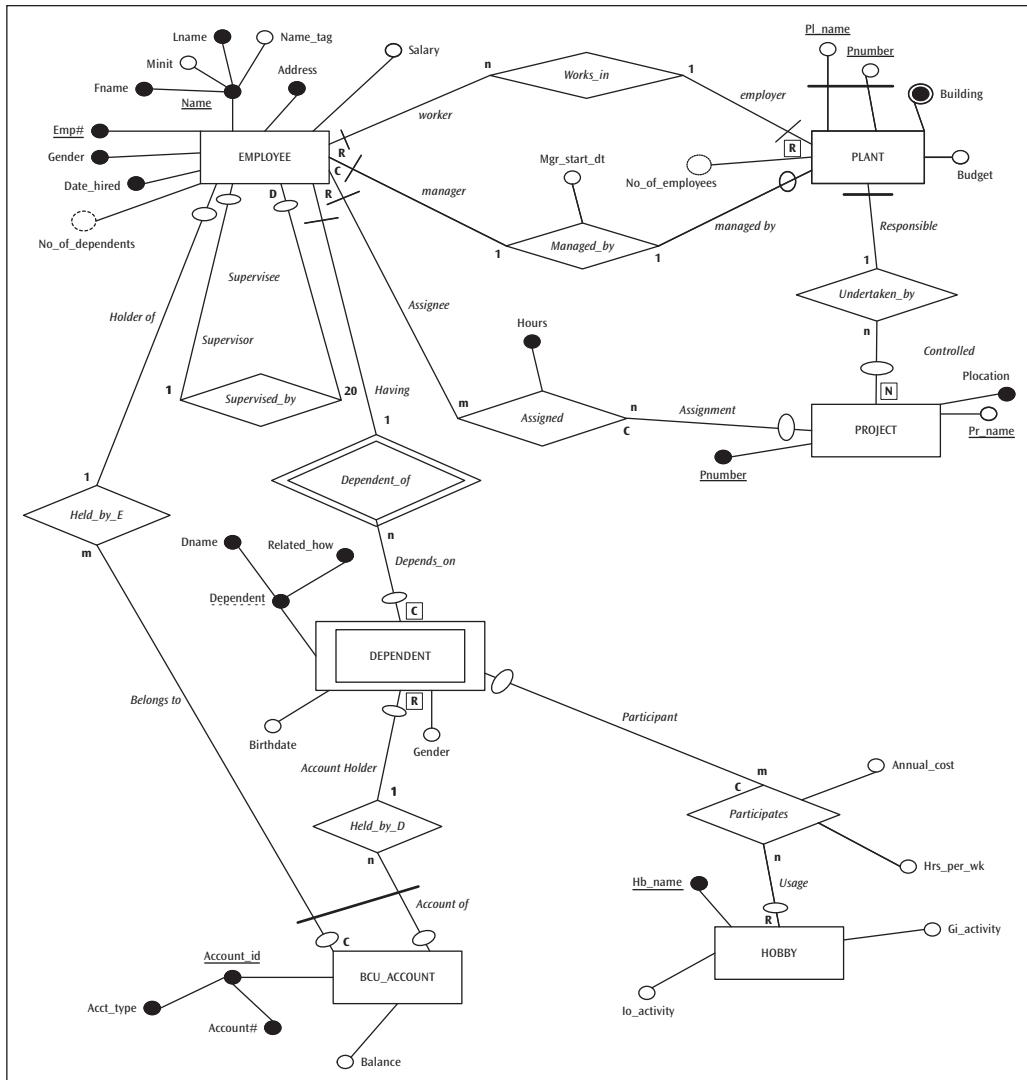
In the Bearcat Incorporated case, a variety of other business rules were introduced in the data requirements to maintain consistency among associated objects. This section discusses a subset of these business rules pertaining to entity deletions. In Chapter 2, we discussed how to specify business rules pertaining to entity deletions (deletion constraints). The next step in the development of the Presentation Layer ERD is to capture the deletion constraints. The business rules pertaining to entity deletions that were culled from the requirements specification stated in the case narrative are listed here:

1. A plant with employees cannot be closed down.
2. If an employee leaves the company, all BCU accounts of the employee must be removed.
3. If a plant is closed down, the projects undertaken by that plant cannot be canceled. The project assignments from a closed plant must be temporarily removed in order to allow the project to be transferred to another plant.
4. The Human Resources Department uses a designated default employee number to replace a supervisor who leaves the company.
5. An employee currently managing a plant cannot be deleted from the database.
6. If a plant is closed down, the employee no longer manages the plant but becomes an employee of another plant.
7. If an employee leaves the company, all dependents and BCU accounts of that employee must be removed.
8. As long as a dependent has a BCU account, deletion of the dependent is prohibited.
9. As long as an employee is assigned to a project, his or her record cannot be removed from the database.
10. If a project is deleted, all assignments of employees to that project must be deleted.
11. If a dependent is deleted, all records of the participation of that dependent in hobbies must be deleted.
12. A hobby with at least one dependent participating in it cannot be deleted.

Figure 3.4 depicts the revised Presentation Layer ERD, in which the deletion constraints have been incorporated. The following observations pertaining to the deletion constraints in Figure 3.4 require the readers' attention:

- The deletion constraint R (Restrict) is always imposed on the parent entity type in a parent-child relationship (PCR).
- Likewise, the deletion constraints C (Cascade), N (set null), and D (Set default) always apply to only the child entity type in the PCR.
- Because there is exactly one child entity type and one parent entity type in a binary relationship with a cardinality constraint of 1:n, there can be only one deletion rule, which applies either on the parent or the child.
- In a binary relationship informed by a 1:1 cardinality constraint, because either one of the participating entity types qualifies as a parent or child, two deletion constraints have been specified (see the *managed\_by* relationship between the entity types EMPLOYEE and PLANT).

## Chapter 3

**FIGURE 3.4** Presentation Layer ER diagram for Bearcat Incorporated—Stage 2

- In a m:n binary relationship, because both participating entity types are parents, the relationship type serves the role of the child (artificial entity type, also known as “associative entity type,” as discussed in Chapter 2), with two parents. Accordingly, once again, the relationship type is informed by two deletion constraints (see the *assigned* relationship between the entity types EMPLOYEE and PROJECT).

Each of the 12 deletion constraints listed here is followed by a description (in italics) of how the relevant deletion rule is indicated in Figure 3.4.

1. A plant with employees cannot be closed down.

*The R adjacent to the PLANT entity type indicates the restriction of the deletion of a plant if the plant has one or more employees. However, the deletion of a plant without employees is permitted.*

2. If an employee leaves the company, all BCU accounts of the employee must be removed.

*The C immediately above the BCU\_ACCOUNT entity type under the relationship Held\_by\_E indicates that the deletion of an employee should cascade through in order to delete all BCU accounts associated with this employee.*

3. If a plant is closed down, the projects undertaken by that plant cannot be canceled. The project assignments from a closed plant must be temporarily removed in order to allow the project to be transferred to another plant.

*The N (representing the “set null” rule) immediately above the PROJECT entity type indicates that a project’s relationship with a plant can be temporarily removed, resulting in the project not being undertaken by (i.e., under the control of) any plant.*

4. The Human Resources Department uses a designated default employee number to replace a supervisor who leaves the company.

*The D (representing the “set default” rule) shown immediately below the EMPLOYEE entity type indicates that if a supervisor is deleted, all supervisees under that supervisor are reassigned to the designated default supervisor.*

5. An employee currently managing a plant cannot be deleted from the database.

*The R adjacent to the EMPLOYEE entity type indicates the restriction of a deletion of an employee if the employee manages a plant since company policy dictates that every plant must have a manager.*

6. If a plant is closed down, the employee no longer manages the plant but becomes an employee of another plant.

*The N adjacent to the EMPLOYEE entity type indicates that an employee’s relationship with a plant as a manager can be removed, resulting in the employee no longer playing the role of a manager.*

7. If an employee leaves the company, all dependents and BCU accounts of that employee must be removed.

*The C immediately above the DEPENDENT entity type indicates that the deletion of an employee should cascade through in order to delete all dependents associated with this employee. Similarly, the C immediately above the BCU ACCOUNT entity type indicates that the deletion of an employee should cascade through in order to delete all BCU accounts associated with this employee.*

8. As long as a dependent has a BCU account, deletion of the dependent is not permitted.

*The R immediately below the DEPENDENT entity type indicates the restriction of the deletion of a dependent if the dependent has one or more BCU accounts. However, the deletion of a dependent without a BCU account is permitted.*

9. As long as an employee is assigned to a project, his or her record cannot be removed from the database.

*The R adjacent to the EMPLOYEE entity type indicates the restriction of a deletion of an employee who is assigned to one or more projects. Once the assignments of the employee have been deleted, then the employee can be deleted.*

10. If a project is deleted, all assignments of employees to that project must be deleted.

*The C adjacent to the Assigned relationship type indicates that the deletion of a project should cascade through in order to delete all related assignments of employees to that project.*

11. If a dependent is deleted, all records of the participation of that dependent in hobbies must be deleted.

*The C above the Participates relationship type indicates that the deletion of a dependent should cascade through in order to delete all related participations of the dependent in hobbies.*

12. A hobby with at least one dependent participating in it cannot be deleted.

*The R immediately above the HOBBY entity type indicates the restriction of a deletion of a hobby if the hobby has one or more participants. Once the participation of all dependents in a particular hobby has been deleted, then the hobby can be deleted.*

A close scrutiny of the business rules pertaining to entity deletions reveals some conflicting rule specifications. In fact, there are three such planted errors in the specification of deletion rules indicated in Figure 3.4.

First, deletion rule 3, which implies a “Set null” constraint, will, if implemented, conflict with the total participation constraint of PROJECT in the *Undertaken\_by* relationship; this is highlighted in Figure 3.4 by enclosing the N in a box. In other words, when every project entity is required to be undertaken by some plant, how can one at the same time specify that a project need not be undertaken by any plant? Thus, either the deletion constraint or the participation constraint must be changed by the user. One way of doing this is to relax the mandatory participation to optional, as implemented in Figure 3.5.

Second, deletion rule 1, which implies a “Restrict” constraint on PLANT, will, if implemented, conflict with the total participation constraint of PLANT in the *Works\_in* relationship; this is highlighted in Figure 3.4 by enclosing the R in a box. In other words, when every plant entity is required to employ at least one (in this case, 100) employee(s), how can one at the same time specify that a plant with employees cannot be deleted? If implemented, this rule will make sure that no plant is ever deleted from the database! The business rule that every plant must have at least 100 employees working in it prevents relaxation of the mandatory participation of PLANT in the *Works\_in relationship* to optional participation. A feasible solution in this case is to replace the deletion constraint R (Restrict) on PLANT with a deletion constraint of C (Cascade) or D (Set default) on EMPLOYEE and propose this as a revision to the business rule to the user community. This solution is reflected in Figure 3.5 using the deletion constraint C.

Third, let us examine the two deletion constraints—C (Cascade) and R (Restrict)—on DEPENDENT in the *Dependent\_of* relationship and the *Held\_by\_D* relationships, respectively. In the list of deletion constraints, business rules 7 and 8 correspond to these two deletion constraints. Accordingly, all dependents of an employee should be removed from the database when an employee is deleted from the database. This works as long as none of the dependents has a BCU\_account. Even if one of the dependents of the employee being deleted from the database happens to have at least one BCU\_account, a rules conflict arises—one rule specifying removal of the dependent from the database and the other rule expecting restriction of the removal. The conflict resolution suggested in Figure 3.5 entails changing the restrict constraint on DEPENDENT in the *Held\_by\_D* relationship to a cascade constraint on BCU\_ACCOUNT.

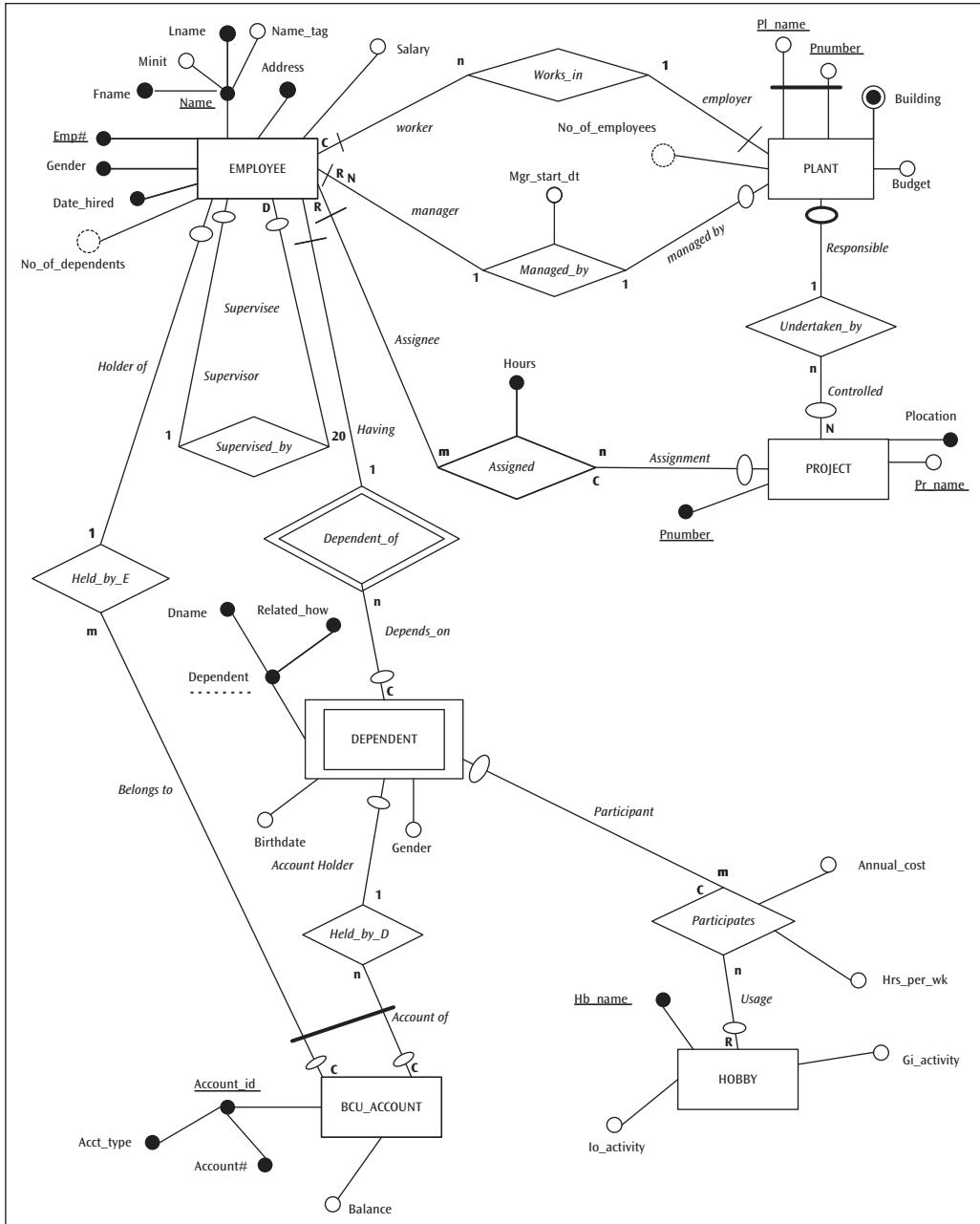


FIGURE 3.5 Presentation Layer ER diagram for Bearcat Incorporated—Final

Please note that in all three cases, alternative solutions are available, too. The reader is encouraged to explore the alternative solutions.

There are several other business rules stated in the requirements specification that still cannot be captured in the Presentation Layer ERD. These rules are expressed as a list of Semantic Integrity Constraints (SICs) and are shown in Table 3.1. The SIC list is a crucial supplement to the ERD in order to preserve all the information conveyed in the data requirements. These Semantic Integrity Constraints are grouped into two categories: attribute-level business rules, and entity-level business rules. For example, the business rule requiring gender to be either male or female cannot be captured in an ERD and therefore must be recorded as a semantic integrity constraint; all domain constraints on attributes usually fall under the category of attribute-level business rules. A constraint of the form “an employee cannot be his or her own supervisor” entails verification involving more than one attribute in the same entity. Such business rules are captured in the SIC list in the entity-level business rules. Next, the business rule that the salary of an employee cannot exceed the salary of the employee’s supervisor cannot be captured as a constraint in the ERD *per se*, nor can it be expressed as an attribute-level or entity-level business rule; therefore, it must be captured as a semantic integrity constraint that takes the form

#### **Attribute-Level Business Rules**

1. Each plant has a plant number that ranges from 10 to 20.
2. The gender of each employee or dependent is either male or female.
3. Project numbers range from 1 to 40.
4. Project locations are confined to the cities of Bellaire, Blue Ash, Mason, Stafford, and Sugarland.
5. Account types are coded as follows: C: checking account; S: savings account; I: investment account.
6. A hobby can be either an indoor activity (I) or an outdoor activity (A).
7. A hobby can be either a group activity (G) or an individual activity (I).

#### **Entity-Level Business Rules**

1. A mother or daughter dependent must be a female, a father or son dependent must be a male, and a spouse dependent can be either male or female.
2. An employee cannot be his or her own supervisor.
3. A dependent may have a joint account only with an employee of Bearcat Incorporated to whom he or she is related.
4. Every plant is managed by an employee who works in the same plant.

#### **Miscellaneous Business Rules**

1. Each plant has at least three buildings.
2. Each plant must have at least 100 employees.
3. The salary of an employee cannot exceed the salary of the employee’s supervisor.

**TABLE 3.1** Semantic integrity constraints for the Presentation Layer ER model

of a miscellaneous business rule. In short, business rules that cannot be expressed as an attribute-level or entity-level business rule are grouped under the miscellaneous category in the SIC list. Together, the Presentation Layer ERD and the associated SIC list constitute the Presentation Layer ER model.

Often, it is difficult to determine if a data element is an attribute or something that should be modeled as an entity type or a relationship type. For instance, shouldn't MANAGER of a plant be modeled as an entity type in the Presentation Layer ERD? If it is not, where will the start date of the manager of a plant be recorded? Although creating a MANAGER entity type appears reasonable, a closer look at the data requirements described in the narrative in Section 3.1 reveals that the only attribute of such an entity type would be manager start date. Furthermore, if a unique identifier for MANAGER is not obvious, one might, as a first step, consider the possibility of modeling plant manager as a weak entity type child of either the EMPLOYEE entity type or the PLANT entity type. Alternatively, given its relationship to both EMPLOYEE and PLANT, one might approach modeling the presence of a plant manager as a relationship between EMPLOYEE and PLANT (see Figure 3.3) and capture **Mgr\_start\_date** as an attribute of the relationship type *Managed\_by*, where the EMPLOYEE entity type plays the role Manager and the PLANT entity type plays the role Managed\_by.

Also, if the **Building** attribute of the PLANT entity type can be modeled as a multi-valued attribute, why can't the dependents of an employee be modeled as a multi-valued attribute of the EMPLOYEE entity type? The answer is that this is possible except for the fact that DEPENDENT is involved in two "external" relationships (one called *Held\_by\_D* with BCU\_ACCOUNT, another called *Participates* with HOBBY). In other words, if DEPENDENT is modeled as a multi-valued attribute of EMPLOYEE instead of the weak entity child of EMPLOYEE, the *Held\_by\_D* and *Participates* relationships could not be shown.

### 3.2.3 The Design-Specific ER Model

While the Presentation Layer ER model serves as a vehicle for the analyst to interact with the user community, additional information is required for the ultimate design and implementation of the database. For instance, additional details about the characteristics of attributes must be obtained from the users in order to further define the nature of each attribute. For each attribute defined in the data requirements, Table 3.2 lists an entity type or relationship type name, abbreviated attribute name, data type, and size. Attribute level business rules from the SIC list of the Presentation Layer ER model are mapped into this table; in addition a domain constraint on **Emp#** was also secured from the user(s) at this time and mapped into the SIC list in the Design-Specific tier (Table 3.2). The entity type names are obtained from the Presentation Layer ERD.

#### 3.2.3.1 The (Min, Max) Notation for the Structural Constraints of Relationships

As noted in Section 3.2.1, a handful of popular notational schemes for the ERD exists. In this section, the **(min, max)** notation, originally prescribed by Abrial (1974) for specifying the structural constraints of a relationship, is introduced. Here, *min* depicts the minimum cardinality of an entity type's participation in a relationship type—the participation constraint—and *max* indicates the maximum cardinality of an entity type's participation

Entity/ Relationship Type Name	Attribute Name	Data Type	Size	Domain Constraint
PLANT	Pl_name	Alphabetic	30	
PLANT	Pnumber	Numeric	2	Integer values from 10 to 20.
PLANT	Budget	Numeric	7	
PLANT	Building	Alphabetic	20	
PLANT	No_of_employees	Numeric	7	
EMPLOYEE	Emp#	Alphanumeric	6	ANNNNN, where A can be letter and NNNNN is a combination of five digits
EMPLOYEE	Fname	Alphabetic	15	
EMPLOYEE	Minit	Alphabetic	1	
EMPLOYEE	Lname	Alphabetic	15	
EMPLOYEE	Name_tag	Numeric	1	Integer values from 1 to 9.
EMPLOYEE	Address	Alphanumeric	50	
EMPLOYEE	Gender	Alphabetic	1	M or F
EMPLOYEE	Date_hired	Date	8	
EMPLOYEE	Salary	Numeric	6	Ranges from \$35,000 to \$90,000
EMPLOYEE	No_of_dependents	Numeric	2	
<i>Managed_by</i>	Mgr_start_dt	Date	8	
DEPENDENT	Dname	Alphabetic	15	
DEPENDENT	Related_how	Alphabetic	12	
DEPENDENT	Birthdate	Date	8	
PROJECT	Pr_name	Alphabetic	20	
PROJECT	Pnumber	Numeric	2	Integer values from 1 to 40.
PROJECT	Plocation	Alphabetic	15	Bellaire, Blue Ash, Mason, Stafford, Sugarland
Assigned	Hours	Numeric	3	

**TABLE 3.2** Semantic integrity constraints for the initial Design-Specific ER model

<b>Entity/ Relationship Type Name</b>	<b>Attribute Name</b>	<b>Data Type</b>	<b>Size</b>	<b>Domain Constraint</b>
BCU_ACCOUNT	Balance	Numeric	(8.2)*	
BCU_ACCOUNT	Account#	Alphanumeric	6	
BCU_ACCOUNT	Acct_type	Alphabetic	1	C=Checking Acct., S=Savings Acct., I=Investment Acct.
HOBBY	Hb_name	Alphabetic	20	
HOBBY	Io_activity	Alphabetic	1	I=Indoor Activity, O=Outdoor Activity
HOBBY	Gi_activity	Alphabetic	1	G=Group Activity, I=Individual Activity
Participates	Hrs_per_wk	Numeric	(2.1)*	
Participates	Annual_cost	Numeric	6	

\*(n<sub>1</sub>.n<sub>2</sub>) is used to indicate n<sub>1</sub> places to the left of the decimal point and n<sub>2</sub> places to the right of the decimal point

**Entity-Level Domain Constraints**

1. A mother or daughter dependent must be a female, a father or son dependent must be a male, and a spouse dependent can be either gender.
2. An employee cannot be his or her own supervisor.
3. A dependent may have a joint account only with an employee of Bearcat Incorporated to whom he or she is related.
4. Every plant is managed by an employee who works in the same plant.

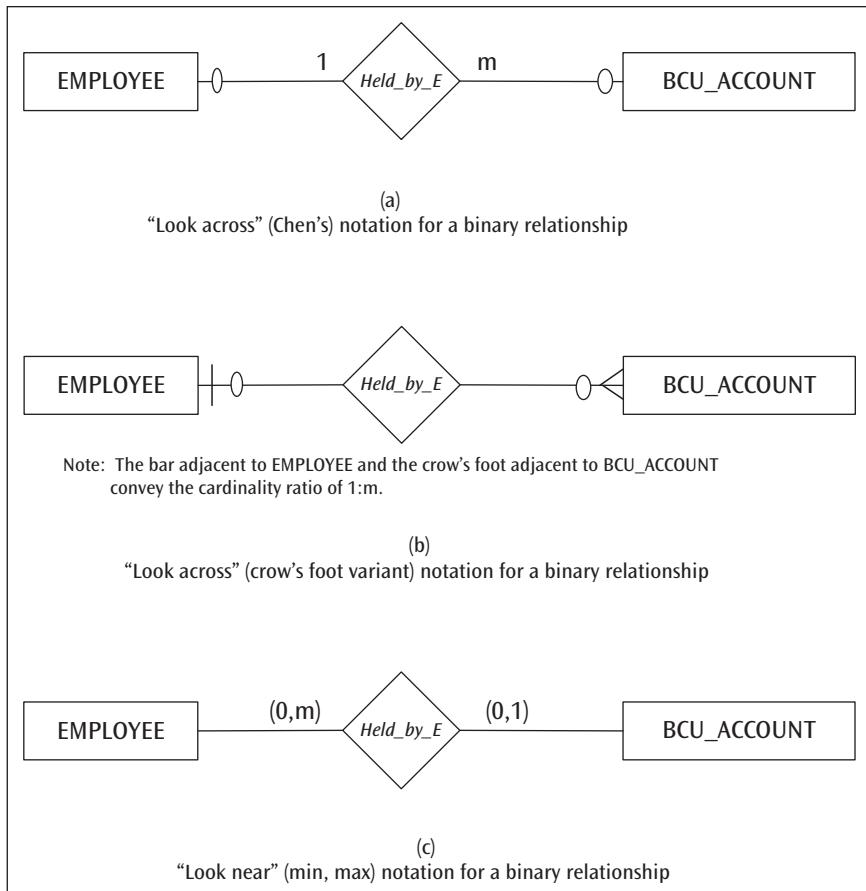
**Remaining Miscellaneous Constraints**

1. Each plant has at least three buildings.
2. The salary of an employee cannot exceed the salary of the employee's supervisor.

**TABLE 3.2** Semantic integrity constraints for the initial Design-Specific ER model (continued)

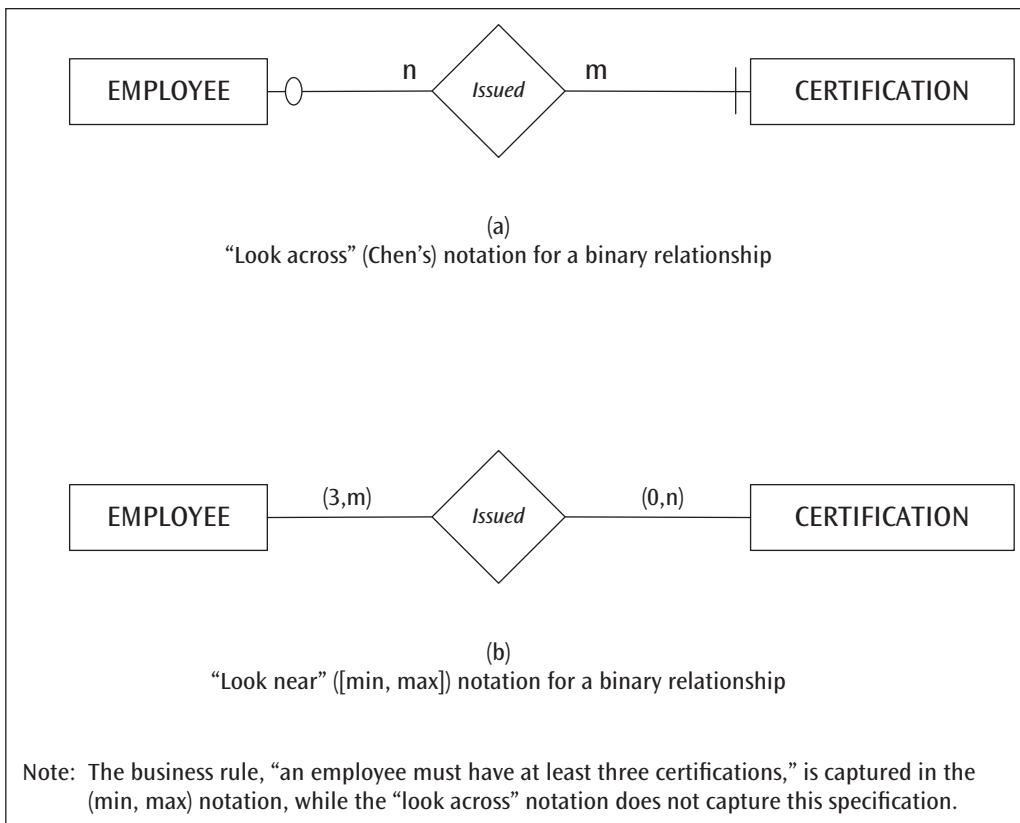
in a relationship, thus reflecting the cardinality ratio. This notation is in general more precise and particularly more expressive for specifying relationships of higher degrees beyond two. In this notation, a pair of finite whole numbers (min, max) is used with each *participation* of an entity type E in a relationship type R without any reference to other entity types participating in the relationship R where  $0 \leq \text{min} \leq \text{max}$  and  $\text{max} > 0$ . The meaning conveyed here is that each entity e in E participates in at least “min” and at most “max” relationships ( $r_1, r_2, \dots$ ) in R. In this notation,  $\text{min}=0$  implies partial or optional participation of E in R, and  $\text{min} \geq 1$  implies total or mandatory participation of E in R.

The mapping of the structural constraints of a relationship from the presentation layer to the Design-Specific layer—in other words, converting Chen’s notation to the (min, max) notation—is demonstrated in Figure 3.6. Figure 3.6a indicates that an employee may have several BCU accounts but need not have any. This is done by “looking across,” from EMPLOYEE to BCU\_ACCOUNT, through the relationship *Held\_by\_E* (Chen’s notation). On the other hand, a BCU account belongs to no more than one employee, and some BCU accounts need not belong to any employee. This is inferred by “looking across,” from BCU\_ACCOUNT to EMPLOYEE, in the relationship *Held\_by\_E*. The same metadata is reflected in Figure 3.6b using the Crow’s Foot notation. In the (min, max) notation shown in Figure 3.6c, the cardinality ratio (maximum cardinality) and participation (minimum cardinality) constraint in the *Held\_by\_E* relationship type are captured in terms of the *participation* of EMPLOYEE in the *Held\_by\_E* relationship type independent of the *participation* of BCU\_ACCOUNT in the *Held\_by\_E* relationship type. An employee participates in at least zero (0) (optional participation) and at most m *Held\_by\_E* relationships, meaning an employee need not have a BCU account, but may have many BCU accounts. Likewise, a BCU account participates in at least zero (0) (optional participation) and at most one (1) *Held\_by\_E* relationship, meaning a BCU account need not belong to any employee, but can belong to a maximum of one employee. Note that the (min, max) notation employs a “look near” approach as opposed to the “look across” approach used in the Presentation Layer ERD.

**FIGURE 3.6** Introduction of (min, max) notation for an 1:m binary relationship

An example of mapping an m:n relationship from the “look across” (Chen) notation to the “look near” notation ([min, max]) is shown in Figure 3.7. Here, two points are noteworthy:

- The fact that an employee must have at least three certifications can be explicitly captured in the (min, max) notation; the “look across” notation cannot capture this specification.
- Because m and n represent different maximum cardinality values, it is imperative that these two values not be arbitrarily placed in the mapping process; an employee cannot be issued more than m certifications, and a specific certification cannot be issued to more than n employees, as indicated in Figure 3.7b.



**FIGURE 3.7** (min, max) notation for an m:n binary relationship

The mapping of the structural constraints specified in the Presentation Layer ERD in Figure 3.5 to the (min, max) notation appears in the initial version of the Design-Specific ERD in Figure 3.8. In contrast to what is shown in Figure 3.5, observe how the (min, max) notation allows the requirement that a plant must have at least 100 employees to be explicitly specified. Also, note that a weak entity type participating as a child in an identifying relationship type, since it also has existence dependency on the identifying parent, will always have a (min, max) value of (1,1) in that relationship.

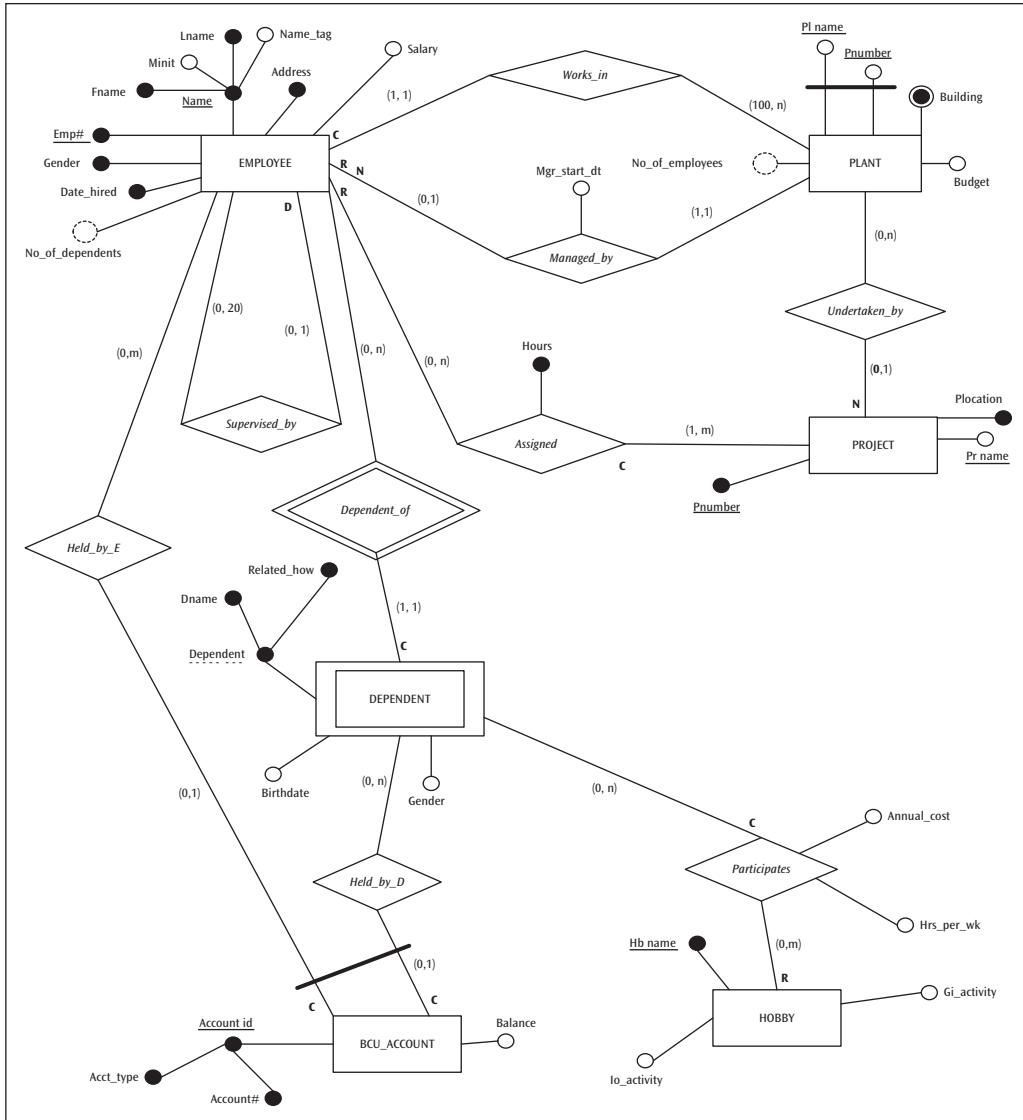


FIGURE 3.8 Design-Specific ER diagram for Bearcat Incorporated—Initial

The other business rules not mapped to the Design-Specific ERD are included with the semantic integrity constraints for the Design-Specific ER model in Table 3.2. Thus, the Design-Specific ER model comprising the ERD in Figure 3.8 and semantic integrity constraints in Table 3.2 fully preserve all constructs and constraints reflected in the Presentation Layer ER model.

Now we have seen how the user-oriented Presentation Layer ER model is translated to a database design orientation. To this end, two specific steps were taken in the process of developing the Design-Specific ER model:

1. Collection of a few characteristics for attributes (e.g., data type and size)
2. Introduction of the technically more precise (min, max) notation for the specification of relationships in the ERD

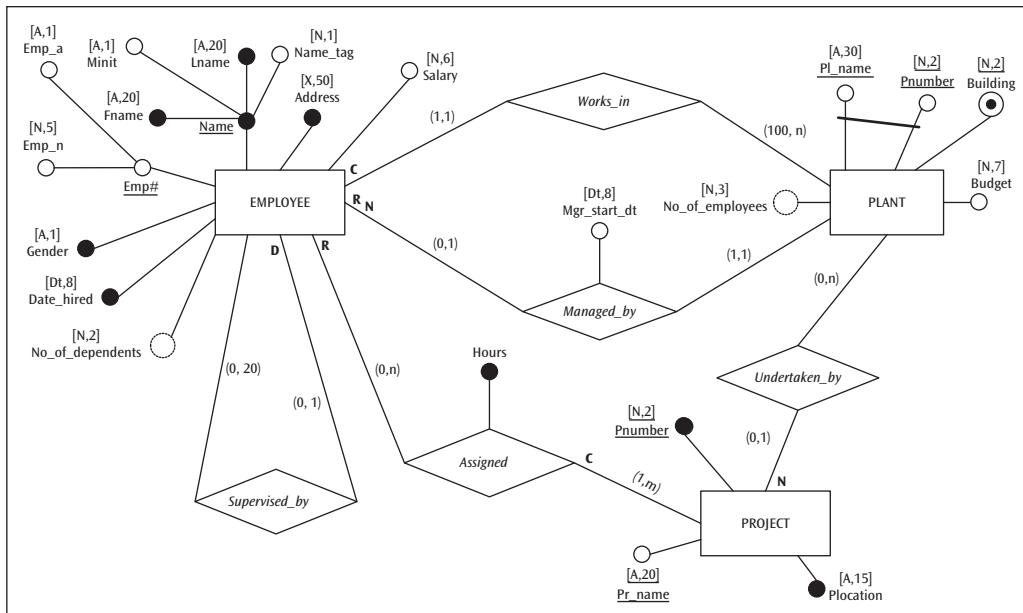
The next section presents the second stage of this line of enquiry in order to render the conceptual schema amenable to direct mapping to the logical level.

### 3.2.4 The Decomposed Design-Specific ER Model

The Design-Specific ER model incorporates data modeling constructs in the conceptual model beyond the Presentation Layer model. However, the Design-Specific ER model, while capturing the conceptual design reflected in the Presentation Layer ER model in all its richness, contains constructs that cannot be directly mapped to certain logical data models. It would be ideal to be able to portray all the specified constructs and constraints—*inherent, implicit, and explicit*—in the ERD itself so that an integrated view of the database design is available in one place. The incremental contributions of the decomposition of the Design-Specific ER model are:

- Mapping the attribute characteristics to the ERD
- Appropriately decomposing constructs in the ERD that cannot be directly mapped to a logical schema—that is, preparing a decomposed Design-Specific ER model for direct mapping to a logical schema

Figure 3.9 presents an example of mapping the attribute characteristics to the ERD.

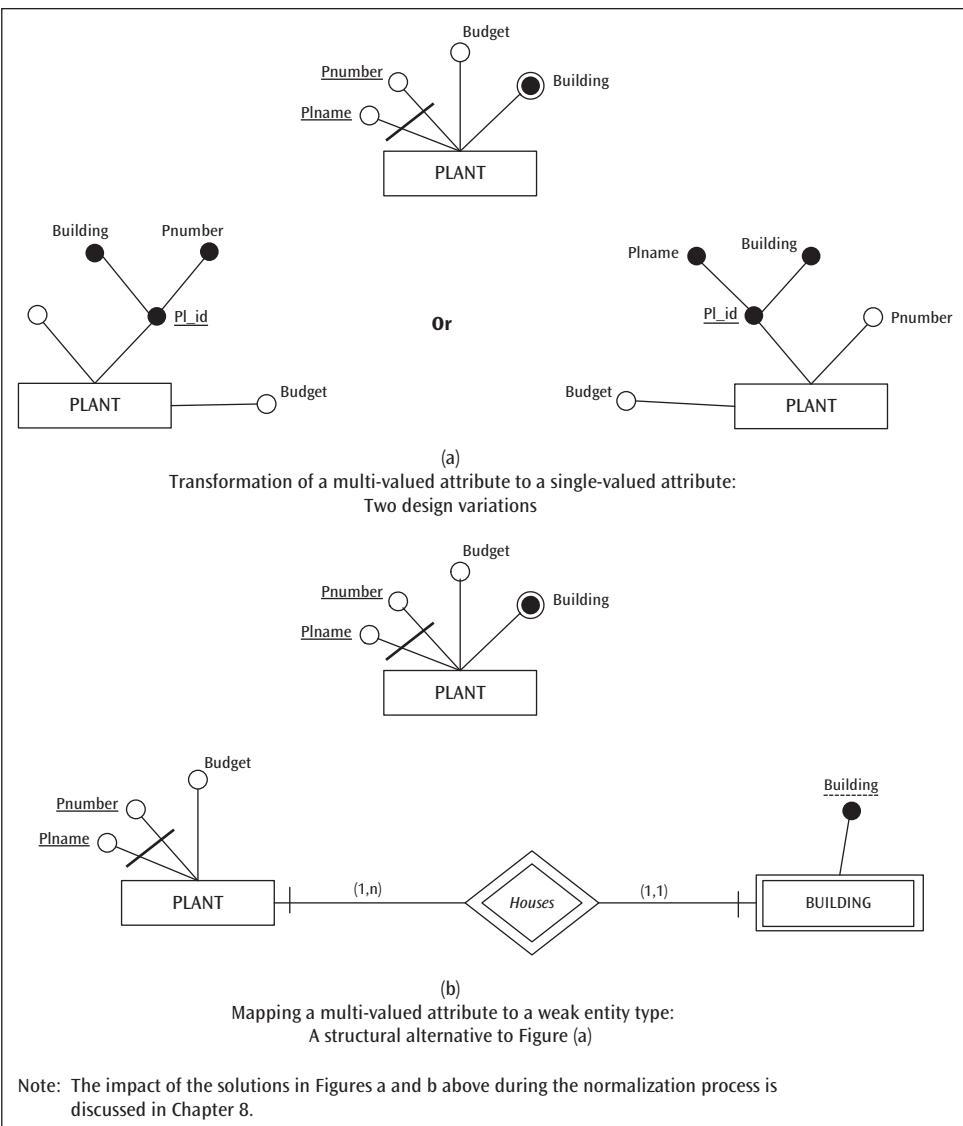
**FIGURE 3.9** Design-Specific ER diagram with attributes mapping—An example

Of the various constructs and constraints supported by the ER modeling grammar, multi-valued attributes and binary relationships of cardinality ratio m:n, while adding to the expressive power of a conceptual schema, cannot be directly implemented in a logical data model. As a consequence, these two constructs will have to be decomposed to lower-level constructs that can be directly mapped to a logical schema.

### 3.2.4.1 Resolution of Multi-Valued Attributes

A multi-valued attribute in an entity type can be transformed to a single-valued attribute by simply including the attribute as part of the unique identifier of the entity type.

Figure 3.10a shows the resolution of the multi-valued attribute **Building** in the entity type PLANT for Bearcat Incorporated. Note that the presence of two unique identifiers in PLANT makes the solution rather interesting in the sense that the attribute **Building** is appended to either **Pnumber** or **Plname** to form a single new unique identifier: **Pl\_id**. Table 3.3a shows sample data for each variation.



**FIGURE 3.10** Two structural alternatives for the resolution of a multi-valued attribute

**Original PLANT Data Set**

Pnumber	Plname	Budget	Building
10	Underwood	3000000	1 2 3
11	Garnett	3000000	1 2
12	Belmont	3500000	1
13	Vanderbilt	3500000	1 2

**Revised PLANT Data Set- Variation 1**

Pnumber	Building	Plname	Budget
10	1	Underwood	3000000
10	2	Underwood	3000000
10	3	Underwood	3000000
11	1	Garnett	3000000
11	2	Garnett	3000000
12	1	Belmont	3500000
13	1	Vanderbilt	3500000
13	2	Vanderbilt	3500000

**Revised PLANT Data Set- Variation 2**

Pnumber	Building	Plname	Budget
10	1	Underwood	3000000
10	2	Underwood	3000000
10	3	Underwood	3000000
11	1	Garnett	3000000
11	2	Garnett	3000000
12	1	Belmont	3500000
13	1	Vanderbilt	3500000
13	2	Vanderbilt	3500000

(a)

Sample data illustrating the transformation of a multi-valued attribute to a single-valued attribute (see Figure 3.10a)

**Original PLANT Data Set**

Pnumber	Plname	Budget	Building
10	Underwood	3000000	1 2 3
11	Garnett	3000000	1 2
12	Belmont	3500000	1
13	Vanderbilt	3500000	1 2

**Revised PLANT Data Set**

Pnumber	Plname	Budget
10	Underwood	3000000
11	Garnett	3000000
12	Belmont	3500000
13	Vanderbilt	3500000

**Revised  
BUILDING  
Data Set**

Building
1
2
3
1
2
1
1
2

(b)

Sample data illustrating the transformation of a multi-valued attribute to a weak entity type (see Figure 3.10b)

**TABLE 3.3** Sample data sets for Figure 3.10

An alternative solution to decomposing a multi-valued attribute to a single-valued attribute is to transform the multi-valued attribute to an entity type. In this case, BUILDING will become a weak entity type child in an identifying relationship with PLANT, and the attribute **Building** will serve the role of the partial key of BUILDING (see Figure 3.10b and Table 3.3b).

The first solution (Figure 3.10a) is apparently simpler and appears more efficient than the alternative solution (Figure 3.10b); however, this solution will pose a data redundancy problem that will then have to be resolved in the logical data model during normalization.<sup>5</sup> Interestingly, such resolution will always result in a schema equivalent to the alternative solution proposed in Figure 3.10b. Therefore, the second solution is used in the final ERD in Figure 3.13. Observe that creation of a new entity type (BUILDING) and relating it to PLANT requires specification of a deletion constraint for the relationship. While rule of thumb suggests a default value of “Restrict” (R), R is not compatible with the total participation of PLANT in this relationship. (Every plant has at least three buildings). Therefore, the “Set default” (D) is assumed by supposing that when a plant is closed the associated buildings will be reassigned to some other plant.

### 3.2.4.2 Resolution of m:n Relationship Types

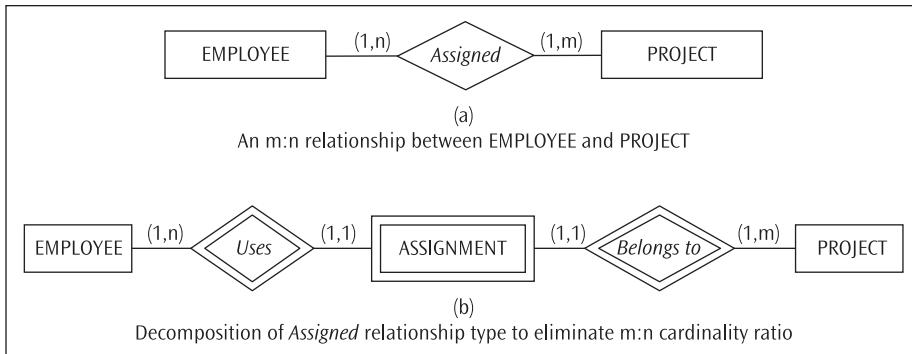
A binary relationship with a cardinality ratio of m:n also poses a problem in that it cannot be represented in a logical data model as is (i.e., as an m:n relationship). The solution for this problem is to decompose the m:n relationship to two relationships (1:n and 1:m), with a newly created entity type serving as a bridge between the two base entity types in the resulting two binary relationships. Since such a bridging entity type at the intersection of the two base entity types is artificially created, it does not have its own natural unique identifier. The entity type thus artificially created is called a **gerund entity type**. Although a gerund entity type structurally mimics the characteristics of a weak entity type, there is a fundamental difference between the two. A weak entity type is always a product of the semantics behind the ER model, mostly a multi-valued attribute translating to a partial key (discriminator) capturing the semantics therein. A gerund entity type, on the other hand, is strictly an artificial entity type resulting from the decomposition of an m:n binary relationship type and so does not contain a partial key.<sup>6</sup> A gerund entity type is sometimes referred to as a **composite entity type** or a **bridge entity type**.

Any attributes of the m:n relationship type now become the attributes of the gerund entity type. The example in Figure 3.11 illustrates the resolution of an m:n cardinality ratio in a binary relationship type. Observe that the relationship type *Assigned* in Figure 3.11a is expressed as a weak entity type ASSIGNMENT in the decomposition shown in Figure 3.11b. In essence, a new (gerund) entity type has been created in place of the relationship type depicting the m:n relationship between EMPLOYEE and PROJECT. The direct relationship between EMPLOYEE and PROJECT no longer exists. The new (gerund) entity type, ASSIGNMENT, is not only related to the EMPLOYEE as well as the PROJECT, both of the relationships are also identifying relationship types. This means that ASSIGNMENT has two identifying parents and has identification dependency on both parents (EMPLOYEE and PROJECT). As a result, the m:n cardinality ratio has been decomposed to a 1:n (*Uses*) and 1:m (*Belongs\_to*) relationship.

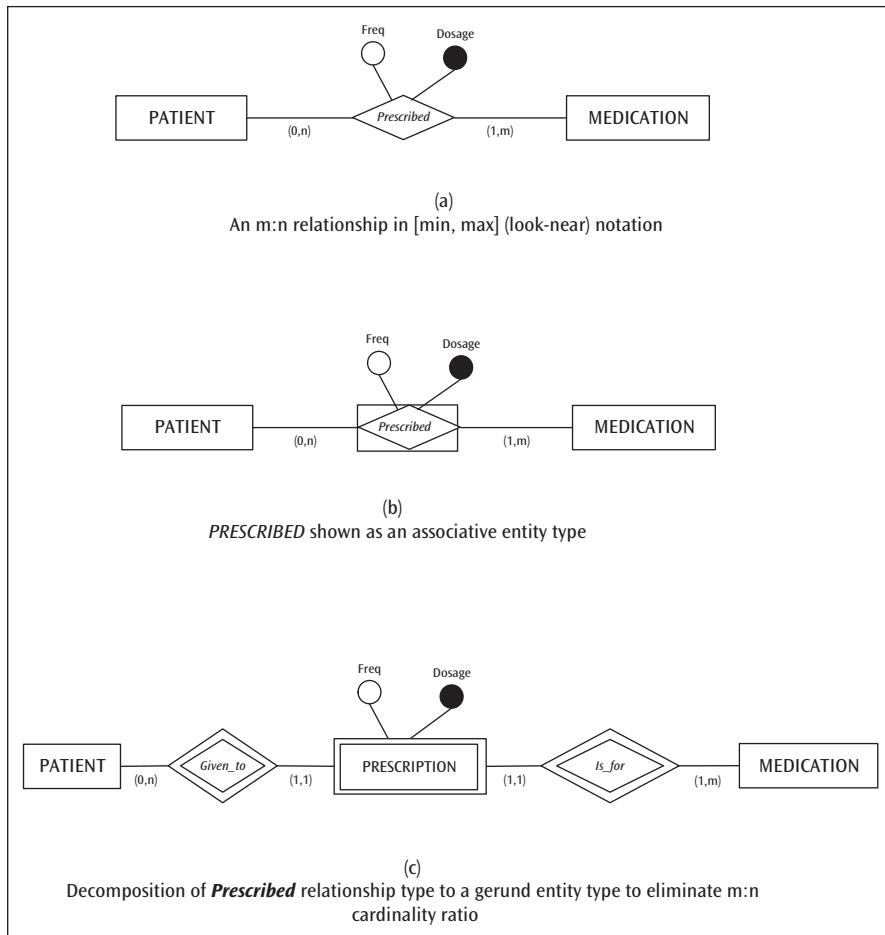
---

<sup>5</sup>Normalization is covered in Chapter 8.

<sup>6</sup>This issue is not unique to binary relationships. Any n-ary relationship (ternary, quaternary, etc.) poses the same problem and has a similar solution. This issue is discussed further in Chapter 5.

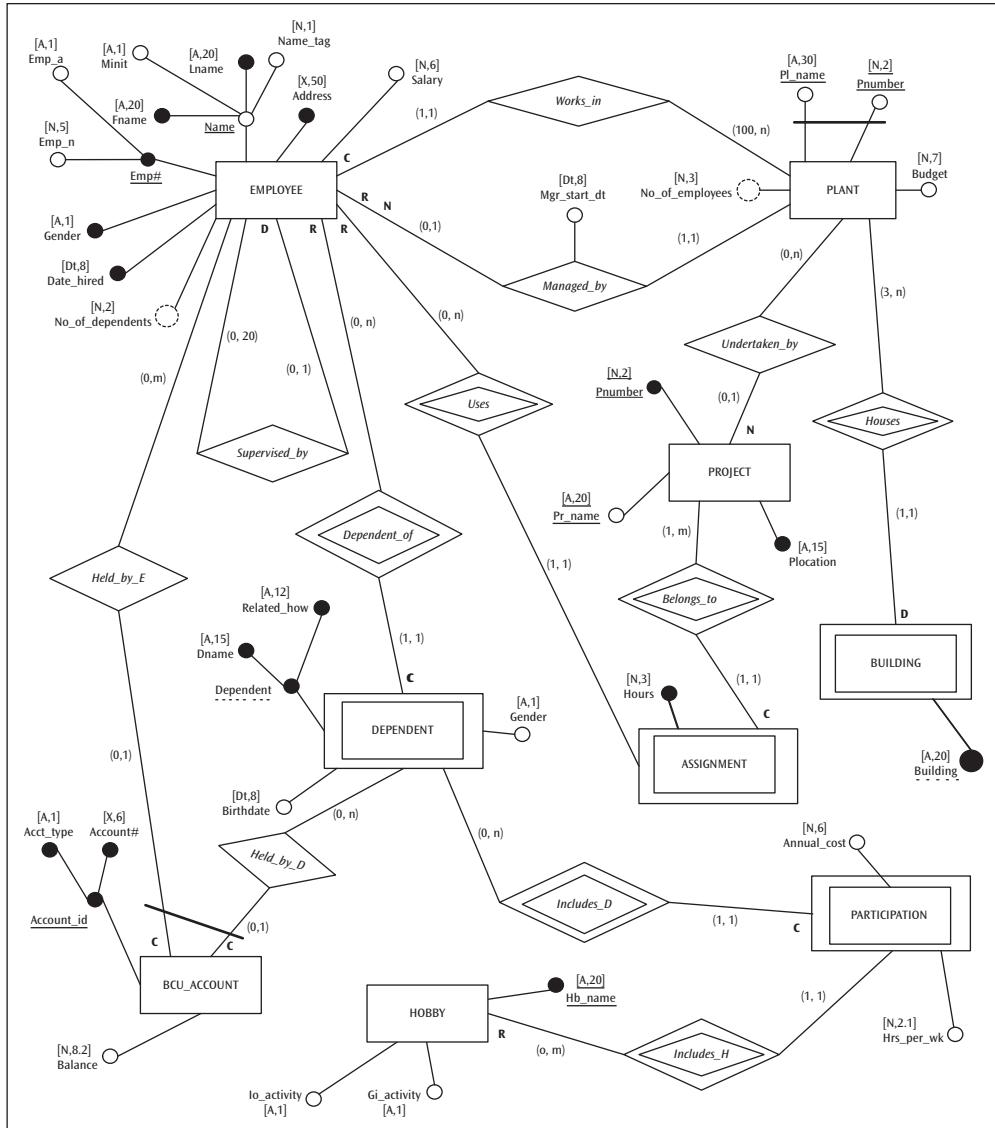
**FIGURE 3.11** Resolution of the m:n cardinality constraint via decomposition

A second example for the decomposition of a m:n relationship type is presented in Figure 3.12.

**FIGURE 3.12** A second example for the decomposition of an m:n relationship

### 3.2.4.3 The Final Design-Specific ER model

The final transformation of the Design-Specific ER model for Bearcat Incorporated, where the multi-valued attribute(s) and relationships with m:n cardinality constraints have been decomposed, is shown in Figure 3.13. Observe that in addition to the transformations discussed earlier (i.e., multi-valued attribute and m:n relationship), this ERD introduces a notation that allows the data type and size associated with each atomic attribute listed in Table 3.2 to be incorporated into the ERD. Enclosed in square brackets ([ ]) adjacent to the name of each atomic attribute is its data type (A: alphabetic data type; N: numeric data type; X: alphanumeric data type; Dt: date data type) followed by its size. When a numeric attribute represents a decimal value, its size is represented by  $n_1.n_2$ , where  $n_1$  is the scale (number of positions to the left of the decimal point) and  $n_2$  is the precision (number of positions to the right of the decimal point).



**FIGURE 3.13** Design-Specific ER diagram for Bearcat Incorporated—Final

Note that the domain constraint on the **Emp#** in Table 3.2 specifies a numeric and alphabetic component for **Emp#**. This requirement is expressed in Figure 3.13 by making **Emp#** a composite attribute composed of the two atomic attributes **Emp\_n** and **Emp\_a**.

The integrity constraints not mapped to the final Design-Specific ERD are included with the semantic integrity constraints for the Design-Specific ER model in Table 3.4. Thus, the final Design-Specific ER model comprising the ERD in Figure 3.13 and constraint specifications in Table 3.4 fully preserve all constructs and constraints reflected in the Presentation Layer ER model.

> Constraint	PLANT.Pnumber	IN (10 through 20)
> Constraint	Nametag	IN (1 through 9)
> Constraint	Gender	IN ("M," "F")
> Constraint	Salary	IN (35000 through 90000)
> Constraint	PROJECT.Pnumber	IN (1 through 40)
> Constraint	Plocation	IN ("Bellaire," "Blue Ash," "Mason," "Stafford," "Sugarland")
> Constraint	Aect_type	IN ("C," "S," "I")
> Constraint	Io_activity	IN ("I," "O")
> Constraint	Gi_activity	IN ("G," "I")
> Constraint	Related_how	IN ("Spouse") OR IN (( "Mother," "Daughter") AND Gender IN ("F")) OR IN (( "Father," "Son") AND Gender IN ("M"))

**Constraints Carried Forward to Logical Design**

1. An employee cannot be his or her own supervisor.
2. A dependent can have a joint account only with an employee of Bearcat Incorporated with whom he or she is related.
3. The salary of an employee cannot exceed the salary of the employee's supervisor.
4. Every plant is managed by an employee who works in the same plant.

**TABLE 3.4** Semantic integrity constraints for the final Design-Specific ER model

### 3.3 DATA MODELING ERRORS

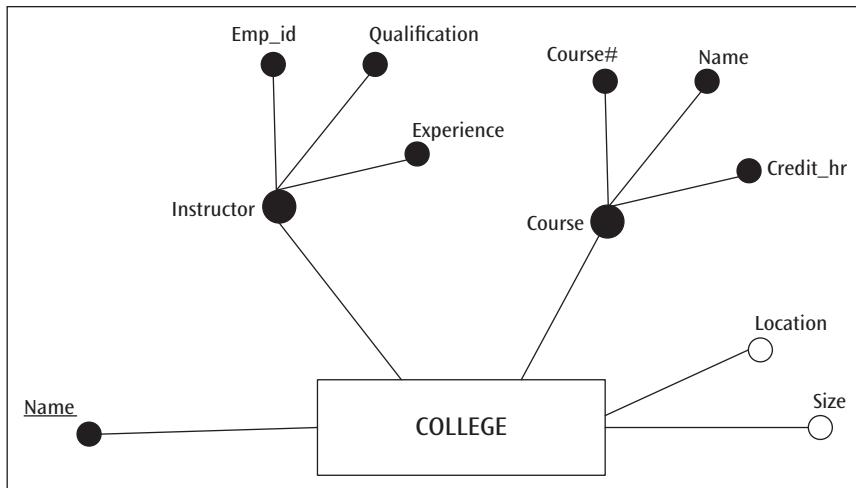
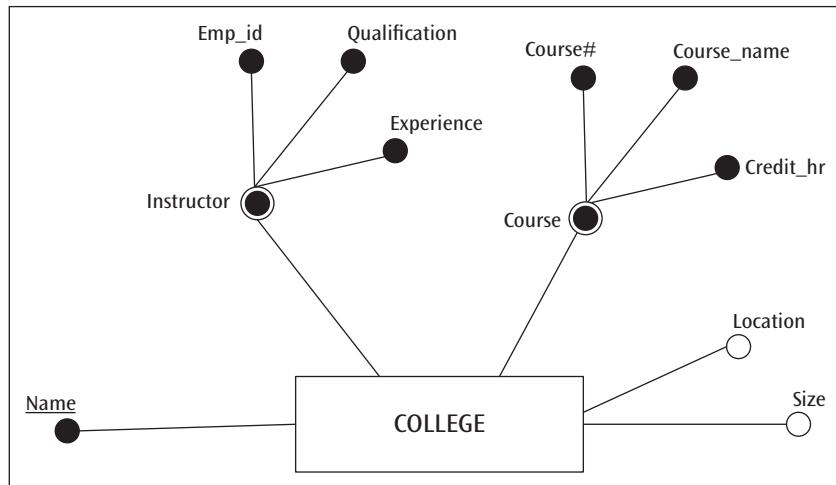
Research in conceptual modeling indicates that modeling errors are not uncommon, especially among novice data modelers. This section presents a few common data modeling errors using vignettes. These errors can be of two kinds: **semantic errors**, which misinterpret the requirements specification, and **syntactic errors**, which violate the grammar of the modeling language. A semantic error entails the incorrect imposition of a business rule that is implicit in the requirements specification and is therefore more difficult to identify since conceptual modeling is not an exact science. Attention to details, repeated practice, and experience over the long term are means by which one can minimize semantic errors in conceptual modeling. As for syntactic errors, they are relatively easy to avoid by simply knowing the grammar rules of the modeling language—in this case, the ER modeling grammar. How to avoid semantic errors in conceptual modeling is a topic addressed throughout Part I of this book.

### 3.3.1 Vignette 1

This vignette is a slightly expanded version of the example about a university's academic program, which was presented in Section 2.3.2.

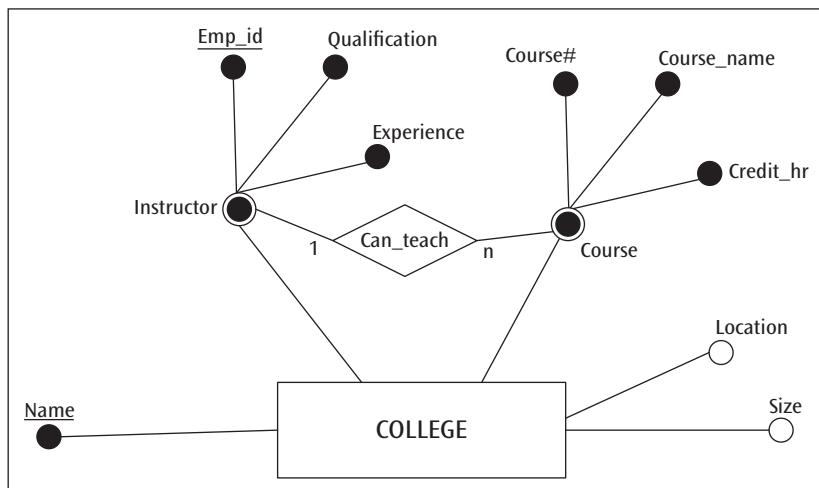
*There are several colleges in the university. Each college has a name, location, and size. A college offers many courses over four college terms or quarters—Fall, Winter, Spring, and Summer—during which one or more of these courses are offered. Course#, name, and credit hours describe a course. No two courses in any college have the same course#; likewise, no two courses have the same name. Terms are identified by year and quarter, and they contain enrollment numbers. Courses are offered during every term. The college also has several instructors. Instructors teach; that is why they are called instructors. Often, not all instructors are scheduled to teach during all terms, but every term has some instructors teaching. Also, the same course is never taught by more than one instructor in a specific term. Furthermore, instructors are capable of teaching a variety of courses offered by the college. Instructors have unique employee IDs, and their names, qualifications, and experience are also recorded.*

To begin with, COLLEGE may be modeled as an entity type since a collection of attributes—namely, **Name**, **Location**, **Size**, **Course**, and **Instructor**—seem to cluster under this title. Figure 3.14a portrays the COLLEGE entity type using the ER modeling grammar. The ERD at this point is syntactically correct. However, there are a couple of problems with reference to the semantics conveyed by this entity type. In Figure 3.14a, COLLEGE is modeled as an entity type with attributes that indicate that every college has one name, one location, one size, one course, and one instructor. Of course, the attributes **Course** and **Instructor** are correctly shown as composite attributes, with their appropriate content of atomic attributes. Nonetheless, this is not quite correct according to the semantics conveyed in the requirements specification. A college indeed offers many courses and also has several instructors. Therefore, these two attributes (**Course** and **Instructor**) must be modeled as multi-valued attributes, as shown in Figure 3.14b. Also, the attribute **Name** is duplicated in COLLEGE, one referring to the name of a college and the other referring to the name of a course. Duplicate attribute names within an entity type are semantically incorrect since the only reference point for the attributes is the entity type. Thus, one of the attributes labeled **Name** is changed to **Course\_name**, as shown in Figure 3.14b. The ERD at this point is both semantically and syntactically correct.

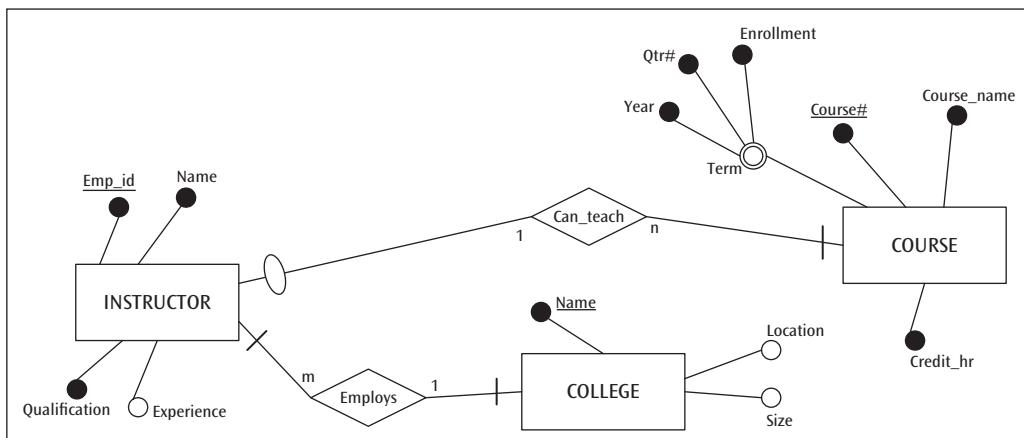
**FIGURE 3.14a** Entity type COLLEGE**FIGURE 3.14b** **Instructor** and **Course** as multi-valued attributes of the entity type COLLEGE

Next, we notice that instructors are capable of teaching a variety of courses. This indicates a relationship between instructors and courses. Given the current portrayal of the **COLLEGE** entity type, the relationship between instructors and courses can be modeled as shown in Figure 3.14c. While the data model shown in Figure 3.14c correctly conveys the intended semantics, the model violates a syntactic rule of the ER modeling grammar. While all attributes of an entity type are implicitly related to one another, an explicit relationship between attributes of an entity type independent of the entity type is not permitted in the ER modeling grammar. This syntactic error can be corrected only by

modeling COURSE and INSTRUCTOR as independent entity types related to the COLLEGE entity type and then establishing the relationship between the INSTRUCTOR and COURSE entity types. The corrected ERD is shown in Figure 3.14d. Since both COURSE and INSTRUCTOR have unique identifiers, they both are modeled as base entity types. Furthermore, since courses are offered every term and there are four terms, **Term** is also modeled here as an optional multi-valued attribute of COURSE; the optional property of the attribute allows for the possibility that some courses are not offered at all even though they may still be on the books. However, when **Term** has a value, it must have the quarter made up of **Year** and **Qtr#**.

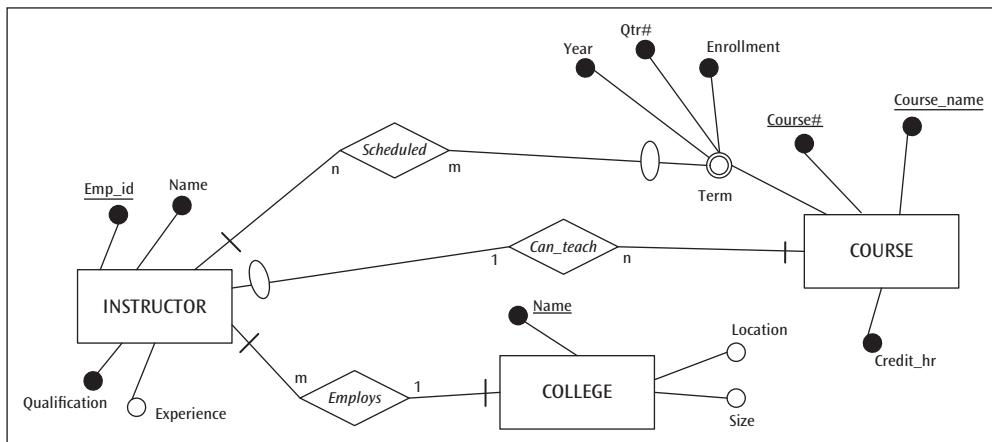


**FIGURE 3.14c** A syntactically incorrect relationship between **Instructor** and **Course** in **COLLEGE**

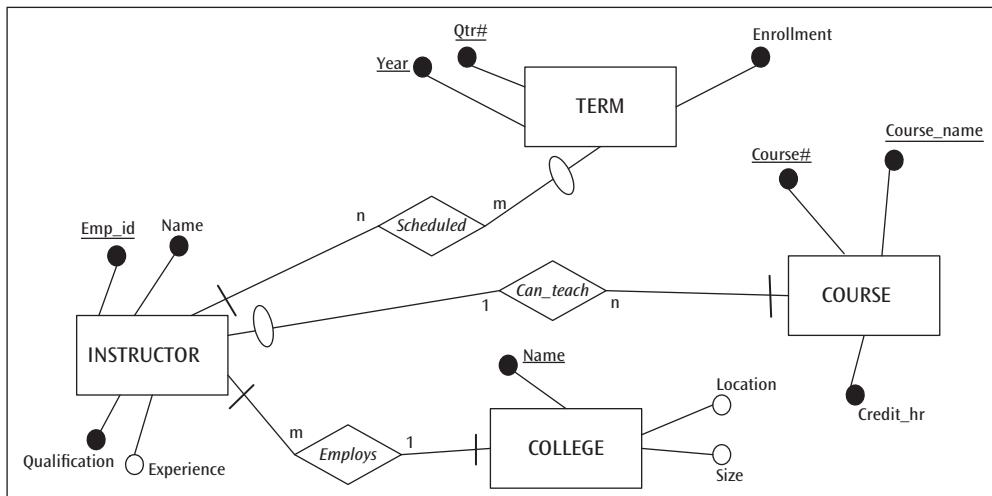


**FIGURE 3.14d** A syntactically correct relationship between **INSTRUCTOR** and **COURSE**

Next, **Course\_name**, according to the requirements specification, is also a unique identifier of COURSE and is not defined so in Figure 3.14d. The underlining of **Course\_name** as in Figure 3.14e corrects this error. We further notice that instructors are scheduled to teach during specific terms. The ERD in Figure 3.14e portrays this relationship. Once again, while semantically correct in expressing the notion of a relationship between terms and instructors, the ER modeling grammar does not permit modeling a relationship between an entity type and an attribute of an entity type. This grammatical error in the model script can be corrected by expressing TERM as a base entity type. The revised ERD is shown in Figure 3.14f.

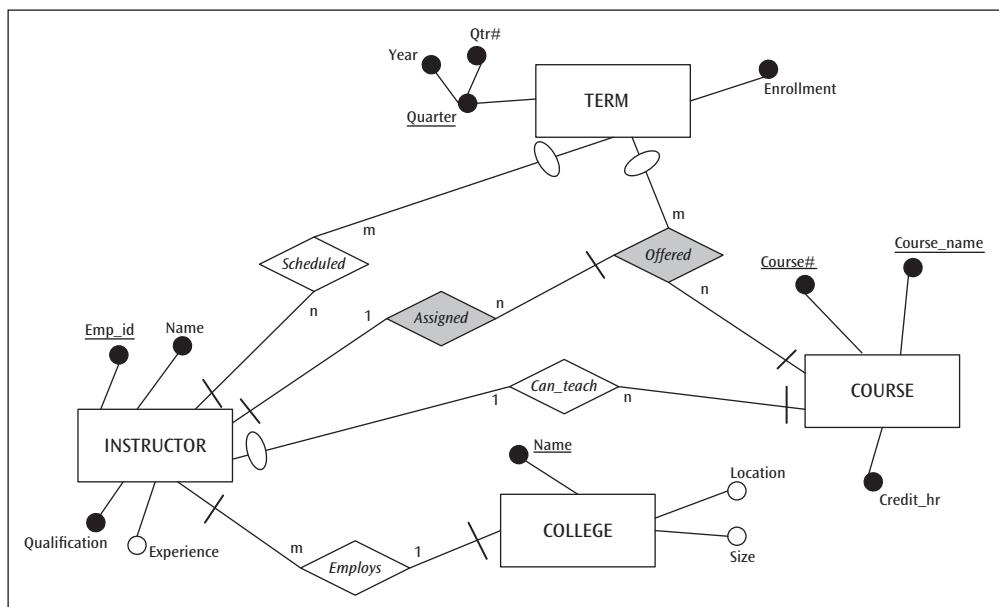


**FIGURE 3.14e** A syntactically incorrect m:n relationship between INSTRUCTOR and TERM



**FIGURE 3.14f** A syntactically correct m:n relationship between INSTRUCTOR and TERM

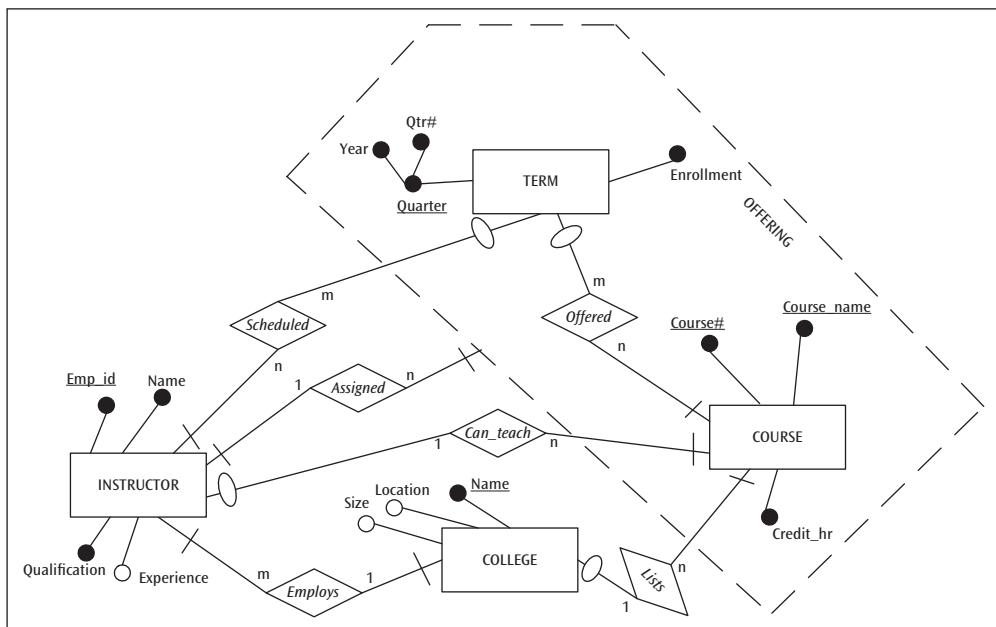
The conceptual model at this point captures the fact that instructors are scheduled to teach in the four terms and the instructors are capable of teaching several courses. However, the fact that courses are offered over the four terms and that in each term one or more of the courses are offered is yet to be incorporated in the ERD. More importantly, the business rule that *the same course is never taught by more than one instructor in a specific term* is not incorporated in the conceptual data model either. An m:n relationship between COURSE and TERM will capture the semantics of the first statement. This is shown in Figure 3.14g. On another note, the unique identifier of TERM is defined as the combination of **Year** and **Qtr#**. However, in Figure 3.14f, these two attributes are shown as two independent unique identifiers of the entity type TERM. This is a syntactic error and is corrected in Figure 3.14g by first constructing a composite attribute **Quarter** whose atomic components are **Year** and **Qtr#** and then labeling (underlining) **Quarter** as the unique identifier of TERM.



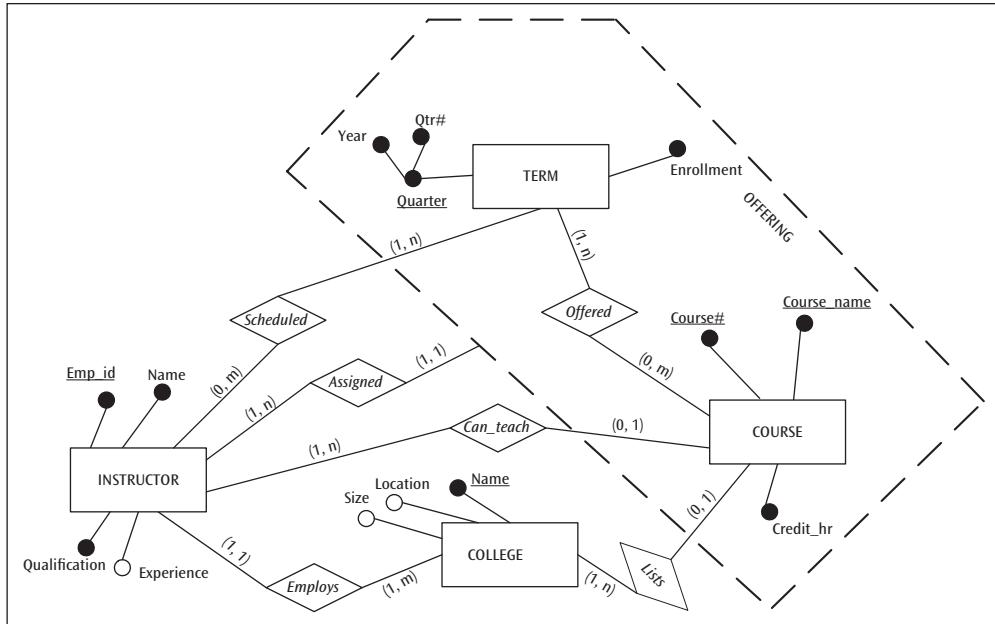
**FIGURE 3.14g** No more than one instructor per course in a term: Syntactically incorrect modeling

In addition, an attempt is made to convey the business rule that *the same course is never taught by more than one instructor in a specific term* by establishing a relationship between INSTRUCTOR and *Offered* (see Figure 3.14g). This relationship does correctly convey the semantics that a course offered in a term is assigned to only one instructor, that an instructor may teach several courses in the same term, and that the instructor may teach the same as well as other courses in other terms. However, there is a violation of the ER modeling grammar in the expression of relationship type *Assigned* since it relates an entity type, INSTRUCTOR, with a relationship type, *Offered*—in other words, there is, in effect, a diamond (*Assigned*) connecting to another diamond (*Offered*). The solution for this syntactic error is not immediately obvious. To get a

better understanding of this scenario, let us review the ERDs that appear in Figure 2.27a and 2.27b in Chapter 2. Following this line of logic, the m:n relationship type *Offered*, along with the entity types TERM and COURSE participating in this relationship, can be modeled as a cluster entity; the relationship type *Assigned* can now relate the base entity type INSTRUCTOR, and the cluster entity type OFFERING will accomplish expression of the intended semantics without violating the rules of the ER modeling grammar. The revised ERD is shown in Figure 3.14h. The final Presentation Layer ERD expressed using the “look near”—(min, max)—notation is shown in Figure 3.14i.

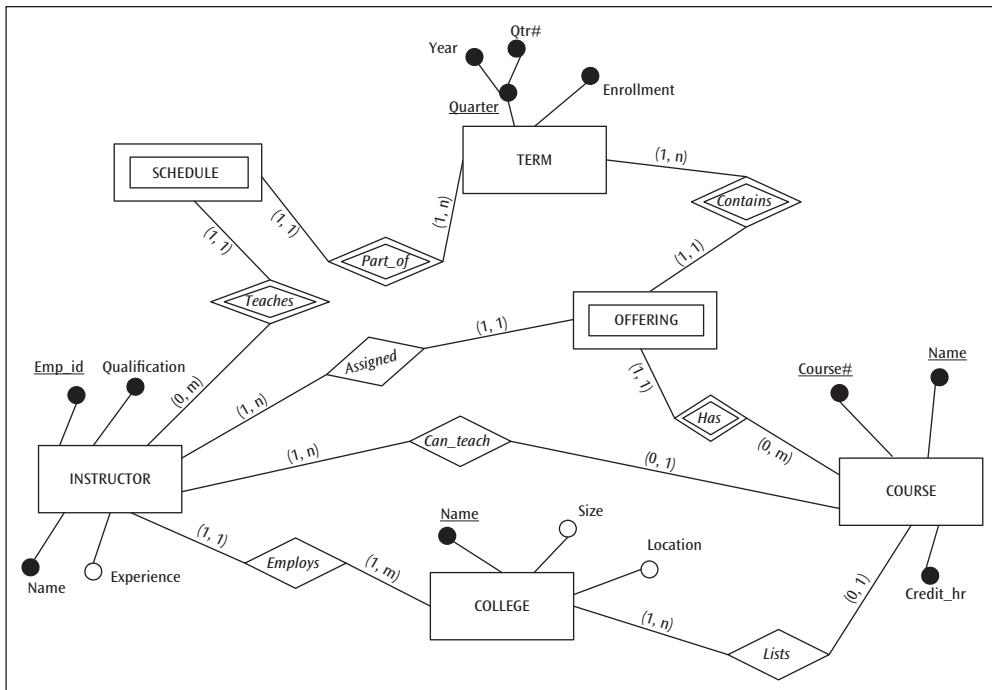


**FIGURE 3.14h** Presentation Layer ERD for vignette 1—“Look across” notation



**FIGURE 3.14i** Presentation Layer ERD for vignette 1—“Look near” (min, max) notation

The next step in the modeling process is to transform the Presentation Layer ERD to the design-specific layer. Since the vignette does not provide the attribute characteristics (data type, size, etc.), the only transformation that can be done at this point is to decompose multi-valued attributes, if any, to weak entity types and then decompose the m:n binary relationships to the gerund entity type with two identifying parents. It can be observed from Figure 3.14i that the Presentation Layer ERD does not contain any multi-valued attributes and that there are two m:n relationship types (*Assigned* and *Scheduled*). Observe the decompositions portrayed in Figure 3.14j, the Design-Specific ERD for vignette 1 transformed from the Presentation Layer ERD shown in Figure 3.14i.



**FIGURE 3.14j** Design-Specific Layer ERD for vignette 1—“Look near” (min, max) notation

As a final note, the scenario described in vignette 1 is not quite complete in its specification. The reader may find it a good exercise to first identify all business rules implicitly expressed in the vignette and then list the ambiguities present in the description of the scenario due to incomplete specification.

### 3.3.2 Vignette 2

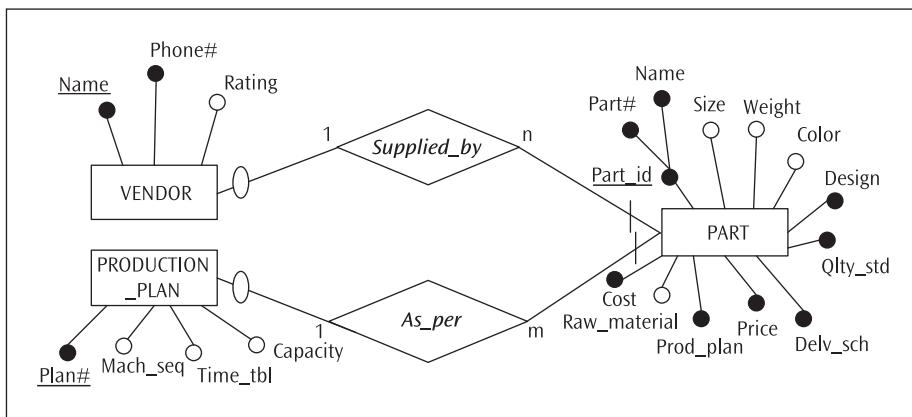
Widget USA is a widgets manufacturer located in Whitefield, Indiana. A widget is an intricate assembly of numerous parts. The assembly and manufacture of a few of the intricate parts are done at Widget USA’s small but highly sophisticated plant in Whitefield. All the other parts of the widget are outsourced to various vendors. Some of the vendors supply more than one part, but a specific part is supplied by only one vendor. A part has a unique part number and a unique name. Therefore, it is enough if a value for one of these two attributes is present for a given part. Other parts attributes are: size, weight, color, design, and quality standard. Manufactured parts have a cost and raw material, and they follow a production plan, whereas purchased parts have a price and delivery schedule. A production plan has a machine sequence, timetable, and capacity, and it is identified by a unique plan number. Vendors are identified by a vendor name. Other information available on a vendor includes its address, phone number, and vendor rating.

At the outset, it is possible to identify VENDOR, PART, and PRODUCTION\_PLAN as base entity types and establish relationships among them, as shown in Figure 3.15a.

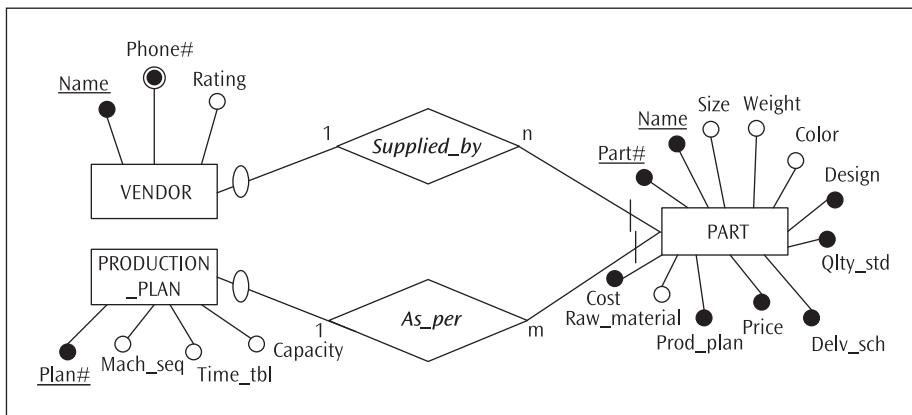
## Chapter 3

The cardinality constraint between VENDOR and PART is 1:n, indicating that a vendor supplies many parts and a part is supplied by only one vendor. Also, the participation constraints (look across) indicate that every vendor must be a supplier of part(s); however, all parts are not supplied by vendors since some are manufactured by Widget USA. A similar relationship exists between PRODUCTION\_PLAN and PART since some but not all parts are manufactured by Widget USA in its own plant. The ERD is syntactically correct.

There are two commonly committed semantic errors seeded in the ERD in Figure 3.15a in order to point out that careful scrutiny of the details present in the requirements specification is crucial to accurate data modeling. First, note that a vendor has telephone numbers (plural) indicating that **Phone#** in the VENDOR entity type should be a multi-valued attribute. Also, every part has a unique part number and a unique name, implying that PART has two unique identifiers. Concatenating **Part#** and **Name** to form a single attribute and labeling that as a single unique identifier of PART is incorrect. These errors are corrected in Figure 3.15b. The fact that an attribute called **Name** occurs in the entity type VENDOR as well as the entity type PART is not an error.



**FIGURE 3.15a** Semantic errors in PART and VENDOR

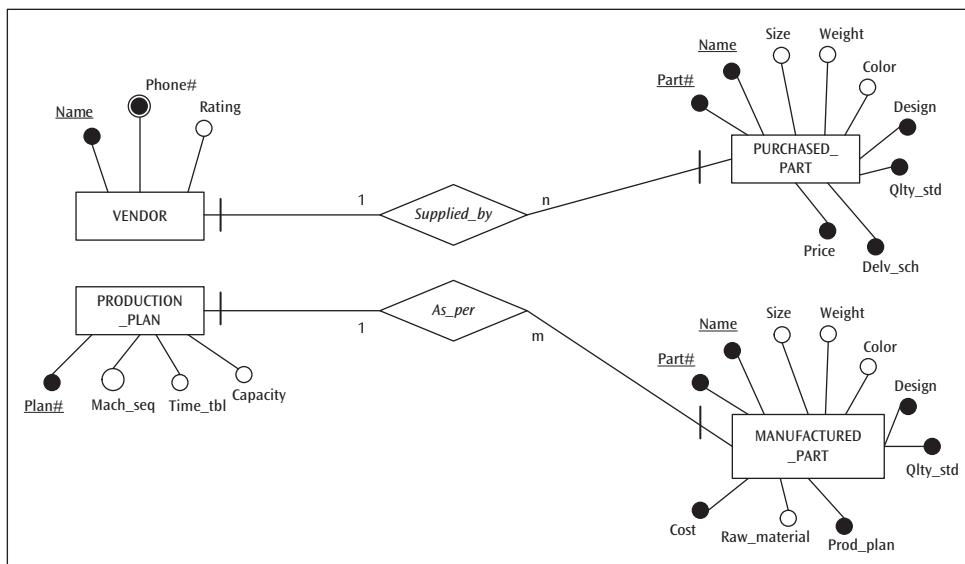


**FIGURE 3.15b** Semantic errors in PART and VENDOR in Figure 3.15a corrected

The conceptual model in Figure 3.15b is still semantically incomplete. What is missing here can be discovered only when the scenario described in the vignette is systematically analyzed for all explicit and implicit business rules conveyed by the story. For instance, a business rule that *a manufactured part is not purchased and vice versa* is implicitly stated in the story. The ERD in Figure 3.15b does not capture this business rule.

Suppose we split PART into two different entity types: PURCHASED\_PART and MANUFACTURED\_PART, as shown in Figure 3.15c. Does this solve the problem? This design certainly offers an opportunity to separate manufactured parts from purchased parts and relate them to production plans and vendors, respectively. However, the data model does not explicitly prohibit a manufactured part from also being a purchased part and vice versa. In other words, inclusion of the same part in both entity sets, even if done inadvertently, will not be considered an error by this design.

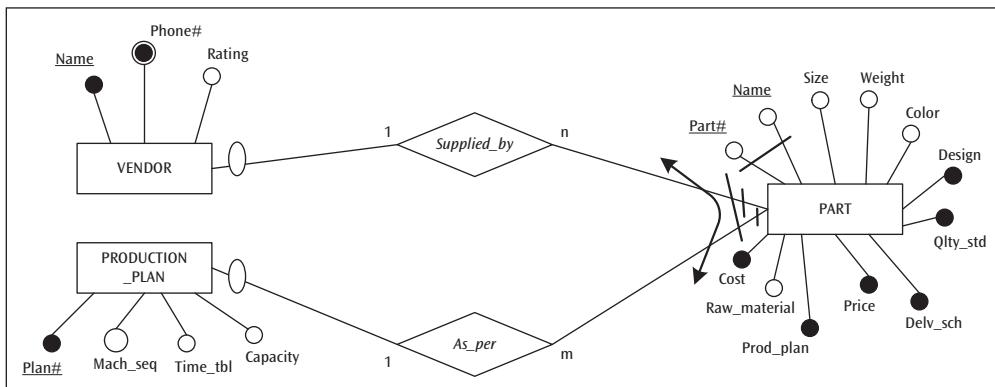
We also notice that a significant number of attributes are duplicated across MANUFACTURED\_PART and PURCHASED\_PART. Does this mean data redundancy? The answer is “No.” As long as the manufactured parts are included in the MANUFACTURED\_PART entity set and purchased parts are present in the PURCHASED\_PART entity set, mere duplication of attribute names does not result in data redundancy. Also, observe that the participation constraint of the entity type MANUFACTURED\_PART in the *As\_per* relationship type and the participation constraint of the entity type PURCHASED\_PART in the *Supplied\_by* relationship type in this design are mandatory—different from the design depicted in Figure 3.15b. All said and done, Figure 3.15c depicts two independent ERDs unconnected to each other; both ERDs being in the same figure do not make them part of a single ERD. This is poor modeling.



**FIGURE 3.15c** Entity type PART in Figure 3.15b split into two entity types

In short, the semantic incompleteness of both designs (Figure 3.15b and 3.15c) with respect to the business rule that *a manufactured part is not purchased and vice versa* is

the same. The ER modeling construct “exclusive arc” is, in fact, capable of capturing this business rule, as can be seen in Figure 3.15d. While the exclusive arc ensures that a part that participates in the relationship type *Supplied\_by* cannot participate in the relationship type *As\_per*, the optional participation of PART in both the relationships may render some parts not participating in both relationships. Once again, the implicit business rule that between purchasing and manufacturing all parts are covered is rather obvious in the vignette narrative. The either/both (hash) constraint can take care of this condition, as shown in Figure 3.15d. In Chapter 4, we will see a more precise and elegant way of specifying both these constraints.

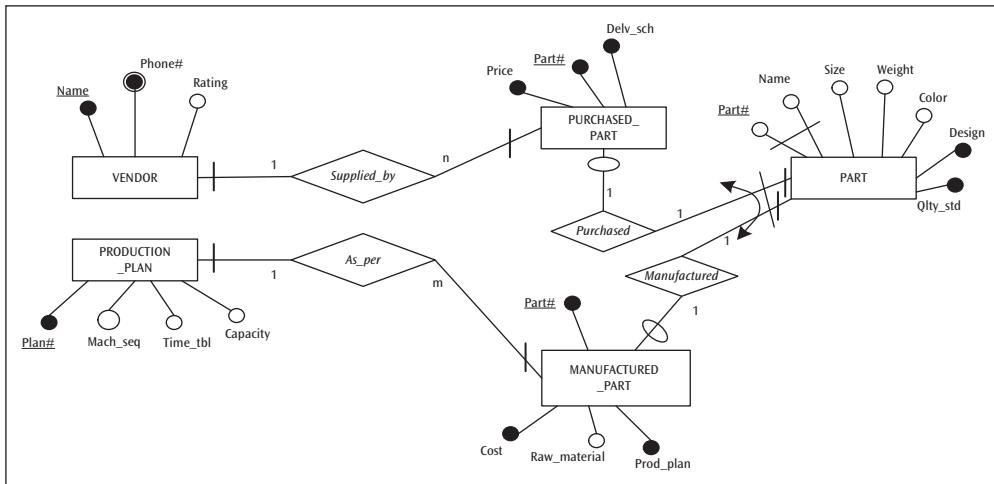


**FIGURE 3.15d** A viable Presentation Layer ER diagram for vignette 2

Let us now explore the following business rule: *It is enough if a value for one of the two attributes, Part# or Name, is present for a given part.* In Figure 3.15b and 3.15c, both **Part#** and **Name** are designated mandatory (dark circle), implying that a value must be present for both of these unique identifiers (underlined attributes) in all entities of the PART entity set. Thus, the design does not reflect the business rule just stated. Designating both unique identifiers as optional (empty circle) in conjunction with the either/both (hash) notation will accomplish the stated objective, as shown in Figure 3.15d. Some data modeling scholars may disagree with this design option because at some point in the design cycle one of the unique identifiers must be designated as the primary means of identifying entities of the entity set and that unique identifier must be a mandatory attribute. However, at the conceptual level of data modeling, there is no reason why all the richness of the scenario cannot be captured.

A further fine-tuning of the ERD is perhaps in order for the following reason. Observe that manufactured parts and purchased parts have several common characteristics (e.g., the attributes **Part#**, **Name**, **Size**, **Weight**, **Color**, **Design**, and **Qty\_std**) as well as attributes that are specific to each of them. Thus, accumulating the individual specific attributes of manufactured parts and purchased parts with the attributes common to both in a single entity type PART entails some inefficiencies pertaining to presence of null values in the specific attributes of manufactured parts and purchased parts; also, additional constraints are necessary to manage different relationships for manufactured parts and purchased parts. This can be resolved by creating separate entity types for MANUFACTURED\_PART,

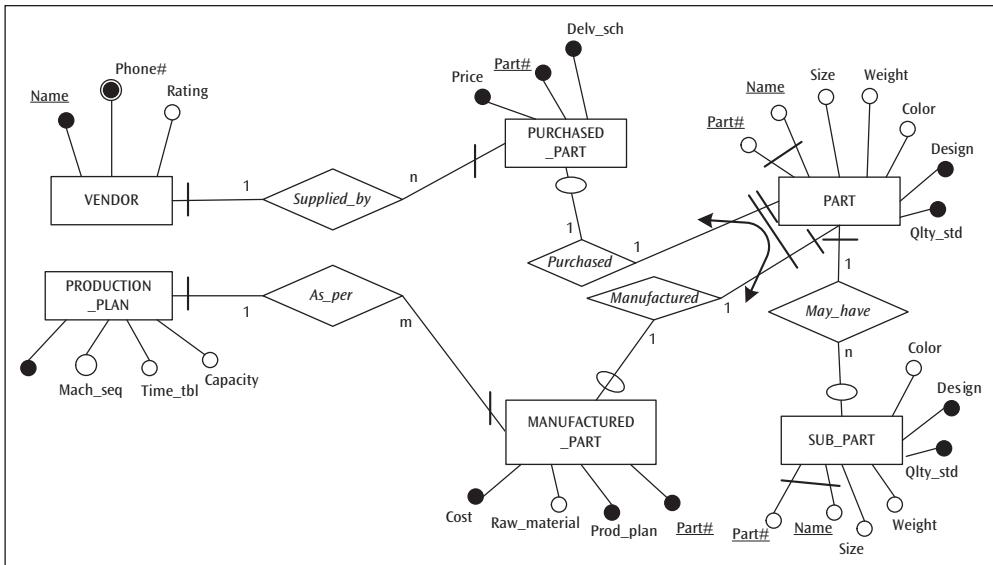
with its specific attributes, and PURCHASED\_PART, with its specific attributes, and independently relating them to PART, which contains the common attributes—essentially, a 1:1 relationship, with partial participation of PART and total participation of MANUFACTURED\_PART and PURCHASED\_PART in the relationship types *Manufactured* and *Purchased*, respectively, as reflected in Figure 3.15e.



**FIGURE 3.15e** A more precise Presentation Layer ER diagram for vignette 2

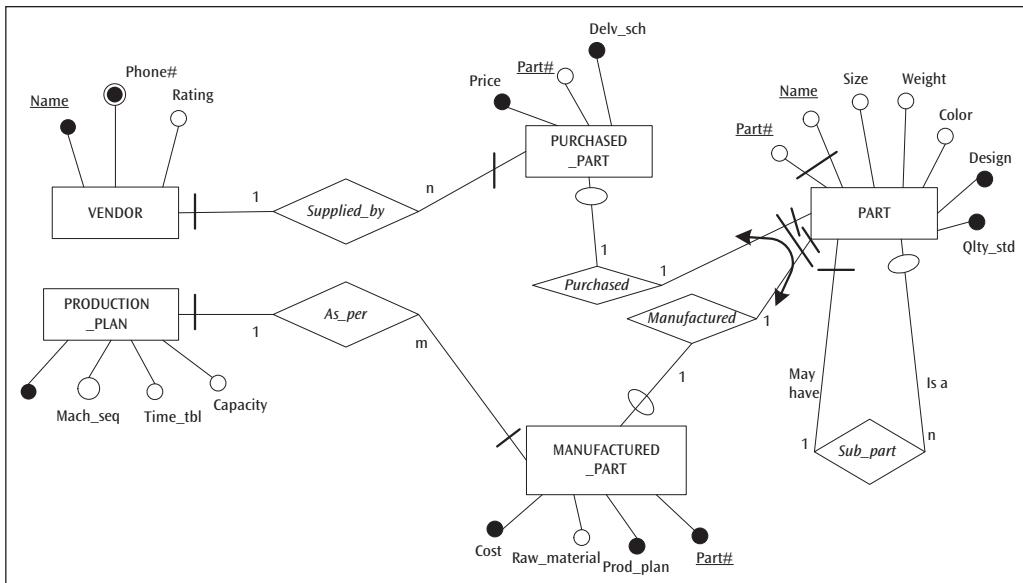
At this point, let us tweak the scenario of this vignette to incorporate the fact that a widget is an intricate assembly of numerous parts. *Some of the parts are actually sub-assemblies, in that they have many subparts. However, a part can be a subpart of only one part—that is, a part can be a subpart in only one subassembly.*

Since we have not been advised to the contrary, the practical assumption to make is that any subpart of a part can be either a manufactured part or a purchased part. Now that the PART entity set includes all the manufactured and purchased parts, a part containing subparts can be modeled by establishing a relationship between PART and SUB\_PART, a mirror image of the entity type PART. The ERD in Figure 3.15f reflecting this relationship is syntactically correct. Is it also semantically correct? The answer is “No.” Since any part can also be a subpart of another part, duplication of several part entities in the parts entity set and the sub-parts entity set is imminent. This is data redundancy and can create semantic problems of data consistency, currency, and correctness in addition to storage inefficiencies during database implementation.

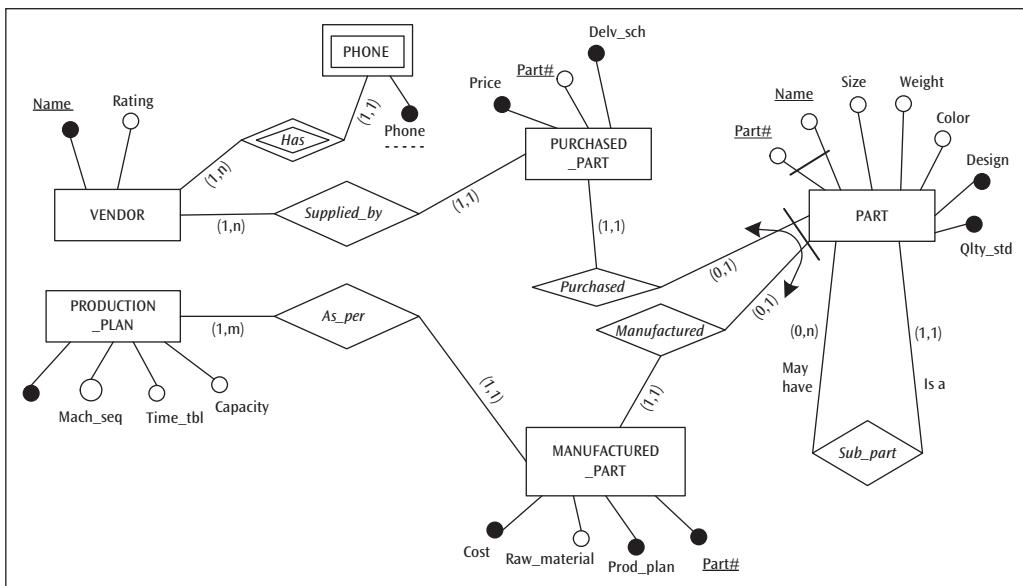


**FIGURE 3.15f** PART-May\_have-SUB\_PART modeled

The correct solution for this scenario is to model *Sub\_part* as a recursive relationship, as shown in Figure 3.15g. The final Design-Specific ER model is presented in the (min, max) notation in Figure 3.15h. The only transformation required here is the decomposition of the multi-valued attribute **Telephone** in the entity type VENDOR to a weak entity type identification-dependent on VENDOR. Observe that neither deletion rules nor the attribute characteristics (data type, size, domain constraints, and any other business rules that cannot be captured in the ERD) are provided in the narrative of the vignette. Accordingly, the final ER model does not have a list of semantic integrity constraints to go with the ERD, and the ERD itself is devoid of deletion rules and attribute characteristics.



**FIGURE 3.15g** The final Presentation Layer ERD with PART-Sub\_part as a recursive relationship



**FIGURE 3.15h** The final Design-Specific ERD for vignette 2 in (min, max) notation

The two vignettes presented in this section used a learning technique of step-by-step development of an ERD with deliberately incomplete stages and seeded errors so that the progressive development can also sensitize the reader to common errors (semantic and syntactic) that occur during the development process of an ER model and the way to correct them.

## Chapter Summary

---

The ER modeling grammar for the conceptual data modeling activity includes an ERD as well as the specification of semantic integrity constraints (SICs) not captured in the ERD. The ER modeling framework presented in this book contains two basic layers: a presentation layer and a design-specific layer.

The case of Bearcat Incorporated, a manufacturing company with several plants located in the northeastern part of the United States, is used to illustrate the ER modeling framework, proceeding from the Presentation Layer ER model to the Design-Specific ER model.

The Presentation Layer ER model is the principal vehicle for communicating with the end-user community. The Presentation Layer ERD contains the initial definition of attributes, entity types, and relationship types using the “look across” specification for the structural constraints of relationships. In addition, it allows for the deletion constraints to be explicitly shown in the ERD. Business rules and data requirements not incorporated into the Presentation Layer ERD are expressed as semantic integrity constraints (domain constraints on an attribute or collection of attributes, deletion constraints, and miscellaneous constraints).

Transformation of the end-user-oriented Presentation Layer ER model to the database-designer-oriented Design-Specific ER model incorporates additional details about the characteristics of attributes obtained from the users. The Design-Specific ERD makes use of the relatively more accurate (min, max) notation to represent the structural constraints. Next, the Design-Specific ER model is further revised by:

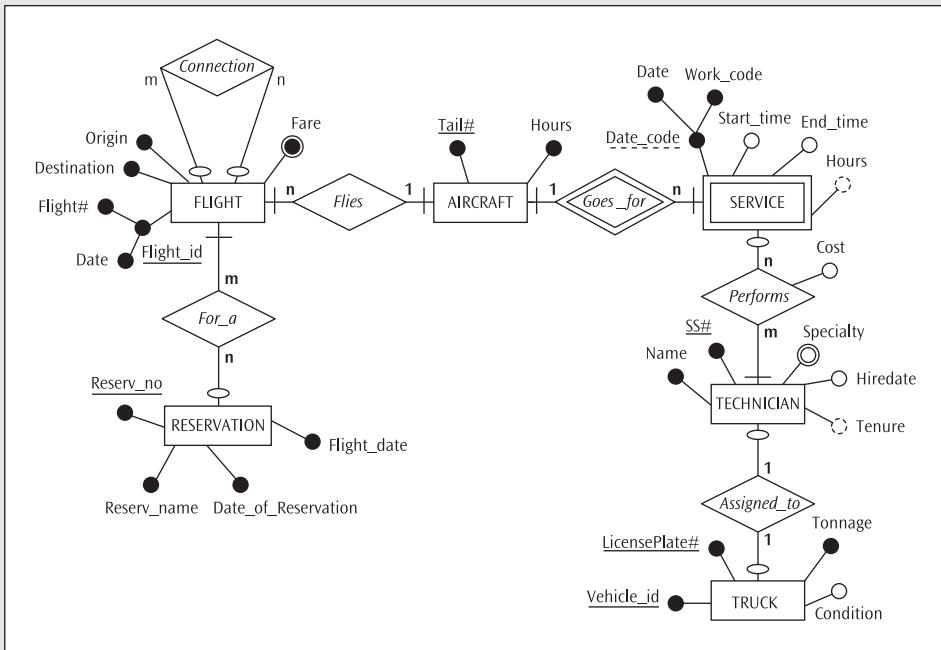
- mapping the attribute characteristics to the ERD and
- decomposing multi-valued attributes and m:n relationship types in order to prepare the conceptual schema for direct mapping to a logical schema

The richness of the original requirements specification is preserved all the way through the conceptual modeling process. Finally, two short vignettes are used to highlight semantic and syntactic errors that usually occur during the development of an ERD and the way to correct them.

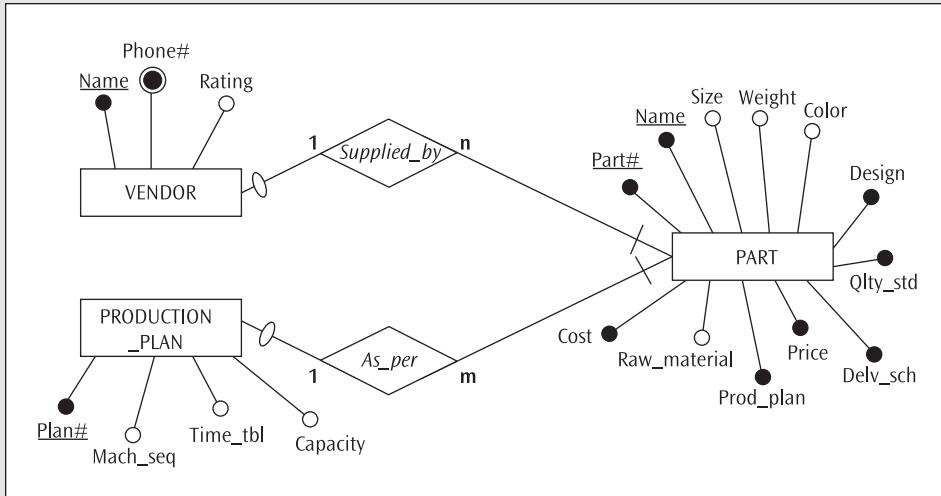
## Exercises

---

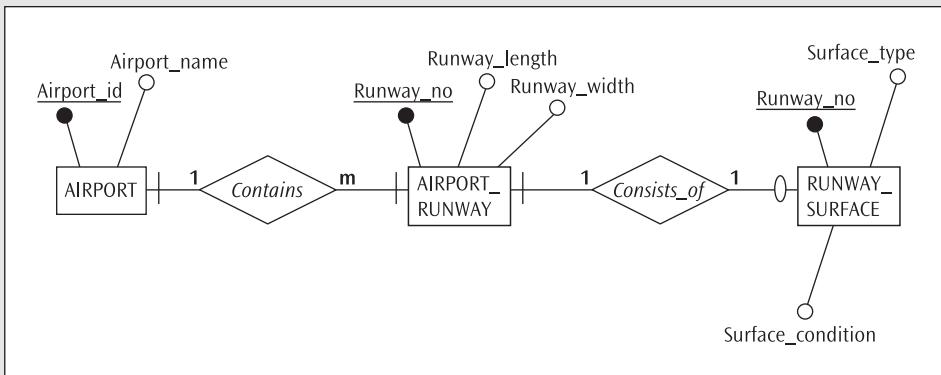
1. What is information preservation and why is it important?
2. Describe what constitutes an ER model.
3. What is the focus of a Presentation Layer ER model?
4. Crow's Foot notation and IDEF1X notation are two other popular notations for the ER modeling grammar. Investigate these two grammars and compare them with the Chen's notation used in this chapter.
5. Examine the CASE tools ERWin and Oracle/Designer and discuss the ER modeling grammar supported by each of them.
6. Give examples of types of business rules that are not reflected in a Presentation Layer ER diagram.
7. Consider the following Presentation Layer ER diagram.



- Identify the base and weak entity types. What is (are) the unique identifier(s) of each entity type? Which unique identifiers are composite attributes?
  - Identify the partial key(s).
  - Identify the optional attributes. Which of these are multi-valued attributes?
  - Identify the derived attributes. What is it about these attributes that allows them to be considered “derived” attributes?
  - Identify the recursive and binary relationship types.
  - Which relationship types exhibit (a) total participation of each entity type, (b) partial participation of each entity type, and (c) total participation of one entity type and partial participation of the second entity type?
  - Describe the nature of each relationship type with (a) a 1:1 cardinality ratio, (b) a 1:n cardinality ratio, and (c) an m:n cardinality ratio.
  - What is the significance of the use of the double diamond in naming the *Goes\_for* relationship type versus the use of a single diamond to name the *Performs* relationship type?
  - Describe an obvious business rule that would be associated with the **Date\_of\_Reservation** attribute in the **RESERVATION** entity type.
  - What does the assignment of the attribute **Cost** to the *Performs* relationship type mean?
8. What is the difference between an exclusive arc and an inclusive arc?
9. What is a deletion constraint?



10. What four deletion rules are applicable to deletion constraints? Which rule(s) refer(s) to an action on the parent and which rule(s) refer(s) to an action on the child?
11. Is the use of the “set null” rule applicable to an identifying relationship type? If yes, explain. If no, definitely explain.
12. What must be done to develop a Design-Specific ER model from a Presentation Layer ER model?
13. When used in the context of data modeling, what is meant by the use of the term “mapping”?
14. What constructs in a Presentation Layer ER diagram cannot be directly mapped to a logical schema? What is required to represent these constructs in a Design-Specific ER diagram?
15. Assume that data are maintained on airports around the country for a company that offers a flight chartering service for college basketball teams. The company gathers information from a wide variety of sources but has had considerable difficulty obtaining data on runway surfaces at airports located in small college towns. Company pilots need access to this information when they land, either at airports such as these or when they must land at small airports that are not located in college towns. As a result, as shown here, they have created a RUNWAY\_SURFACE entity type separate from a RUNWAY entity type.



- a. What does the cardinality ratio and participation constraint suggest about some airport runways in the *Consists\_of* relationship?
  - b. How might the Presentation Layer ER diagram be revised to make the relationship between an airport runway and runway surface more efficient?
16. Transform the Presentation Layer ER diagram for Exercise 7 to a Design-Specific ER diagram.
17. Develop a Presentation Layer Entity-Relationship (ER/EER) model for building a database for the Indian Hill Company described in the following narrative. Indian Hill Company is a factory manufacturing miscellaneous spare parts for the farm equipment industry. The ER diagram should be fully specified, with unique identifier(s) and other attributes for each entity type, and relationship(s) among the entity types. The narrative is complete. However, if you discover any ambiguities in the narrative, make up reasonable assumptions to complete the story and state the assumptions made. Note that no assumption you make can contradict the specifications contained in the narrative. Also, business rule(s) not incorporated in the ERD, if any, must be explicitly stated as semantic integrity constraint(s). A business rule captured in the ERD should not be restated as a Semantic Integrity Constraint and vice versa. You should also list any ambiguities/conflicts in the stated specifications.

Caution: Do not read any extra meanings into the story and make it more complicated than the simple one given below.

The factory has several departments. A department may have many employees but must have at least seven. Every employee works for one and only one department. Every department has a manager—only one manager per department. Clearly, a manager is an employee of the company, but all employees are not managers. For an employee to be the manager of a department, that employee must belong to that particular department. If a department is closed down, all employees of that department are laid off.

A department may have many machines, and every machine is assigned to a specific department. A machine may go for maintenance numerous times. Maintenance is performed on a machine only once on a given day. Some machines are so new that they may not have gone for maintenance yet. Maintenance tasks are outsourced to contractors. Every contractor performs at least one maintenance task, often more. If a contractor quits,

the association of that contractor with its maintenance tasks are temporarily suspended; after all, the maintenance task itself cannot go away! A maintenance task is often done by one contractor but may sometimes involve up to three contractors. When a machine is retired from service, all the associated maintenance records are erased.

Products are produced on machines. A product can be an assembly of several different components (products) or a single piece. Also, a product cannot be a component of more than one product. A product cannot be a component of itself. Every product (component) goes through one or more machines for appropriate production operations. Likewise, several products may go through a particular machine for a production operation. If a product is deleted (due to obsolescence), all operations on that product can be discarded. Operations have precise specifications; so, every operation of a product on a machine is specified by a designer.

Designers design the products and/or specify production operations. Some designers may design more than one product; others may specify more than one operation, and some may do both. Of course, all designers are employees of the factory. Operators, who are also employees of the factory, operate the machines. Due to multiple shifts, several operators will operate the same machine. All operators are routinely assigned to work on only one machine, and no operator is kept idle. A machine is never kept idle except when it is out for maintenance. The same employee cannot be a designer as well as an operator. The factory also has employees other than designers and operators.

The ER model should capture employee's name, which will include first name, last name, and middle initial [o]. It should also capture gender, address, and salary [o]. An employee number uniquely identifies an employee. Likewise, department number [o], department name [o], type, and location [o] must be captured. The department number and department name are both unique identifiers of a department. (Note that it is enough if one of these two is present for any particular department.) Every machine will have a unique machine number. It will also have other attributes, like name of machine, type [o], and vendor's name [o].

When a machine goes for maintenance, the maintenance date for that machine must be captured since a maintenance activity is identified by the date of maintenance for each machine. The attributes of maintenance activity are time taken and cost. A product is identified by its component ID. Component name, description [o] must also be recorded. It should be possible to compute the number of components in a product. When a component goes through machining operation, the starting time [o] and completion time [o] for each product on every machine must be captured, from which the hours of machining operation for a particular product in a specific machine can be computed. The information about designer includes his/her qualifications [o], specialization field, and experience [o] in years. Operators, who are responsible for operating the machines, belong to a labor union [o] and have certain skill sets (at least one skill) associated with them. Contractors are identified by their names. Additional attributes captured for a contractor are experience and expertise.

Note: [o] indicates optional attribute; where specification of deletion rule is missing, the default value of Restrict should be considered first; if found problematic, an alternative may be suggested and used.

Hint: No more than nine entity types are needed to complete this design. It is possible to model all but two business rules in the ER diagram, if the ER modeling grammar is fully

employed; otherwise, a few more business rules may have to be stated in the semantic integrity constraints.

18. The NCAA (National Collegiate Athletic Association) wants to develop a database to keep track of information about college basketball. Each university team belongs to only one conference (the University of Houston belongs to Conference USA; the University of Cincinnati belongs to the Big East Conference, etc.), but a team may not belong to any conference. A conference has several teams; no conference has less than five (5) teams. Each team can have a maximum of 20 players and a minimum of 13 players. Each player can play for only one team. Each team has from three (3) to seven (7) coaches on its coaching staff, and a coach works for only one team. Lots of games are played in each university location every year, but a game between any two universities is played at a given location only one time a year. Three referees from a larger pool of referees are assigned to each game. A referee can work several games; however, some referees may not be assigned to any game. Players are called players because they play in games—in fact, several games. A game involves at least 10 players. It is possible that some players simply sit on the bench and do not play in any game. Player performance statistics (i.e., points scored, rebounds, assists, minutes played, and personal fouls committed) are recorded for each player for every game. Information collected about a game includes the final score, the attendance, and the date of the game. During the summer months, some of the players serve as counselors in summer youth basketball camps. These camps are identified by their unique campsite location (e.g., Mason, Bellaire, Kenwood, League City, etc.). Each camp has at least three (3) players who serve as counselors, and a player serving as a counselor may work in a number of camps.

A player can be identified by student number (i.e., Social Security number) only. The other attributes for a player include name, major, and grade point average. For a coach, relevant attributes include name, title (e.g., head coach, assistant coach), salary, address, and telephone number. Attributes for a referee include name, salary, years of experience, address, telephone number, and certifications. Both coaches and referees are identified by their personal NCAA identification number. A team is identified by the name of the university (i.e., team). Other team attributes include current ranking, capacity of home court, and number of players. Each conference has a unique name, number of teams, and an annual budget. For the basketball camps, data is available on the campsite (i.e., location) and the number of courts.

Develop a Presentation Layer ER model for the NCAA database. The ERD should be fully specified with the unique identifiers, other attributes for each entity type, and the relationship types that exist among the various entity types. All business rules that can be captured in the ERD must be present in the ERD. Any business rule that cannot be captured in the ERD should be specified as part of a list of semantic integrity constraints.

*Hint:* No more than seven entity types are needed to complete this design.

19. This exercise contains additional information in the form of deletion rules that will enable us to develop a Design-Specific ER diagram for the NCAA database in Exercise 18.

When a referee retires, all links to the games handled by that referee should be removed. Likewise, if a game is cancelled, all links to the referees for that game should be dropped. Although it does not happen often, a university may sometimes leave the conference of

which it is a member. Naturally, we want to keep the team in the database since the university could decide to join another conference at a later date. However, if a team (university) leaves the NCAA altogether, all players and coaches of that team should be removed from the database along with the team. In all other relationships that exist in the database, the default value of “Restriction of Deletion” should be explicitly indicated.

- a. Incorporate the above business rules in the Presentation Layer ER diagram for the NCAA database developed in the previous exercise.
- b. Transform the design in (a) to a Design-Specific ER diagram. Note that attribute characteristics are not provided and thus need not appear in the diagram.

# CHAPTER 4

# ENHANCED ENTITY-RELATIONSHIP (EER) MODELING

Enhanced entity-relationship (EER) modeling is an extension to ER modeling that incorporates additional constructs to enhance the capability of the ER modeling grammar for conceptual data modeling. Sometimes, the EER model is referred to as the extended entity-relationship model. The EER model's additional semantic modeling constructs were developed in response to the demands of more complex database applications in the 1980s and the inadequacy of the ER model to meet these demands. Some of these constructs/concepts were also independently developed in related areas, such as software engineering (object modeling) and artificial intelligence (knowledge representation).

This chapter introduces the Superclass/subclass (SC/sc) relationship as the basis for the EER modeling specialization/generalization and categorization constructs. Section 4.1.1 presents a scenario that exposes the inadequacy in the expressive power of ER modeling constructs. Section 4.1.2 introduces the concept of the intra-entity class relationship and the SC/sc relationship as the building block to model such intra-entity class relationships. Sections 4.1.3 through 4.1.8 present descriptions of EER constructs that extend the power of the ER model. Section 4.1.3 elaborates on the general properties of an SC/sc relationship, while Section 4.1.4 engages in an in-depth coverage of specialization/generalization. Section 4.1.5 pertains to the development of the specialization/generalization construct in a hierarchy and network of relationships. The categorization construct is discussed in Section 4.1.6. Specialization/generalization and categorization essentially serve two different modeling needs. Section 4.1.7 gives some guidelines on how to determine which construct to use in a business database design scenario. Although aggregation is often discussed as a construct in the object-oriented paradigm, a presentation of this construct appears in Section 4.1.8 because aggregation adds to the expressive power of the EER modeling grammar.

Section 4.2 describes the process of converting the Presentation Layer EER diagram to the Design-Specific EER diagram as a preliminary step for mapping the conceptual schema to a logical schema. Section 4.3 begins with an extension to the Bearcat Incorporated story that incorporates business rules requiring the use of EER modeling constructs. Next, Section 4.4 takes the reader through the conceptual data modeling

layers for Bearcat Incorporated. Finally, Section 4.5 specifies deletion rules and incorporation of the corresponding deletion constraints for intra-entity class relationships.

## 4.1 SUPERCLASS/SUBCLASS RELATIONSHIP

---

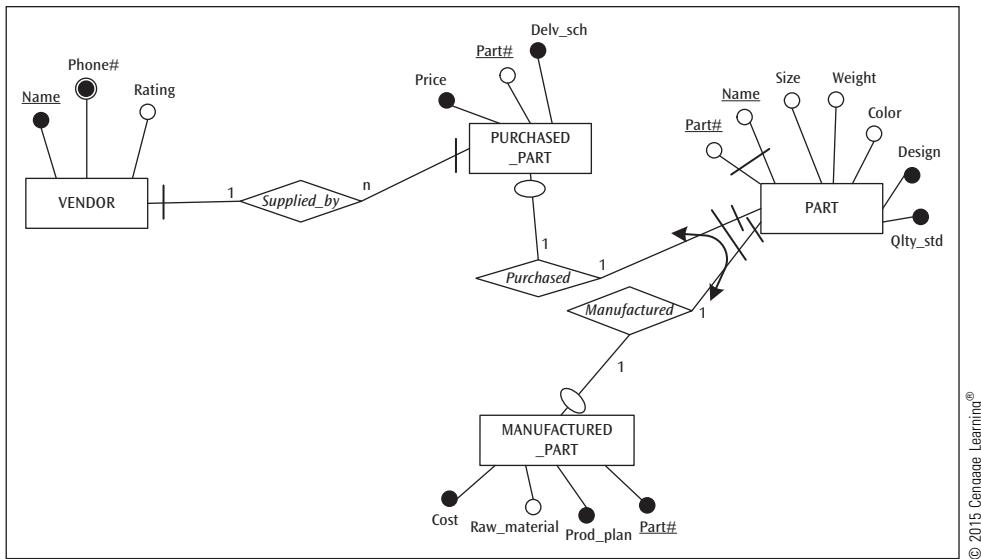
The scenario presented below in Section 4.1.1 followed by vignette 1 set the stage for introducing the building block of enhanced ER modeling constructs—namely, the Superclass/subclass relationship—as an extension to the ER modeling grammar.

### 4.1.1 A Motivating Exemplar

Let us use Figure 4.1, a subset of the ERD developed for vignette 2 in Chapter 3 (Section 3.3.2) to set the stage for introducing the intra-entity class relationship constructs—an enhancement to the ER modeling grammar. Incidentally, the relationships depicted in an ER diagram are intended to capture inter-entity class relationships. Since these modeling grammar enhancements are very much relevant to the Presentation Layer ER model, the “look across” notation is used in the ER diagram.

The cardinality constraint of both the relationships, *Manufactured* and *Purchased*, is 1:1, indicating that one part can be related to only one manufactured\_part and only one purchased\_part and vice versa. The exclusive arc ensures that the same part is not related to a manufactured part as well as a purchased part. The ERD also informs the total participation of MANUFACTURED\_PART in the *Manufactured* relationship type and PURCHASED\_PART in the *Purchased* relationship type. Observe, though, that the participation of PART in the relationship type *Manufactured* is partial, meaning that a part need not be related to a manufactured part; likewise, a part need not be related to a purchased part (partial participation of PART in the *Purchased* relationship type). However, this can create an inadvertent condition of some parts not participating in either relationship, meaning that there are some parts that are neither manufactured nor purchased. The “**Either/both**” (**Hash**) construct makes sure that this possibility is eliminated. Thus, the ER diagram is essentially robust. Suppose we want to allow any part to be manufactured as well as purchased; this can be accomplished by removing the exclusive arc from the ERD.

Figure 4.1 specifies another inter-entity class relationship labeled *Supplied\_by* between the PURCHASED\_PART and VENDOR entity types. What is the difference between this relationship and the other two relationships—that is, *Purchased* and *Manufactured*? First, the entity types participating in the *Supplied\_by* relationship—that is, VENDOR and PURCHASED\_PART—belong to two different entity classes. However, the entity types PURCHASED\_PART and PART, which participate in the relationship *Purchased* semantically belong to the same entity class, and the same is true regarding MANUFACTURED\_PART and PART in the *Manufactured* relationship. While the expression of these two relationship types in Figure 4.1 is syntactically correct, the semantics conveyed by these relationships is questionable. For instance, the model indicates that a part is related to at most one manufactured part or one purchased part, as if a part is different from a manufactured part or a purchased part. The idea that purchased parts and manufactured parts are indeed parts to begin with and so possess all the attributes of PART is not explicitly specified in the inter-entity class relationships shown in the ERD. In short, the inter-entity class relationship types *Purchased* and *Manufactured* are at best an awkward expression of the intended semantics.



© 2015 Cengage Learning®

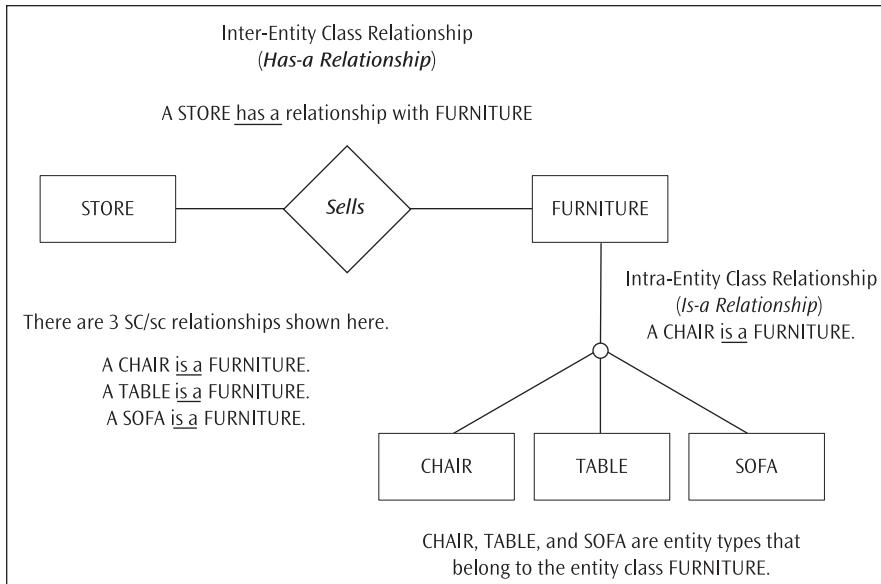
**FIGURE 4.1** Relationship of PART with PURCHASED\_PART and MANUFACTURED\_PART

The following sections of this chapter discuss enhancements to the ER modeling grammar by way of additional constructs to express sharper relationships among entity types, especially the ones that belong to the same entity class. The building block for these modeling constructs is the fundamental construct called the Superclass/subclass relationship.

#### 4.1.2 Introduction to the Intra-Entity Class Relationship Type

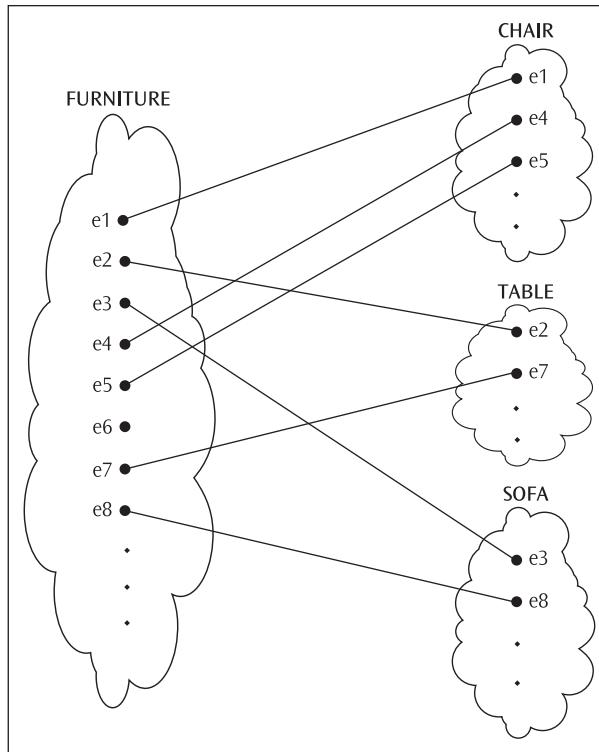
Section 2.2 introduced the entity class as a construct that conceptualizes an object class. For example, stores selling furniture can be modeled as a relationship between a STORE entity type and a FURNITURE entity type. Here, STORE and FURNITURE represent entity types of different generic entity classes because they represent different independent object types and the relationship between the two is a binary relationship that represents an **inter-entity class relationship**. Because STORE *has* a relationship with FURNITURE, an inter-entity class relationship is often called a *Has-a relationship*.

Now, consider the fact that FURNITURE can be chairs, tables, desks, sofas, beds, and so on. Although FURNITURE and STORE do not share any common properties (attributes), chairs, tables, and sofas are all furniture and possess some common properties. In this case, CHAIR, TABLE, and SOFA are referred to as entity types that belong to the generic entity class FURNITURE, and a relationship between CHAIR and FURNITURE (or TABLE and FURNITURE, or SOFA and FURNITURE) is called an **intra-entity class relationship** (see Figure 4.2). Because CHAIR (or TABLE or SOFA) is FURNITURE, this relationship is also referred to as an *Is-a relationship* (see Figure 4.2).



**FIGURE 4.2** Inter-entity and intra-entity class relationships

Because FURNITURE represents the generic class of entity that includes one or more entity type occurrences (CHAIR, TABLE, SOFA), FURNITURE is labeled a **Superclass (SC)** entity type. Because CHAIR (or TABLE or SOFA) represents an entity type that is a subgroup of FURNITURE, it is labeled a **subclass (sc)** entity type. The relationship between a superclass and any one of the subclasses is called an **SC/sc relationship**. There are three SC/sc relationships present in the intra-entity class relationship shown in Figure 4.2. Note that FURNITURE and CHAIR (or TABLE or SOFA) are separate entity types, although they are not independent, like FURNITURE and STORE are. Figure 4.3 shows that entities belonging to the CHAIR, TABLE, and SOFA subclasses also belong to the FURNITURE superclass. Note that an entity that belongs to a subclass represents the same entity that is connected to it from the FURNITURE superclass.



**FIGURE 4.3** Superclass/subclass entity instances

It is not necessary that CHAIR, TABLE, and SOFA be abstracted to an SC called FURNITURE. There may be situations in which direct relationships between CHAIR and STORE, TABLE and STORE, and SOFA and STORE are modeled. Likewise, instead of creating subclasses, the generic entity type FURNITURE may incorporate all the attributes of CHAIR, TABLE, and SOFA in a single entity type. These modeling variations are context dependent.

#### 4.1.3 General Properties of a Superclass/subclass Relationship

There are two basic kinds of SC/sc relationships: specialization/generalization and categorization. Specialization/generalization is discussed in Sections 4.1.4 and 4.1.5, and categorization is discussed in Section 4.1.6.

The general properties of an SC/sc relationship are as follows:

- One superclass (SC) is related to one or more subclasses (sc) (specialization/generalization), or one subclass (sc) is related to one or more superclasses (SC) (categorization).
- An entity<sup>1</sup> that exists in a subclass can be associated with only one superclass entity.

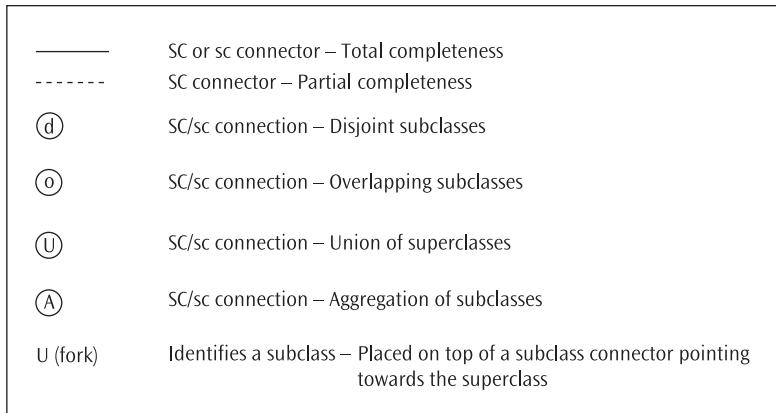
---

<sup>1</sup>Note that in Section 2.2, the term “entity” is defined as an instance (occurrence) of an entity type.

- An entity cannot exist in the database merely by being a member of a subclass; it must also be a member of an associated superclass.
- An entity that is a member of a superclass can be optionally included as a member of any number of its subclasses.
- It is not required that every entity (member) of a superclass be a member of a subclass.
- A subclass inherits all the attributes of the superclass to which it is related. In addition, it inherits all the relationship types in which the superclass participates. This is known as the **type inheritance property**.
- A subclass may possess its own *specific attributes*<sup>2</sup> in addition to the attributes inherited from the superclass to which it is related. Likewise, a subclass may have its own specific relationship(s) with (an)other entity type(s) (i.e., it may have its own inter-entity class relationships).

In Section 2.3.4, we asserted that a relationship type is not fully specified until both structural constraints—that is, cardinality ratio and participation constraints—are specified. While that rule applies to both of the EER constructs mentioned earlier, the cardinality ratio of any SC/sc relationship in both is always 1:1. Likewise, the participation of a subclass in an SC/sc relationship is always total.

In order to represent the EER constructs diagrammatically, the ER diagramming notation set is expanded with a few additional symbols, shown in Figure 4.4. Use of these symbols is discussed in the remainder of Section 4.1.



**FIGURE 4.4** EER diagram notation

#### 4.1.4 Specialization and Generalization

Generating subgroups (“sc’s) of a generic entity class (SC) by specifying the distinguishing properties (attributes) of the subgroups is called **specialization**. **Generalization**, on the

---

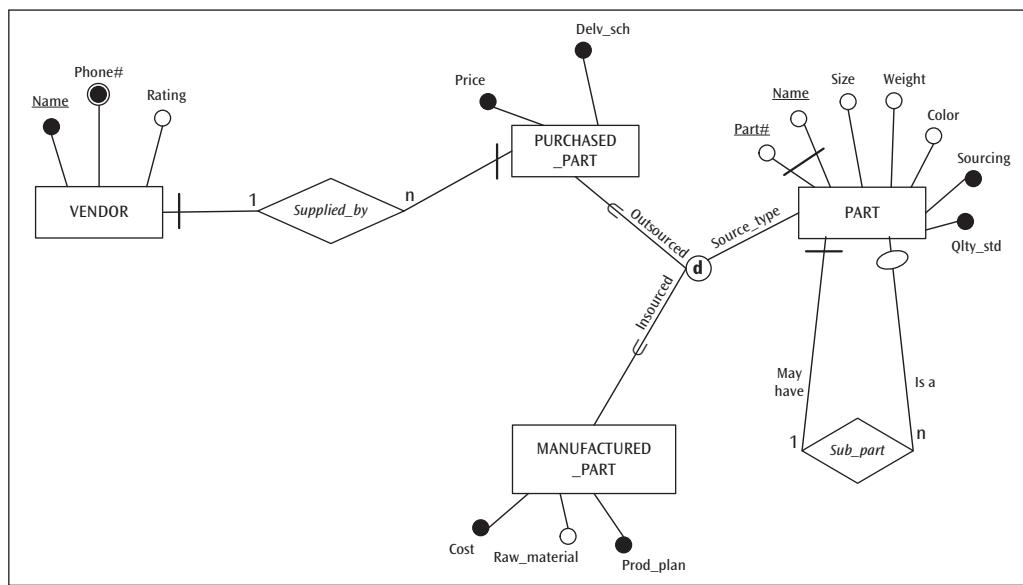
<sup>2</sup>These attributes need not be unique to the subclass.

other hand, pulls together the common properties (attributes) shared by a set of entity types ("sc"s) into a generic entity type (SC). In other words, generalization is the reverse process of specialization in which the differences among a set of entity types are suppressed and the common features are "generalized" into a single superclass of which the original source entity types become subclasses. Put another way, specialization is a top-down approach for describing an SC/sc relationship, whereas generalization is a bottom-up approach for describing the same relationship. Specialization and generalization can be thought of as two sides of the same coin; therefore, all discussions about specialization apply equally well to generalization and vice versa.

#### 4.1.4.1 Motivating Exemplar Revisited

At this point, let us revisit the motivating exemplar introduced in Section 4.1.1 and see if the design reflected in Figure 4.1 can be better expressed using the EER construct, specialization.

Because a purchased part *is a* part (the same is true for a manufactured part), the design shown in Figure 4.1 can indeed be modeled as a specialization. Figure 4.5 is a diagrammatic depiction of this scenario and takes the form of a specialization. The subclasses that participate in the specialization, PURCHASED\_PART and MANUFACTURED\_PART, are represented as base entity types (as is the case in Figure 4.5) and are collectively connected to the parent superclass PART via a circle symbol that signifies the relationship of specialization. A fork (U) symbol placed on each subclass connector (the line connecting the subclass to the circular specialization symbol) opens toward the superclass. A solid line flowing from a subclass to the back of the fork denotes total participation of that subclass in that particular specialization. The cardinality ratio (1:1) is inherent to an SC/sc relationship and so is not overtly indicated in the diagram.



**FIGURE 4.5** Modeling PART as intra-entity class relationships

The participation of the superclass in the specialization is referred to as the **completeness constraint** and can assume one of two values: total or partial. **Total specialization** means that every entity of the superclass must participate in this specialization relationship. This is indicated in the diagram by drawing a solid line from the superclass to the circular specialization symbol. A dotted line, on the other hand, connotes **partial specialization**, meaning that there may be entities present in the superclass that do not participate in this specialization. Each subclass may have its own specific attributes that the superclass does not have. For example, in Figure 4.5, PURCHASED\_PART has the attributes **Price** and **Delv\_sch**, and MANUFACTURED\_PART has the attributes **Cost**, **Raw\_material**, and **Prod\_plan**. That each subclass inherits all the attributes of the superclass and all the relationship types in which the superclass participates is implicit in this specialization construct. Accordingly, both MANUFACTURED\_PART and PURCHASED\_PART inherit attributes **Part#**, **Name**, **Size**, **Weight**, **Color**, **Sourcing**, and **Qty\_std** from the superclass entity type PART. Furthermore, both subclasses in Figure 4.5 implicitly inherit the recursive relationship type *Sub\_part* of PART.

In the example shown in Figure 4.5, the solid line for the **superclass connector** indicates that every part is either a purchased part or a manufactured part. Had this been a dotted line, it would have conveyed that there are some parts that are neither manufactured nor purchased.

Another constraint pertaining to SC/sc participation in a specialization is called the **disjointness constraint**, which is used to specify that the subclasses of a specialization must bear the value “disjoint”; it is conveyed by placing the letter “d” in the circular specialization symbol. This means that an entity of the superclass cannot be a member of more than one subclass of the specialization. If, however, the subclasses in a specialization are not constrained to be “disjoint,” their sets of entities may *overlap* across the subclasses in this specialization. This is indicated by placing the letter “o” in the circular specialization symbol. In Figure 4.5, the value “d” for the disjointness constraint says that a purchased part cannot also be a manufactured part and vice versa. Had this value been an “o,” it would have meant that a part can be manufactured as well as purchased; such conditions may arise from time to time when a firm has insufficient production capacity. It is important to note that the completeness constraint and the disjointness constraint are two independent yet complementary constraints and that a specialization construct is not fully specified until *both* of these constraints are specified.

Frequently, the membership of an entity in a subclass can be exactly determined based on a specified condition reflected by the value of some attribute(s) in the superclass of the specialization. In this case, the subclasses in the specialization are called **predicate-defined** (or **condition-defined**) subclasses. The condition itself is called the **defining predicate** or **defining condition** and is shown beside the superclass connector, while the values of the defining predicate are coded next to the subclass connectors. In Figure 4.5, *Source\_type* is the defining predicate<sup>3</sup> and can have the values “Outsourced” or “Insourced.” This models the semantics that all part entities in the superclass PART marked by the defining predicate “Insourced” will also be members of the subclass MANUFACTURED\_PART;

---

<sup>3</sup>It should be noted that a defining predicate is not an attribute; it is a condition based on the value(s) of one or more attributes in the superclass.

likewise, the predicate “outsourced” defines the membership of PURCHASED\_PART in the specialization.

Often, attribute(s) in the superclass are used to define the predicate. If all values of the predicate are determined by the same attribute in the superclass, this attribute is referred to as the **defining attribute**<sup>4</sup> and the specialization itself is labeled **attribute-defined specialization**. The mandatory attribute **Sourcing** in PART plays this role in Figure 4.5. There are times when the membership of a superclass entity in the subclass(es) of its particular specialization cannot be exactly determined by a specified condition that can be evaluated automatically. Often in these cases, the membership of the entity is determined by the end-user individually for each entity, based on some not necessarily well-defined procedure. The specialization then is called a **user-defined (or procedure-defined) specialization**.

Had the process of generalization been used to develop Figure 4.5, the data modeler would have begun by observing a number of common attributes<sup>5</sup> (**Part#, Name, Size, Weight, Color, Sourcing, and Qty\_std**) in the process of defining each of the individual entity types, PURCHASED\_PART and MANUFACTURED\_PART. These common attributes suggest that each of the two entity types really belongs to a more general entity type. Thus, the superclass PART emerges. The result enables the attributes common to the two subclass entity types to be highlighted in the superclass while at the same time preserving the attributes that are specific to each of the subclasses.

#### 4.1.4.2 Vignette 1

*This vignette is about Division I collegiate sports programs in which universities provide academic support in the form of academic advising, tutoring, career counseling, and so on to student-athletes who participate in the sports programs sponsored by the university. Note that a university provides this support to numerous student-athletes, whereas each student-athlete receives the support from only the university that he or she attends. Every student-athlete participates in one or more sponsored sports. Football, basketball, and baseball are among the many sponsored sports. Attributes common to all student-athletes, regardless of sport, include student# (a unique identifier), name, major, grade point average, eligibility, sport (a multi-valued attribute), weight, and height. For student-athletes participating in the sports of football, basketball, and baseball, attributes specific to each sport are also collected. These include:*

- For football players: touchdowns, position, speed, and uniform#
- For basketball players: uniform# (a unique identifier), position (a multi-valued attribute), points scored per game, assists per game, and rebounds per game
- For baseball players: uniform# (a unique identifier), position (a multi-valued attribute), batting average, home runs, and errors

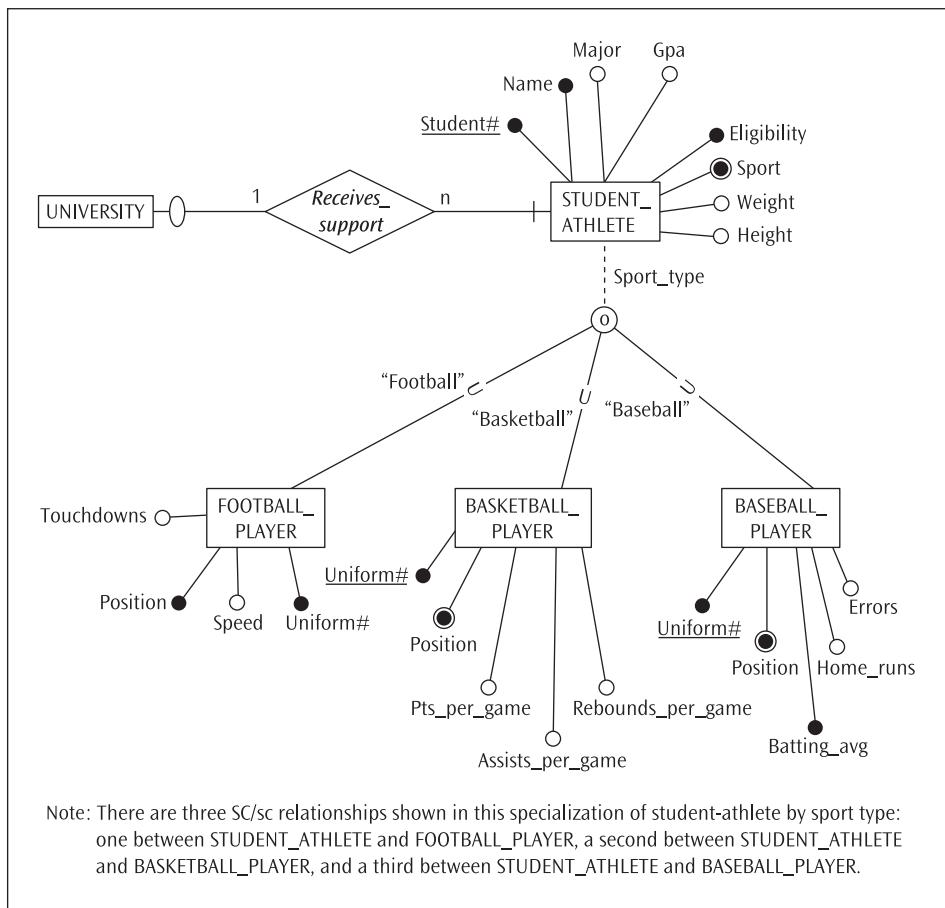
---

<sup>4</sup>Sometimes, the term “subclass discriminator” is used instead.

<sup>5</sup>Commonality implies that *all* characteristics of the individual attributes across the entity types are the same.

For other sports, such sport-specific attributes are not available. Also, note that uniform# is not a unique identifier of football players because it is possible for two football players to have the same uniform number in squads with more than 100 players (one may be an offensive player and the other a defensive player).

Since football players, basketball players, and baseball players are all student-athletes, the relationships can indeed be modeled as a specialization. Figure 4.6 depicts this scenario using the specialization construct of the enhanced ER (EER) modeling grammar. The subclasses that participate in the specialization, FOOTBALL\_PLAYER, BASKETBALL\_PLAYER, and BASEBALL\_PLAYER, are represented as base entity types and are collectively connected to the parent superclass, STUDENT\_ATHLETE, using the specialization notation.



**FIGURE 4.6** Modeling STUDENT\_ATHLETE using intra-entity class relationships

As we know now, each subclass may have its own specific attributes that the superclass does not. For example, in Figure 4.6, FOOTBALL\_PLAYER has the attributes **Speed**, **Position**, **Touchdowns**, and **Uniform#**; BASKETBALL\_PLAYER has the attributes

**Uniform#** (unique identifier), **Pts\_per\_game**, **Position** (multi-valued), **Assists\_per\_game**, and **Rebounds\_per\_game**; and **BASEBALL\_PLAYER** has the attributes **Position** (multi-valued), **Uniform#** (unique identifier), **Batting\_avg**, **Home\_runs**, and **Errors**. Notice that **Uniform#** is not a unique identifier in **FOOTBALL\_PLAYER** because, in squads that have more than 100 players, the same uniform number will be assigned to more than one player. That each subclass inherits all the attributes of the superclass and all the relationship types in which the superclass participates is implicit in the specialization construct. Using the sample data sets in Table 4.1, each subclass (**FOOTBALL\_PLAYER**, **BASKETBALL\_PLAYER**, **BASEBALL\_PLAYER**) inherits the attributes **Student#**, **Name**, **Major**, **Gpa**, **Eligibility**, **Sport** (multi-valued), **Height**, and **Weight** from the **STUDENT\_ATHLETE** superclass. Furthermore, each of the three subclasses in Figure 4.6 implicitly inherits the inter-entity class relationship type *Receives\_support* with the entity type **UNIVERSITY**.

STUDENT_ATHLETE Data Set							
Student#	Name	Major	Gpa	Eligibility	Sport	Weight	
345212	Routt	Sociology	3.25	2	Football Track	185	
672333	Evans	Communications	2.68	2	Football	205	
502123	Gettys	Business	3.65	0	Basketball Baseball	215	
230543	Francis	Psychology	2.58	4	Basketball	200	
902341	Pakilana	Communications	3.72	0	Swimming	115	
324543	Oliver	Communications	3.12	1	Basketball Soccer	125	
FOOTBALL_PLAYER Data Set							
Student#	Uniform#	Touchdowns	Speed	Position			
345212	6	0	4.23	Cornerback			
672333	6	12	4.39	Halfback			
BASKETBALL_PLAYER Data Set							
Student#	Uniform#	Position	Pts/Game	Ast/Game	Reb/Game		
502123	10	Pt Guard	12	9	6		
230543	1	Sh Guard Sm Forward	7	2	5		
324543	44	Sh Guard	8	4	6		
BASEBALL_PLAYER Data Set							
Student#	Uniform#	Position	Average	HomeRuns	Errors		
502123	10	First Base Outfield	.320	6	0		

Note: The shaded Student# in the FOOTBALL\_PLAYER, BASKETBALL\_PLAYER, and BASEBALL\_PLAYER data sets is used to illustrate which student-athletes participate in the sports of football, basketball, and baseball. Observe that Routt and Oliver participate in these sports plus other sports as well. Pakilana, on the other hand, is a student-athlete who participates in a sport other than football, basketball, and baseball and thus data for Pakilana appears only in the STUDENT\_ATHLETE data set.

**TABLE 4.1** Sample data sets for Figure 4.6

The dotted line for the **superclass connector** in Figure 4.6 indicates that there are student-athletes who are neither football players, nor basketball players, nor baseball players; they participate in other sports. The reason these other sports are not depicted as

subclasses is that they do not have any specific attributes beyond what they inherit from the superclass. Had this been a solid line, it would have conveyed that there are no student-athletes playing anything other than football, basketball, or baseball. Notice in Figure 4.6 that the value “o” for the disjointness constraint conveys that a football player can also be a basketball player and/or baseball player. Had this value been a “d,” it would have prohibited a student-athlete from participating in more than one of these three sports.

A closer inspection of Figure 4.6 may suggest that the attributes **Uniform#** and **Position** should be included among the attributes of the STUDENT\_ATHLETE entity type instead of each of the individual subclasses. However, there are a variety of factors that prohibit adding these attributes to STUDENT\_ATHLETE. First, the intent is to show **Uniform#** as a unique identifier in two of the subclasses. But **Uniform#** cannot be shown as a unique identifier of STUDENT\_ATHLETE because two student-athletes in baseball and basketball respectively may have the same uniform number. Then the requirement that **Uniform#** is a unique identifier in BASKETBALL\_PLAYER and BASEBALL\_PLAYER would not be preserved. In addition, designation of **Uniform#** as a unique identifier in STUDENT\_ATHLETE will propagate the same property to FOOTBALL\_PLAYER, which would also be incorrect. The multi-valued attribute **Position** poses a similar problem because a student-athlete participating in multiple sports may play the same position in more than one sport—a center in football could also be a center in basketball.

#### 4.1.4.3 Vignette 1 Extended

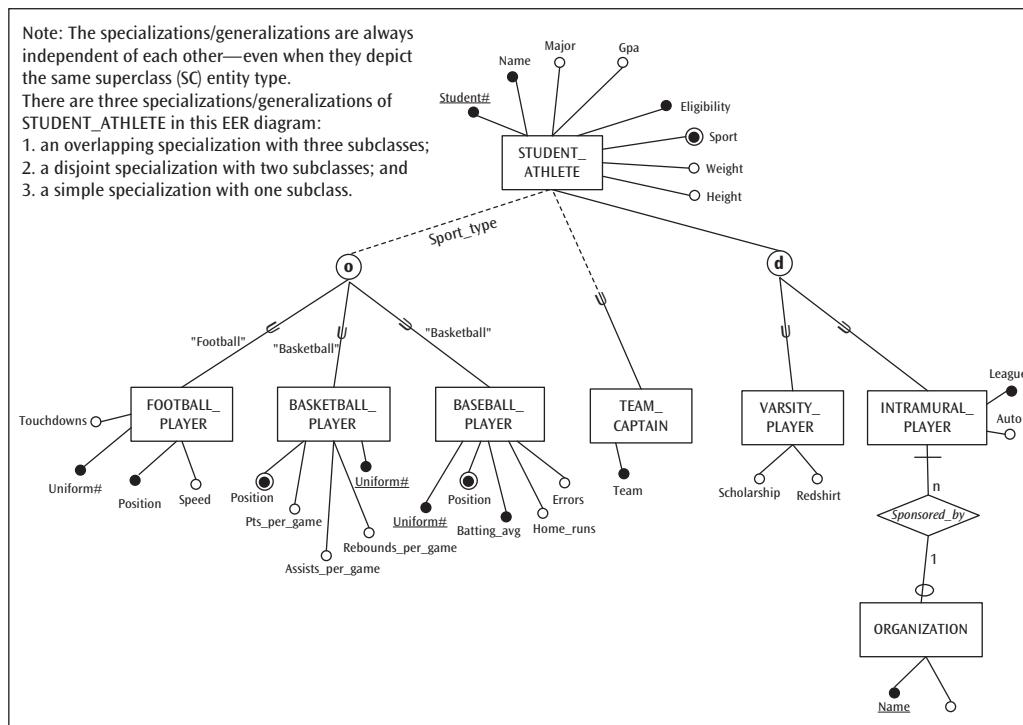
Vignette 1 can be extended to demonstrate two other properties of a specialization/generalization:

- multiple specialization of the same entity type (SC)
- specific relationship of a subclass entity type

First, it is also possible for an entity type to participate as a superclass in multiple independent specializations. For example, suppose a student-athlete, in addition to participating in various sports programs, may also serve as a captain of a team and participate as either a varsity-level player or intramural-level player. In addition, let us say that an intramural-level player may be sponsored by an organization (e.g., fraternity, sorority, or club), as is demonstrated in Figure 4.7. That is, STUDENT\_ATHLETE specialized as the subclass set {FOOTBALL\_PLAYER, BASKETBALL\_PLAYER, BASEBALL\_PLAYER} can also be independently specialized as TEAM\_CAPTAIN and the subclass set {VARSITY\_PLAYER, INTRAMURAL\_PLAYER}. In essence, STUDENT\_ATHLETE is participating as a superclass in three independent specializations—the first one with three subclasses, the second with just one subclass, and the third with two subclasses. Note that since TEAM\_CAPTAIN is the only subclass in that specialization, the disjoint property is irrelevant. Since not all student-athletes are team captains, the dotted line emanating from STUDENT\_ATHLETE in this specialization indicates partial specialization (i.e., the completeness constraint in this case is “partial”). Likewise, the {VARSITY\_PLAYER, INTRAMURAL\_PLAYER} specialization of STUDENT\_ATHLETE is a disjoint specialization, as indicated by the “d” in the circular specification circle, meaning that a student-athlete can be either a varsity player or an intramural player but not both. In addition, the solid line emanating from STUDENT\_ATHLETE in this specialization denotes total

specialization (i.e., the completeness constraint is “total”), meaning that every student-athlete *must* be either a varsity player or intramural player.

Also, in Figure 4.7, the inter-entity class relationship depicted by *Sponsored\_by* between INTRAMURAL\_PLAYER and ORGANIZATION illustrates that an entity type participating in a specialization as a subclass may also have specific (external) relationship(s) beyond the specialization. Sample data representing this extension of vignette 1 appears in Table 4.2. Observe that in both Figure 4.7 and Table 4.2, the subclasses INTRAMURAL\_PLAYER and VARSITY\_PLAYER also have their own specific attributes.



**FIGURE 4.7** Multiple specializations of the same superclass (SC) STUDENT\_ATHLETE

STUDENT_ATHLETE Data Set							
Student#	Name	Major	Gpa	Eligibility	Sport	Weight	Height
345212	Routt	Sociology	3.25	2	Football Track	185	72
672333	Evans	Communications	2.68	2	Football	205	70
502123	Gettys	Business	3.65	0	Basketball	215	78
230543	Francis	Psychology	2.58	4	Basketball	200	79
902341	Pakilana	Communications	3.72	0	Swimming	115	65
324543	Oliver	Communications	3.12	1	Basketball	125	70
454212	Newman			2	Soccer		
					Baseball		

FOOTBALL_PLAYER Data Set				
Student#	Uniform#	Touchdowns	Speed	Position
345212	6	0	4.23	Cornerback
672333	6	12	4.39	Halfback

BASKETBALL_PLAYER Data Set					
Student#	Uniform#	Position	Pts/Game	Ast/Game	Reb/Game
502123	10	Pt Guard	12	9	6
230543	1	Sh Guard	7	2	5
324543	44	Sm Forward			
		Sh Guard	8	4	6

BASEBALL_PLAYER Data Set		
Student#	Uniform#	Position
502123	10	First Base
		Outfield
454212	19	Pitcher

VARSITY_PLAYER Data Set		
Student#	Scholarsp	Redshirt
345212	Y	N
672333	Y	N
502123	Y	N
230543	N	N
902341	Y	N
324543	N	N

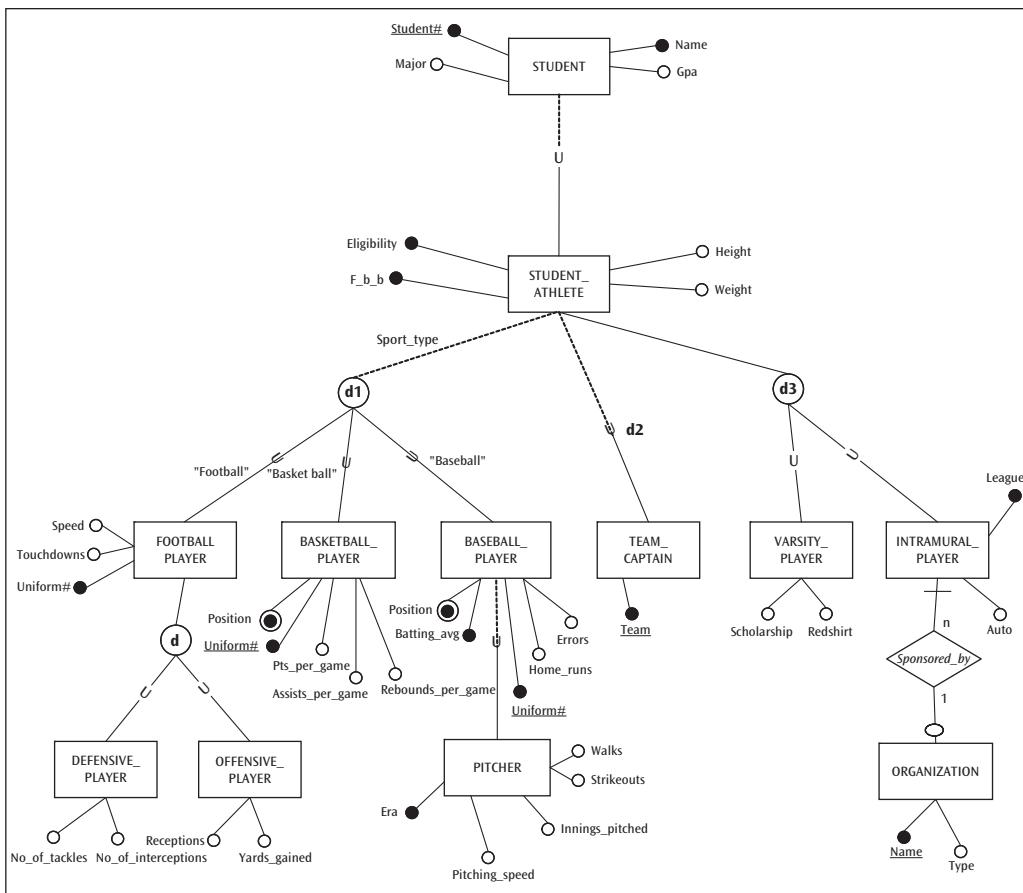
INTRAMURAL_PLAYER Data Set		
Student#	League	Auto
454212	Baseball	Camry

Note: Two student-athletes (Francis and Oliver) are non-scholarship varsity players. Francis participates in basketball while Oliver participates in basketball and soccer. Newman participates in baseball but only as an intramural player.

TABLE 4.2 Sample data sets for Figure 4.7

#### 4.1.5 Specialization Hierarchy and Specialization Lattice

Just as an entity type can be a superclass in multiple specializations, an entity type participating as a subclass in a specialization can serve as a superclass of another specialization. When this happens, there is a hierarchy of specializations. Often, a **specialization hierarchy** will have tiers of specialization cascading through multiple levels, as shown in Figure 4.8. Vignette 2 describes a situation to which a specialization hierarchy can be applied.



**FIGURE 4.8** A specialization/generalization hierarchy

#### 4.1.5.1 Vignette 2

Vignette 2 is a continuation of vignette 1, which was introduced in Section 4.1.4.2. It is common today for a football player to specialize in either offense or defense. Attributes collected for offensive players include receptions and yards gained. Attributes collected for defensive players include number of tackles and number of interceptions. Likewise, some baseball players are also pitchers. Attributes collected about pitchers include earned run average (ERA), pitching speed, innings pitched, strikeouts, and walks.

The structure of a specialization hierarchy is constrained to an inverted tree in that an entity type must not participate as a child in more than one specialization. In informal terms, a child entity type cannot have more than one parent. Notice that Figure 4.8

exhibits a three-level specialization hierarchy [STUDENT → STUDENT\_ATHLETE → FOOTBALL\_PLAYER → {DEFENSIVE\_PLAYER → OFFENSIVE\_PLAYER}].<sup>6</sup> There are no entity types in this structure that participate as a subclass in more than one specialization. Clearly, in such a hierarchy, the type inheritance rule means that a subclass inherits the attributes and relationship types of not just the immediate parent, but also of the predecessor superclasses in the hierarchy all the way up to the root of the specialization hierarchy.

#### 4.1.5.2 Vignette 3

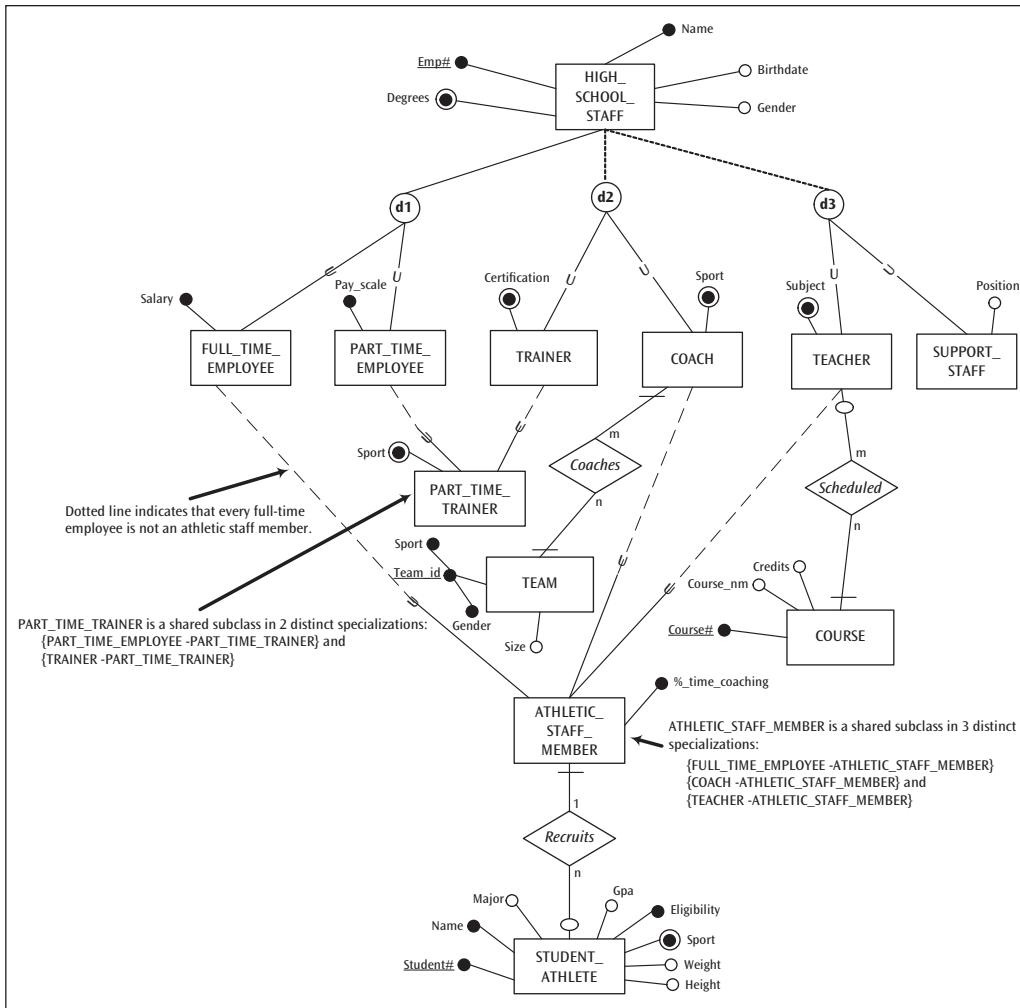
Vignette 3 describes a scenario suitable for modeling as a specialization lattice. Each member of the high school staff at Homer Hanna High School is either a full-time or part-time employee. At the same time, a staff member may be either a trainer or coach and a member of the teaching staff or support staff. A part-time employee may be a part-time trainer, and each part-time trainer is also a trainer. Finally, a member of the athletic staff is a full-time employee, a coach, and a teacher.

This scenario describes a situation in which an entity type can participate as a subclass in more than one specialization; in simple terms, a child can have more than one parent. Such a specialization is called a **specialization lattice**. Since a specialization can involve only one superclass, each parent in a specialization lattice comes from a different specialization. The subclass itself inherits all the attributes and relationship types from the superclasses of all the specializations participating in the specialization lattice and the predecessor hierarchy of all these superclasses. This is called **multiple type inheritance**, and the subclass in the specialization lattice is referred to as a **shared subclass** since it is participating as a subclass in multiple specializations. As a rule, any attribute or relationship type inherited more than once via different paths in the specialization lattice is not duplicated in the shared subclass.

Figure 4.9 illustrates a specialization lattice. Here, the entity type ATHLETIC\_STAFF\_MEMBER is a shared subclass in three distinct specializations: [FULL\_TIME\_EMPLOYEE → ATHLETIC\_STAFF\_MEMBER], [COACH → ATHLETIC\_STAFF\_MEMBER], and [TEACHER → ATHLETIC\_STAFF\_MEMBER] and inherits the specific attributes of FULL\_TIME\_EMPLOYEE, COACH, and TEACHER. In addition, ATHLETIC\_STAFF\_MEMBER inherits the attributes of HIGH SCHOOL\_STAFF, but only once, even though the inheritance itself occurs via three paths. It should be noted that the relationship types *Scheduled* between TEACHER and COURSE and *Coaches* between COACH and TEAM are also inherited by ATHLETIC\_STAFF\_MEMBER. At the same time, the specific attributes of ATHLETIC\_STAFF\_MEMBER and the specific relationship type *Recruits* belong only to ATHLETIC\_STAFF\_MEMBER.

---

<sup>6</sup>The symbol → is used here to convey the hierarchical predecessor and successor.



**FIGURE 4.9** Two occurrences of a specialization lattice

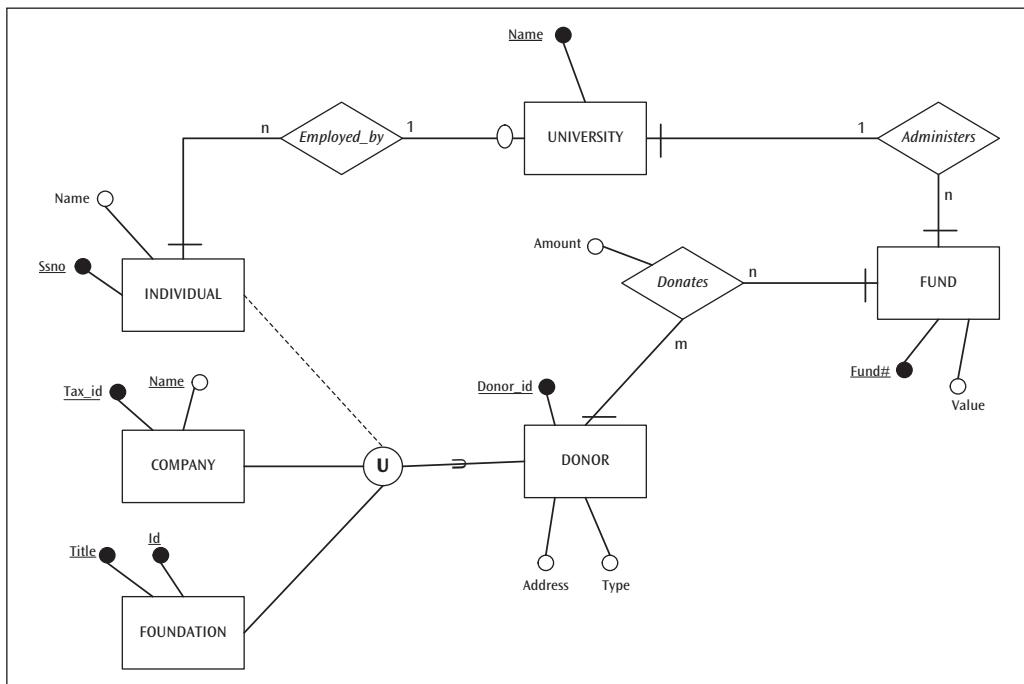
It is important to observe that specializations constituting a lattice are always partial specializations; any total specialization participating in a lattice will create a circularity in the relationships that will contradict the business rules of the requirements specification. If for any reason such a business rule is part of the specification, then the modeled specialization lattice will be incorrect. Finally, while specialization is used as the basis for all discussions in Sections 4.1.4 and 4.1.5, all the concepts covered apply equally to generalization, given that these two constructs portray the same EER modeling features viewed from opposite ends.

#### 4.1.6 Categorization

A fundamental characteristic of the specialization/generalization construct is that there can be only one superclass in the construct. For this reason, there are intra-entity

class (*Is-a*) relationships that cannot be modeled as a specialization/generalization. For example, donors that make donations to a university can be individuals, companies, or foundations. Since INDIVIDUAL, COMPANY, and FOUNDATION are entity types of different entity classes and any one or more individuals, companies, and foundations can be donors, this relationship requires a construct that involves three superclasses and a subclass in a *single* relationship type. Therefore, this relationship cannot be modeled as a specialization/generalization. In other words, the donor set is a collection of entities that is a subset of the union of the three superclass entity sets. Thus, this relationship construct captures a concept different from a specialization/generalization and is called **categorization**. The participating subclass (in this case, DONOR) is labeled a **category**. It is important to note that an entity that is a member of the category (subclass) must exist in only one of the superclasses in the categorization relationship.

Diagrammatically, as shown in Figure 4.10, categorization is represented similarly to specialization except that the circle symbol that signifies the categorization relationship contains the letter “U” to indicate that the membership in the subclass set results from the *union* of multiple superclasses. Also, one or more superclass connectors radiate from the relationship (circle symbol) connecting the participating superclasses. A characteristic of categorization is that there is only one subclass in each categorization. The cardinality ratio in categorization is 1:1 within and across the SC/sc relationships, and the participation of the subclass in the categorization is always total (solid line from the category to the back of the fork). Once again, these properties are inherent to the categorization construct and so are not overtly indicated in the diagram. Each superclass in a categorization may exhibit total participation (solid line from the entity type to the circle symbol) or partial participation (dotted line from the entity type to the circle symbol).

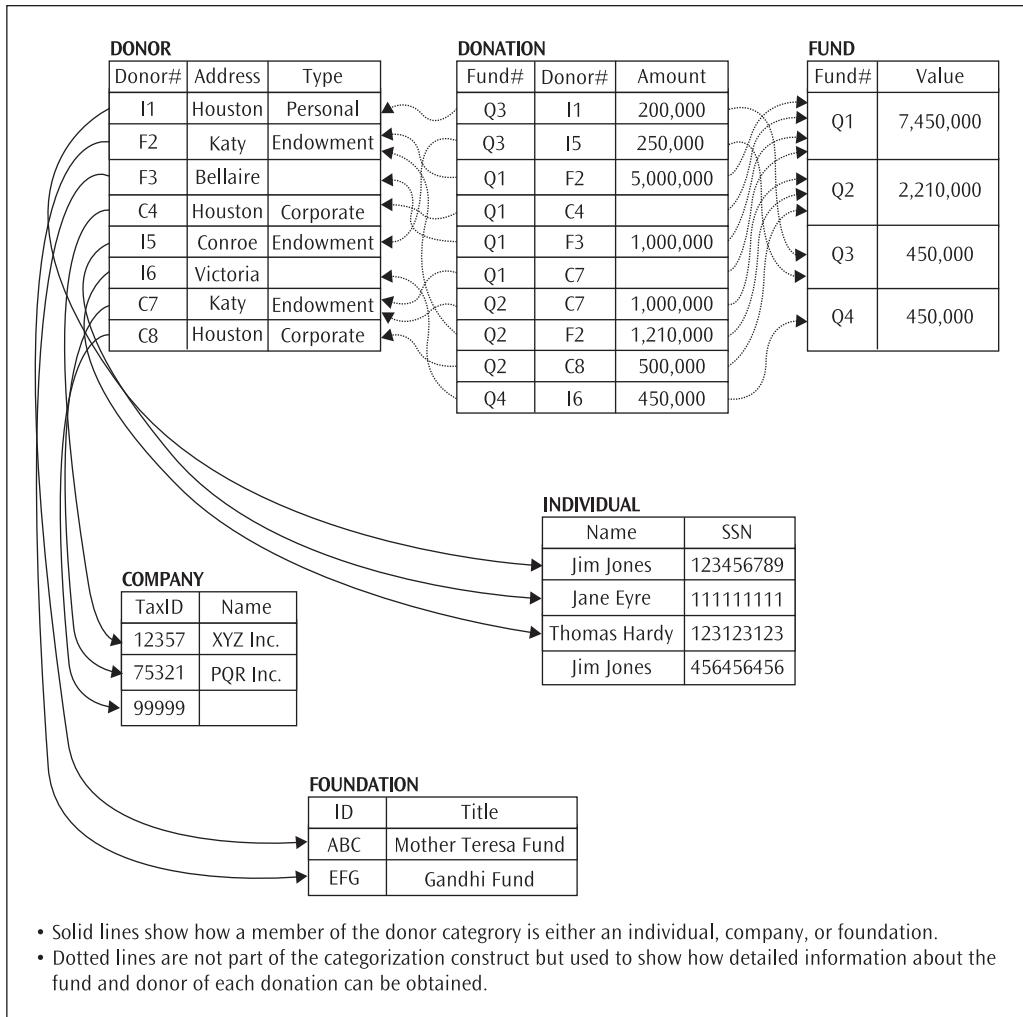


**FIGURE 4.10** An example of categorization

In the example shown in Figure 4.10, the participation of COMPANY and FOUNDATION is total (every company and foundation must be a donor), whereas the participation of INDIVIDUAL is partial (some individuals are not donors). On the other hand, a donor is either a company, a foundation, or an individual. If the completeness constraint for all superclasses in the categorization exhibits total participation, then the category (subclass) itself is called a **total category**. That is, the category set is a union of all the three superclass entities. Likewise, if the completeness constraint is partial, the category itself is referred to as a **partial category** (i.e., the category set is a proper subset of the union of all the superclass entities). Finally, type inheritance in categorization is selective. That is, members of the category (subclass) selectively inherit attributes and relationships of the superclass entity based on the SC/sc relationship in the categorization in which the member participates. This is often referred to as the **selective type inheritance** property of a category. Observe that this is diametrically opposite to the property of multiple type inheritance exhibited by a shared subclass in a specialization lattice.

Sometimes, a category may not have a unique identifier. For instance, DONOR does not have a unique identifier stated as part of the data specification. While type inheritance, even when selective, will furnish the category (in this case, DONOR) with a unique identifier, the properties of the unique identifier will vary across category instances, depending on the superclass from which the attributes are inherited. This situation is often simplified by specifying a “manufactured” surrogate key for the category to serve the role of unique identifier. The attribute **Donor\_id** shown in the EERD in Figure 4.10 is the surrogate key artificially created by the modeler.

The sample data shown in Figure 4.11 illustrates the relationship among INDIVIDUAL, COMPANY, FOUNDATION, DONOR, and FUND shown in Figure 4.10. Observe that DONOR is a partial category because, while the participation of COMPANY and FOUNDATION is total, the participation of INDIVIDUAL is partial (i.e., the Jim Jones with Social Security number 456456456 is not a donor). In Figure 4.11, the surrogate key “manufactured” for each donor begins with a one-character code that represents the donor category (F = Foundation, C = Company, I = Individual) followed by the donor number within that category. In addition, the DONATION data reflects the fact that each donor makes a donation to at least one fund and that each fund has at least one donation from a donor.

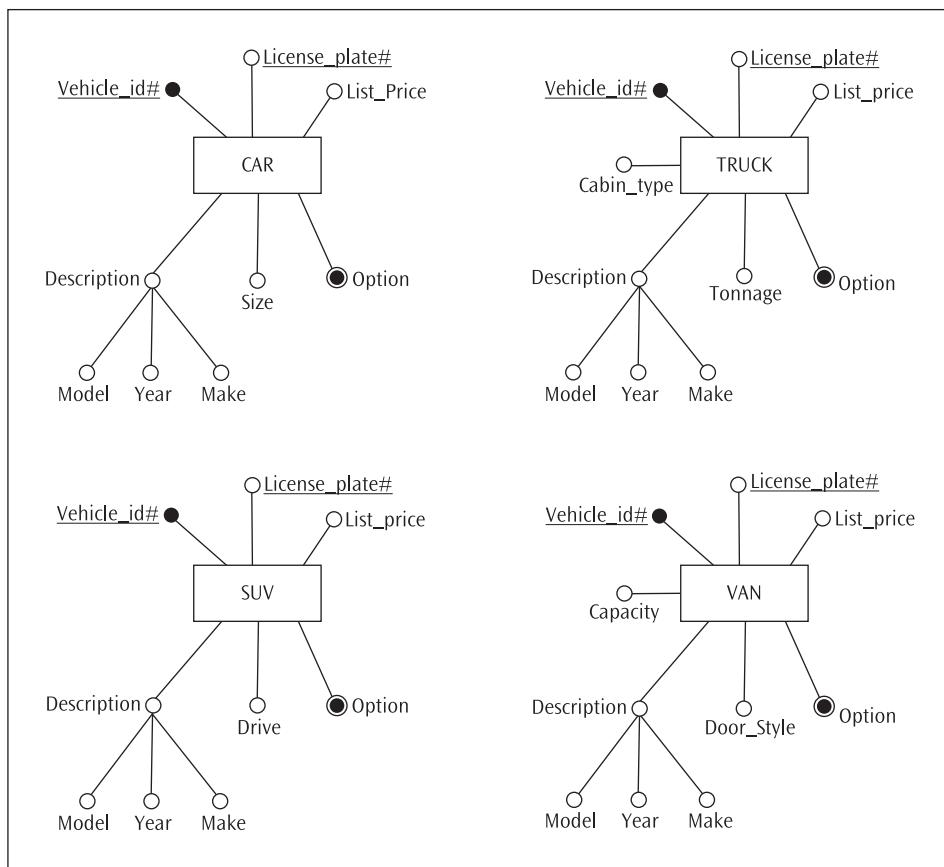
**FIGURE 4.11** Sample data sets for the categorization example in Figure 4.10

#### 4.1.7 Choosing the Appropriate EER Construct

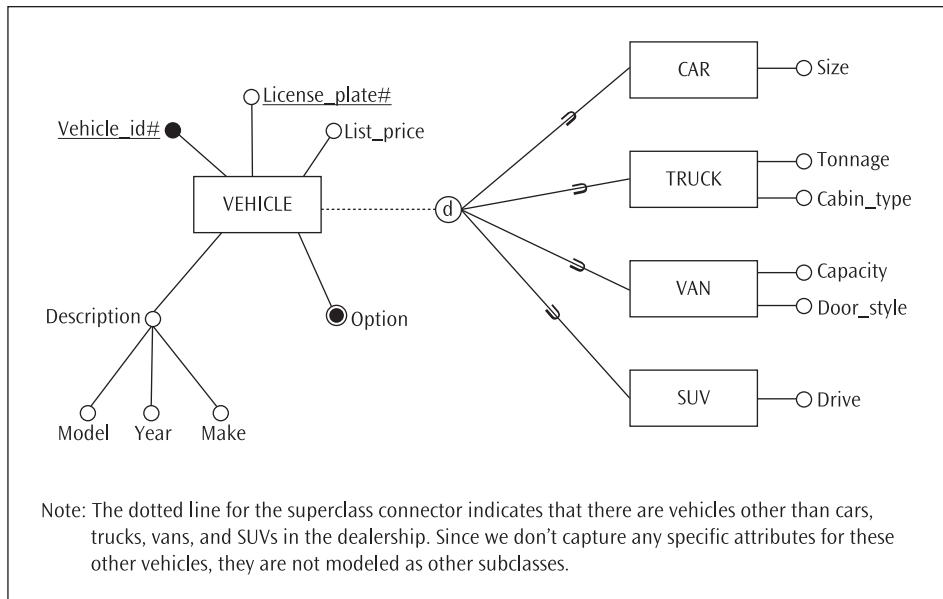
It should be clear at this point that specialization/generalization and categorization serve different modeling needs. Are there any rules of thumb that will help determine which to use in a given situation? Yes, there are, and the principles can best be illustrated with an example.

Consider a scenario of an automobile dealership in the state of Texas. The dealership typically stocks cars, trucks, vans, and sport utility vehicles (SUVs). Figure 4.12 displays the entity types CAR, TRUCK, VAN, and SUV. The generalization of these entity types to the entity type VEHICLE that contains the attributes common to cars, trucks, vans, and SUVs appears in Figure 4.13. The result is a generalization/specialization in which

VEHICLE represents the entity class (superclass) to which the entity types (subclasses) CAR, TRUCK, VAN, and SUV belong. Notice that the completeness constraint indicates that the dealership has other vehicles not captured explicitly in this generalization and the only data captured on these vehicles is that which is common to all vehicles. It is also important to realize that an entity *cannot* be a member of one of the subclasses {CAR, TRUCK, VAN, SUV} unless it exists in the superclass VEHICLE.

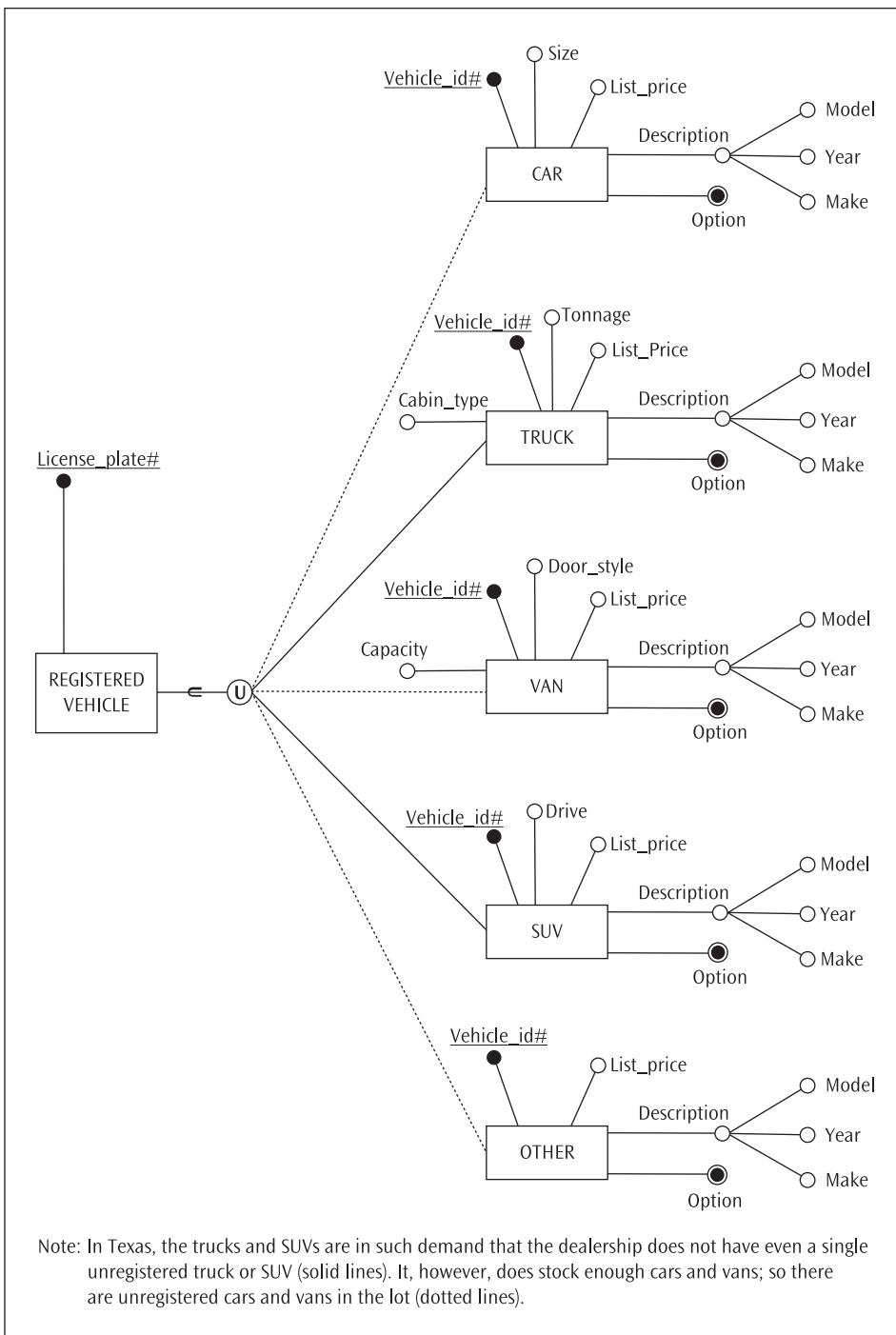


**FIGURE 4.12** A set of entity types: CAR, TRUCK, VAN, and SUV



**FIGURE 4.13** Generalization of subclasses CAR, TRUCK, VAN, and SUV to a VEHICLE superclass

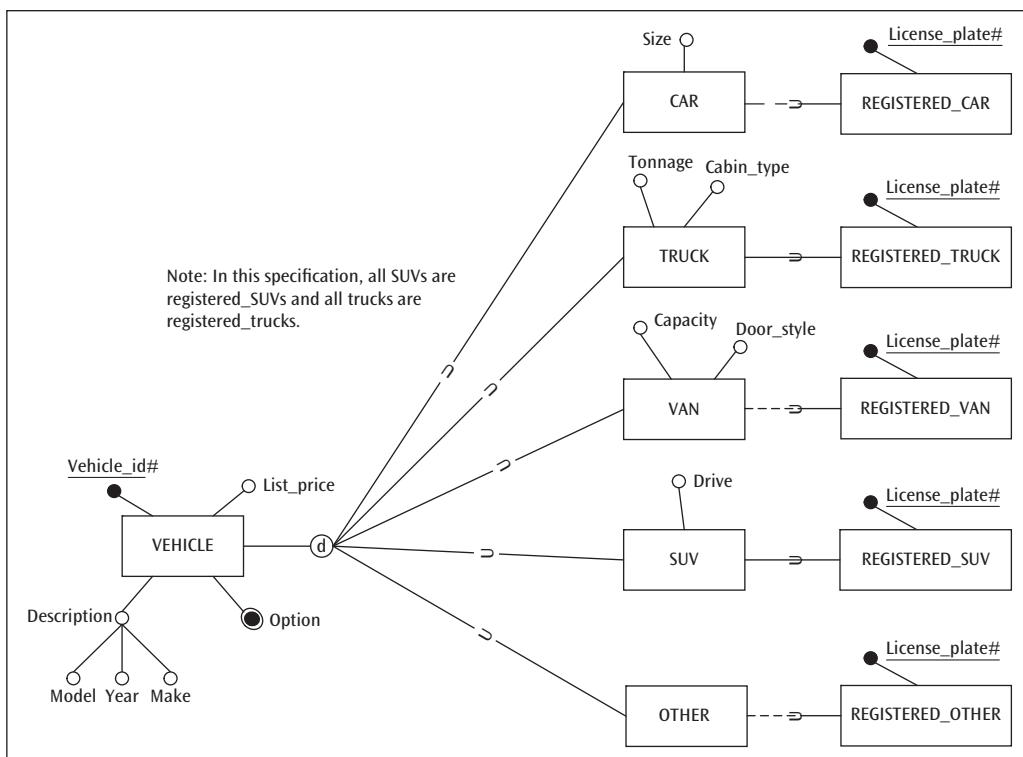
Not all vehicles in the dealership are registered vehicles, though. In other words, the dealership has in its inventory lots of vehicles not yet registered; they do not have an assigned license plate number. How would you model REGISTERED\_VEHICLE? The crucial issue here is that the model should allow for the possibility of some unregistered cars, trucks, vans, and/or SUVs to be present. This property cannot be captured in the generalization/specialization construct shown in Figure 4.13. One method for modeling this situation involves the use of the categorization construct. Figure 4.14 presents this scenario. Note that REGISTERED\_VEHICLE is the subclass in this categorization, while CAR, TRUCK, VAN, and SUV are superclasses of the categorization. Participation of CAR and VAN in this relationship is partial, while TRUCK and SUV exhibit total participation. This is indicated in Figure 4.14 by the dotted lines and solid lines for the completeness constraint. That there are other vehicles in the lot and some of them may be registered vehicles is incorporated via the superclass OTHER with a partial participation in the categorization. Also, since the category REGISTERED\_VEHICLE has a unique identifier `License_plate#`, there is no need to create a surrogate key for REGISTERED\_VEHICLE.



**FIGURE 4.14** Categorization of superclasses CAR, TRUCK, VAN, and SUV to a REGISTERED\_VEHICLE subclass

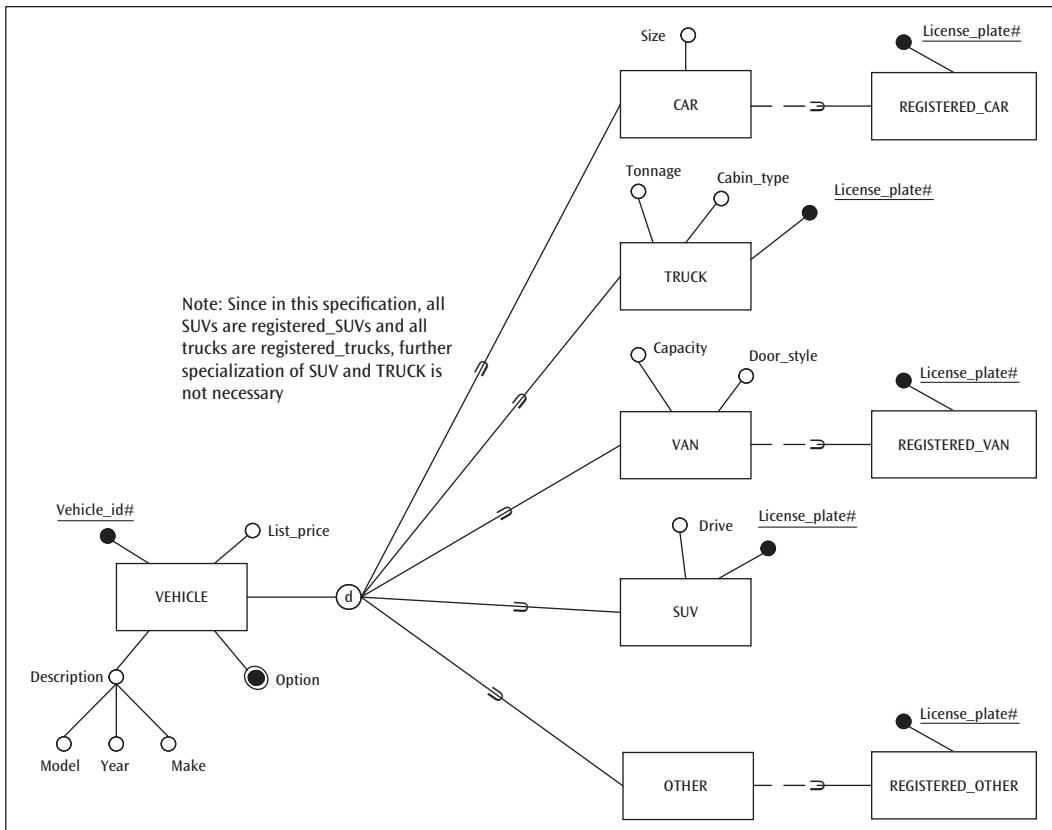
## Chapter 4

Note that if REGISTERED\_VEHICLE is a “total category” (i.e., if the participation of all the superclasses in the categorization is total), then a generalization/specialization can replace the categorization. But then, in this case, there is no difference between VEHICLE (Figure 4.13) and REGISTERED\_VEHICLE (Figure 4.14). In essence, with a total category, categorization and generalization/specialization are mutually substitutable constructs. If REGISTERED\_VEHICLE is a “partial category,” an alternative design of the following form is possible. You could portray the registered subset of CAR, TRUCK, VAN, SUV, and OTHER as subclasses REGISTERED\_CAR, REGISTERED\_TRUCK, REGISTERED\_VAN, REGISTERED\_SUV, and REGISTERED\_OTHER of the respective vehicle types CAR, TRUCK, VAN, SUV, and OTHER. CAR, TRUCK, VAN, SUV, and OTHER can participate as subclasses of a generalization in which VEHICLE serves as the superclass, thus creating a generalization hierarchy that includes both vehicles and registered vehicles, as shown in Figure 4.15.



**FIGURE 4.15** The partial category REGISTERED\_VEHICLE in Figure 4.14 expressed in a specialization hierarchy

The ERD in Figure 4.15 can be rendered a little more efficient by eliminating the subclasses REGISTERED\_SUV and REGISTERED\_TRUCK from the ERD since, according to the specifications provided, all trucks and SUVs are indeed registered. The revised ERD appears in Figure 4.16.



**FIGURE 4.16** A revised version of the disjoint specialization hierarchy of VEHICLE displayed in Figure 4.15

In general, it may not be efficient to replace a partial category with a generalization/specialization construct. A total category, on the other hand, can always be alternatively modeled as a generalization/specialization—total or partial. Although the choice is subjective and context specific, if the superclasses belong to the same semantic class of entity—that is, if they share numerous attributes that are common, sometimes including unique identifiers with the same properties—generalization/specialization may be a more appropriate modeling construct to adopt. Otherwise, one must resort to categorization as the modeling construct.

#### 4.1.8 Aggregation<sup>7,8</sup>

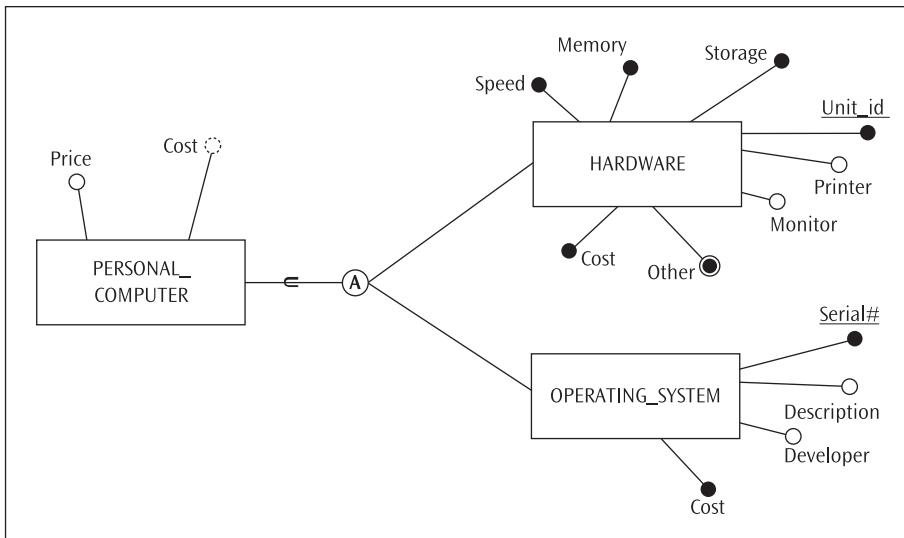
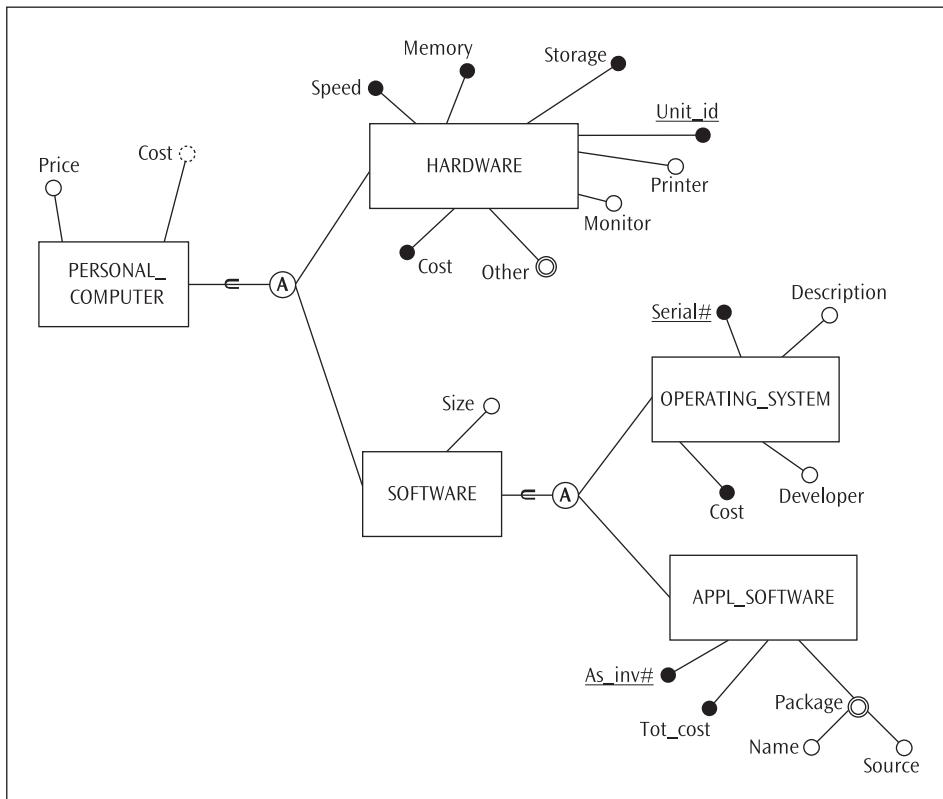
While categorization is capable of expressing a modeling variation that generalization/specialization cannot incorporate, there are other constraints that categorization imposes in order to sharpen its expressive power. In categorization, an entity that is a member of a category (subclass) must exist in only one of its superclasses. With aggregation, this constraint is relaxed. In fact, it is not only relaxed, it is reversed in the aggregation construct, thereby enriching the capabilities of an EER modeling domain.

**Aggregation** allows modeling a “whole/part” association as an “*Is-a-part-of*” relationship between a superclass and a subclass. An aggregate here is a subclass that is a subset of the aggregation of the superclasses in the relationship. In other words, an entity in the aggregate *contains* superclass entities from *all* SC/sc relationships in which it participates. Therefore, in this case the type inheritance is collective, as opposed to categorization, where it is selective. **Collective type inheritance** connotes inheritance of attributes and relationships from all superclass entities contained in the specific aggregation.

A diagrammatic representation of the aggregation construct in the EER diagram is shown in Figure 4.17. The notation is similar to that of categorization except that the union indicator (U) is replaced by the aggregation indicator (A). In the example in Figure 4.17, the subclass PERSONAL\_COMPUTER is the aggregate of which HARDWARE and OPERATING\_SYSTEM are parts. While a category can be total or partial, an aggregate can never be partial (no connector is a dotted line). That is, all hardware and operating system entities are “part of” some personal computer. Further, a hardware entity or an operating system entity can belong to only one personal computer entity. Figure 4.18 portrays an aggregation hierarchy.

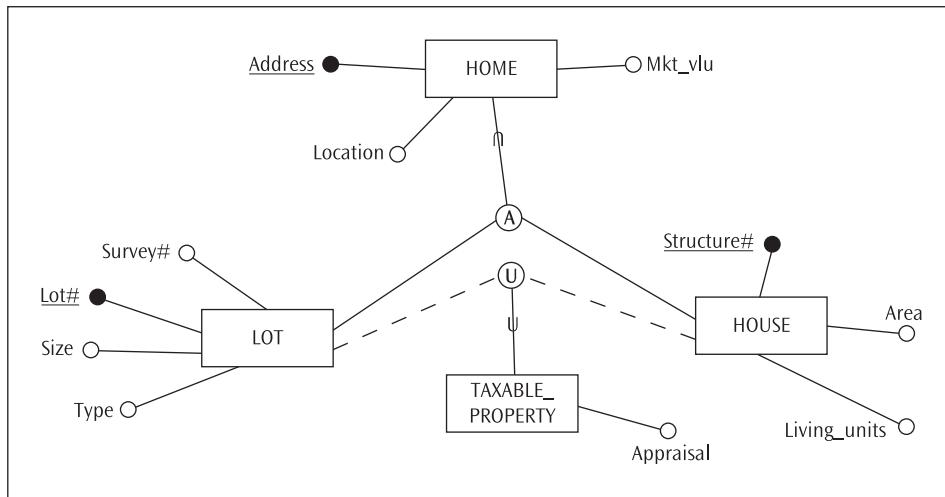
<sup>7</sup>Unified modeling language (UML) for object-oriented modeling distinguishes between simple aggregation, which is entirely conceptual, and composition, which is classified as a variation of simple aggregation and does add some valuable semantics (Booch, Rumbaugh, and Jacobson, 2005). Composition changes the meaning of navigation across the association between the whole and its parts and links the lifetimes of the whole and its parts. Aggregation, as described in this section, corresponds to UML’s composition construct.

<sup>8</sup>The term “aggregation” is also used in inter-entity class relationships to indicate a cluster of related entity types, which is referred to as an aggregate entity type or a cluster entity type. This will be addressed in Chapter 5.

**FIGURE 4.17** An example of the EER aggregation construct**FIGURE 4.18** An aggregation hierarchy

## Chapter 4

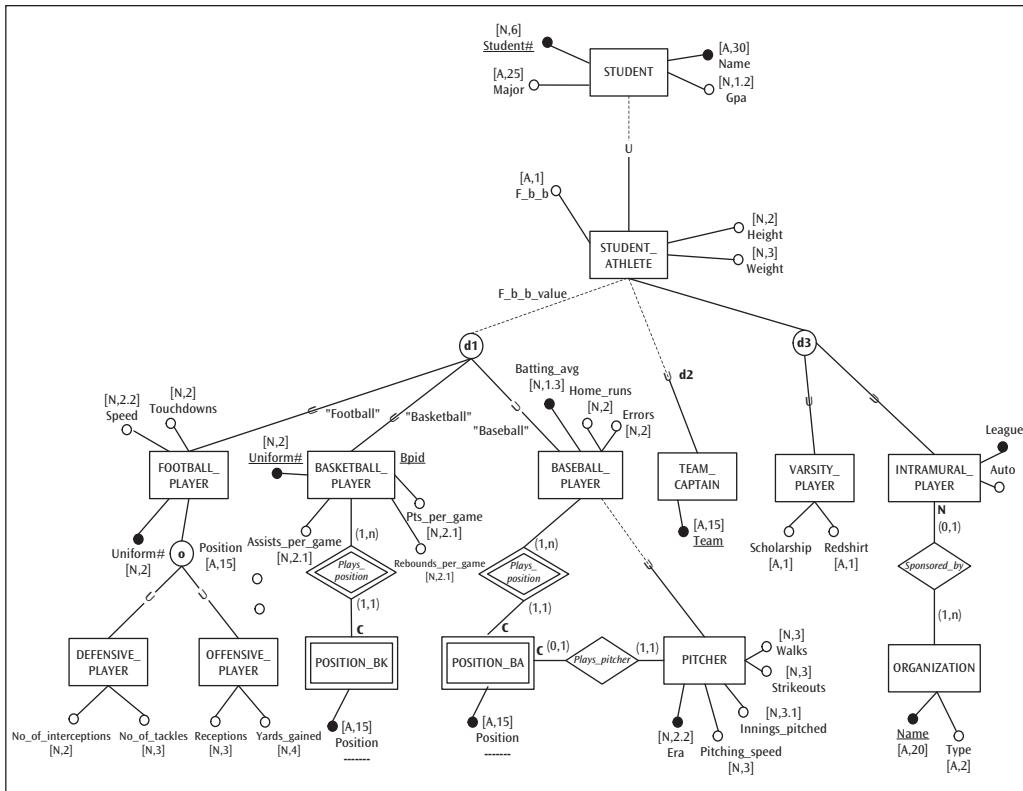
Figure 4.19 depicts an aggregation and a categorization involving the same set of entity types. Every taxable property is a lot or a house (not a lot *and* a house together as a single taxable property). Some lots and some houses are not taxable properties (superclass connectors are dotted lines, denoting partial completeness). Each lot and each house is recorded as a separate taxable property. On the contrary, a lot and a house are “parts of” a home—that is, a HOME entity includes both a LOT entity and a HOUSE entity. All lots and houses participate in the aggregation. A house and a lot cannot belong to more than one home.



**FIGURE 4.19** A category and an aggregate contrasted

## 4.2 CONVERTING FROM THE PRESENTATION LAYER TO A DESIGN-SPECIFIC EER DIAGRAM

Recall that the Design-Specific ER model transforms the user-oriented Presentation Layer ER model to a database designer orientation while at the same time fully preserving all constructs and constraints that are inherent, implicit, and explicit in the Presentation Layer ER model. Figure 4.20 illustrates the representation of the Presentation Layer ER diagram of Figure 4.8 as a Design-Specific ER diagram. Observe that the converted diagram contains (a) the data type and size of each atomic attribute, (b) the transformation of the multi-valued attributes in the Presentation Layer ER diagram into weak entity types, and (c) deletion constraints that govern inter-entity class relationships.



**FIGURE 4.20** A Design-Specific EER diagram

In the Presentation Layer ER diagram shown in Figure 4.8, the multi-valued attribute **Position** in BASKETBALL\_PLAYER and BASEBALL\_PLAYER indicates that both baseball and basketball players can play more than one position within that sport. Observe in Figure 4.20 the multi-valued attribute **Position**, which appears in both BASKETBALL\_PLAYER and BASEBALL\_PLAYER, has been replaced by the weak entity types POSITION\_BK (for BASKETBALL\_PLAYER) and POSITION\_BA (for BASEBALL\_PLAYER), with the attribute **Position** serving as its partial key in each weak entity type. In BASEBALL\_PLAYER, replacing the multi-valued attribute **Position** with the weak entity type POSITION\_BA also requires that a relationship *Plays\_pitcher* between the POSITION\_BA and PITCHER be established, where each POSITION\_BA entity is associated with at most one PITCHER entity and each PITCHER entity is associated with exactly one POSITION\_BA entity. The SC/sc relationship between BASEBALL\_PLAYER and PITCHER remains intact.

The deletion constraints associated with the inter-entity class relationship types in Figure 4.20 are listed here, followed by a description, *in italics*, of how the deletion constraint is shown in the figure.

- If an organization stops sponsoring intramural players, then intramural players sponsored by that organization are retained in the database for

possible future affiliation with another organization. *The N (representing the set null rule) located immediately below INTRAMURAL\_PLAYER indicates that an intramural player entity is retained when the related organization is deleted.*

- When a basketball player does not play any longer, all records of the positions s/he plays are removed along with the player entity. *The C (representing the cascade rule) located immediately above the weak entity type POSITION\_BK specifies the deletion of related entities in the POSITION\_BK entity set when a basketball player is deleted.*
- When a baseball player does not play any longer, all records of positions s/he plays are removed along with the player entity. *The C (representing the cascade rule) located immediately above the weak entity type POSITION\_BA specifies the deletion of related entities in the POSITION\_BA entity set when a baseball player is deleted.*
- When a pitcher is no longer in service, the associated position entity in POSITION\_BA is removed along with the pitcher entity. *The C (representing the cascade rule) located immediately next to the weak entity type POSITION\_BA specifies the deletion of the related entity in the POSITION\_BA entity set when a pitcher is deleted.*

### 4.3 BEARCAT INCORPORATED DATA REQUIREMENTS REVISITED

---

Recall that the discussion of the ER model in Chapter 3 is framed in the context of data requirements for Bearcat Incorporated, a manufacturing company with several plants located in the northeastern region of the United States. This section introduces additional requirements that will exemplify the incorporation of EER modeling constructs to the conceptual model of Bearcat Incorporated.

Bearcat Incorporated provides significant recreational opportunities for the dependents of its employees. However, in order to do so, the company's employee association aggressively solicits sponsors for various hobbies. The sponsor of a hobby can be one or more individuals, schools, or churches. Although each hobby need not have a sponsor, each individual, school, and church is involved in sponsoring one or more hobbies. A Social Security number is used to identify each individual sponsor. Other data captured about individuals include name, address, and phone number. Schools and churches are identified by their names. For a church, its denomination and pastor are recorded; for a school, its size and the name of its principal are recorded. The church denominations (when available) are Catholic, Baptist, Methodist, or Lutheran.

Many of the sponsors of hobbies are not-for-profit organizations; for these organizations, the type, exempt ID, and annual operating budget are recorded. Some of the schools are public schools and therefore are also classified as not-for-profit organizations. For the public schools, the name of the school district and its tax base are recorded.

A plant's project may be done in-house or outsourced to one or more contracted vendors. However, a vendor can participate in only one outsourced project at a time. A plant employee is assigned to an in-house project, and an in-house project involves one or

more employees. An employee is involved in no more than seven in-house projects but need not be involved in any project. For both in-house and outsourced projects, a description of the project is gathered. Data gathered about each vendor include a vendor name, address, phone number, and contact person. Vendor name is used to identify each vendor.

Because the same vendors are often contracted for future projects, when an outsourced project is removed from the system, the vendor information should be retained for future use. If a hobby is removed from the recreation portfolio of Bearcat Incorporated, its relationship with any sponsor is removed as well. Likewise, when a sponsor is removed from the recreation portfolio of Bearcat Incorporated, its relationship with any hobby is removed.

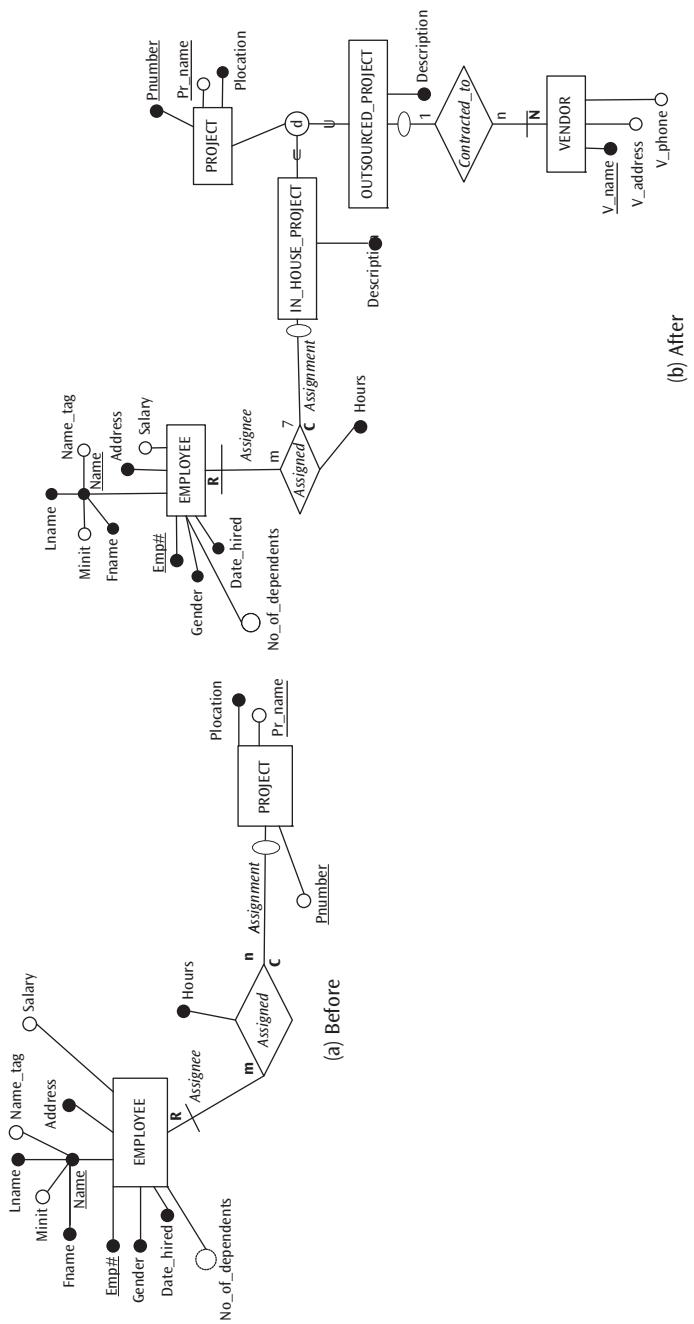
## 4.4 ER MODEL FOR THE REVISED STORY

---

The revision to the specifications presented in Section 4.3 is essentially an extension to the specifications given in Chapter 3. A systematic incorporation of the additional specifications from Section 4.3 into the ER model developed in Chapter 3 is discussed in the following paragraphs. Two boxes are used in which relevant excerpts from the specifications narrated in Section 4.3 are included, and the corresponding diagrammatic expression is developed using the enhanced ER modeling constructs covered in this chapter. This method of development is the same one used in Chapter 3 for the development of the original ER model. Once again, the Presentation Layer ER diagram is used in order to emphasize that the enhanced ER modeling constructs do indeed belong in the presentation layer.

The first subset of excerpts from the specifications stated in Section 4.3 appears at the top of Box 1 on next page. The existing inter-entity relationship between EMPLOYEE and PROJECT from the original ER diagram is shown on the left side of Box 1. The revisions to the ERD based on the specifications subset is shown on the right side of Box 1. Because projects are done either in-house or outsourced, observe how PROJECT now plays the role of a superclass entity type (SC) in a total, disjoint specialization, with IN\_HOUSE\_PROJECT and OUTSOURCED\_PROJECT as the newly introduced subclass entity types (sc). Because a project may be done either in-house or outsourced (not both), the circular specialization symbol contains the letter "d." Furthermore, because vendors are contracted to do the outsourced projects, the inter-entity relationship type *Contracted\_to* has been created between OUTSOURCED\_PROJECT and VENDOR. The deletion rule specified for this relationship type is also incorporated in the ERD.

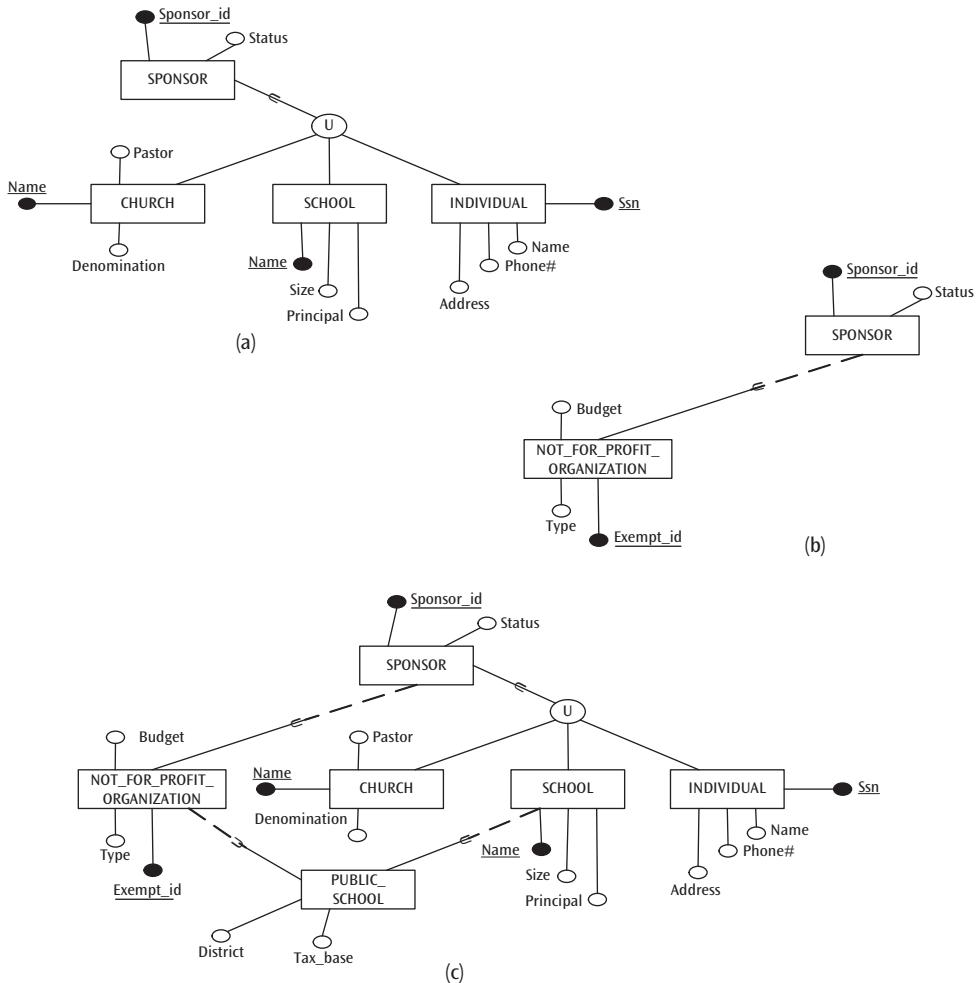
A plant's project may be done in-house or outsourced to one or more contracted vendors. However, a vendor can participate in only one outsourced project at a time. A plant employee is assigned to an in-house project, and an in-house project involves one or more employees. An employee is involved in no more than seven in-house projects but need not be involved in any project. For both in-house and outsourced projects, a description of the project is gathered. Data gathered about each vendor include a vendor name, address, phone number, and contact person. Vendor name is used to identify each vendor. Because the same vendors are often contracted for future projects, when an outsourced project is removed from the system, the vendor information should be retained for future use.

**BOX 1**

The sponsor of a hobby can be one or more individuals, schools, or churches. Although each hobby need not have a sponsor, each individual, school, and church is involved in sponsoring one or more hobbies. A Social Security number is used to identify each individual sponsor. Other data captured about individuals include name, address, and phone number. Schools and churches are identified by their names. For a church, its denomination and pastor are recorded; for a school, its size and the name of its principal are recorded.

Many of the sponsors of hobbies are not-for-profit organizations; for these organizations, the type, exempt ID, and annual operating budget are recorded.

Some of the schools are public schools and therefore are also classified as not-for-profit organizations. For the public schools, the name of the school district and its tax base are recorded.



## BOX 2

The excerpts from the requirements specification stated in Section 4.3 that are displayed at the top of Box 2 depict three distinct, yet interrelated concepts:

- a) Sponsors of hobbies comprise churches, schools and individuals.
- b) Some of the sponsors are not-for-profit organizations.
- c) All public schools are not-for-profit organizations.

First, CHURCH, SCHOOL, and INDIVIDUAL represent different entity classes. Thus, the fact that these three entity types are the sponsors cannot be represented by a specialization simply because the entity types participating in a specialization relationship type must, by definition, belong to the same entity class. Categorization is the EER modeling construct designed to express the union of entity types from different entity classes (SCs) selectively representing a subclass entity type SPONSOR. Box 2(a) presents this scenario using the intra-entity class relationship “Categorization.” Observe that the requirements specification does not provide a unique identifier for the “category” (subclass in this relationship). Although, through selective inheritance, SPONSOR does indeed inherit the respective unique identifier from each of CHURCH, SCHOOL, and INDIVIDUAL, none of them can serve the role of the unique identifier of all the entities in the SPONSOR entity set due to incompatible attribute characteristics across them. Under these circumstances, the modeler has no choice but to specify a surrogate key for the category; thus, **Sponsor\_id** is the surrogate key manufactured by the modeler for the entity type SPONSOR participating as the subclass in the categorization that has three superclass entity types; see Box 2(a).

The second concept (b) stated earlier is captured in Box 2(b). This is a simple partial specialization depicting the business rule that some of the sponsors are not-for-profit organizations.

Finally, Box 2(c) first combines the ERDs in (a) and (b). Next, the entity type PUBLIC\_SCHOOL is modeled as the shared subclass participating in two distinct partial specializations—one in which SCHOOL is the superclass and the other in which NOT\_FOR\_PROFIT\_ORGANIZATION is the superclass. Thus, the specialization lattice results. Observe that the entity type SPONSOR plays the role of a subclass in one intra-entity class relationship (categorization) while also assuming the role of a superclass in another intra-entity class relationship (specialization).

Figure 4.21 and Table 4.3 incorporate the additional requirements specified in Section 4.3 into the Presentation Layer ER model for Bearcat Incorporated.

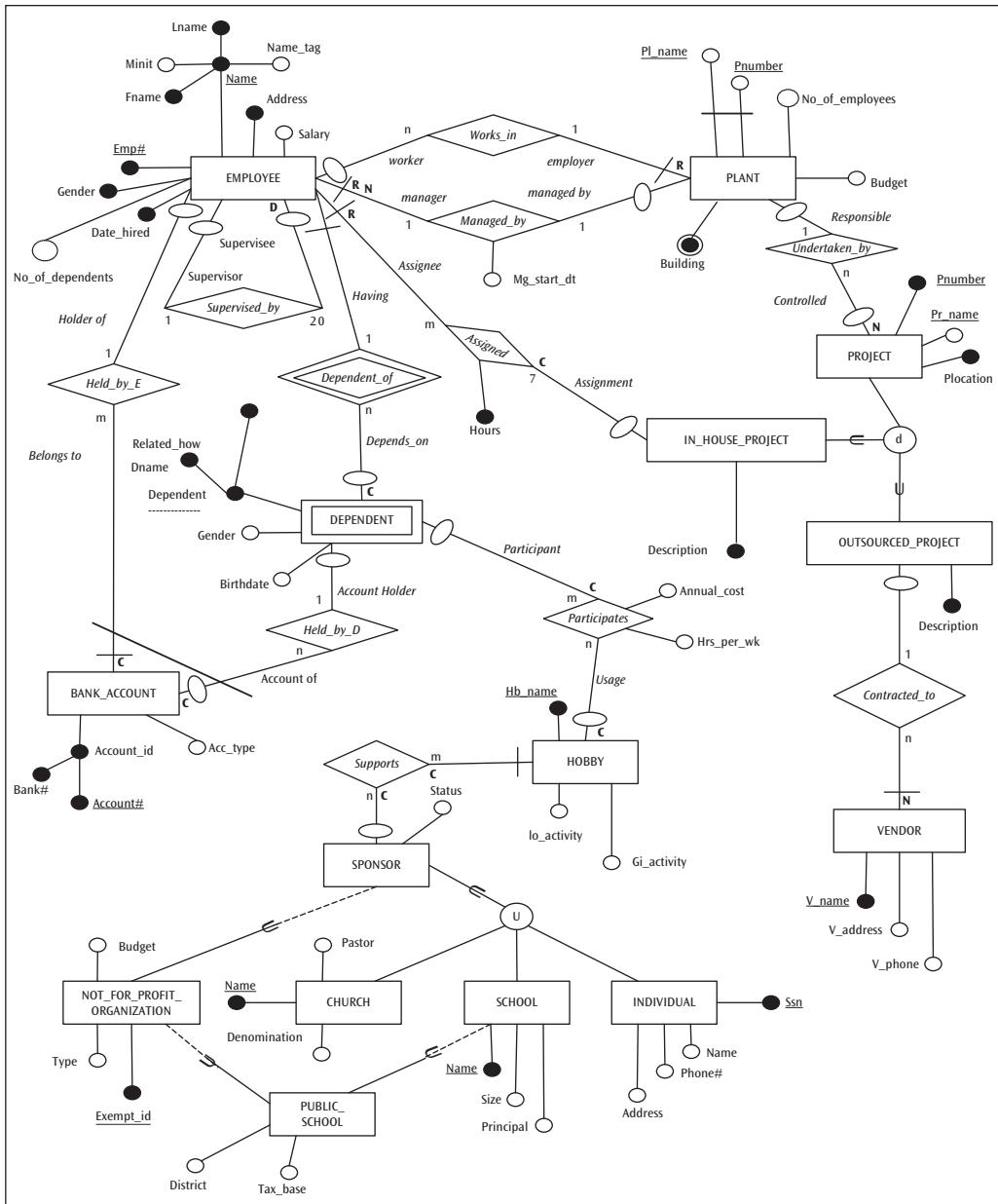


FIGURE 4.21 Presentation Layer Enhanced ER (EER) diagram for Bearcat Incorporated

**Attribute-Level Business Rules**

1. Each plant has a plant number that ranges from 10 to 20.
2. The gender of each employee or dependent is either male or female.
3. Project numbers range from 1 to 40.
4. Project locations are confined to the cities of Bellaire, Blue Ash, Mason, Stafford, and Sugarland.
5. Account types are coded as C (checking account), S (savings account), or I (investment account).
6. A hobby can be either I (indoor activity) or O (outdoor activity).
7. A hobby can be a G (group activity) or I (individual activity).
8. Church denomination, if available, is confined to Baptist, Catholic, Lutheran, or Methodist.
9. The status of a sponsor can be either A (active) or I (inactive).

**Entity-Level Business Rules**

1. A mother or daughter dependent must be a female, a father or son dependent must be a male, and a spouse dependent can be either a male or female.
2. An employee cannot be his or her own supervisor.
3. A dependent may have a joint account only with an employee of Bearcat Incorporated to whom he or she is related.
4. Every plant is managed by an employee who works in the same plant.

**Miscellaneous Business Rules**

1. Each plant has at least three buildings.
2. Each plant must have at least 100 employees.
3. The salary of an employee cannot exceed the salary of the employee's supervisor.

(Note: Shaded statements represent constraints originally established in Chapter 3.)

**TABLE 4.3** Presentation Layer semantic integrity constraints for expanded Bearcat Incorporated scenario

Table 4.4 represents the first step in the transition from the Presentation Layer ER model to the Design-Specific ER model—the collection of attribute characteristics (data type, size, and domain constraints) for the incremental specifications added for the Bearcat Incorporated case in Section 4.3.

Entity/Relationship Type Name	Attribute Name	Data Type	Size	Domain Constraint
PLANT	Pl_name	Alphabetic	30	
PLANT	Pnumber	Numeric	2	Integer values from 10 to 20.
PLANT	Budget	Numeric	7	
PLANT	Building	Alphabetic	20	
PLANT	No_of_employees	Numeric	7	
EMPLOYEE	Emp#	Alphanumeric	6	
EMPLOYEE	Fname	Alphabetic	15	
EMPLOYEE	Minit	Alphabetic	1	
EMPLOYEE	Lname	Alphabetic	15	
EMPLOYEE	Name_tag	Numeric	1	Integer values from 1 to 9.
EMPLOYEE	Address	Alphanumeric	50	
EMPLOYEE	Gender	Alphabetic	1	M or F
EMPLOYEE	Date_hired	Date	8	
EMPLOYEE	Salary	Numeric	6	Ranges from \$35,000 to \$90,000
EMPLOYEE	No_of_dependents	Numeric	2	
<i>Managed_by</i>	Mgr_start_dt	Date	8	
DEPENDENT	Dname	Alphabetic	15	
DEPENDENT	Related_how	Alphabetic	12	
DEPENDENT	Birthdate	Date	8	
PROJECT	Pr_name	Alphabetic	20	
PROJECT	Pnumber	Numeric	2	Integer values from 1 to 40.
PROJECT	Plocation	Alphabetic	15	Bellaire, Blue Ash, Mason, Stafford, Sugarland
<i>Assigned</i>	Hours	Numeric	3	
BANK ACCOUNT	Bank#	Numeric	2	Integer values from 10 to 90.
BANK ACCOUNT	Acct#	Alphanumeric	6	
BCU_ACCOUNT	Acct_type	Alphabetic	1	C = Checking Acct., S = Savings Acct., I = Investment Acct.
HOBBY	Hb_name	Alphabetic	20	
HOBBY	Io_activity	Alphabetic	1	I = Indoor Activity, O = Outdoor Activity

**TABLE 4.4** Semantic integrity constraints for Initial Design-Specific ER model for expanded Bearcat Incorporated scenario—Initial

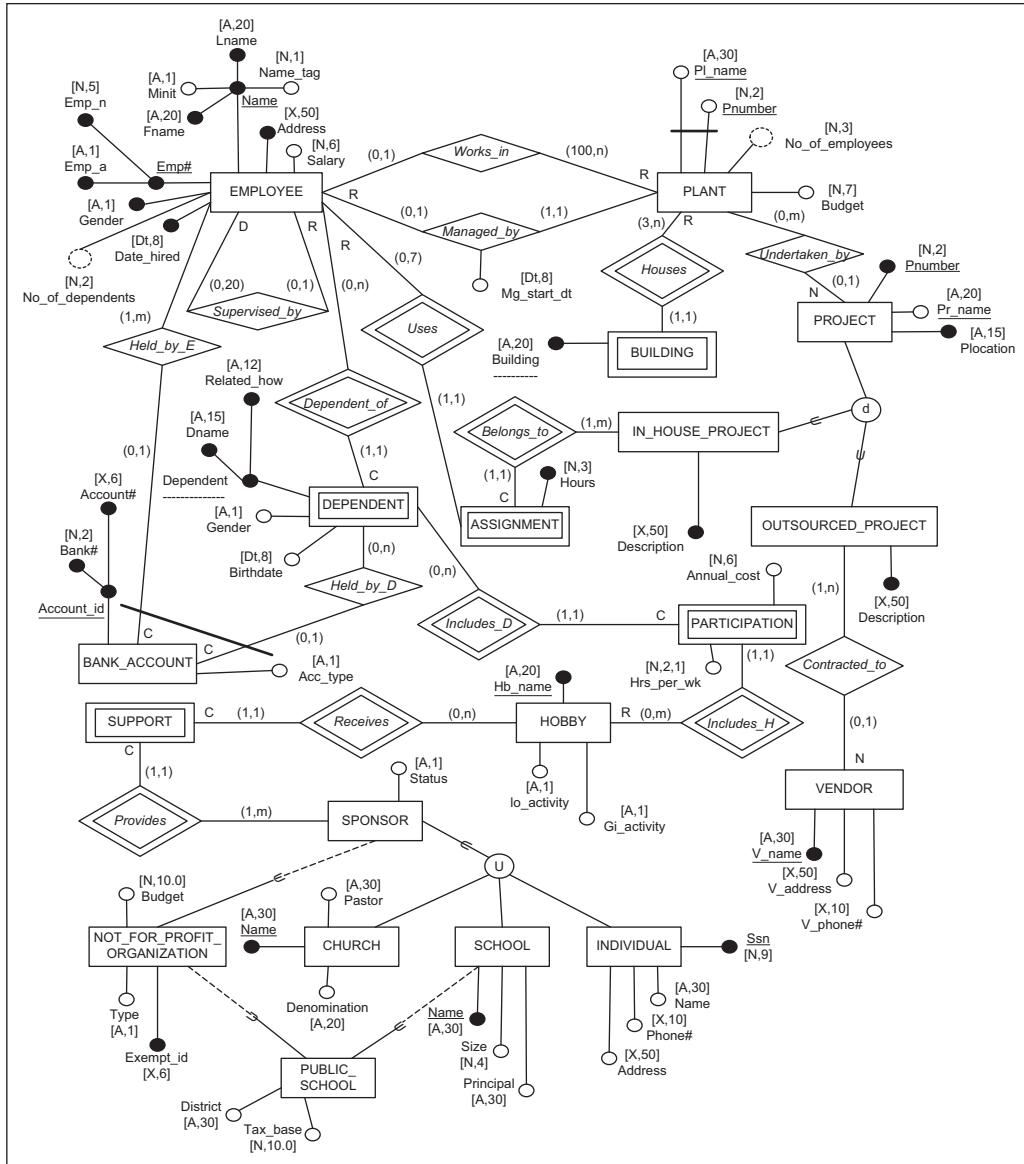
Entity/Relationship Type Name	Attribute Name	Data Type	Size	Domain Constraint
HOBBY	Gi_activity	Alphabetic	1	G = Group Activity, I = Individual Activity
Participates	Hrs_per_wk	Numeric	(2.1)*	
Participates	Annual_cost	Numeric	6	
IN_HOUSE_COMPONENT	Proj_pet	Numeric	(3.1)	
IN_HOUSE_COMPONENT	Component	Alphanumeric	50	
OUTSOURCED_COMPONENT	Proj_pet	Numeric	(3.1)	
OUTSOURCED_COMPONENT	Component	Alphanumeric	50	
VENDOR	V_name	Alphabetic	30	
VENDOR	V_address	Alphanumeric	50	
VENDOR	V_phone#	Alphanumeric	10	
SPONSOR	Status	Alphabetic	1	A = Active, I = Inactive
NOT_FOR_PROFIT_ORGANIZATION	Exempt_id	Alphanumeric	9	
NOT_FOR_PROFIT_ORGANIZATION	Budget	Numeric	(10.0)	
NOT_FOR_PROFIT_ORGANIZATION	Type	Alphabetic	1	
CHURCH	Name	Alphabetic	30	
CHURCH	Pastor	Alphabetic	30	
CHURCH	Denomination	Alphabetic	20	Baptist, Catholic, Lutheran, Methodist
SCHOOL	Name	Alphanumeric	40	
SCHOOL	Size	Numeric	4	
SCHOOL	Principal	Alphabetic	30	
INDIVIDUAL	Ssno	Numeric	9	
INDIVIDUAL	Address	Alphanumeric	50	
INDIVIDUAL	Name	Alphabetic	30	
INDIVIDUAL	Phone#	Alphanumeric	10	
PUBLIC_SCHOOL	District	Alphabetic	30	
PUBLIC_SCHOOL	Tax_base	Numeric	(10,0)	
*(n,m) is used to indicate n places to the left of the decimal point and m places to the right of the decimal point				

**TABLE 4.4** Semantic integrity constraints for Initial Design-Specific ER model for expanded Bearcat Incorporated scenario—Initial (continued)

<u>Entity-Level Domain Constraints</u>
1. A mother or daughter dependent must be a female, a father or son dependent must be a male, and a spouse dependent can be either a male or female.
2. An employee cannot be his or her own supervisor.
3. A dependent can have a joint account with only an employee of Bearcat Incorporated with whom he or she is related.
4. Every plant is managed by an employee who works in the same plant.
<u>Remaining Miscellaneous Constraints</u>
1. Each plant has at least three buildings.
2. The salary of an employee cannot exceed the salary of the employee's supervisor.
(Note: Shaded areas represent constraints originally established in Chapter 3.)

**TABLE 4.4** Semantic integrity constraints for Initial Design-Specific ER model for expanded Bearcat Incorporated scenario—Initial (continued)

The next step is to transform the Presentation Layer EER model to the Design-Specific tier. Here, the majority of the mapping has already been done in Chapter 3 and appears in Figure 3.13. One of the incremental mappings pertains to relating IN\_HOUSE\_PROJECT to the gerund entity type ASSIGNMENT in place of PROJECT. Likewise, OUTSOURCED\_PROJECT, the new subclass entity type resulting from the specialization of PROJECT is now related to a new entity type, VENDOR, in an inter-entity class relationship. Figure 4.22 contains the ER diagram for the Design-Specific ER model.



**FIGURE 4.22** Design-Specific EER diagram for Bearcat Incorporated

Because the data type and size of attributes are now captured in the EER diagram, the associated list of semantic integrity constraints accordingly shrinks. The updated semantic integrity constraints are displayed in Table 4.5.

> Constraint	PLANT.Pnumber	IN (10 through 20)
> Constraint	Nametag	IN (1 through 9)
> Constraint	Gender	IN ("M," "F")
> Constraint	Salary	IN (35000 through 90000)
> Constraint	PROJECT.Pnumber	IN (1 through 40)
> Constraint	Plocation	IN ("Bellaire," "Blue Ash," "Mason," "Stafford," "Sugarland")
> Constraint	Bank#	IN (10 through 90)
> Constraint	Account_type	IN ("C," "S," "I")
> Constraint	Io_activity	IN ("I," "O")
> Constraint	Gi_activity	IN ("G," "I")
> Constraint	Related_how	IN ("Spouse") OR
> Constraint	Related_how	IN ("Mother," "Daughter") AND Gender IN ("F")) OR IN ("Father," "Son") AND Gender IN ("M"))
> Constraint	Building	COUNT (not < 3)
> Constraint	No_of_employees	NOT < 100
> Constraint	Status	IN ("A," "I")
> Constraint	Denomination	IN (" ", "Baptist," "Catholic," "Lutheran," "Methodist")
<b>Constraints Carried Forward to Logical Design</b>		
1. An employee cannot be his or her own supervisor. 2. A dependent can have a joint account only with an employee of Bearcat Incorporated with whom he or she is related. 3. Every plant is managed by an employee who works in the same plant. 4. The salary of an employee cannot exceed the salary of the employee's supervisor.		
(Note: Shaded areas represent constraints originally established in Chapter 3.)		

**TABLE 4.5** Semantic integrity constraints for the Design-Specific ER model for expanded Bearcat Incorporated scenario—Final

Similar to the gerund entity type ASSIGNMENT created as a result of the m:n relationship between EMPLOYEE and IN\_HOUSE\_PROJECT, the gerund entity type SUPPORT appears in Figure 4.22 as a result of the decomposition of the m:n relationship between HOBBY and SPONSOR. This makes it possible to tie the support of a certain sponsor of a specific hobby to its two identifying parents (SPONSOR and HOBBY). Furthermore, that an employee cannot be assigned to more than seven project components is captured by the maximum cardinality shown on the edge connecting employee to the *Uses* relationship.

## 4.5 DELETION RULES FOR INTRA-ENTITY CLASS RELATIONSHIPS

As was discussed earlier in this chapter, the basic building block of intra-entity class relationships is the Superclass/subclass (SC/sc) relationship. The essential properties of SC/sc relationships are:

- The cardinality constraint is always 1:1.
- The participation constraint of the subclass (sc) in the relationship is always total—that is, the subclass is existent-dependent on the relationship.

The first property suggests that an SC/sc relationship must have two deletion rules, based on the fact that either one of the participating entity types can play the role of the parent or the child. The second property suggests that when the subclass plays the role of parent, the deletion rule **R** (restrict) is invalid because it will contradict the total participation constraint of the sc in the relationship; and, in the role of a child in the relationship, the subclass cannot have the deletion rule **N** (set null) because the deletion rule will contradict the total participation constraint of the subclass in the relationship.

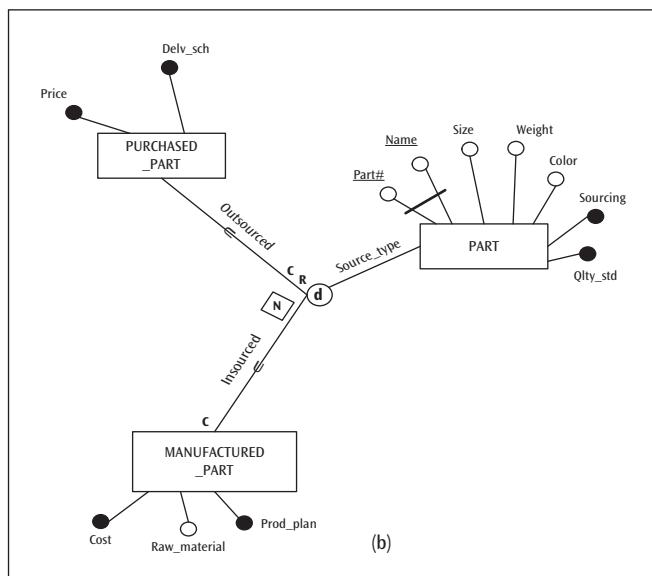
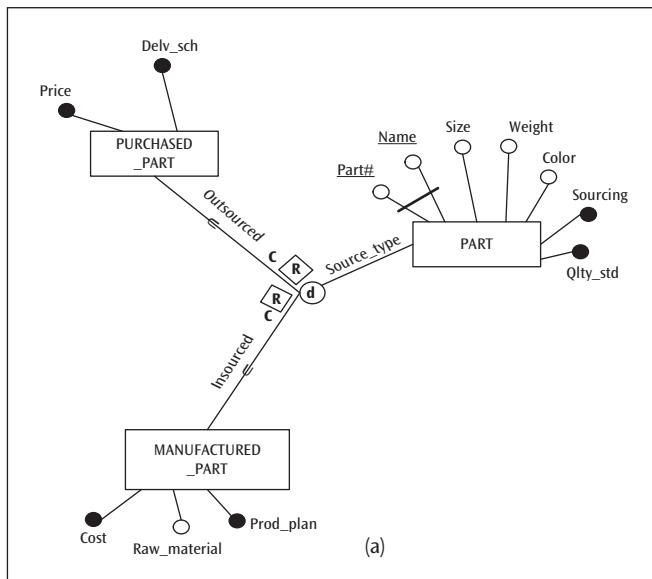
From the perspective of the superclass in the relationship, the deletion rule **R** is certainly valid when the completeness constraint is partial; sometimes, when multiple subclasses are present—particularly in a total specialization, though the deletion rule **R** may work in a technical sense—it is best to avoid this situation. Likewise, when the completeness constraint is total, the deletion rule of **N** is invalid for a SC in the role of a child in the SC/sc relationship.

Finally, the very semantics of the intra-entity class relationship is incompatible with the deletion rule of **D** (set default) and so cannot be used in any SC/sc relationship. In essence, the only deletion rule that is compatible with all the different conditions in the SC/sc relationships of intra-entity class relationships is the cascade (**C**) rule. Thus, a data modeler should consider the deletion rule **C** as the preferred default deletion constraint, even though, under certain conditions, the constraints **R** and **I** are legal. At the same time, indiscriminate use of the deletion constraint **C** can create unexpected results across other SC/sc relationships in the same intra-entity class relationship—be it specialization or categorization. The following examples should clarify the use of deletion rules in intra-entity class relationships.

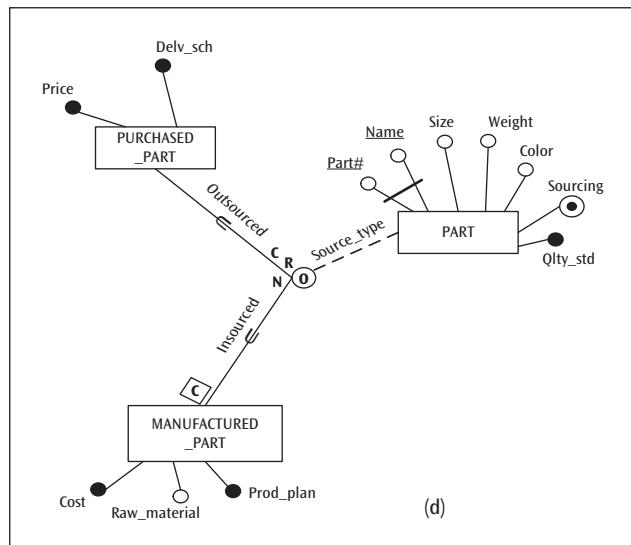
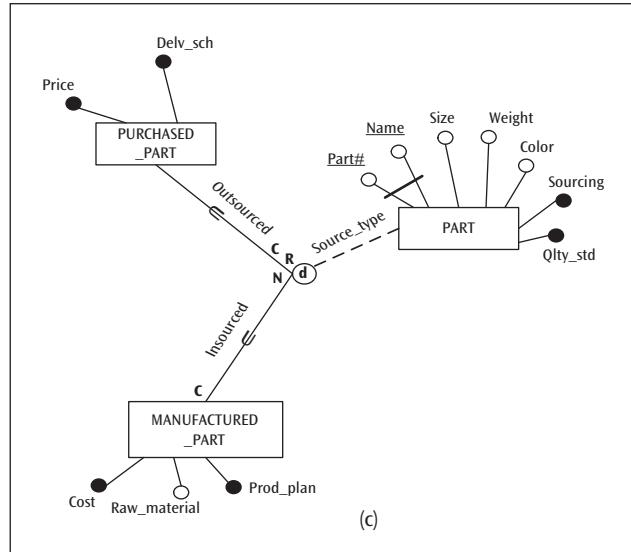
The first example deals with the specialization/generalization construct and demonstrates several alternatives pointing out inadvertent errors that one is prone to commit in the specification of deletion rules/constraints. This is done using a series of five ERDs labeled as Figure 4.23. First, because the SC/sc relationships always have a cardinality constraint of 1:1, two deletion constraints are needed to cover the alternating roles of parent and child of the two participating entity types in each SC/sc relationship.

Figure 4.23a models the SC entity type PART in a total specialization. If its deletion is restricted by the deletion constraint **R** in all the SC/sc relationships in the specialization, a deadlock will occur; that is, any attempt to delete a Part entity will be restricted as long as the Part entity participates in the specialization and all Parts do indeed participate in the specialization because of the total participation constraint on PART and hence the deadlock. The cascade (**C**) deletion constraint on the child entity PART, when the sc is considered in the role of the parent, is clearly a valid specification. Figure 4.23b attempts to

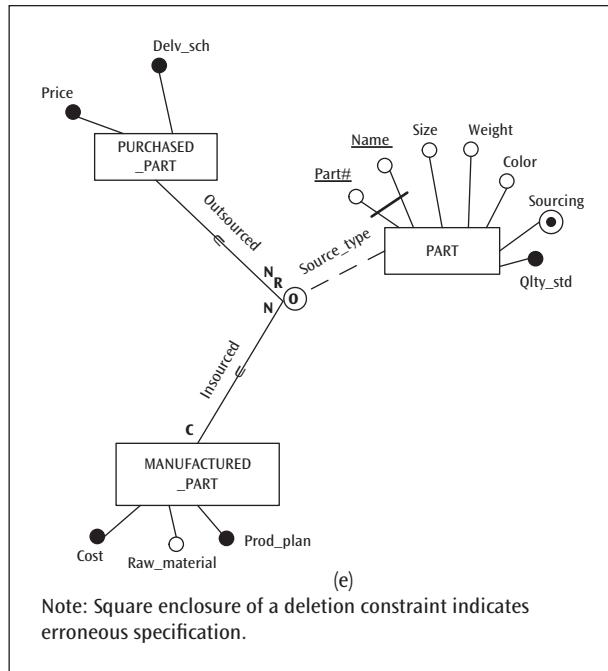
correct this error by revising one of the two restrict (**R**) constraints to a cascade (**C**); note that the cascade constraint is applied on the child entity in this SC/sc relationship. Because this is a disjoint specialization (**d**), as long as at least one part is manufactured, the participation of PART in the relationship with PURCHASED\_PART will be partial. Therefore, the restrict (**R**) constraint becomes legal. The revision of the cascade constraint on PART in the role of child in the relationship with MANUFACTURED\_PART to a “set null” (**N**) is incorrect since the total specialization (solid line on the SC edge) contradicts with the deletion constraint **N**.



**FIGURE 4.23a,b** Deletion rules in Specialization EER construct—Stage 1



**FIGURE 4.23c,d** Deletion rules in Specialization EER construct—Stage 2



**FIGURE 4.23e** Deletion rules in Specialization EER construct—Final

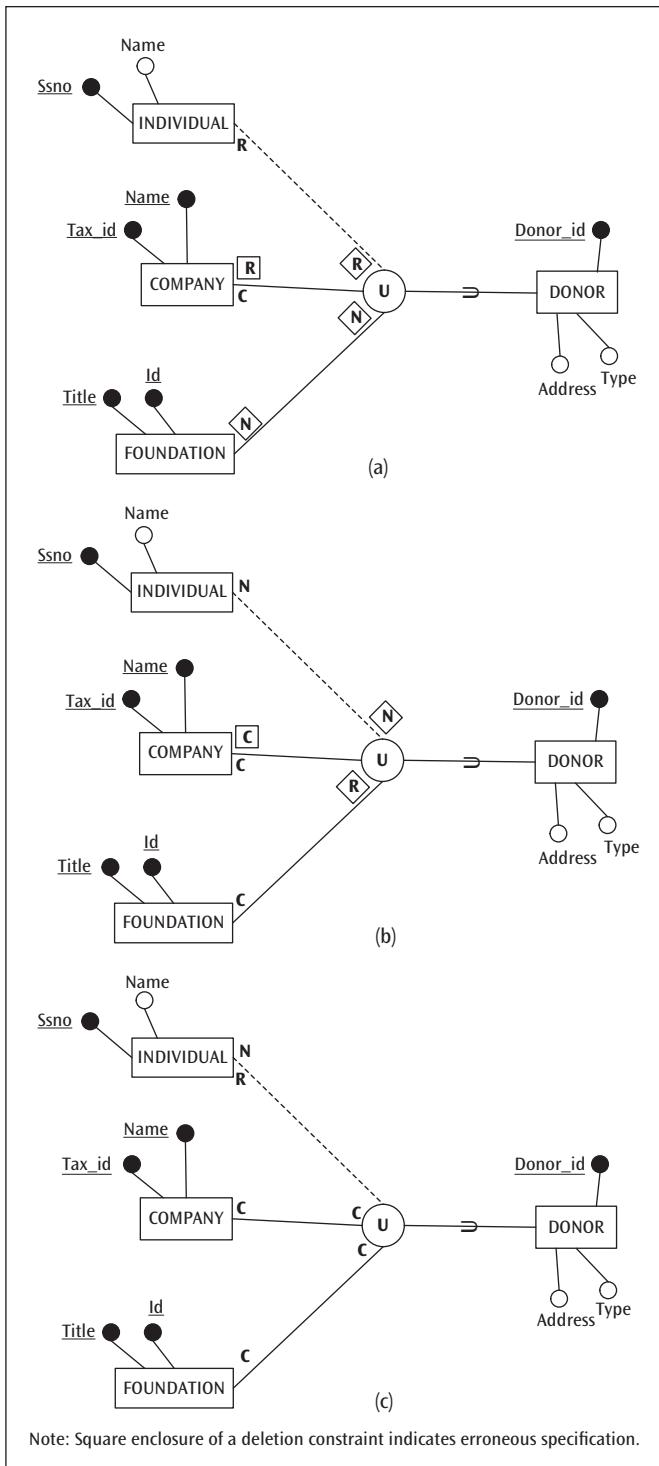
The only revision made in Figure 4.23c that differentiates it from Figure 4.23b is the partial specialization of PART. Because this revision implies that there can be Part entities that are neither manufactured nor purchased, the deletion constraint **N** now becomes legal. Next, what if while some Parts are manufactured and some are purchased, some others are insourced as well as outsourced—that is? manufactured as well as purchased? Perhaps some Parts are large-quantity items and neither source has sufficient capacity to supply the quantity needed. There can be other reasons for such situations; what is important from a data modeling perspective is that the model should be able to handle this condition.

In fact, the overlapping specialization (disjointness constraint = **O**) in the intra-entity class relationship of the EER modeling grammar is specifically intended to serve this need. Note that, in Section 4.1.4.1, the story line for this example presents this model as an attribute-defined specialization where the role of subclass discriminator (defining attribute) is played by the mandatory attribute **Sourcing** in the entity type PART. The overlapping specialization then requires that this attribute be a mandatory multi-valued attribute. These revisions appear in Figure 4.23d. When any revision to an ER/EER model is made, one must carefully review the model for any unintended consequences. The current example (see Figure 4.23d) is a case in point for such an occurrence. Observe that the cascade deletion constraint on MANUFACTURED\_PART was legal in the previous stage (Figure 4.23c); however, it is shown as problematic in Figure 4.23d. That is because, in itself, the cascade rule on the child entity type in the SC/sc relationship depicted by

the defining predicate “insourced” is correct. However, the change in the semantics conveyed through the change in the specialization characteristic from disjoint to overlapping raises an unintended consequence. According to the current model, deletion of a Purchased\_part entity will trigger deletion of the associated part from the PART entity type. This is perfectly alright for a disjoint specialization (Figure 4.23c) because the specific Part entity deleted will not also be a Manufactured\_part. When the specialization is overlapping (Figure 4.23d), the deletion of a Part entity triggered by the deletion of a Purchased\_part entity can further trigger the deletion of the corresponding Manufactured\_part entity if this Part happens to be obtained from both sources. What if the intention was to only stop outsourcing the specific Part and continue insourcing it? Then, an inadvertent error has occurred by the automatic deletion of the corresponding Manufactured\_part due to the cascade deletion constraint imposed on that sc entity type. That is why, in Figure 4.23d, the deletion constraint **C** on MANUFACTURED\_PART is labeled an error. The revision of the cascade deletion constraint to the “set null” deletion constraint on PART in the SC/sc relationship defined by the defining predicate “outsourced” remedies this error (Figure 4.23e). It must be noted that this correction is possible only because

the specialization is partial; if this is total, an alternative solution must be sought.

The second example pertaining to the role of deletion constraints in the intra-entity class relationship types demonstrates the incorporation of deletion rules for the EER construct “Categorization.” As a quick review, categorization entails an intra-entity class relationship where a sc *is-a* subset of the union of one or more SCs (for a detailed review, see Section 4.1.6). Figure 4.24 is an excerpt from an example introduced in Section 4.1.6 (Figure 4.10). The example here (Figure 4.24a) is a partial categorization since all entities of the entity type INDIVIDUAL are not Donors. The “category” DONOR being a subclass in this relationship, by definition it always has total participation in each of the SC/sc relationships in the categorization. Thus, the “set null” (**N**) specification on FOUNDATION end of the edge conflicts with the participation constraint of DONOR in this relationship. Likewise, since FOUNDATION also exhibits total participation (solid edge) in this relationship, the deletion constraint **N** on the DONOR end of the edge is not valid either. Next, since all Company entities are Donors (solid edge implying total participation), the deletion rule **R** on COMPANY will create a deadlock in that a deletion action cannot be directly initiated on COMPANY. The deletion of a Donor entity triggering the deletion of the related Company entity through the deletion rule **C** is perfectly legal. Once again, the deletion rule **R** on INDIVIDUAL is compatible with the partial participation of INDIVIDUAL in the relationship. On the same grounds, the total participation of DONOR in the relationship conflicts with the deletion constraint **R** on DONOR in this SC/sc relationship. The reader should now be able to interpret the revisions shown in Figure 4.24b and the associated errors identified and the final solution shown in Figure 4.24c.

**FIGURE 4.24a,b,c** Deletion rules in categorization EER construct

## Chapter Summary

---

With the advent of complex database applications, the constructs available in the entity-relationship (ER) modeling grammar were found inadequate to fully capture the richness of the conceptual design. The enhanced entity-relationship (EER) modeling grammar extends the original ER modeling grammar to include a few new constructs: specialization/generalization, specialization hierarchies and lattices, categorization, and aggregation. Each of these new constructs can be displayed in an EER diagram.

A fundamental unit of these intra-entity class relationships is the Superclass/subclass (SC/sc) relationship where the superclass represents a generic entity type for a group of entity types (subclasses). Since the generic entity type subsumes the group, it is also referred to as an entity class.

Specialization and generalization can be viewed as two sides of the same coin. Specialization involves the generation of subgroups (i.e., subclasses) of a generic entity class by specifying the distinguishing attributes of the subgroups, whereas generalization consolidates common attributes shared by a set of entity types into a generic entity type. Two main constraints apply to specialization/generalization: the completeness constraint, which can be total or partial, and the disjointness constraint, which can be disjoint or overlapping.

A specialization/generalization can take the form of a hierarchy or a lattice. In a specialization hierarchy, an entity type can participate as a subclass in only one specialization, whereas in a specialization lattice an entity type can participate as a subclass in more than one specialization. A category allows for the modeling of a situation where a subclass can be a subset of the union of several superclasses. A constraint imposed by the use of a category is that an entity that is a member of a category (subclass) must exist in only one of its superclasses. On the other hand, aggregation allows for the relaxation of this constraint and requires that an entity that is a member of an aggregate must exist in all of its superclasses.

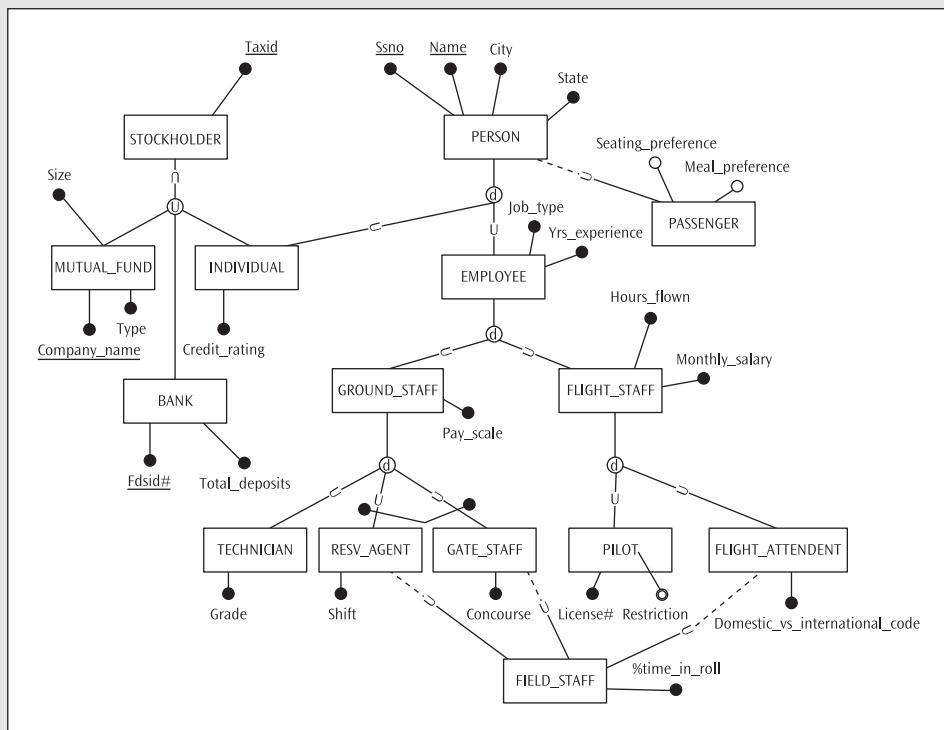
Finally, the incorporation of deletion rules in intra-entity class relationships is articulated.

## Exercises

---

1. What is the difference between an inter-entity class relationship and an intra-entity class relationship?
2. What is a subclass and when is a subclass used in data modeling?
3. Under what circumstances in a specialization is it possible for one superclass to be related to more than one subclass and one subclass to be related to one or more superclasses?
4. What is the type inheritance property?
5. What is the difference between specialization and generalization? Why is this difference not reflected in ER diagrams?
6. What is the difference between total specialization and partial specialization, and how is each reflected in an ER diagram?
7. What is the difference between a specialization hierarchy and a specialization lattice?
8. How does categorization differ from specialization?
9. What is the difference between a total category and a partial category?
10. In categorization, what is meant by the property of selective type inheritance?

11. Explain how a total category and specialization are mutually substitutable constructs.
12. Contrast categorization and aggregation.
13. Consider the following Presentation Layer EER diagram.

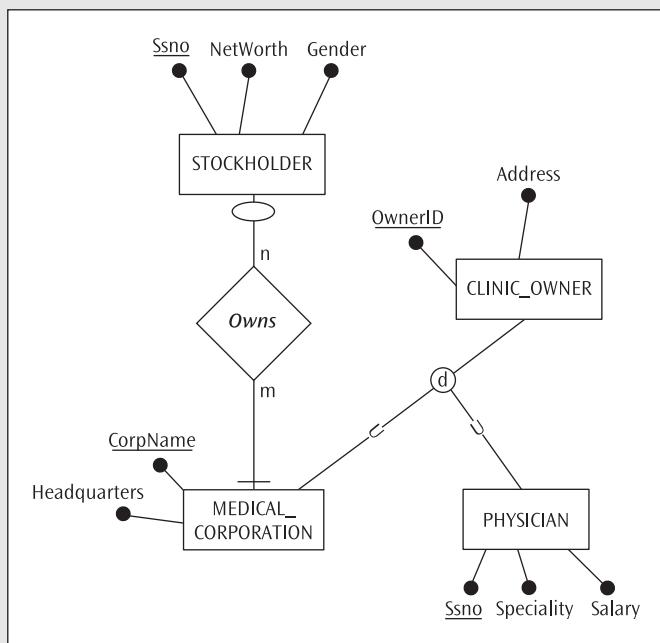


189

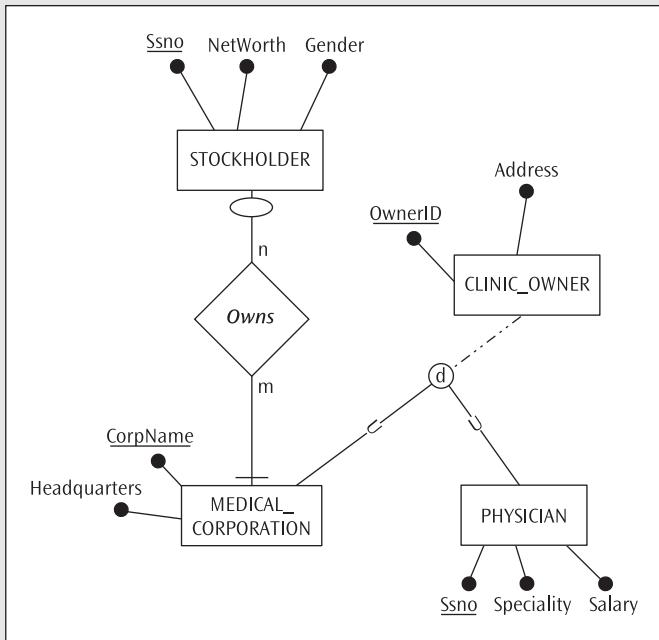
Presentation Layer EER diagram for Exercise 13

- a. Identify all entity types that at one time or another function as a superclass.
- b. List the superclass entity type and subclass entity type(s) in each distinct specialization.
- c. Identify an entity type that functions as a shared subclass. What type of specialization does this entity type reflect?
- d. Which entity types comprise the entire specialization hierarchy? For each level in this specialization hierarchy, define which entity types serve as a superclass and which serve as a subclass.
- e. Which entity type functions as a superclass in more than one specialization?
- f. How many Superclass/subclass(SC/sc) relationships appear in the ER diagram?
- g. How many specializations appear in the ER diagram?
- h. Which entity types in the diagram form a category?
- i. What is the meaning of the arc in the diagram?
- j. List the attributes inherited by the entity type **RESV\_AGENT**.

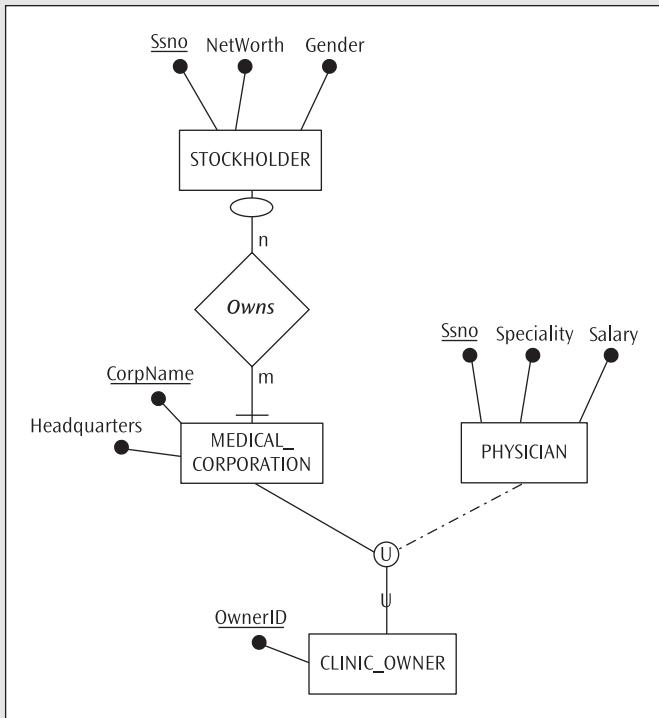
- k. List the attributes inherited by the entity type INDIVIDUAL.
- l. List the attributes inherited by FIELD\_STAFF.
- m. Should all members of the GATE\_STAFF also be members of FIELD\_STAFF? If the answer is no, what modification must be made to EERD to enforce such a business rule?
- n. List the attributes inherited by MUTUAL\_FUND.
- o. Which, if any, entity type(s) function as both a superclass and a category?
- p. What is the identifier of the entity type MUTUAL\_FUND?
- q. Which entity type(s) *require(s)* the use of a surrogate key as an identifier?
- r. Which entity types(s) function(s) as a superclass in one SC/sc relationship and as a subclass in another SC/sc relationship?
- s. Is it possible for an individual stockholder to be an employee of the airline company? Explain your answer.
- t. Suppose it is possible for an employee to own shares in a mutual fund that is an owner of the airline. Modify the ER diagram to incorporate this enhancement.
14. Revise Figure 4.10 to allow (i.e., require) each individual to be a donor. Show how your revision allows the categorization construct to be replaced by the specialization/generalization construct. Explain why is it not possible to make a similar revision to the original version of Figure 4.10.
15. This exercise is based on the three Presentation Layer ER diagrams that follow.



Presentation Layer EER diagram number 1 for Exercise 15



Presentation Layer EER diagram number 2 for Exercise 15

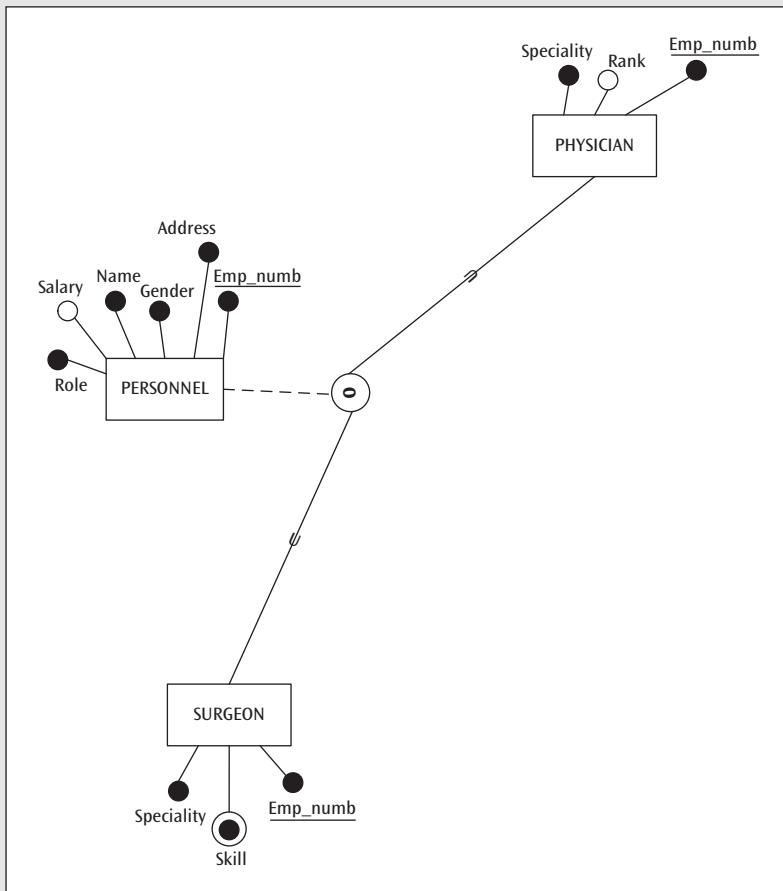


Presentation Layer EER diagram number 3 for Exercise 15

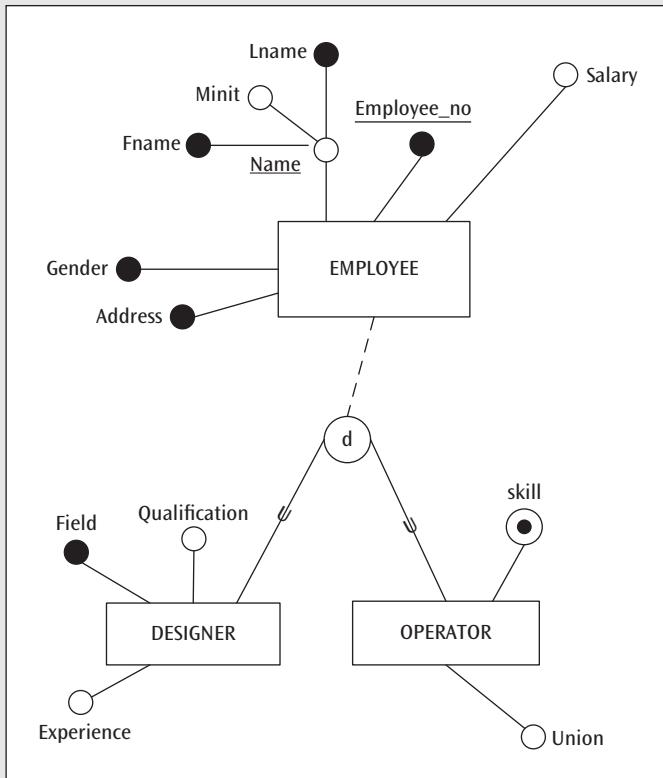
- a. Describe what is reflected by the entity types and relationship types in Presentation Layer ER diagram number 1. In other words, please tell the basic facts of the story. An explanation of the various attributes need not be part of your story.
  - b. How do the basic facts of the story change with the change noted in Presentation Layer ER diagram number 2?
  - c. How many data sets would be required to illustrate the nature of the information requirements in Presentation Layer ER diagram number 1?
  - d. What would be the name and attributes associated with each data set identified in question (c)?
  - e. Using Presentation Layer ER diagram number 1, assume there are four clinic owners, at least one of which is a physician and at least one of which is a medical corporation. How many entities (instance of each entity type) would appear in the MEDICAL\_Corporation, PHYSICIAN, and CLINIC\_OWNER data sets? It is possible that there may be more than one correct answer to this question.
  - f. How would your answer to question (e) change if it were based on illustrating the difference between Presentation Layer ER diagram number 1 and Presentation Layer ER diagram number 2? Please justify your answer.
  - g. Discuss how the basic facts of the story change if the story is based on Presentation Layer ER diagram number 3.
16. Consider the Presentation Layer ER diagram that appears in Figure 4.21.
    - a. What makes SCHOOL part of the specialization lattice involving SCHOOL, NOT\_FOR\_PROFIT\_ORGANIZATION, and PUBLIC\_SCHOOL and, at the same time, part of the SPONSOR category that involves CHURCH, SCHOOL, and INDIVIDUAL? What role (i.e., subclass, superclass, category, aggregate) do SCHOOL and NOT\_FOR\_PROFIT\_ORGANIZATION serve in each of these structures?
    - b. Does SPONSOR take the form of a total category or a partial category?
    - c. Is it possible to redraw the SPONSOR category as a specialization and still retain the participation of SCHOOL in the specialization? If the answer is "yes," redraw this portion of Figure 4.21.
  17. This exercise contains additional information to the information given in Exercises 17 and 18 in Chapter 3 in order to give you an opportunity to work with various types of enhanced ER modeling constructs.
    - Both coaches and referees are basketball professionals who have chosen different careers within this profession.
    - Players no longer serve as counselors in summer youth basketball camps. Instead, some of the players and all of the coaches do voluntary service as trainers in the summer youth basketball camps.

Incorporate this additional information into the Presentation Layer ER diagram that you developed for Exercise 17 in Chapter 3.

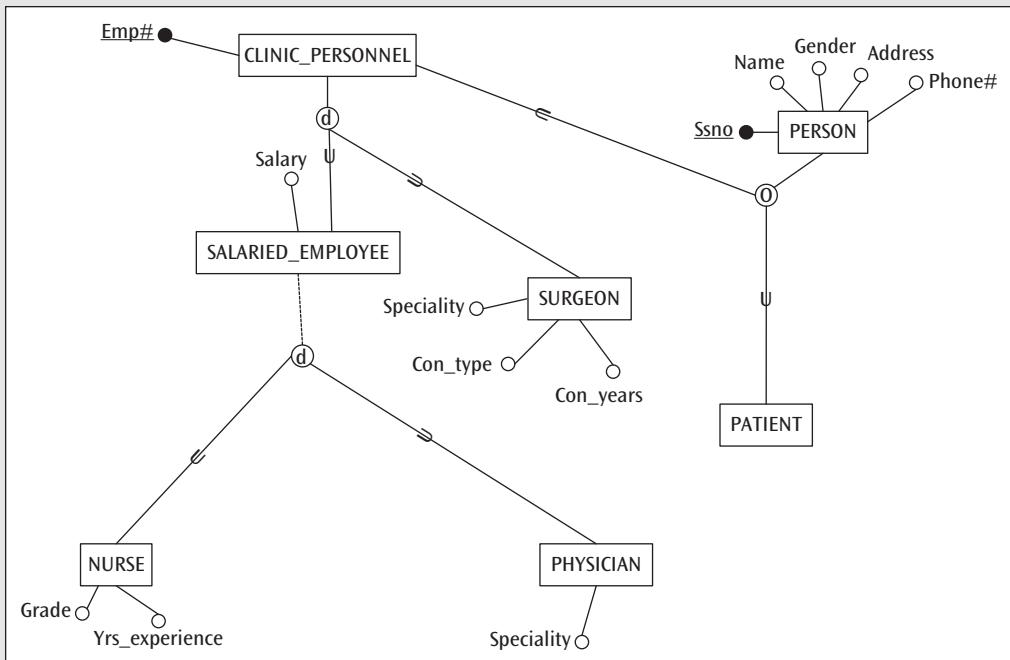
18. Develop valid deletion rules and incorporate the associated deletion constraints in the EERDs shown next. Then explain the meaning of the incorporated deletion constraints, clarifying the absence of errors in your specifications.



Presentation Layer EER diagram number 1 for Exercise 18



Presentation Layer EER diagram number 2 for Exercise 18



Presentation Layer EER diagram number 3 for Exercise 18

19. Develop valid deletion rules and incorporate the associated deletion constraints for the EERD in diagram 3 of Exercise 15. Then explain the meaning of the incorporated deletion constraints, clarifying the absence of errors in your specifications.



# CHAPTER 5

# MODELING COMPLEX RELATIONSHIPS

The ER modeling grammar for conceptual data modeling presented in Chapters 2, 3, and 4 can portray many rich circumstances that represent business scenarios. Yet these designs are insufficient to fully express some of the intricate yet real data relationships in an enterprise. Although it is practically impossible to express *all* business rules of an application in an ER diagram (ERD), a few additional ER modeling constructs are available and can be used to model more business rules. In addition, existing constructs can be used in interesting ways to incorporate certain business rules in the ERD instead of relegating the business rules to additional textual semantic constraints of the ER model. This chapter presents several application scenarios as vignettes and demonstrates how the business rules pertaining to these scenarios can be modeled in an ERD using advanced modeling techniques.

This chapter begins with a presentation of the ternary relationship type. Two vignettes depicting real-world examples are used in Section 5.1 to describe the need for and use of relationships of degree three. Section 5.2 extends this discussion to examine relationship types beyond degree three. This is done by imposing additional constraints on the scenario of the first two vignettes supplemented by vignettes 3 and 4. The weak relationship type and its practical applications are introduced in Section 5.3. In Section 5.4, innovative use of weak relationship types is presented through composites of weak relationship types that reflect inclusion and exclusion dependencies pertaining to relationship types. Following the format of Chapters 3 and 4, Section 5.5 discusses how to decompose some of the complex relationship types preparatory to logical schema mapping in the next step. Chapter 5 concludes with two important sections. Section 5.6 addresses the need for the conceptual modeling process to include a careful validation of the semantics captured in an ERD. Using revisions to vignettes introduced previously in the chapter, semantic errors caused by the misinterpretation of relationships are identified. In cases where semantic errors are of significance in the context of the requirements specification, alternatives for restructuring the ERD are discussed. Section 5.7 begins with a narrative of a new story for a small medical clinic, once again seeking to model a real-world scenario. The rest of the section is devoted to the development of a Presentation Layer ERD as well as a Design-Specific ERD for the medical clinic, Cougar Medical Associates.

## 5.1 THE TERNARY RELATIONSHIP TYPE

In Chapter 2, we learned that the degree of a relationship is the number of entity types that participate in a relationship type. Accordingly, a **binary relationship** and a **ternary relationship** indicate a degree of two and three, respectively.

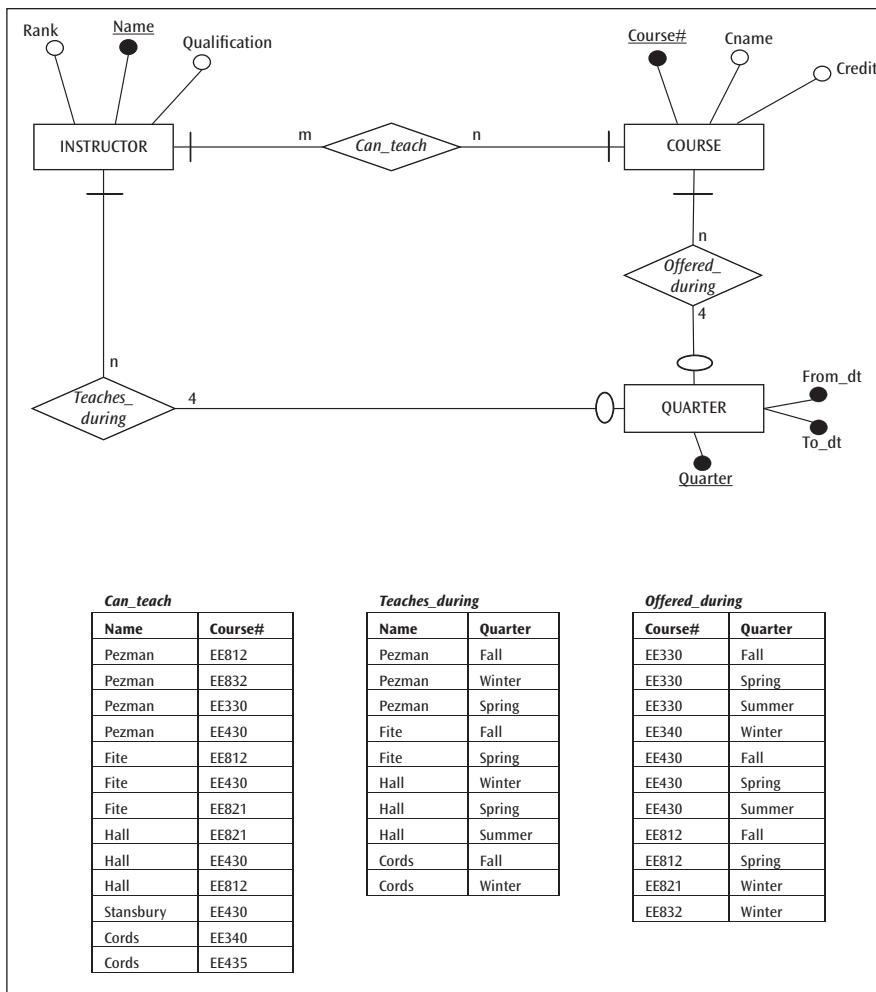
It is sometimes argued that relationship types beyond degree two (binary) are rarely found in real-world applications and thus are not crucial for data modeling and database design. This argument is far from the realities of the business world. Costly errors of expression may occur in database design when a genuinely ternary relationship is inadvertently expressed as a set of binary relationships among the three entity types taken two at a time. Likewise, combining independent binary relationships to a ternary (or n-ary) relationship is also problematic. Contemporary CASE tools tend to sacrifice expressive power in database design by not providing for relationships beyond degree two. As a consequence, designers often are not afforded an opportunity to consider relationships beyond degree two during the data modeling and database design phases. In this section, several examples are presented to illustrate the value of ternary relationship types in conceptual data modeling.

### 5.1.1 Vignette 1—Madeira College

*Madeira College offers many courses, and a college term is divided into four quarters—fall, winter, spring, and summer—during which one or more of these courses may be offered. Every quarter, at least one course is offered. The college also has several instructors. Often, not all instructors teach during all quarters. Furthermore, instructors are capable of teaching a variety of courses that the college offers. Likewise, at least one, but often more than one, instructor can teach a specific course. A course may be offered during some or all quarters; some courses may not be offered at all. Finer specifications, such as the minimum number of instructors teaching in a quarter, the minimum number of quarters in which an instructor teaches, and so on could be stated. For now, let us just say that at least one instructor teaches per quarter and that some instructors may not teach any course in any quarter—they may be doing research.*

A representation of this vignette as a Presentation Layer ERD is shown in Figure 5.1, along with sample data sets.

Is it possible to infer from this ERD which instructor teaches what course during which quarter? The answer is “no” from a semantic perspective. Semantically, the information on what an instructor “can teach” can never be used to infer what an instructor actually teaches. Just because an instructor is capable of teaching a course does not mean that he or she indeed teaches that particular course. For example (using the data sets for *Can\_teach*, *Teaches\_during*, and *Offered\_during*), since both Pezman and Fite can teach EE812, since both teach during the Fall and Spring quarters, and since EE812 is offered during the Fall and Spring quarters, the inference is that both Pezman and Fite teach EE812 during the Fall and Spring quarters. However, suppose (as per the vignette’s story line) Pezman actually teaches EE812 only during the Spring quarter while Fite teaches EE812 only during the Fall quarter.

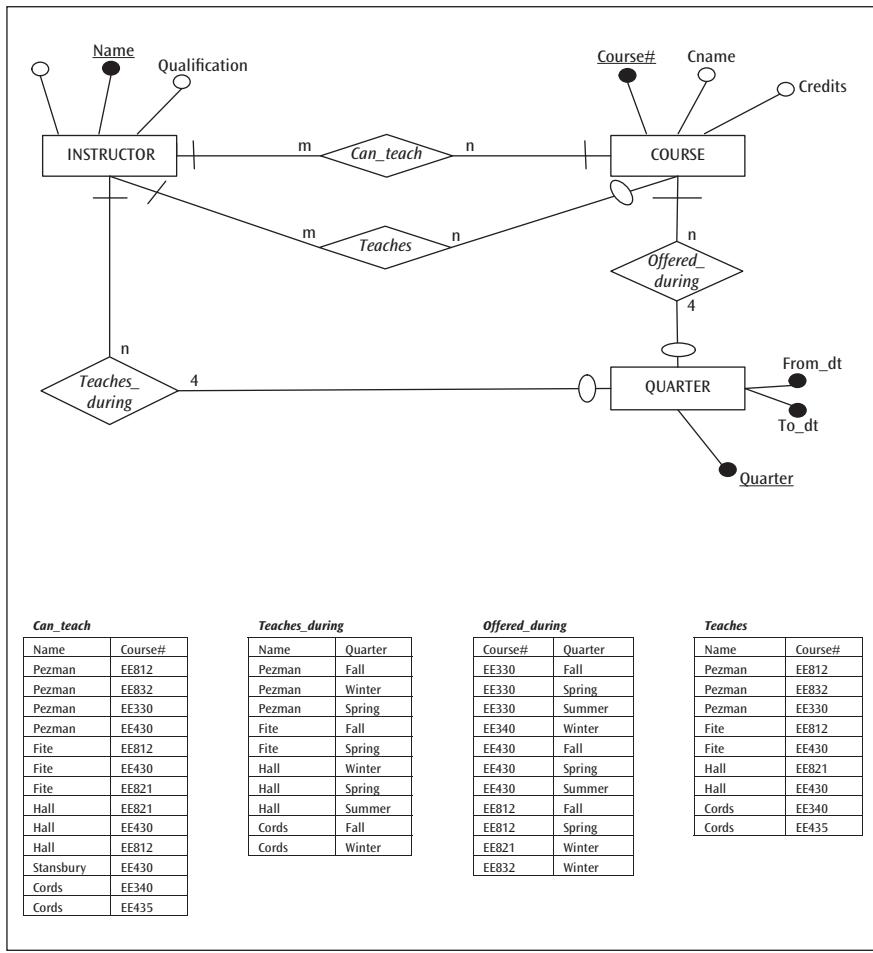


© 2015 Cengage Learning®

**FIGURE 5.1** An initial ERD and sample data sets for vignette 1

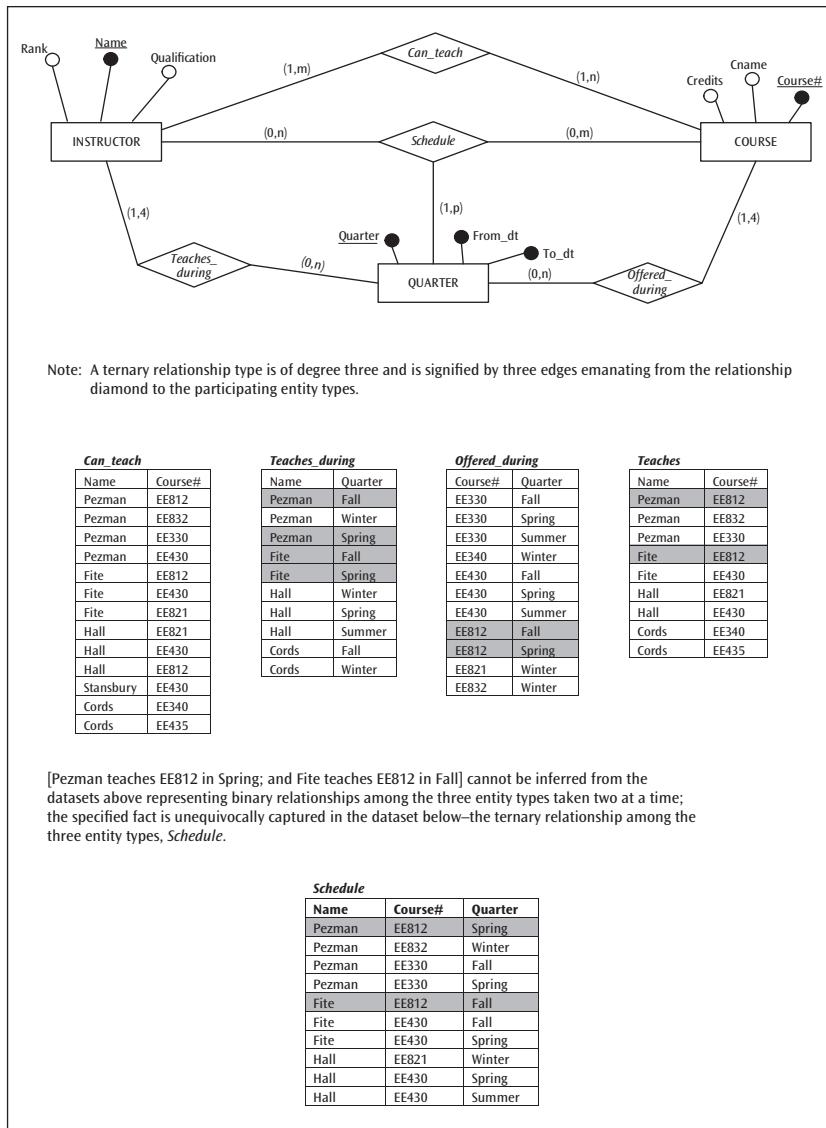
Now, what if we add another binary relationship, *Teaches*, between INSTRUCTOR and COURSE and that *Teaches* captures the courses an instructor actually teaches irrespective of what he or she is capable of teaching (see Figure 5.2)? Is it now possible to infer which instructor teaches what course during which quarter? The answer now is “not always.” For example (using the data sets for *Teaches*, *Teaches\_during*, and *Offered\_during* shown in Figure 5.2), it is still not possible to determine that Pezman actually teaches EE812 *only* during the Spring quarter whereas Fite teaches the course *only* during the Fall quarter. There are indeed times when one may be able to infer the course/teaching schedule with certainty from the three data sets, *but not always*. For example, had there been a business rule requiring that an instructor who teaches a course must be

teaching that course if it is offered during the quarter when he or she teaches, then both Pezman and Fite would be scheduled to teach EE812 in both Fall and Spring. This result can, in fact, be derived from the three binary relationship types *Teaches*, *Teaches\_during*, and *Offered\_during* that are shown in Figure 5.2. Absent such a business rule, it is possible, as stated in the story line, for Pezman to teach EE812 in the Spring quarter and for Fite to teach EE812 in the Fall quarter; this can never be derived from the three binary relationship types *Teaches*, *Teaches\_during*, and *Offered\_during* that are shown in Figure 5.2.



**FIGURE 5.2** A second ERD and sample data sets for vignette 1

What is the solution to this problem? When can one unequivocally infer who teaches what and when? It is possible to capture this condition precisely via a ternary relationship type among INSTRUCTOR, COURSE, and QUARTER. This relationship type is shown as *Schedule* in Figure 5.3, with a supporting sample data set.



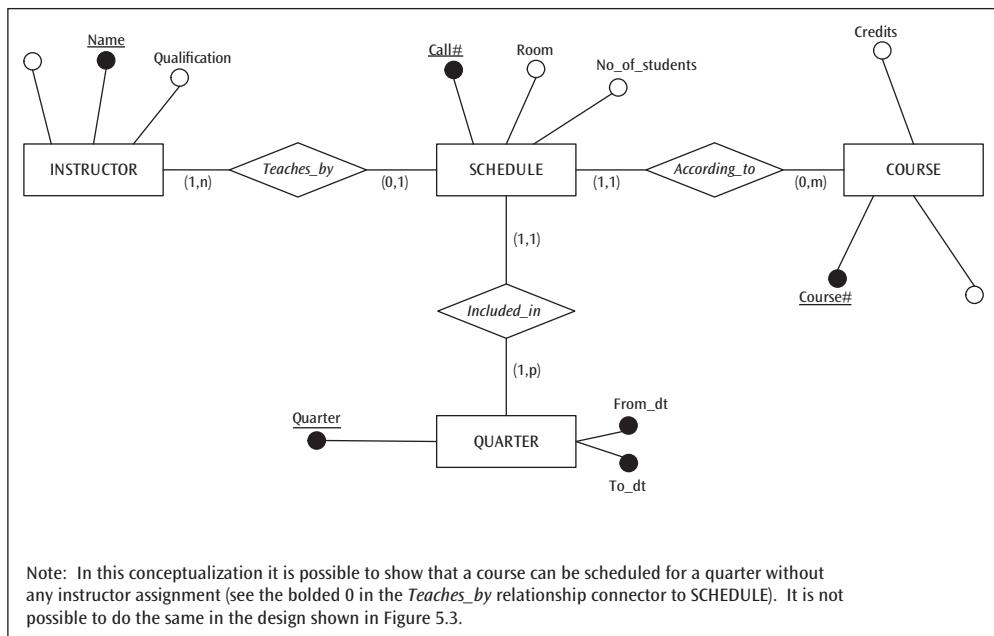
**FIGURE 5.3** The ternary relationship type *Schedule* and associated sample data set

Note that ER modeling grammars that use the “look across” notation (Chen’s notation employed in the Presentation Layer ERD) to express a cardinality ratio cannot accurately capture the cardinality ratio of any relationship type beyond degree two. In a binary relationship type, there is only one entity type present when looking across the relationship type. For instance, in Figure 5.1, it is possible to state unambiguously that one instructor entity is related to a maximum of n course entities. However, looking across from the **INSTRUCTOR** in a ternary relationship type *Schedule* (Figure 5.3), we see both **COURSE** and **QUARTER**, and the maximum cardinality should actually reflect the maximum

combination of courses and quarters per instructor. The (min, max) notation, which uses the “look near” approach to express the structural constraints of a relationship type, is able to express the maximum cardinality precisely in a relationship type of any degree. For this reason the (min, max) notation is used here to specify the structural constraints of relationship types beyond degree two; in the interest of consistency, the same notation will be used for recursive and binary relationship types henceforth.

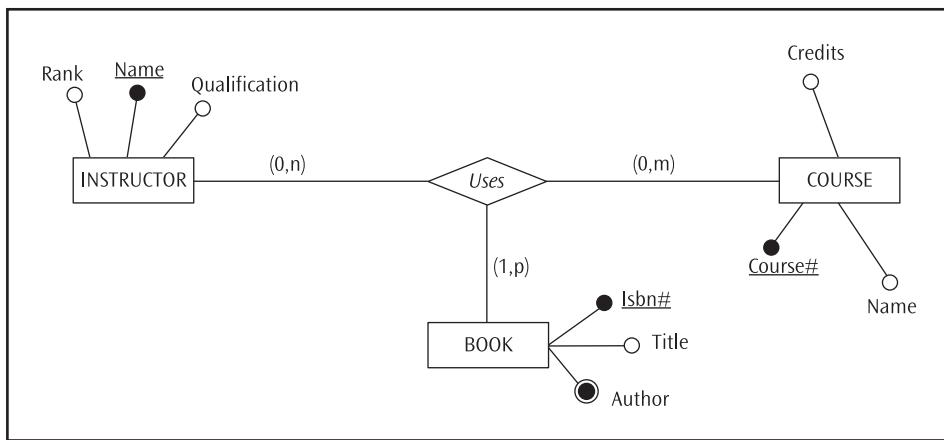
In the ERD that appears in Figure 5.3, the (0, n) on the edge labeled “Teaches\_by” indicates that an instructor need not teach any course in any quarter (0 for the min value) and that an instructor may teach up to n {course, quarter} pairs (n for the max value). For example, using the data sets in Figure 5.3, while Stansbury can teach EE430 (see the *Can\_teach* data set), the data set *Schedule* indicates (by his absence) that he is not scheduled to teach a course during any of the four quarters. Likewise, Cords can teach EE435 but is not scheduled to teach a course during any of the four quarters either. Hall, on the other hand, who can teach three different courses, is scheduled to teach two of them. Looking at the (1, p) on the “Included\_in” edge associated with the ternary relationship type *Schedule* reveals that a quarter must be related to at least one {instructor, course} pair (1 for the min value) and that a quarter may be related to up to p {instructor, course} pairs (p for the max value). The data set *Schedule* in Figure 5.3 indicates that there is at least one {instructor, course} pair related to each quarter and that each quarter has one or more instructors teaching one or more courses.

One can also argue that *Schedule* can be conceptualized as a base entity type and, in effect, preempt the need for formulating a ternary relationship type. The ERD in Figure 5.4 models this viewpoint. This is a valid argument since *SCHEDULE* in Figure 5.4 lends itself to conceptualization as a base entity type with its own unique identifier, (**Call#**).



**FIGURE 5.4** Modeling the ternary relationship type *Schedule* of Figure 5.3 as a base entity type

However, there are situations where the conceptualization of a ternary relationship as a base entity type is not semantically obvious. For instance, consider possible relationships among INSTRUCTOR, COURSE, and BOOK. It is quite conceivable that one may need to know which instructor uses what book for which course. After all, one instructor may use a certain book for a course and a different instructor may teach the same course using a different book, and so on. While binary relationships among the three entity types taken two at a time are meaningful, they cannot even collectively capture the semantics stated above. In this case, a ternary relationship type *Uses* among INSTRUCTOR, COURSE, and BOOK becomes genuinely necessary. However, a base entity type to preempt this ternary relationship is not semantically obvious (see Figure 5.5), essentially demonstrating that a ternary relationship type is not just a syntactically valid construct, it is a semantically valuable construct in the ER modeling grammar.



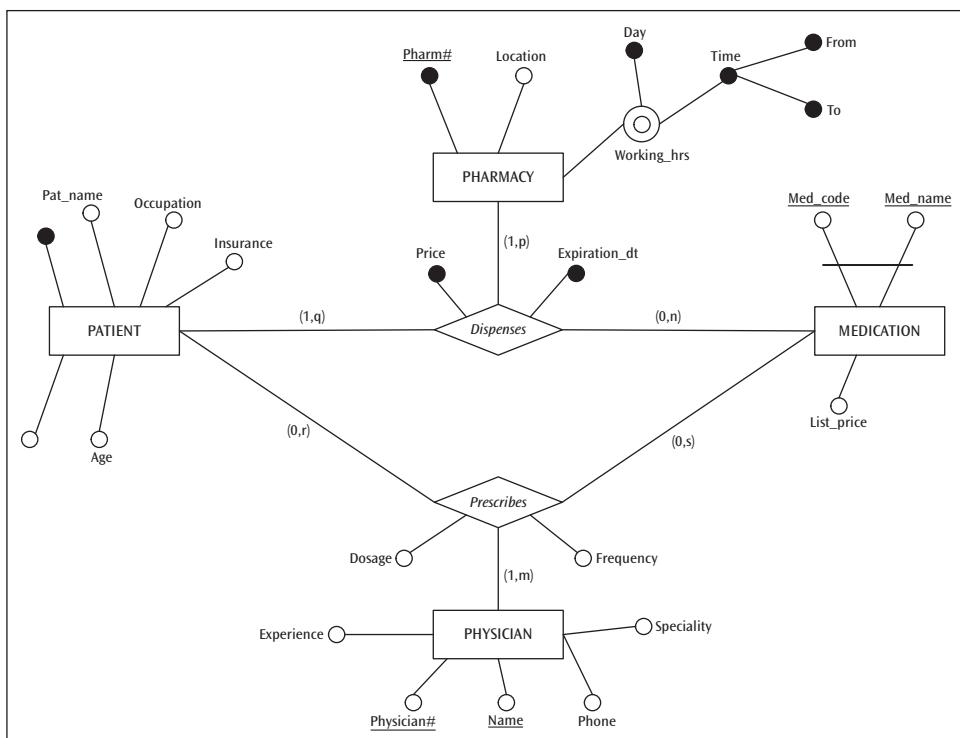
**FIGURE 5.5** The ternary relationship type *Uses*

Let us now examine another scenario to signify the utility of a ternary relationship.

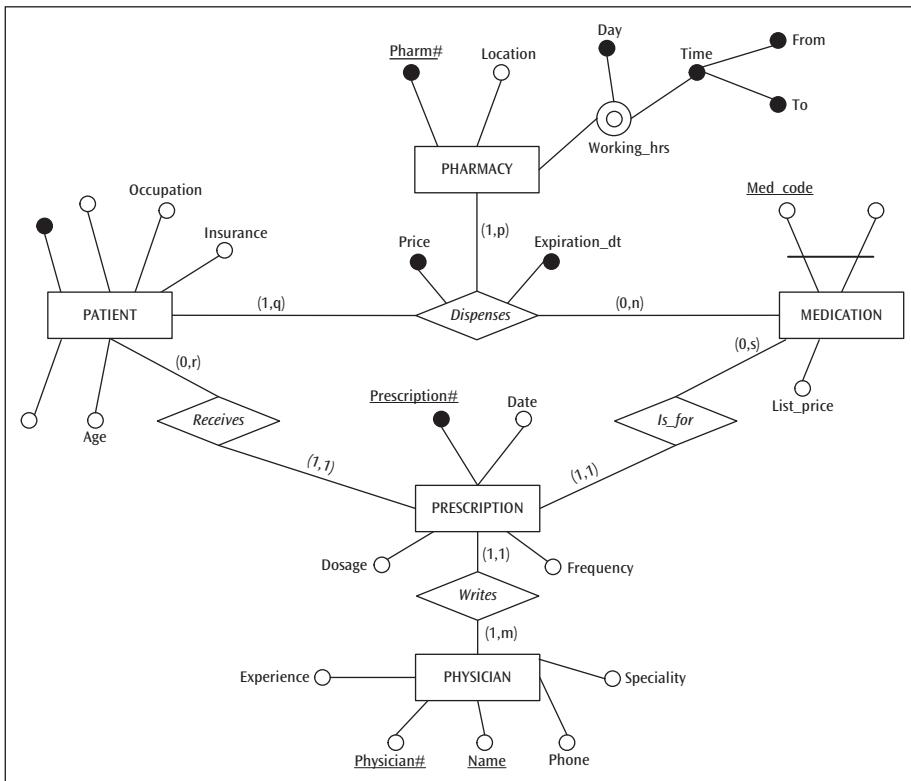
### 5.1.2 Vignette 2—Get Well Pharmacists, Inc.

Get Well Pharmacists, Inc. has numerous pharmacies across the state of Ohio. A pharmacy dispenses medication to patients. It is imperative that the records at Get Well, Inc. should always have the data on which of its pharmacies dispensed what medication to which patient. In addition, every pharmacy stocks numerous different medicines and the same medicine is carried in several pharmacies. A patient often takes one or more medicines, and the fact that the same medicine may be used by at least one and often many patients can also be a meaningful relationship. Finally, a pharmacy typically has one or more patients as customers, and some patients use one or more pharmacies. To make the story a little more interesting, let us impose the additional business rule (V2R1) that a particular physician prescribes a certain medication to a specific patient.

At the outset, it is seen that a series of binary relationships as well as a ternary relationship exist among the PHARMACY, PATIENT, and MEDICATION entity types. It is not always possible to capture the import of the business rule V2R1 by specifying binary relationships among PATIENT, MEDICATION, and PHYSICIAN taken two at a time. But it is always possible with certainty for a ternary relationship among the three to precisely frame this relationship (V2R1). Figure 5.6 shows two ternary relationship types: *Dispenses* among PHARMACY—PATIENT—MEDICATION and *Prescribes* among PHYSICIAN—PATIENT—MEDICATION. Once again, it is conceivable that the ternary relationship *Prescribes* lends itself to be conceptualized as a base entity type with the unique identifier **Prescription#**, thus possibly preempting the need for the ternary relationship. However, the same is not true for *Dispenses*. This alternative appears in Figure 5.7.



**FIGURE 5.6** Two ternary relationship types *Dispenses* and *Prescribes*



**FIGURE 5.7** Representing the *Prescribes* relationship type as the PRESCRIPTION base entity type

## 5.2 BEYOND THE TERNARY RELATIONSHIP TYPE

Sometimes, business rules defining real-world occurrences spontaneously convey relationships even beyond degree three. Although the frequency of such occurrences may not be high, a data modeler/database designer must be sensitive to these possibilities lest such occurrences go unnoticed and compromise the richness of expression of a data model. The question of implementability is a different issue and should not erode the expressive power of a conceptual data model, which is supposed to be technology-independent.

### 5.2.1 The Case for a Cluster Entity Type

Entity clustering as an ER modeling construct is a useful abstraction to present an ERD at a broader level of conceptualization. The technique incorporates object clustering ideas to produce a bottom-up consolidation of natural groupings of entity types. A **cluster entity**

type literally emerges as a result of a grouping operation on a collection of entity types and relationship(s) among them. Some database design applications lend themselves to a repeated roll-up to higher levels of abstraction through clustering and provide an opportunity to conceptualize a layered set of ERDs. This can be especially useful for large database design projects where the different layers within the Presentation Layer ERD can be used to inform the end-user hierarchy (for example, executives, managers, and staff). Even for an office staff working at the detail level of a business, a layered presentation can facilitate quicker understanding of the semantics captured by the overall design when presented through a cascading set of ERDs. Cluster entity type has been introduced, though briefly, in Chapter 2 (Section 2.3.6). The rest of the section 5.2 illustrates the varieties of deployment of this valuable construct in ER modeling.

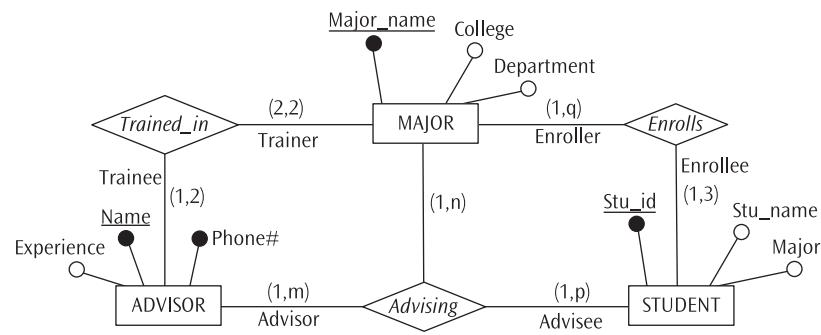
### 5.2.2 Vignette 3—More on Madeira College

*Madeira College, introduced in Vignette 1, has students as well. Students must declare a major field of study and in fact can enroll in up to three majors. Every major has some students. The advising office of the college has staff specially trained in advising in each major. An advisor is restricted to two majors by training. Every major has exactly two trained advisors. A student can have multiple advisors for each major he or she enrolls in but must have at least one advisor per major. Therefore, it is imperative that information about which advisor advises which student for what major is recorded.*

Notice in the ERD of Figure 5.8a that in order to capture all aspects of this scenario, a ternary relationship type *Advising* among ADVISOR, STUDENT, and MAJOR is necessary but not sufficient. This is why two additional binary relationship types, *Trained\_in* between ADVISOR and MAJOR and *Enrolls* between STUDENT and MAJOR, are included. There is no story line that requires the presence of a binary relationship type between ADVISOR and STUDENT, nor can such a relationship type, collectively with the other two binary relationship types, replace the ternary relationship type *Advising* as demonstrated through the previous two examples. The data set in Figure 5.8b contains representative data for the *Advising* relationship.

Suppose we introduce a new business rule (Rule V3R1) in vignette 3: *An advisor may advise a student for only one major.*

Will this be accomplished if the maximum cardinality of the edge “Advisee” is changed from a **p** to a **1**? The answer here is “yes,” but a “yes” with an unintended side effect. The maximum cardinality of 1 on the edge “Advisee” (see Figure 5.9) does indeed limit an advisor to advising a student for only one major; but it also limits the relationship of a student entity to no more than one {major, advisor} pair. This essentially implies that a student may have up to three majors (as indicated on the edge “Enrollee”) but no advisor for more than one major—in fact, no more than one advisor, period. The data set in Figure 5.9 shows the effect of restricting a student to one {major, advisor} pair. The reader should note that changing the maximum cardinality of the “Advisor” edge or the “Advised\_for” edge to a **1** instead of the “Advisee” edge (as discussed above) also does not yield a correct solution to the implementation of Rule V3R1 in the ERD.



ER diagram for the scenario described in vignette 3

Note: The “Advisee” edge of the *Advising* relationship type indicates that a student must be related to at least 1 {major, advisor} pair and may be related to up to  $p$  {major, advisor} pairs (e.g., SID 123 in Figure 5.8b has three different major/advisor combinations and SID 345 also has three different major/advisor combinations but only two different majors and two different advisors).

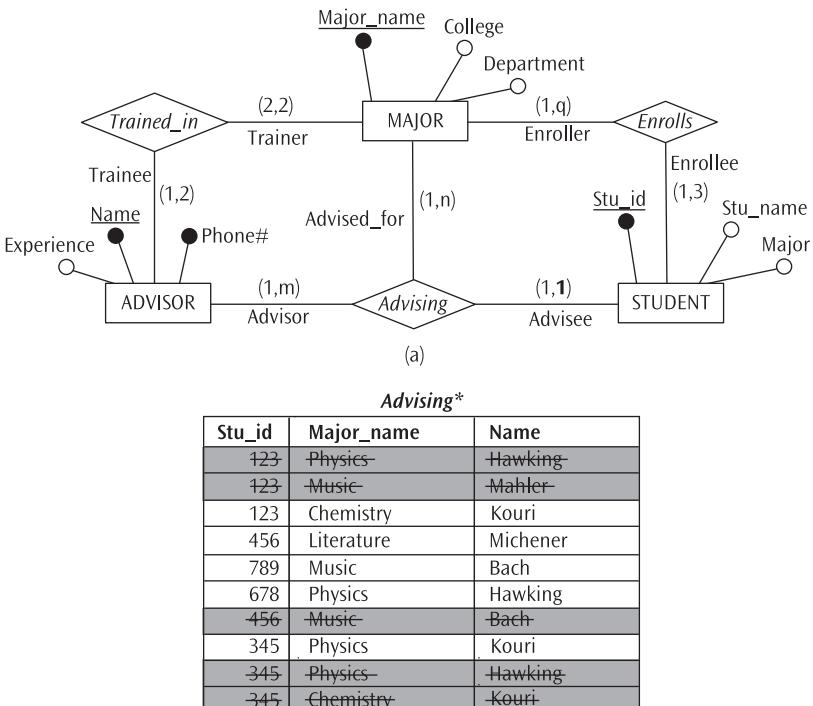
*Advising*

Stu_id	Major_name	Name
123	Physics	Hawking
123	Music	Mahler
123	Chemistry	Kouri
456	Literature	Michener
789	Music	Bach
678	Physics	Hawking
456	Music	Bach
345	Physics	Kouri
345	Physics	Hawking
345	Chemistry	Kouri

(b)

Data set for the *Advising* relationship in the Figure 5.8a

**FIGURE 5.8** The ternary relationship type *Advising* plus two binary relationship types *Trained\_in* and *Enrolls*



Note: A student is restricted to one major/advisor combination.

\*Removal of shaded rows from the data set enforces the restriction of one major/advisor pair per student

(b)

**FIGURE 5.9** Changing the maximum cardinality of the “Advisee” edge in Figure 5.8a from  $p$  to 1

Rule V3R1 requires that whereas a student may have multiple advisors and also multiple majors, the student is not permitted to have the same advisor for more than one major. This essentially means that a {student, advisor} pair must be related to no more than one major while a major may be related to many such pairs. This constraint is reflected in the revised *Advising* data set in Table 5.1.

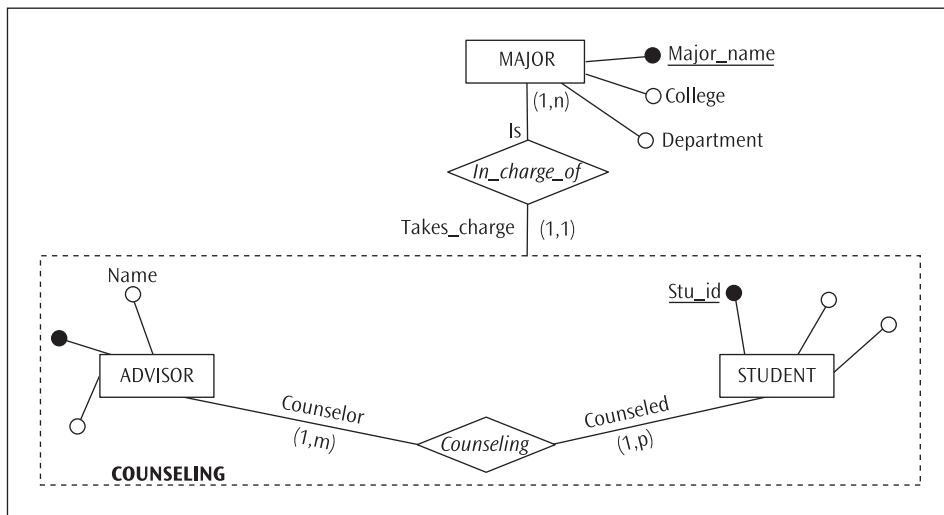
As previously discussed, Rule V3R1 cannot be incorporated in the ERD by changing the structural constraints of the ternary relationship type *Advising* in Figure 5.8. The implication of this rule is that one {student, advisor} pair can be related to only one major. This does not prohibit the presence of several {student, advisor} pairs, nor does it preclude a major being related to several {student, advisor} pairs. Rule V3R1 can then be interpreted as a relationship between MAJOR and an emergent entity type—say, COUNSELING—that results from the cluster {ADVISOR—*Counseling*—STUDENT}, as shown in Figure 5.10.

Advising Data Set*		
SID	Major	Advisor
123	Physics	Hawking
123	Music	Mahler
123	Chemistry	Kouri
456	Literature	Michener
789	Music	Bach
678	Physics	Hawking
456	Music	Bach
345	Physics	Kouri
345	Physics	Hawking
345	Chemistry	Kouri

Note: An advisor is limited to advising a student for only one major.

\*Removal of the shaded row enables a constraint that limits an advisor to advising a student in only one major.

TABLE 5.1 Revised Advising data set

FIGURE 5.10 The relationship type *In\_charge\_of* with the cluster entity type COUNSELING

COUNSELING here is called a cluster entity type,<sup>1</sup> indicated in the ERD by the dotted rectangle. Note that each cluster entity “COUNSELING” represents an inter-related {Student, Advisor} pair. Rule V3R1 is implemented in the ERD by replacing the ternary relationship type *Advising* in Figure 5.8 with the cluster entity type COUNSELING and its relationship with MAJOR (Figure 5.10). The maximum cardinality of 1 on the edge labeled “Takes\_charge” is also required to specify the constraint implied by the Rule V3R1.

The other two relationship types (*Trained\_in* and *Enrolls*) shown in Figure 5.8 also persist in the ERD. They are not shown in Figure 5.10 in order to enhance the clarity of expression of modeling this particular business rule.

Finally, consider the addition of one more business rule (V3R2) to vignette 3: *In order to minimize advising snafus, the college mandates that no more than one advisor can advise the same student for the same major.*

The *Advising* data set in Table 5.2 reflects Rule V3R2. This rule can be interpreted as one advisor per {student, major} pair triggering the emergence of another cluster entity type ENROLLMENT as the product of the relationship type *Enrolls* between STUDENT and MAJOR. ENROLLMENT then is related to ADVISOR.

Advising Data Set*		
SID	Major	Advisor
123	Physics	Hawking
123	Music	Mahler
123	Chemistry	Kouri
456	Literature	Michener
789	Music	Bach
678	Physics	Hawking
456	Music	Bach
345	Physics	Kouri
345	Physics	Hawking
345	Chemistry	Kouri

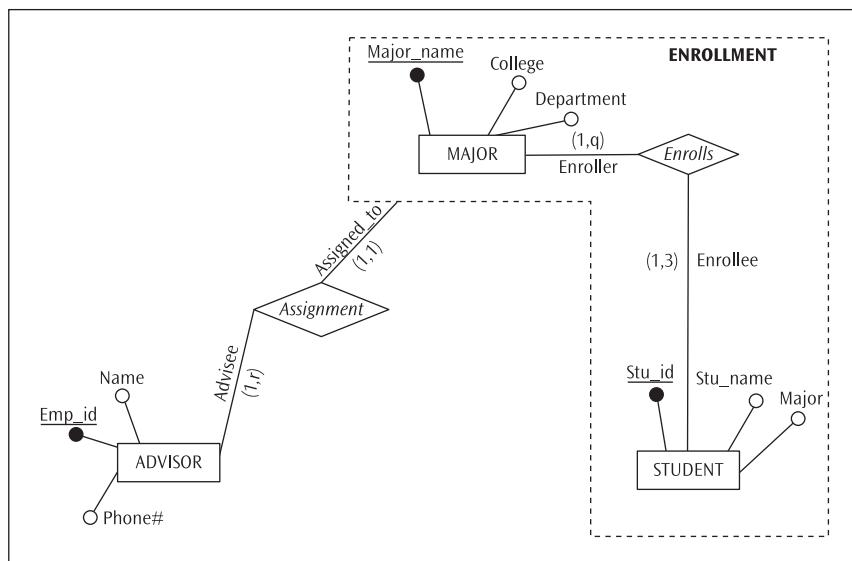
Note: An student is limited to only one advisor in one major.

\*Row shaded allows the restriction that limits a student to only one advisor in each major to be enforced.

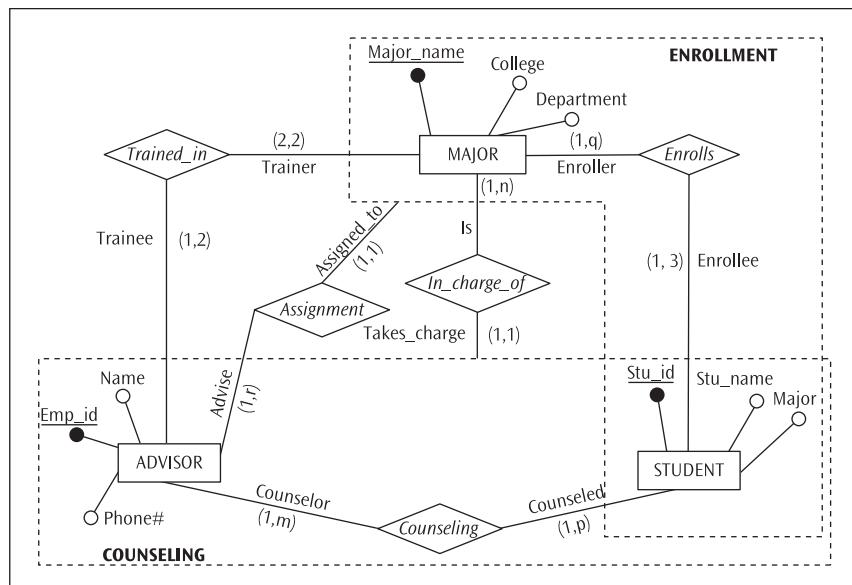
TABLE 5.2 Advising data: next revision

<sup>1</sup>“Aggregate entity type” is also a popular term for this. Sometimes, this is referred to as a composite (molecular) object. However, in order to distinguish this from an aggregate arising from an aggregation construct (see Section 4.1.6), the term “cluster entity type” is used. Notice that this entity type is essentially virtual (not real) and does not have a real existence, like a base or weak entity type.

Figure 5.11 portrays the cluster entity type ENROLLMENT and its relationship type *Assignment* with ADVISOR. Once again, note that the maximum cardinality on the edge labeled “Assigned\_to” indicates a 1 in order to enforce the constraint of no more than one advisor per {student, major} pair in the ERD. Figure 5.12 is a consolidated view of the two cluster entity types ENROLLMENT and COUNSELING, along with the rest of the scenario from the beginning of the episode in vignette 1 and the embellishment added in vignette 3.



**FIGURE 5.11** The relationship type *Assignment* with the cluster entity type ENROLLMENT



**FIGURE 5.12** Two cluster entity types: COUNSELING and ENROLLMENT

### 5.2.3 Vignette 4—A More Complex Entity Clustering

Vignette 3 demonstrated the use of a new ER modeling grammar construct—namely, the cluster entity type—using a simple example. In the vignette that follows, a slightly more complex entity clustering is demonstrated.

*Surgeons perform surgeries on patients to treat illnesses. Also, a surgery event pertains to a certain surgery type, and there can be numerous surgeries of a certain surgery type. Suppose that, for insurance purposes, there is a need to keep track of which surgeons perform what surgery(ies) on which patient(s) to treat what illness(es). In fact, the primary insurance covers such treatments. However, all treatments are not necessarily covered by insurance.*

How do we model this scenario? This story line naturally lends itself to being modeled as a **quaternary relationship** type (a relationship of degree four) among PATIENT, SURGEON, SURGERY, and ILLNESS in addition to other possible meaningful lower-degree relationships among some of these entity types not indicated in this vignette. Notice that the relationship type *Performs* does not spontaneously lend itself to be modeled as a base entity type, thus inducing the emergence of a quaternary relationship type. In addition, SURGERY can be modeled as a weak entity type identity-dependent on SURGERY\_TYPE. Figure 5.13a depicts the quaternary relationship type *Performs* along with the other relevant relationship types specified in the vignette. As a matter of convenience, the attributes in the ERD as well as the various participation constraints in the relationship types are arbitrarily created since they are not the primary focus of this vignette.

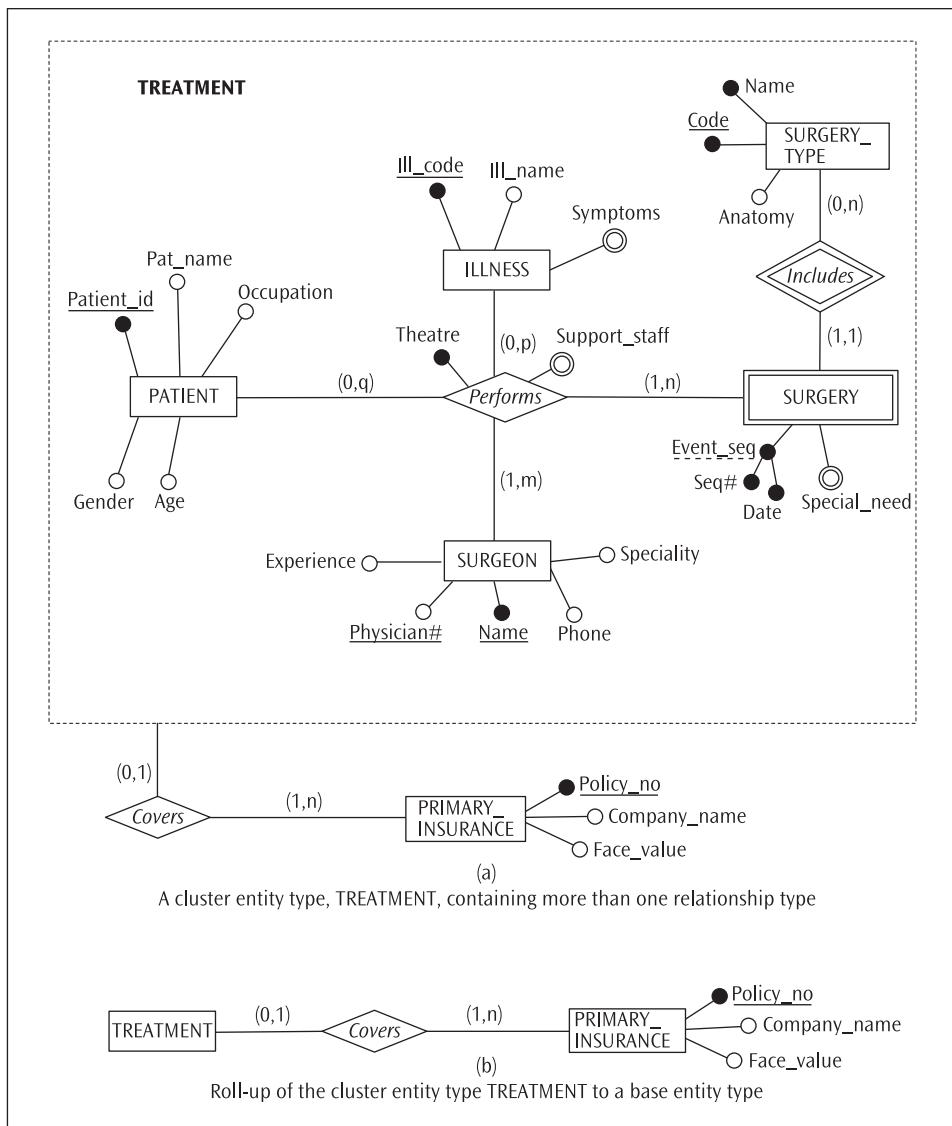
Observe that the insurance does not cover the PATIENT or SURGERY or ILLNESS individually. The insurance coverage pertains to the surgery performed on the patient by a surgeon to treat an illness. Thus, the cluster entity emerging from the *Performs* relationship is the one that is covered by insurance. The ERD in Figure 5.13a captures this by creating the cluster entity type TREATMENT, which exhibits a relationship with an entity type PRIMARY\_INSURANCE outside the cluster. In Figure 5.13b, the roll-up of the cluster entity type TREATMENT to a base entity type is demonstrated.

### 5.2.4 Cluster Entity Type—Additional Examples

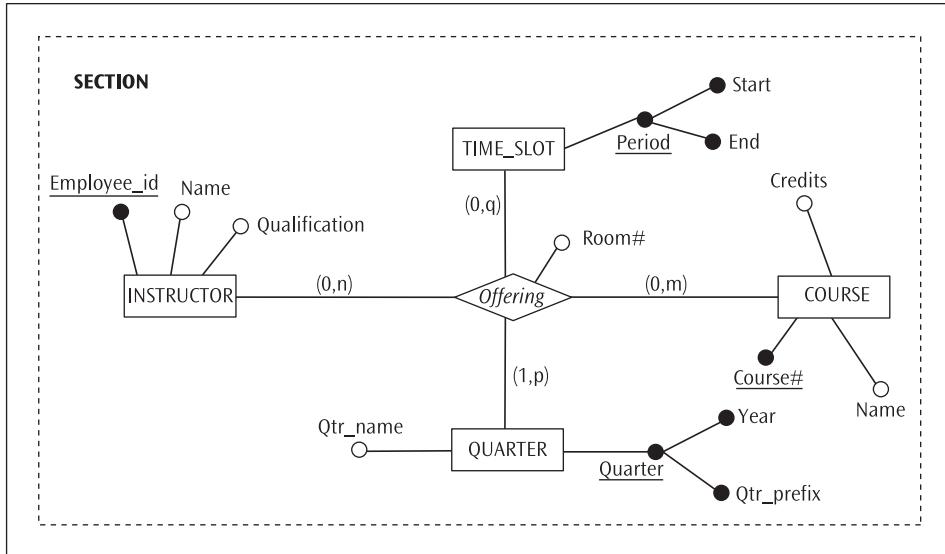
Let us now evaluate a few nuances that can be teased out of the Madeira College story (see vignette 1). Suppose a course is offered by a specific instructor during a certain quarter at a particular time slot. This enables the same course to be offered at the same time slot during the same quarter by more than one instructor. This story line is modeled as a quaternary relationship type *Offering* among COURSE, INSTRUCTOR, QUARTER, and TIME\_SLOT, as shown in Figure 5.14. On closer inspection, one may see an entity type emerging as a product of the quaternary relationship type *Offering*. In other words, the cluster of entity types involved in the quaternary relationship type *Offering* lends itself to the state of an entity type called SECTION. SECTION then is a cluster entity type.

If TIME\_SLOT has no attributes other than **Period**, one can question if it is necessary to model it as a base entity type. Accordingly, an alternative way of modeling the cluster entity type SECTION is to treat **Time\_slot** as a multi-valued attribute of the *Offering* relationship type. This conceptualization is shown in Figure 5.15.<sup>2</sup>

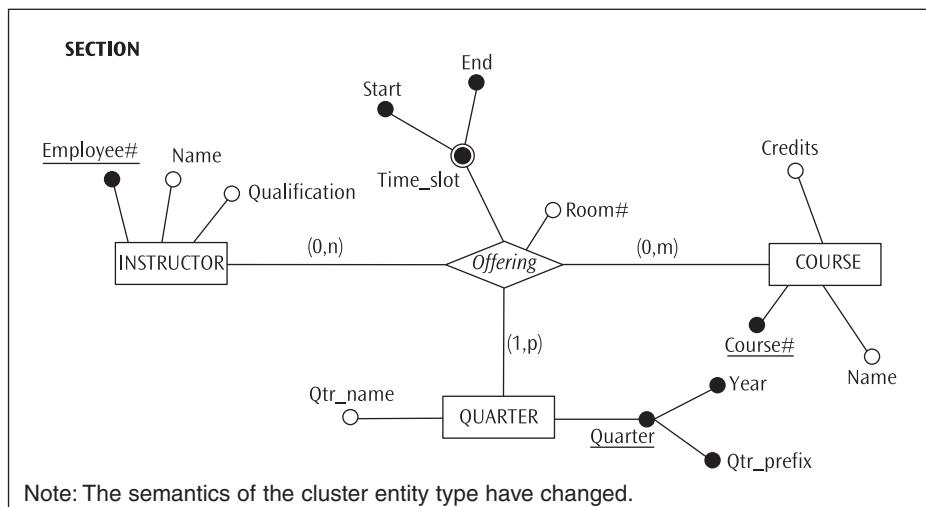
<sup>2</sup>While specification of attributes for a relationship type in ER modeling grammar is an accepted practice, designation of a multi-valued attribute for a relationship type is somewhat uncommon. An alternate solution essentially resulting from the decomposition of this multi-valued attribute of a relationship type is presented in Section 5.5.2.



**FIGURE 5.13** Modeling vignette 4 with a cluster entity type



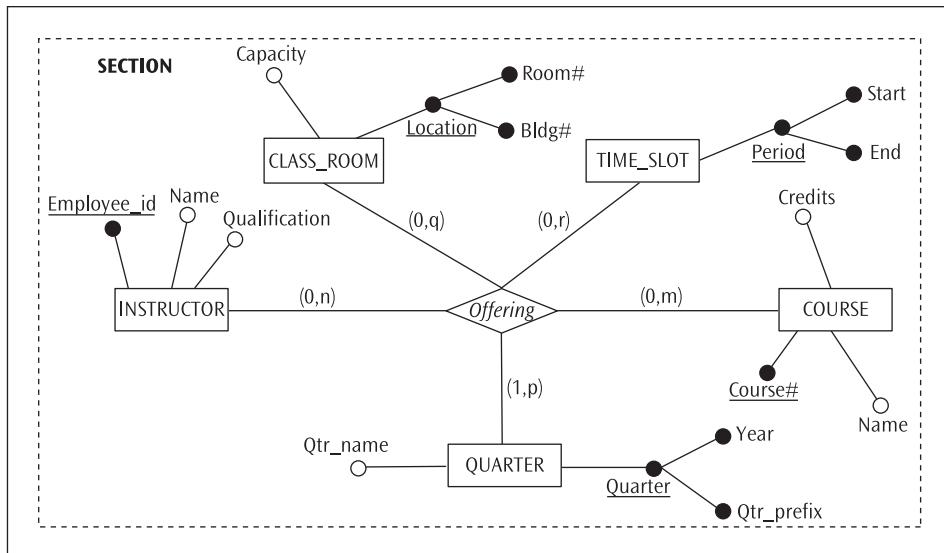
**FIGURE 5.14** An example of the cluster entity type **SECTION** emerging as a product of a quaternary relationship type



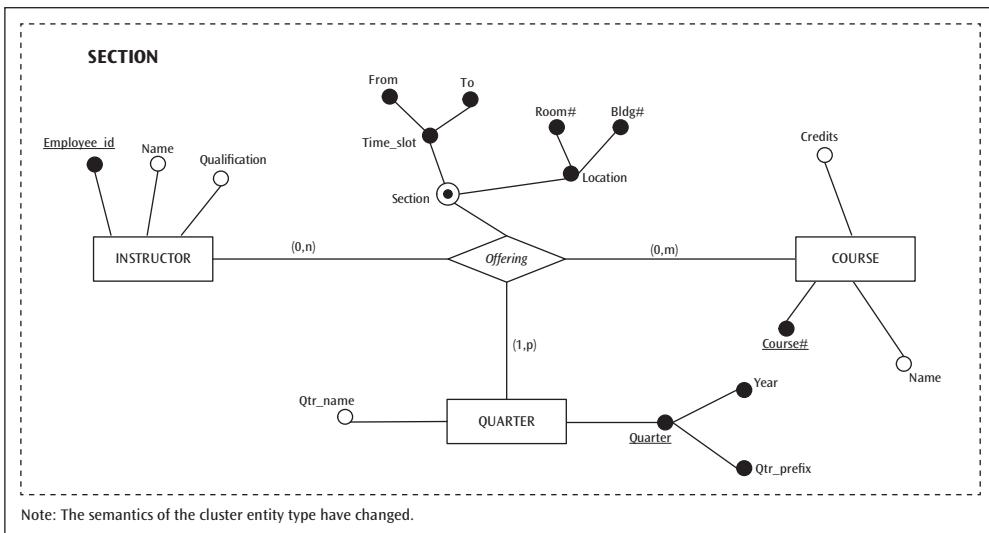
**FIGURE 5.15** An alternate representation of the cluster entity type **SECTION** (**TIME\_SLOT** reduced to a multi-valued attribute of **Offering**)

Suppose we also model **CLASS\_ROOM** as a component of the cluster entity type **SECTION**. This is accomplished by including **CLASS\_ROOM** as an entity type in the **Offering** relationship type. **Offering** now becomes a **quintary relationship**, a relationship of degree five, illustrated in Figure 5.16. This relationship can be alternatively modeled

similar to Figure 5.15, with **Location** and **Time\_slot** becoming part of the multi-valued attribute of the **Offering** relationship type, as long as the entity types **CLASS\_ROOM** and **TIME\_SLOT** need not be represented as entity types. This design appears in Figure 5.17.



**FIGURE 5.16** The quintary relationship type *Offering*



**FIGURE 5.17** Reducing *Offering* to a ternary relationship type with a multi-valued attribute

Observe that the cluster entity type SECTION (ERD in Figure 5.16 or 5.17) in its current form implies that “team teaching” of a course section is possible—that is, a particular course in a certain time slot in a given room during a specific quarter can be offered by more than one instructor. Observe that the current design also permits other semantically impractical states; for example, an instructor (or another instructor) may teach a different course in the same room at the same time slot during a specific quarter. Nonetheless, let us presently focus on the “team teaching” issue because the purpose of the design in Figure 5.16 is limited to exemplifying the possibility of a relationship type of degree five.

### 5.2.5 Madeira College—The Rest of the Story

Let us entertain, at this time, the imposition of a series of additional business rules on the Madeira College scenario and observe the unfolding of the ER modeling variations to implement these business rules in the design.

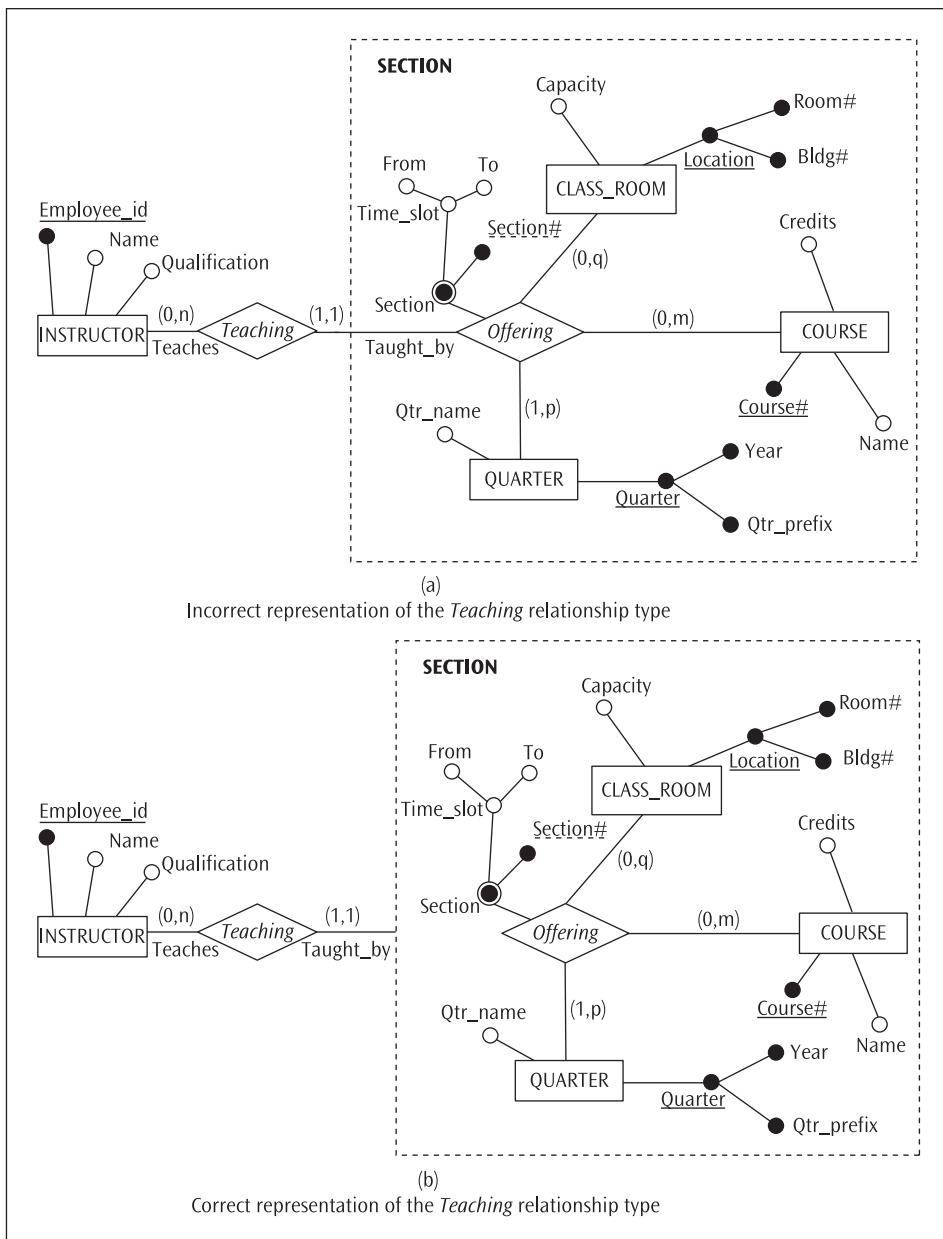
To begin with, consider a new business rule (V4R1) that *team teaching is not practiced in Madeira College*. In other words, no two instructors can teach the same course, at the same time slot, in the same room during a quarter. This implies that INSTRUCTOR is no longer a component of the cluster entity type SECTION, shown in Figure 5.16. Instead, there is a 1:n relationship between INSTRUCTOR and SECTION. In addition, **Time\_slot**, while correct, is a rather clumsy composite attribute to work with. Thus, from a practical perspective, one may choose to make up a **Section#** to serve as a surrogate for **Time\_slot** as the partial key in *Offering*, in which case **Time\_slot** need not be a mandatory attribute anymore. This design variation is shown in Figure 5.18.

Note, however, that the ERD in Figure 5.18a is incorrect because in the ER modeling grammar a relationship type cannot be directly related to another relationship type (i.e., *Teaching* cannot be related to *Offering*).<sup>3</sup> The correct rendition of this design appears in Figure 5.18b. Observe that the edge “Taught\_by” emanating from the *Offering* relationship type in Figure 5.18a has been replaced by the edge “Taught\_by” in Figure 5.18b emerging from the cluster entity type SECTION. Also, in Figure 5.18b, in order to enforce the “no team teaching” business rule, the maximum cardinality reflected on the edge labeled “Taught\_by” in the *Teaching* relationship type must become a **1**. At this point, this design permits use of multiple classrooms for the same Section, which is not a semantically practical option.

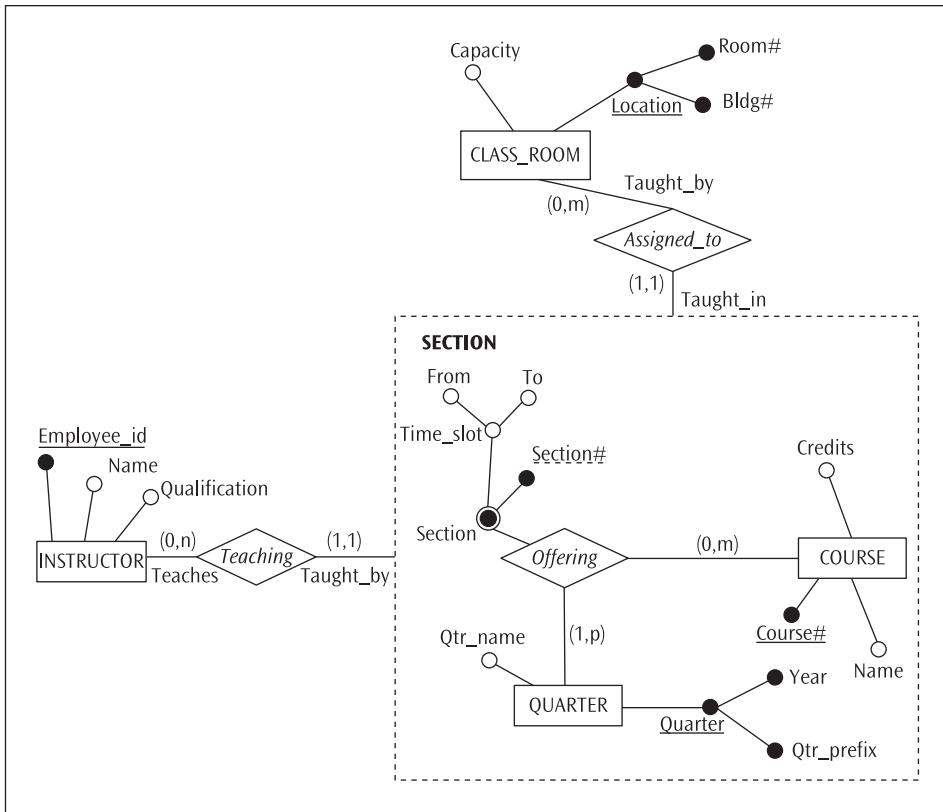
Suppose, at this point, we introduce a new business rule (V4R2): *A Section cannot span multiple classrooms*. This change is accomplished by removing CLASS\_ROOM from the cluster entity type SECTION, modeling a 1:m relationship type *Assigned\_to* between CLASS\_ROOM and SECTION, and restricting the classrooms used for a section to **1** through the specification of maximum cardinality of **1** on the *Taught\_in* edge of the *Assigned\_to* relationship type (see Figure 5.19).

---

<sup>3</sup>The ER modeling grammar does allow a weak relationship type to be involved in a relationship with another relationship type. A weak relationship type is discussed in Section 5.3.



**FIGURE 5.18** Prohibition of “team-teaching” modeled in the *Teaching* relationship type



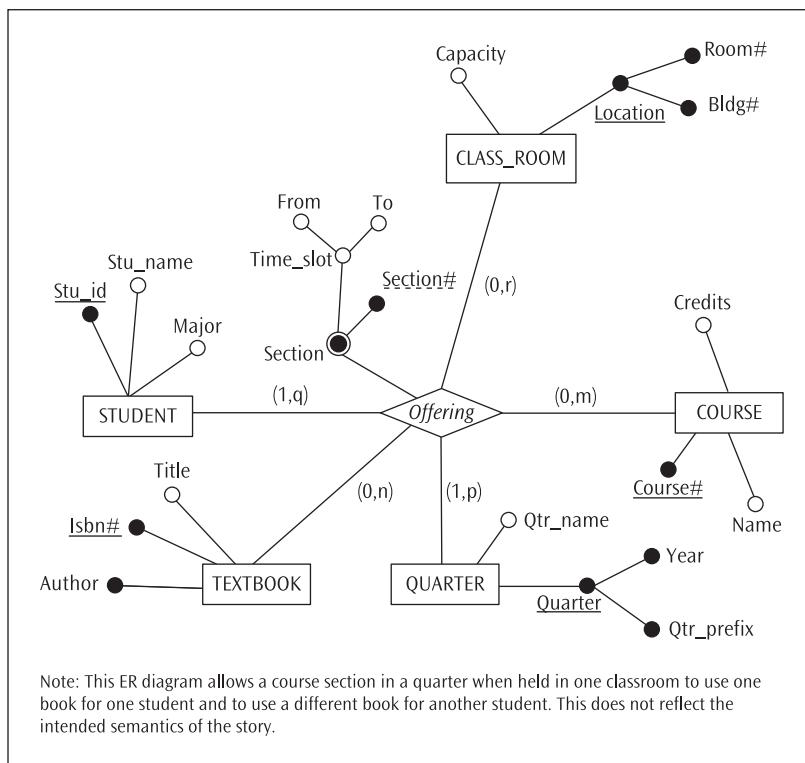
**FIGURE 5.19** Prohibition of a section from being assigned to multiple classrooms

Interestingly, the design in Figure 5.19 does not prevent sections of two or more different courses from being taught in the same classroom at the same time during a quarter. One can argue that this is intended to permit cross-listing of multiple courses. Suppose cross-listing of courses is not permitted. How should the ERD in Figure 5.19 be altered to handle such a business rule? This is left as an exercise for the reader.

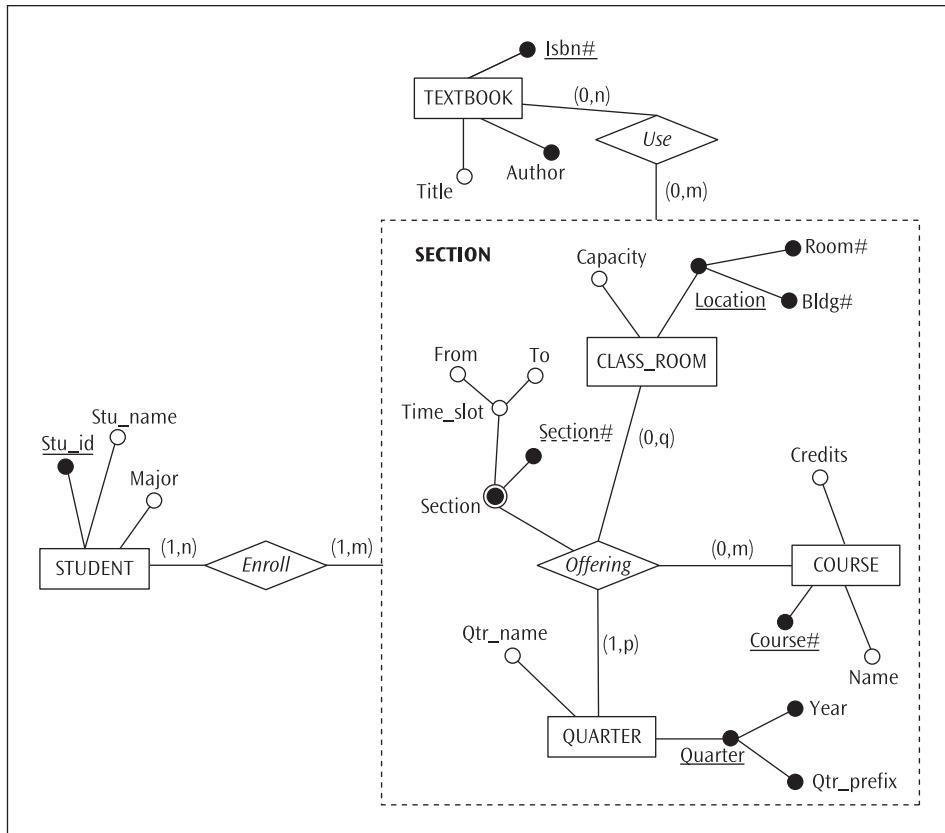
Now, let us embellish the Madeira College story further by stating a couple of additional business rules: (1) While a course section need not use a textbook, it is also possible that a course section may sometimes use more than one textbook and that a textbook may be used in multiple course sections (V4R3). (2) Likewise, a student must enroll in one or more course sections and a course section must have more than one student (V4R4).

Does the ERD shown in Figure 5.20 (a relationship type of degree five) accurately describe this scenario? Not unless it is okay for a course section in a quarter, when held

in one or more classrooms, to use one or more textbooks for one or more students and use a different textbook for another student or students, and also okay for a course section in a quarter to use a different textbook (or textbooks) for the same student(s) in a different classroom (or classrooms). Clearly, while the ERD is syntactically correct, it does not reflect the semantics conveyed by the business rules stated above. SECTION, to begin with, should be a cluster entity type akin to the design shown in Figure 5.18b. Then, it is possible to establish an m:n relationship type between SECTION and TEXTBOOK as well as between SECTION and STUDENT. The ERD in Figure 5.21 is a correct rendition of this scenario in which a student enrolls in a course section and uses the same textbook(s) that all other students in that course section are using, and the classroom location of the course section does not change with the student(s) or textbook(s).

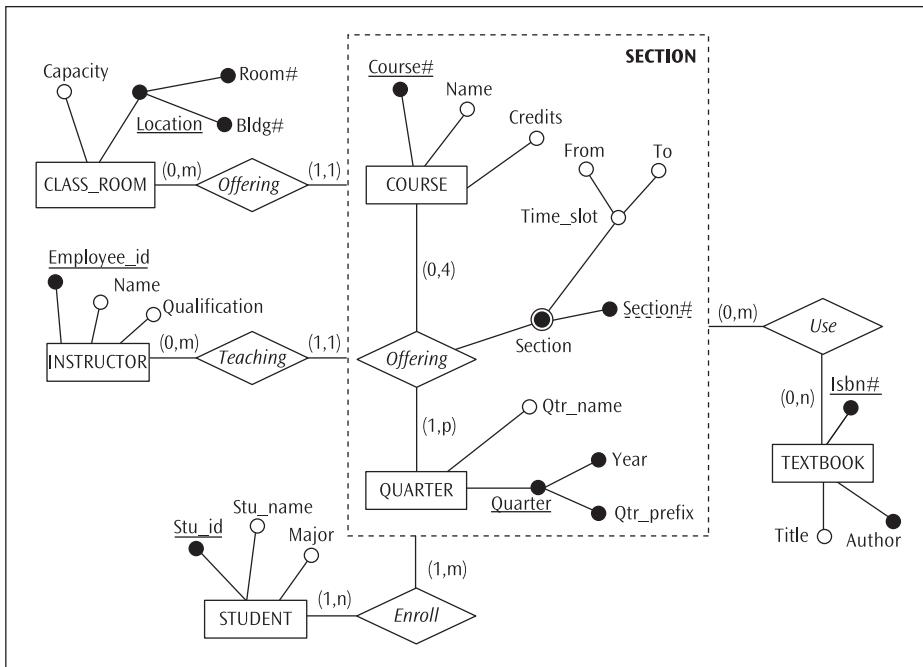


**FIGURE 5.20** STUDENT enrolled in a course *Offering* using a TEXTBOOK modeled as a quintary relationship type



**FIGURE 5.21** A correct rendition of the design intended in Figure 5.20

The final ERD reflecting all the business rules stated in the rest of the story for Madeira College appears in Figure 5.22.

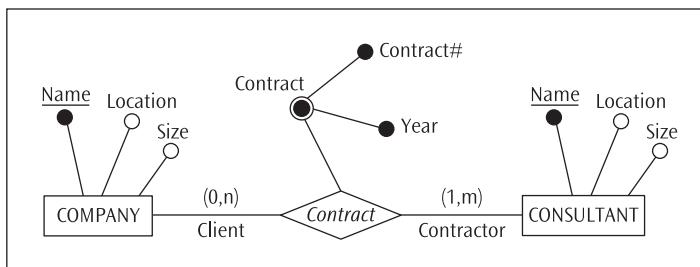


**FIGURE 5.22** ERD for Madeira College—The rest of the story

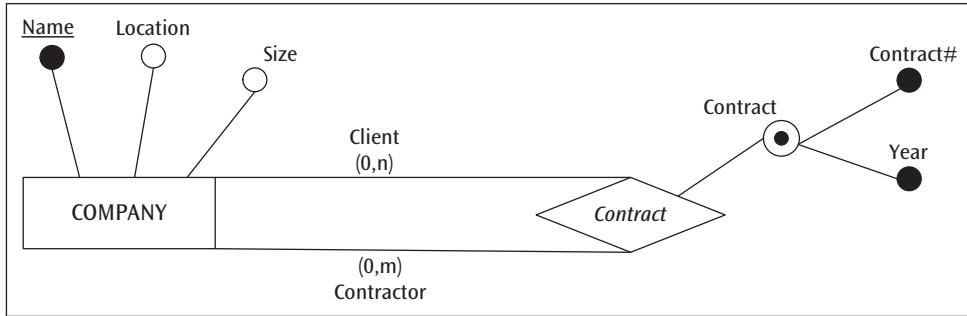
### 5.2.6 Clustering a Recursive Relationship Type

ER modeling constructs lend themselves to use in innovative ways. Let's look at a couple of examples that illustrate interesting ways to use the ER modeling grammar.

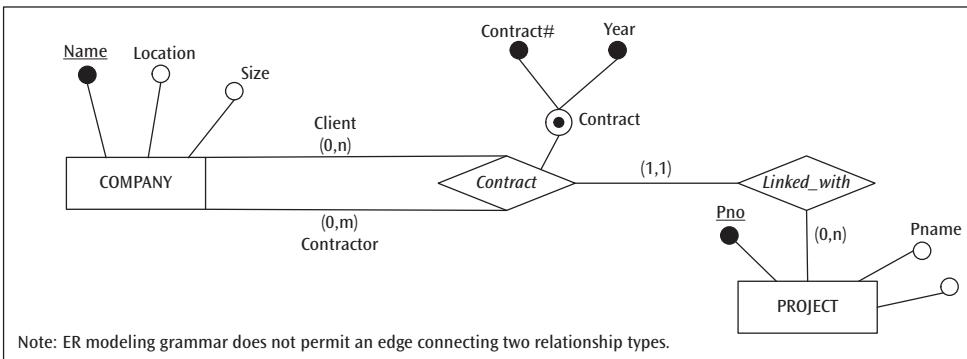
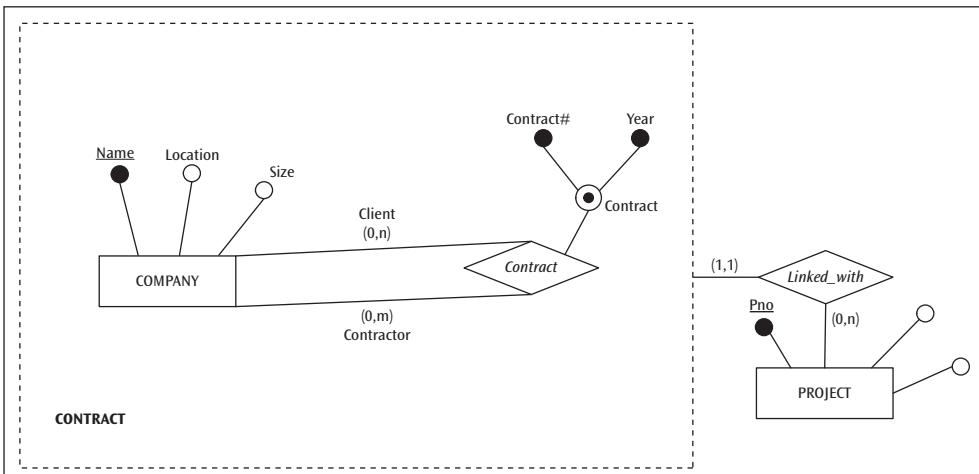
Companies often use consultants on a contract basis to do some work and often have different types of boilerplate contracts that can be simply reviewed on a year-to-year basis. This can be modeled in the form of the binary relationship shown in Figure 5.23. As one can see in this figure, a consultant is also a company (the common attribute names indicate that the entity types are identical). In this case, the ERD reduces to the recursive relationship type shown in Figure 5.24. Here is a subtle point to note: In Figure 5.23 the participation constraint of CONSULTANT indicates total participation in the *Contract* relationship type; however, it is semantically incorrect to carry forward the same constraint in the recursive relationship type *Contract* simply because it will mean that every COMPANY must be a contractor. Accordingly, observe the partial participation of COMPANY in the role of contractor in Figure 5.24.



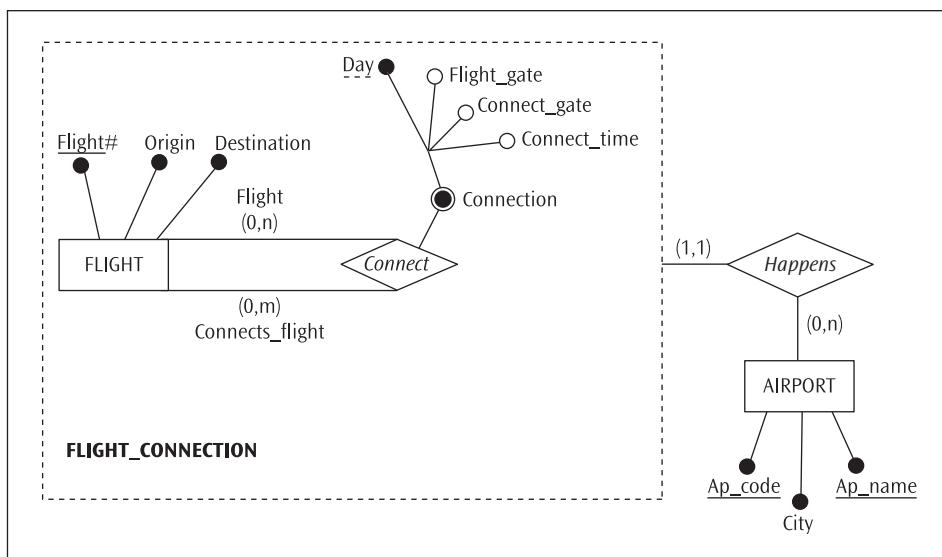
**FIGURE 5.23** Contract as a binary relationship type

**FIGURE 5.24** Contract as a recursive relationship type

Now, what if each contract is associated with one project and a project may include several contracts? Because the ER modeling grammar does not permit a relationship to be established between two relationship types (see Figure 5.25a), creating the cluster entity type CONTRACT permits the relationship type *Linked\_with* to be specified (see Figure 5.25b).

**FIGURE 5.25a** Syntactic error in the representation of the *Linked\_with* relationship type**FIGURE 5.25b** Resolution of the syntactic error in Figure 5.25a using a cluster entity type CONTRACT

Here is another real-world scenario similar to the one described in the previous example: A flight often connects to several other flights at an airline's hub. Thus, a flight's arrival information and the connecting flight's departure information are crucial to the airline and its passengers. Figure 5.26a depicts this scenario.



**FIGURE 5.26a** Syntactically correct representation of the *Happens* relationship type

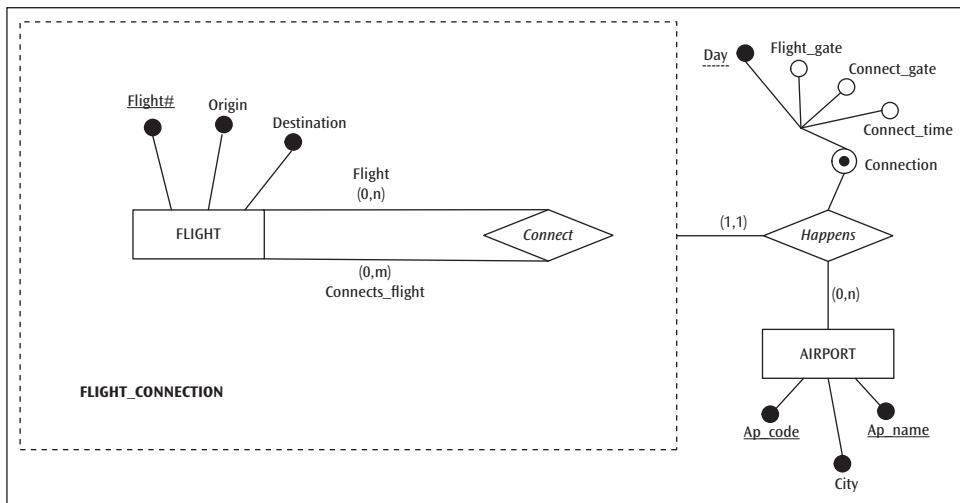
A technical examination of the ERD yields the following facts captured in the ERD:

- An airport need not be a place for any flight connections; then again, as many as “n” flight connections can happen in an airport.
- A flight connection happens in exactly one airport—no more, no less, and not in thin air, either.
- A flight connection is defined as a flight (say, Flight A) connecting to another flight (say, Flight B) on a given day; accordingly, Flights A and B connect more than once a week, and for each of these connections, the **Flight\_gate**, the **Connection\_gate**, and the **Connection\_time** can be the same or different.

A closer scrutiny reveals that the current model will not prevent a connection between Flights A and B occurring at different airports on different days. Since the business rule forming the basis for this ERD is not available, one may certainly question the intent behind this ERD. It is perhaps more practical to assume that Flights A and B connect at the same airport on all days they connect. If that is true, then this ERD is in error.

Figure 5.26b offers a solution to this problem. The structural constraints (cardinality constraint and participation constraint) of the binary relationship between **AIRPORT** and the cluster entity type **FLIGHT\_CONNECTION** remain the same. In fact, the only difference between the ERDs in Figures 5.26a and 5.26b is that the multi-valued, composite attribute **Connection** in Figure 5.26b is on the relationship **Happens** instead of on the relationship **Connect**. Now, a flight connection is between two flights (Flight A and Flight B, independent of the day of flight), and a specific flight connection (say, between Flights A and B) happens not only at just one airport but at the same airport, irrespective of the day.

of flight. The fact that Flights A and B connect more than once a week and that, for each of these connections, the **Flight\_gate**, the **Connection\_gate**, and the **Connection\_time** can be the same or different, is independent of the fact that the two flights connect in not just one but the same airport. The difference between the two ERDs will get further clarified after the final decomposition of the two models later in this chapter (Section 5.5).



**FIGURE 5.26b** Semantically superior modeling of the *Happens* relationship type

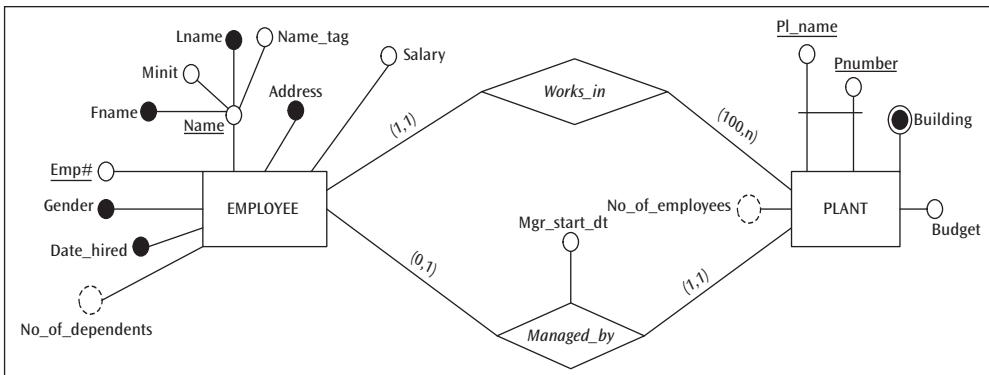
So far, we have seen several real-world scenarios that require relationship types beyond the conventional binary relationship types. The presentation is by no means exhaustive because other innovative ways of combining the various modeling constructs of the ER/EER modeling grammar are possible. An understanding of the domain of the business problem normally leads to the emergence of such uses. The objective of this section has been to sensitize data modelers and database designers to the rich modeling opportunities available at their disposal via the ER modeling grammar.

### 5.3 INTER-RELATIONSHIP INTEGRITY CONSTRAINT

Recall that the business rules for Bearcat Incorporated (Section 3.1) indicate that a plant has employees and every employee works in one and only one plant. Also, some plant employees hold the position of managers of these plants. Let us now change this business rule as follows: *All employees need not be working at the plants, because some are in the corporate office and others are in the regional offices of Bearcat Incorporated. However, at present, we are modeling just the activities in the plants.* Also, let us add a new business rule: *A plant manager should be a plant employee.*

Figure 5.27 is an excerpt from the Design-Specific ERD for Bearcat Incorporated developed in Chapter 3 (Figure 3.13) that pertains to this scenario.

In order to incorporate the change in business rule, the participation constraint of EMPLOYEE in the *Works\_in* relationship type must be made partial (*min = 0*) since not



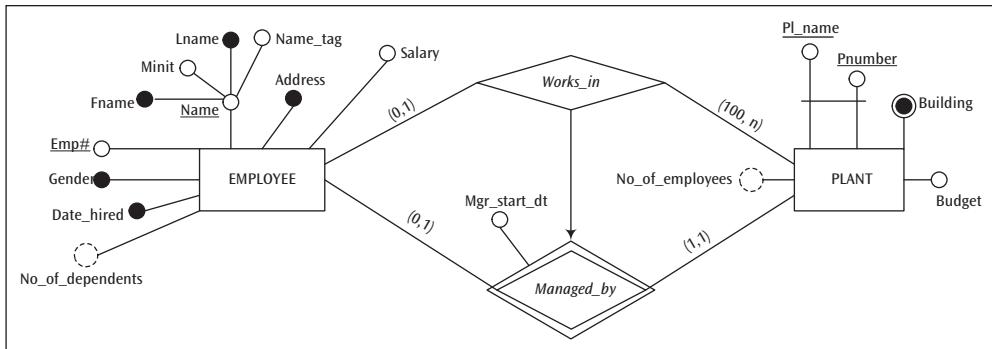
**FIGURE 5.27** The *Works\_in* and *Managed\_by* relationship types from Figure 3.13

all Bearcat employees work at the plants. The new business rule essentially suggests a precedence relationship between *Works\_in* and *Managed\_by*. In other words, for a *Managed\_by* relationship instance between an employee and a plant to exist, a corresponding *Works\_in* instance between the same two entities must be present. In 1999, Debabrata Dey, Veda Storey, and Terry Barron published an article in the journal *ACM Transactions on Database Systems* in which they introduced a new ER modeling construct called the **weak relationship type** that indicates an inter-relationship integrity constraint. The symbol used to denote a weak relationship type is the same as the identifying relationship type (a double-diamond symbol; see Figure 3.2), as shown in Figure 5.28.<sup>4</sup> A solid arrow from a regular relationship type to the weak relationship type indicates an inter-relationship integrity constraint, implying that the latter relationship set is included in (i.e., is a subset of) the former relationship set. That is, in order for an instance of the *Managed\_by* relationship type to occur, an instance of the *Works\_in* relationship type between the same entity pair should be present; essentially, a manager of a plant must work in the same plant. This constraint specification is referred to as **inclusion dependency**.<sup>5</sup> Dey, Storey, and Barron define a weak relationship type as "... a relationship, the existence of whose instances depends on the [presence of] instances of (one or more) other relationships."<sup>6</sup> The inclusion dependency is shown as  $\text{Managed\_by} \subseteq \text{Works\_in}$ .

<sup>4</sup>This does not cause any conflict in the ER modeling grammar because an identifying relationship type can exist only between a weak entity type and its identifying parent entity type(s), while a weak relationship type can relate only to a regular relationship type. Furthermore, interpretation in context will clarify if a double diamond is an identifying relationship type or a weak relationship type or both.

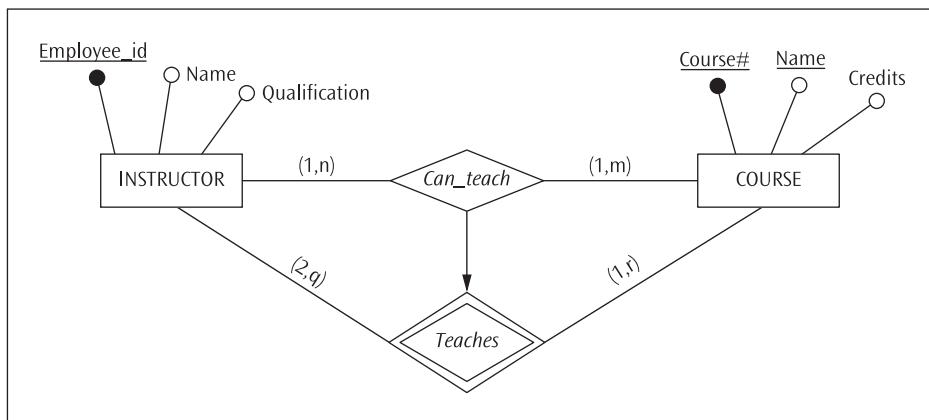
<sup>5</sup>Inclusion dependency is different from the inclusive arc construct of the ER modeling grammar in that it conveys directionality through a subset relationship, whereas the inclusive arc conveys the idea of mutuality in inclusiveness. Furthermore, an inclusive arc pertains to the participation of an entity type in any two relationship types, whereas inclusion dependency is about the inter-relationship between two relationship types among the same two entity types. Inclusion dependency when mapping a conceptual schema to a logical schema is discussed in Chapter 6.

<sup>6</sup>Dey, D., V. C. Storey, and T. M. Barron. "Improving Database Design through the Analysis of Relationships." *ACM Transactions on Database Systems*, 24, 4 (December) 453–486, 1999.



**FIGURE 5.28** *Managed\_by* as a (condition-precedent) weak relationship type

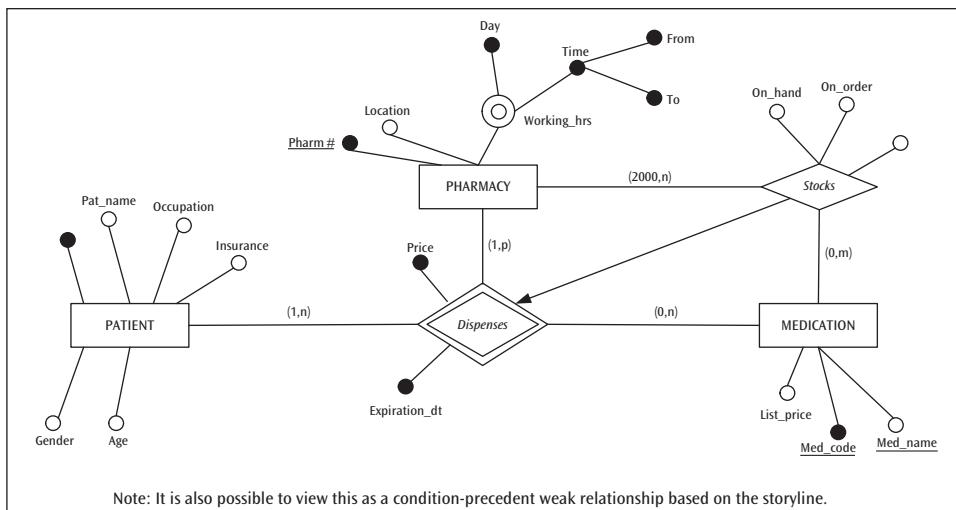
A weak relationship type arises in many real-world situations when two relationship types are linked by (1) a **condition precedence sequence** (based on meeting a condition) or (2) an **event precedence sequence** (based on the occurrence of an event). The weak relationship type *Managed\_by* in Figure 5.28 is a condition-precedent weak relationship type because the condition that one has to be an employee of the plant in order to be a manager of that plant semantically precedes that employee being the manager of that plant. Another opportunity for a condition-precedent weak relationship can be seen in the scenario reflected in Figure 5.2. An appropriate excerpt from Figure 5.2 is presented in Figure 5.29. Consider a business rule: *In order for an instructor to teach a course, he or she must be capable of teaching that course*. This refinement of the story is incorporated in the ERD in Figure 5.29. *Teaches* here is a condition-precedent weak relationship type that is inclusion-dependent on *Can\_teach*, as shown in (*Teaches*  $\subseteq$  *Can\_teach*). This is captured in the ERD by the solid arrow drawn from *Can\_teach* to *Teaches*.



**FIGURE 5.29** *Teaches* as a (condition-precedent) weak relationship type

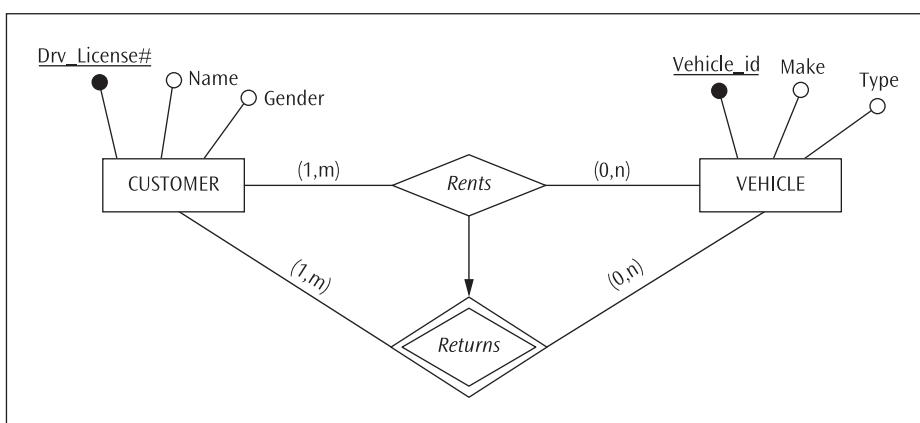
Figure 5.30 is a subset of the ERD for the story narrated in vignette 2 about Get Well Pharmacists, Inc. (see Section 5.1.2 and Figure 5.6) where a new constraint is imposed via this business rule: *A medication must be stocked by a pharmacy before it*

can be dispensed to a patient. The inclusion dependency *Dispenses*  $\subseteq$  *Stocks* captures this business rule and is shown in the ERD by the solid arrow drawn from *Stocks* to *Dispenses*. *Dispenses* here is labeled as an event-precedent weak relationship type, but it may also be viewed as a condition-precedent weak relationship based on the story line.<sup>7</sup>



**FIGURE 5.30** Dispenses as an (event-precedent) weak relationship type

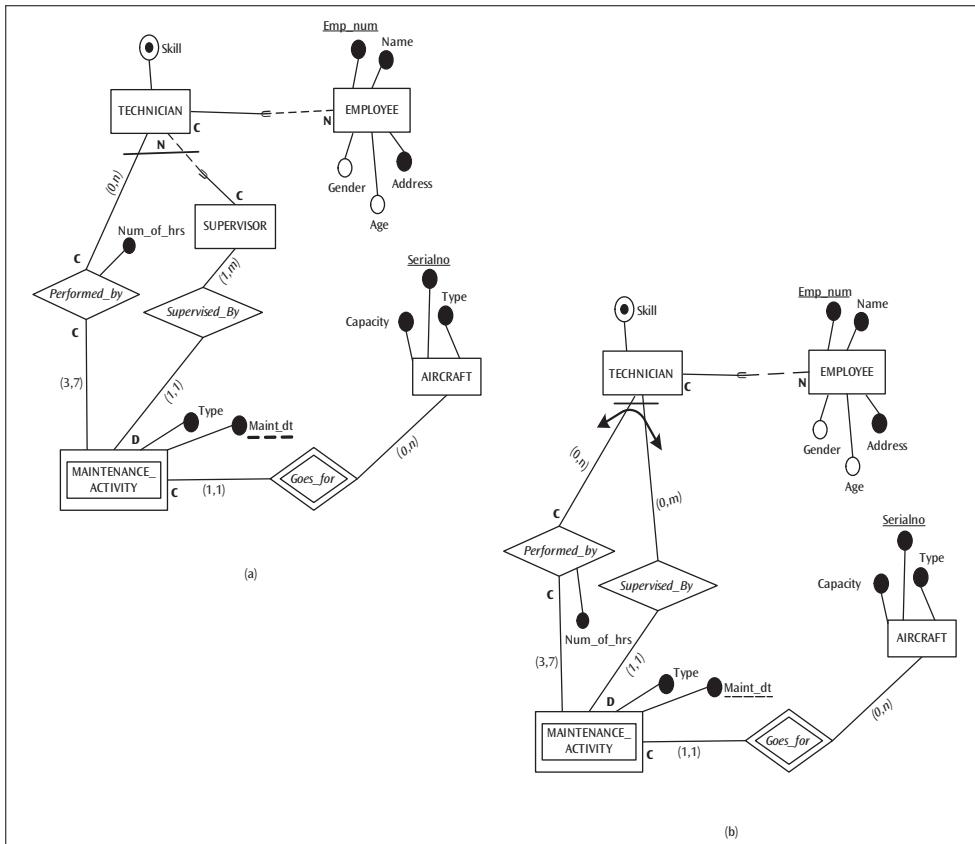
Suppose a rental agency rents an array of vehicles (e.g., cars, trucks, vans, boats). A plausible business rule in this context is: *Before the event “return of a vehicle by a customer” happens, the event “rental of that specific vehicle by that particular customer” should transpire*. Figure 5.31 captures this requirement as an event-precedent weak relationship type.



**FIGURE 5.31** Returns as an (event-precedent) weak relationship type

<sup>7</sup>What is important is that this indicates a weak relationship type. Interpretation of whether it is event-precedent or condition-precedent has only amusement value.

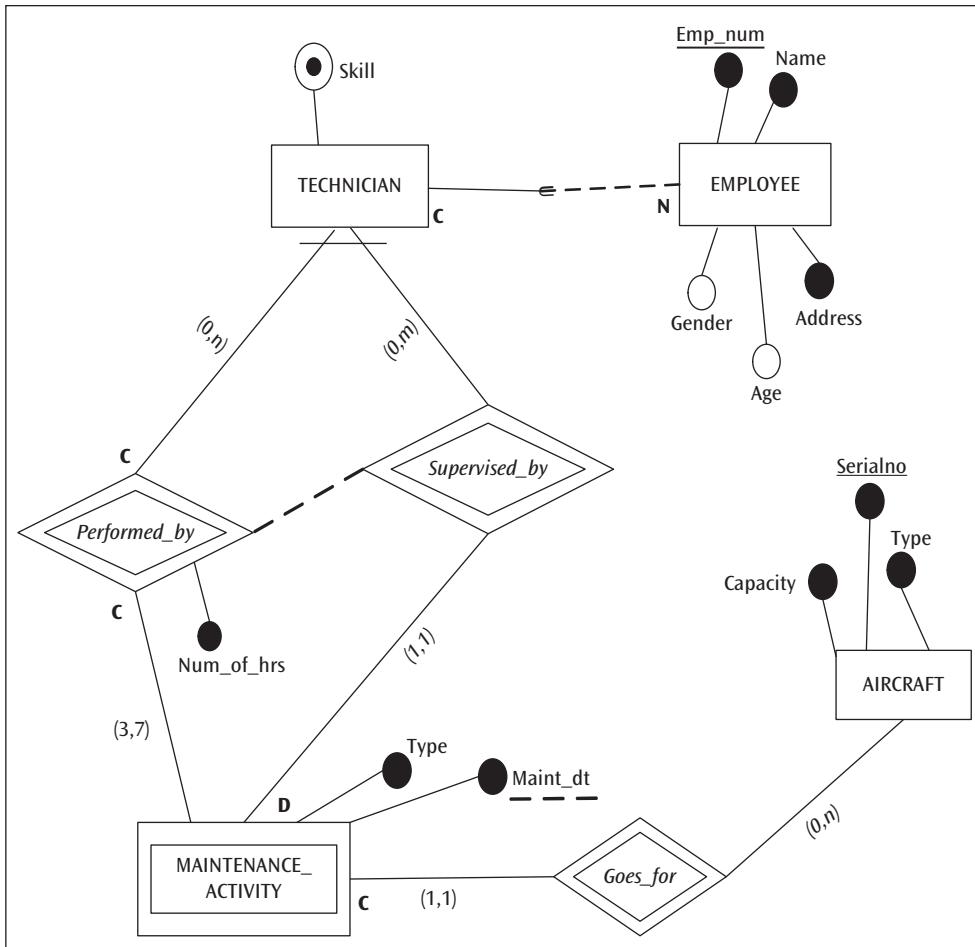
While the weak relationship type can be used to indicate inclusion dependency in terms of a unidirectional subset relationship between two relationship types, it can also be used to model scenarios of bidirectional (mutual) exclusion between two relationship types. Referred to as **exclusion dependency**, this construct in the ER modeling grammar may appear to be equivalent to the ER modeling construct “exclusive arc” defined in Chapter 3 (see Figure 3.2), but it is not. Thus, these two constructs are not mutually substitutable. As an example, consider an excerpt from a larger ERD, displayed in Figure 5.32, in which a technician performs and/or supervises a maintenance activity on an aircraft. Figure 5.32a models some of the technicians to be supervisors in addition to being technicians. Only the supervisors supervise the maintenance activities. And, being a technician, a supervisor can also engage in performing a maintenance activity. In fact, the modeling here permits a technician to also engage in performing the maintenance activity s/he supervises.



**FIGURE 5.32** Modeling mutuality in relationship exclusion for an entity type

It is not unreasonable to expect the supervisor to be competent as a technician in order to assume the role of a supervisor. In other words, a supervisor should be a technician to begin with. However, suppose it is not acceptable for a technician to supervise a maintenance activity s/he is engaged in performing. The ERD in Figure 5.32a does not support this business rule. The ER modeling grammar construct “Exclusive arc” is employed in Figure 5.32b to ensure that a technician participating in the *Supervise\_by* relationship type does not participate in the *Performed\_by* relationship type. But then, this constraint goes far beyond the enforcement of preventing a supervisor from engaging in the performance of the maintenance activity s/he is supervising. It simply prohibits a technician who is participating in the role of a supervisor of maintenance activities from engaging in the performance of any maintenance activity. In other words, here (Figure 5.32b), a technician can either perform maintenance activities or supervise maintenance activities—never both, even if the supervised maintenance activity is different from the maintenance activity performed.

The use of weak relationship types in specifying inclusion dependency between two relationship types was presented earlier in this section. In the example currently under discussion, we have an opportunity to use weak relationship types to model scenarios of bidirectional (mutual) exclusion between two relationship types. A dashed line with no directional indicator connects the two relationship types *Performed\_by* and *Supervised\_by* in Figure 5.33. Moreover, both *Supervised\_by* and *Performed\_by* are represented here as weak relationship types (double-diamond symbol). The semantics conveyed by this exclusion dependency constraint is that the two relationships are mutually exclusive; that is, if a maintenance task is performed by a technician, the same maintenance task cannot be supervised by the same technician. This is precisely the objective of the business rule we were seeking to specify. Note that the exclusion dependency constraint here permits a technician who is carrying out the role of a supervisor on a maintenance activity to engage in the performance of a different maintenance activity that s/he is not supervising.



**FIGURE 5.33** Modeling exclusion dependency between two relationships among the same two entity types

## 5.4 COMPOSITES OF WEAK RELATIONSHIP TYPES

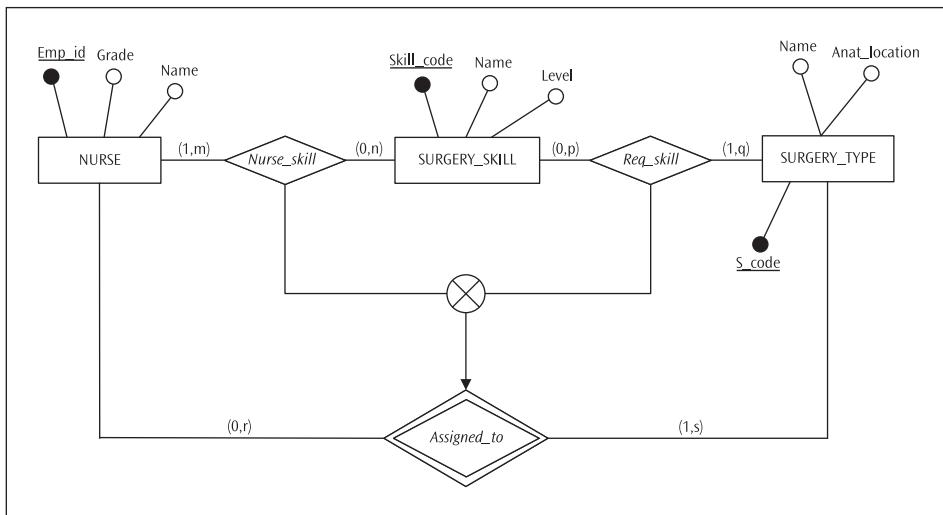
Weak relationship types also enable the data modeler to capture richer business situations in an ERD, as illustrated in the following examples.

### 5.4.1 Inclusion Dependency in Composite Relationship Types

Consider this scenario: A particular surgery type requires certain skills and only nurses possessing these skills can be assigned to this type of surgery. The set of skills required for assisting in all surgery types as well as the list of skills possessed by each nurse are also available.

Figure 5.34 depicts this scenario. The relevant entity types and their attributes are arbitrarily assigned. In this case, the composite relationship type (*Nurse\_skill*  $\otimes$  *Req\_skill*) captures which nurse(s) has/have the skills to assist in what surgery type(s). Therefore, the relationship type *Assigned\_to* is transformed to a weak relationship type and the

inclusion dependency  $Assigned\_to \subseteq (Nurse\_skill \otimes Req\_skill)$ ,<sup>8</sup> as denoted by the solid arrow drawn from the composite  $(Nurse\_skill \otimes Req\_skill)$  to  $Assigned\_to$  in the ERD, ensures incorporation of the business rule in the ERD.



**FIGURE 5.34** A weak relationship type inclusion-dependent on a composite relationship type

Next, consider this scenario:

*Restaurants cater to banquets. A banquet contains a menu of food items, and a restaurant caters various food items. Unless a restaurant is capable of preparing the set of food items contained in a banquet's menu, the restaurant cannot cater that particular banquet.*

The ERD in Figure 5.35 portrays the scenario for this story line. The relevant entity types, their respective attributes, and their relationship types are arbitrarily assigned—for example, a restaurant “can cater” to at least 50 banquets and a banquet can be catered by at least seven restaurants, etc. The specific business rule stated above is captured through the weak relationship *Can\_cater*. Note that the ERD reflects the inclusion dependency  $(Can\_cater \subseteq Caters \otimes Contains)$ , meaning that in order for a relationship to exist in the *Can\_cater* set, it should be a subset of the composite set of *Caters* and *Contains*.

#### 5.4.2 Exclusion Dependency in Composites of Weak Relationship Types

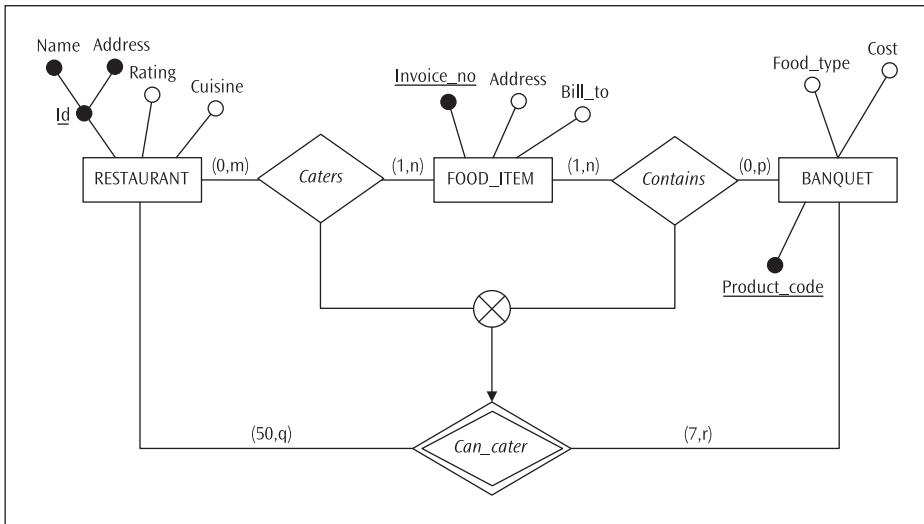
The expressive power of an exclusive arc is limited to portraying mutual exclusiveness between two relationship types. In other words, an exclusive arc is not capable of dealing with composites of relationship types similar to the ones illustrated in the previous section. The exclusion dependency construct, however, is capable of handling composites of relationship types.

As an example, consider this scenario:

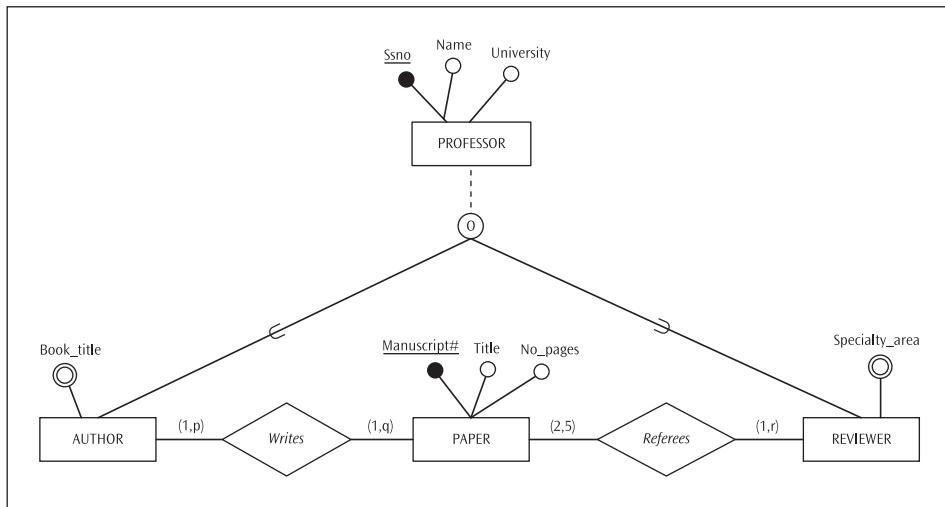
*Professors can be authors of papers and/or reviewers of papers. That is, any professor can be an author as well as a reviewer of papers.*

<sup>8</sup>The symbol  $\otimes$ , referred to as a “composite” in  $A \subseteq (B \otimes C)$ , implies a projection from the natural join of B and C that is union-compatible with A.

Figure 5.36 depicts this scenario as an ERD. The relevant entity types and their attributes are arbitrarily assigned.



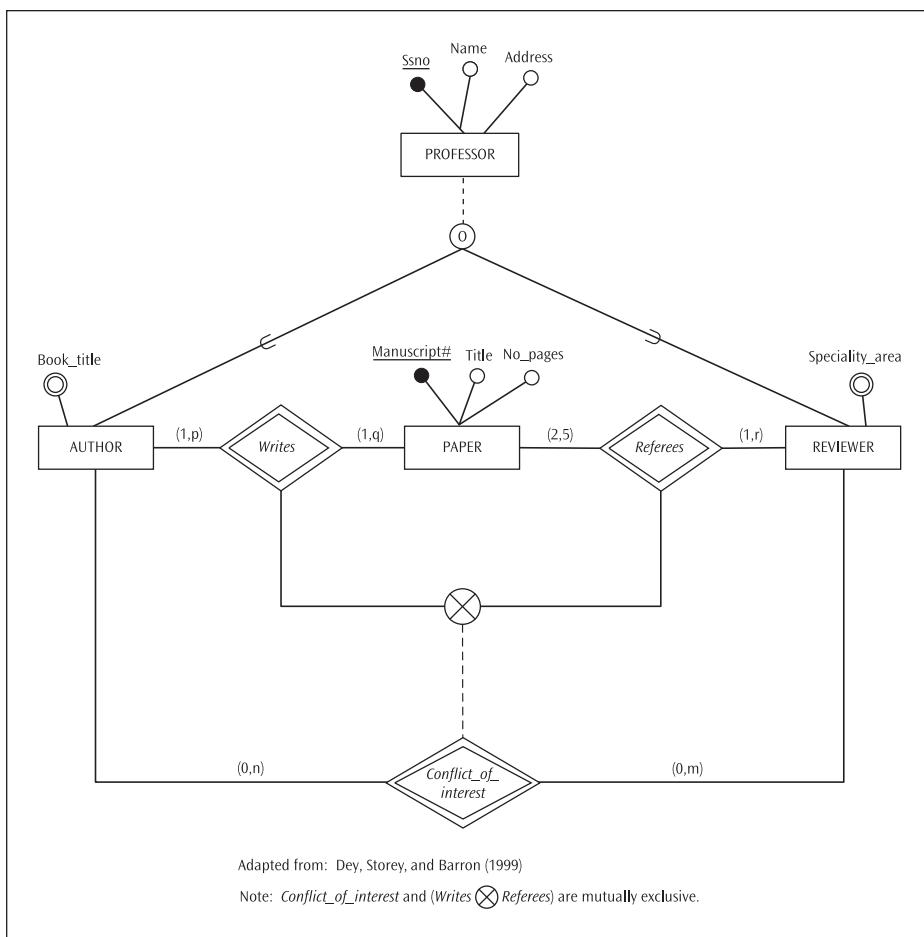
**FIGURE 5.35** A weak relationship type inclusion-dependent on a composite relationship type: A second example



**FIGURE 5.36** An ERD depicting a professor as an author and reviewer of papers

Suppose we want to impose a business rule: *A refereeing assignment cannot be made if there is a conflict of interest between an author and a reviewer.* Clearly, the author reviewing his or her own paper or the paper written by his or her advisor, student, etc., amounts to a conflict of interest. Can this conflict of interest be captured in this ERD?

Following Dey, Storey, and Barron (1999), this story line can be expressed as an exclusion dependency between (*Writes*  $\otimes$  *Referees*) and *Conflict\_of\_interest*, as shown in Figure 5.37. Here, the composite (*Writes*  $\otimes$  *Referees*) represents the reviewer who referees an author's paper, and *Conflict\_of\_interest* captures the reviewer who has a conflict of interest with the author. That is, *Conflict\_of\_interest* and (*Writes*  $\otimes$  *Referees*) are mutually exclusive. In other words, if *Conflict\_of\_interest* exists, then the composite (*Writes*  $\otimes$  *Referees*) cannot exist and thus the author cannot be assigned to review the paper. On the other hand, if the composite (*Writes*  $\otimes$  *Referees*) exists, then the *Conflict\_of\_interest* relationship cannot exist. Thus, the exclusion is bidirectional. Notice that all three relationships (*Writes*, *Referees*, and *Conflict\_of\_interest*) are modeled as weak relationship types since, unlike in inclusion dependency, there is no directionality in expressing an exclusion dependency. In other words, the order of the relationship types is immaterial. However, a relationship instance cannot be part of both the composite (weak) relationship type and the other weak relationship type at the same time. The exclusion dependency itself is indicated by a dotted line with no directional pointer connecting the weak relationship type *Conflict\_of\_interest* and the composite (weak) relationship type (*Writes*  $\otimes$  *Referees*).



**FIGURE 5.37** An example of exclusion dependency in an ERD

## 5.5 DECOMPOSITION OF COMPLEX RELATIONSHIP CONSTRUCTS

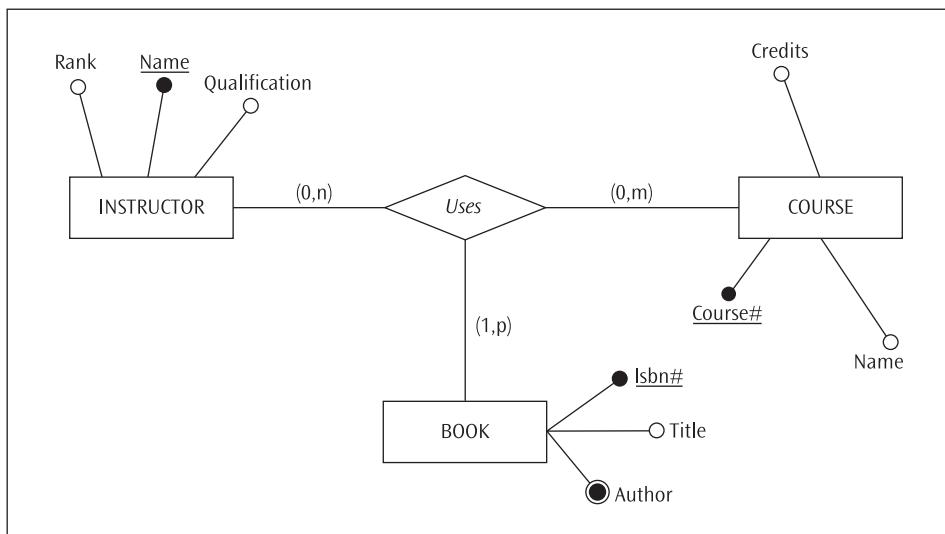
Some of the relationship constructs presented so far in this chapter require further decomposition before they are ready for mapping to the logical schema. In this section, these constructs are identified and their decomposition to the final form of Design-Specific layer is demonstrated.

234

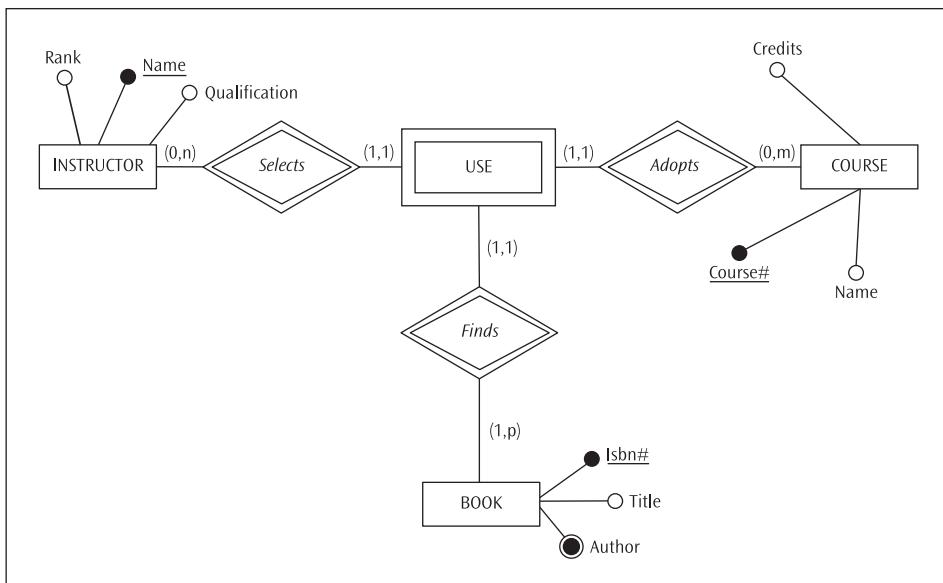
### 5.5.1 Decomposing Ternary and Higher-Order Relationship Types

In Section 3.2.4, we saw how a binary relationship type with an m:n cardinality ratio cannot be mapped to a logical schema “as is.” Therefore, the relationship was decomposed to a gerund entity type. The concept is the same for any n-ary relationship type. A relationship beyond degree two cannot be expressed “as is” in a logical schema. Thus, the decomposition of a ternary relationship type should transform the relationship so that the resulting transformation contains nothing beyond a set of binary relationship types and each of these binary relationships has cardinality ratio of the form 1:m. Converting the relationship type to a gerund entity type with all the participating base entity types as its identifying parents will accomplish this.

Consider the simple ternary relationship type *Uses* among INSTRUCTOR, COURSE, and BOOK depicted in the Design-Specific ERD shown in Figure 5.5. For the reader’s convenience, this diagram is reproduced in Figure 5.38. Based on the rationale stated above, the decomposition to the final form of the Design-Specific ERD is shown in Figure 5.39. Observe that the gerund entity type USE is a weak entity type with no partial key and three identifying parents (since *Use* is a ternary relationship type). The participation of USE in each of the three identifying relationship types *Selects*, *Adopts*, and *Finds* is total because USE exhibits existence dependency (i.e., min = 1) in all three identifying relationships.



**FIGURE 5.38** The ternary relationship type *Uses*



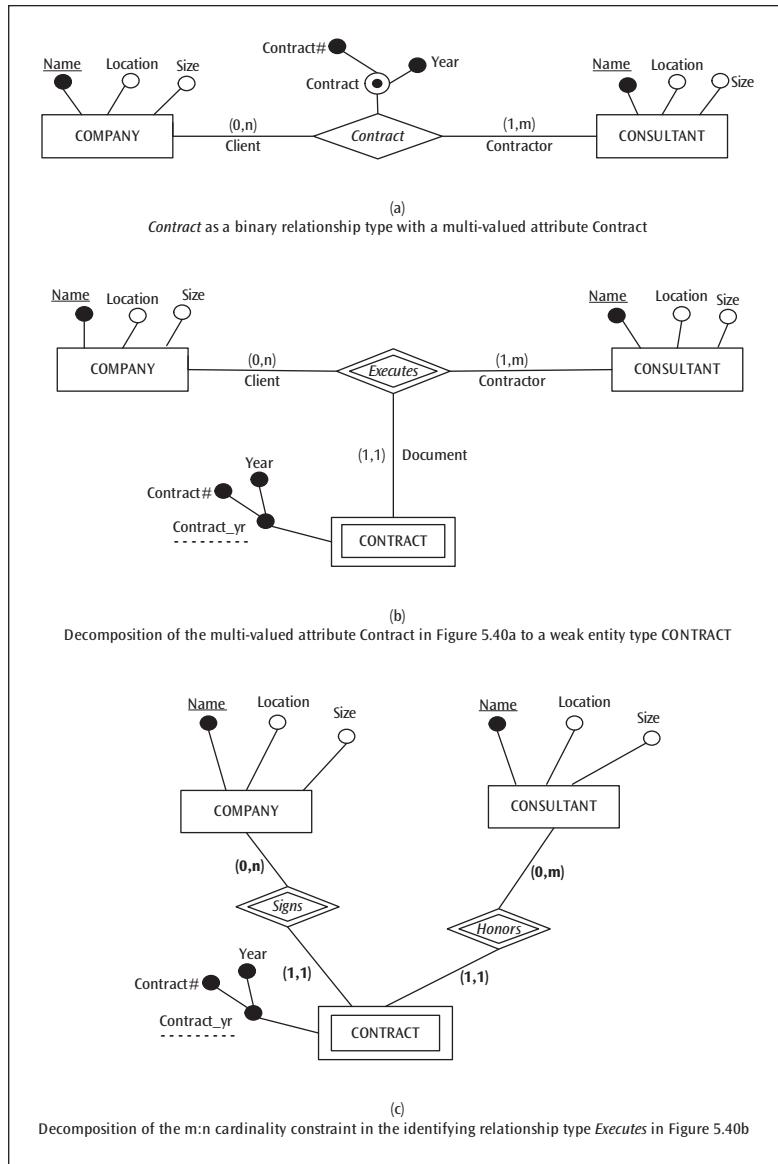
**FIGURE 5.39** Decomposition of the ternary relationship type *Uses* to the gerund entity type *USE*

The transformation process is the same for quaternary (degree four), quintary (degree five), and higher-order relationship types. The reader may, as an exercise, map the quaternary relationship *Performs* (Figure 5.13) to a gerund entity type *PERFORMS*.

### 5.5.2 Decomposing a Relationship Type with a Multi-Valued Attribute

Relationship types with a multi-valued attribute present another interesting case. Consider the *Contract* relationship type shown in Figure 5.23. For convenience, this figure is reproduced as Figure 5.40a. First of all, the cardinality ratio in this binary relationship type is of the form m:n. Thus, the relationship needs decomposition preparatory to a logical schema mapping. In addition, the relationship type has a multi-valued attribute. In the absence of the multi-valued attribute **Contract** in *Contract*, the relationship type would have been decomposed to a gerund entity type *CONTRACT* with two identifying parents, *COMPANY* and *CONSULTANT*. With the composite multi-valued attribute, the decomposition is somewhat similar except that the multi-valued attribute becomes the partial key of the weak entity type *CONTRACT*, as shown in Figure 5.40b. The decomposition at this point is not quite complete since the m:n relationship type is still present. The final decomposition appears in Figure 5.40c.

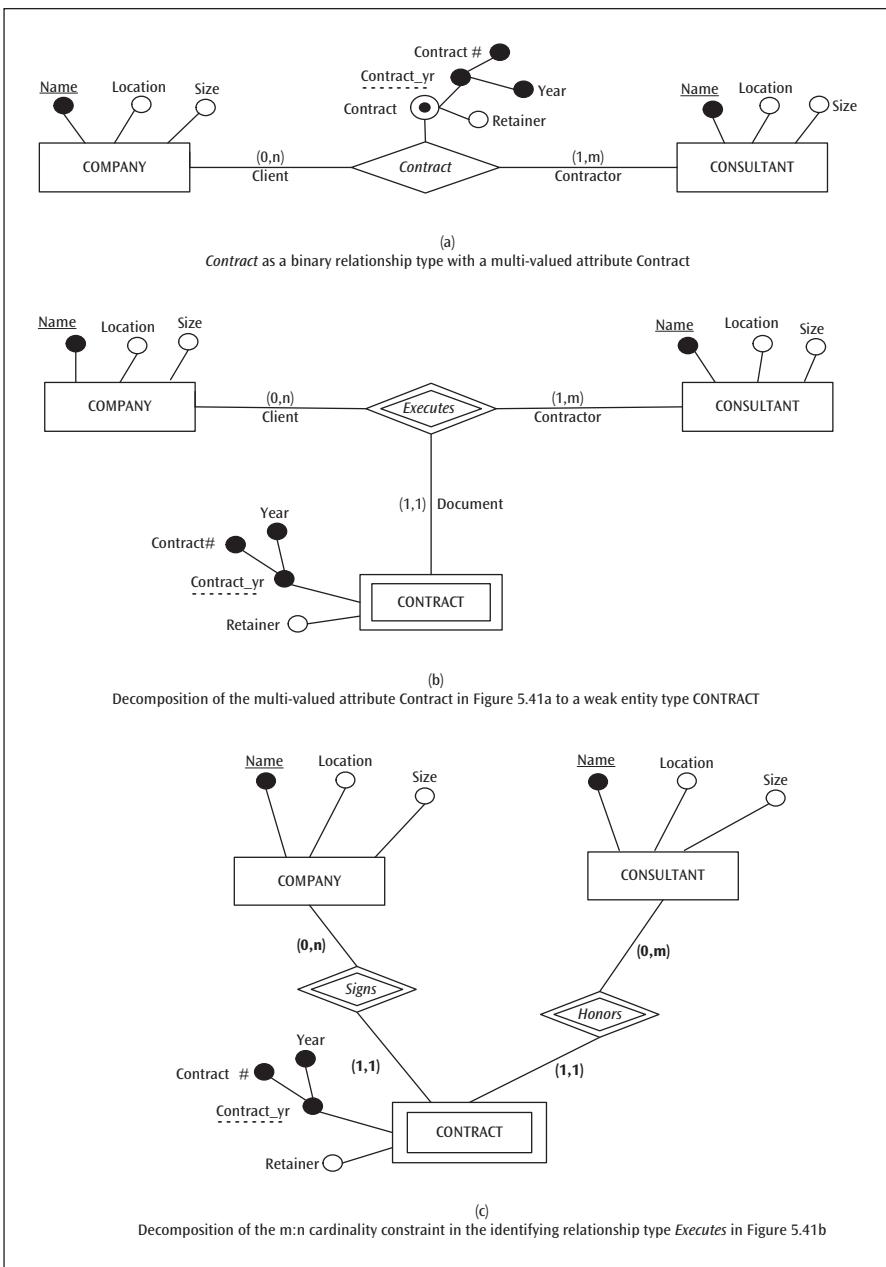
Let us build on the simple scenario of this contract between companies and consultants using additional business rules. While the contract may be boilerplate, as described in Section 5.2.6, let us just say that *as part of the contract there is a retainer fee that some companies pay their consultant(s), which can change as part of the contract provisions*. This entails only a minor change in the design ERD, as shown in Figure 5.41a. Since the **Retainer** is specified as a part of the contract, it is included as part of the composite multi-valued attribute **Contract**. Observe that since only part of the attribute **Contract** is necessary to serve as the discriminator, this part is explicitly identified here as the partial key (dotted underline). The decomposition is shown in Figure 5.41b. Although the decomposition here as well as in Figure 5.40b has a superficial appearance of a ternary relationship type, the semantics



**FIGURE 5.40** Decomposition of a m:n relationship type with a multi-valued attribute

conveyed by them are the same as in Figures 5.41a and 5.40a, respectively.<sup>9</sup> In fact, the degree of this relationship in the decomposition is not three. In order for this to qualify as a ternary relationship, *Execute* cannot be an identifying relationship type.

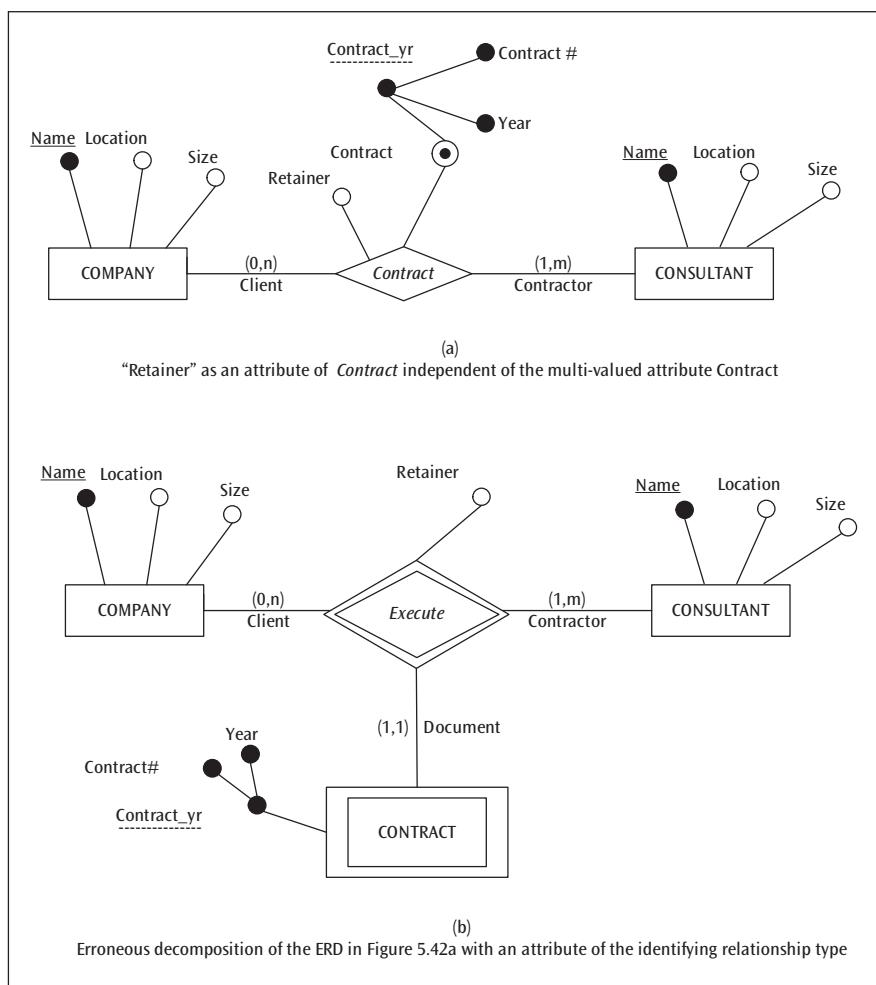
<sup>9</sup>In Chapter 6, we will see how the mapped logical schema conveys the semantics of the conceptual schema correctly.



**FIGURE 5.41** Decomposition of a relationship type with a partial key in a multi-valued attribute

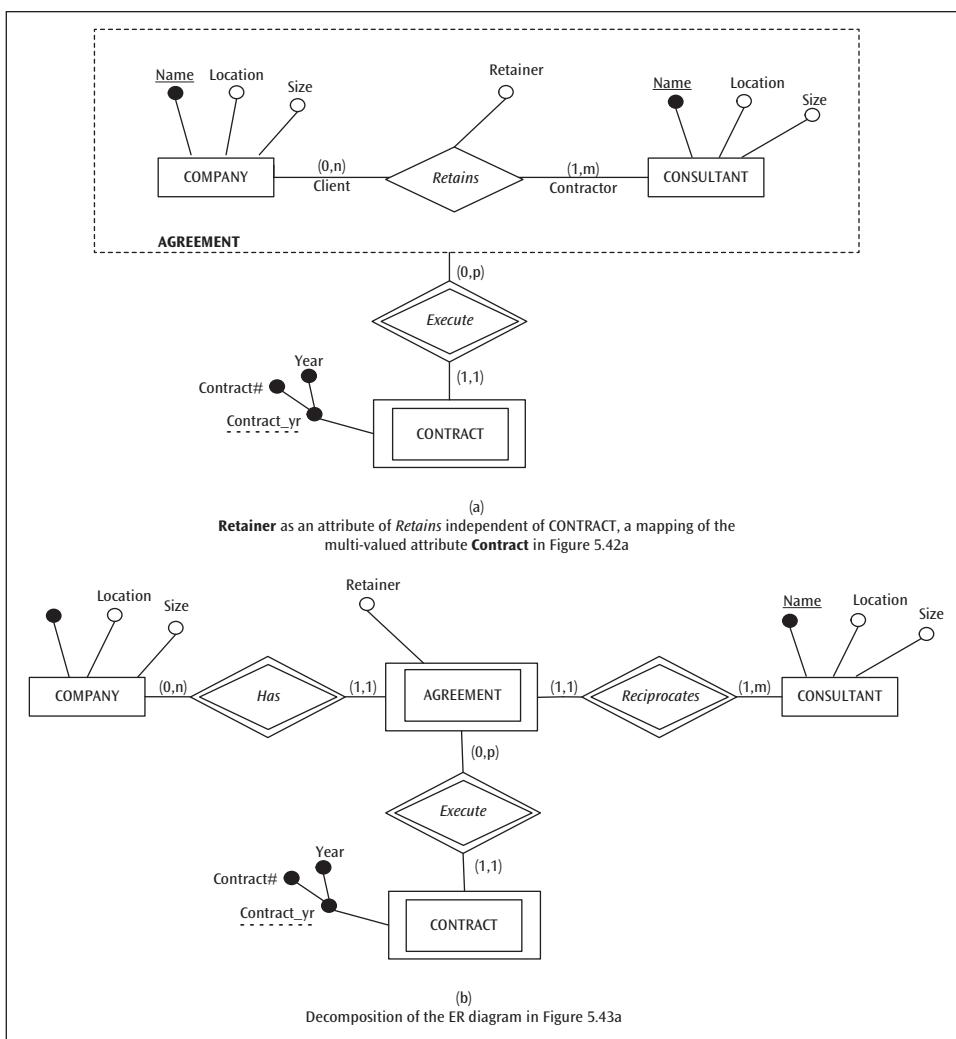
Let us now tweak the story line to refine the business rule about the retainer: *The retainer is independent of the contract*. That is, a company pays a retainer (a fixed amount) to a consultant for availability at short notice with or without a contract; this amount may change, but not as a part of the contract provisions. Moving the attribute **Retainer** out of the composite attribute **Contract**, as in Figure 5.42a, is the obvious solution. By decomposing the multi-valued

attribute of the m:n relationship type *Contract*, as was done in the previous scenario (see Figures 5.40 and 5.41), we end up with a weak entity type CONTRACT, with the two identifying parents COMPANY and CONSULTANT. Since **Retainer** is independent of CONTRACT, it cannot be an attribute of CONTRACT; instead, it is mapped as an attribute of the identifying relationship type *Execute*, as shown in Figure 5.42b. Is this mapping correct? While syntactically acceptable, this mapping generates a semantic error. Given that CONTRACT is the only child entity type in the relationship type *Execute*, **Retainer** (the attribute of this relationship type) will end up as an attribute of CONTRACT because it is the child entity type of the relationship type *Execute*. Then, the decomposition is no different from the ERD in Figure 5.41b. But then, the source ERDs from which the decompositions shown in Figures 5.42b and 5.41b are not semantically equivalent (see Figures 5.42a and 5.41a, respectively). In short, the mapping shown in Figure 5.42b fails to honor the business rule that *The retainer is independent of the contract*, which is expressed in Figure 5.42a. This essentially generalizes to a semantic rule that an identifying relationship type cannot have an attribute. Thus, Figure 5.42b is in error.



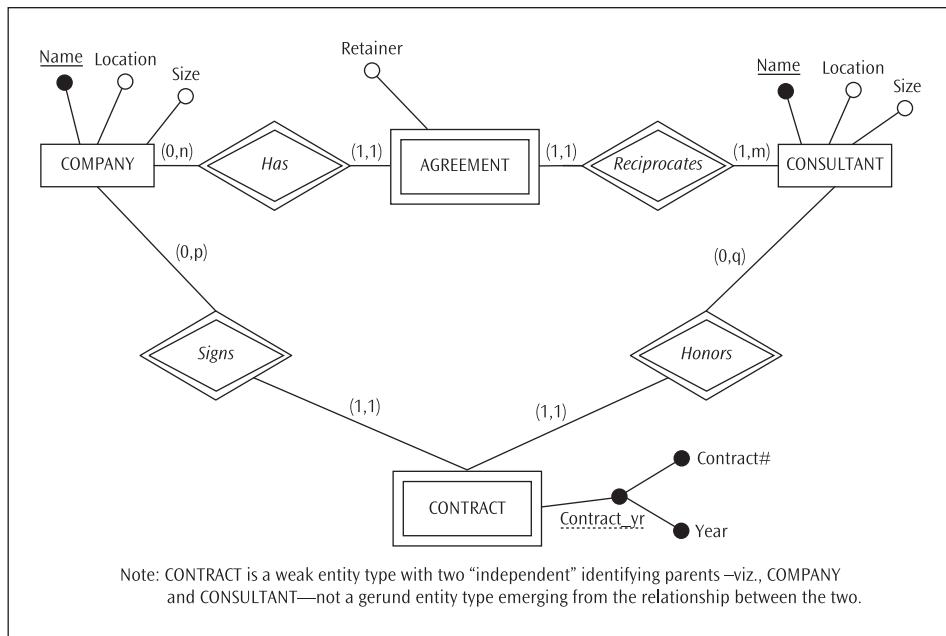
**FIGURE 5.42** ERD with a single-valued as well as a multi-valued attribute of a relationship type

An alternative two-step solution to the decomposition of the ERD of Figure 5.42a to a final Design-Specific ERD is presented below. As a first step, the m:n relationship type between COMPANY and CONSULTANT is preserved, with **Retainer** as the attribute of the relationship type *Retains*. The multi-valued attribute **Contract** is decomposed as a weak entity child of the cluster entity type AGREEMENT resulting from the *Retains* relationship type between COMPANY and CONSULTANT. This ERD appears in Figure 5.43a. In this design, **Retainer** exists independent of the CONTRACT even if no contract is in place, thus preserving the stated business rule of independence between **Retainer** and the multi-valued attribute **Contract**. The second step of the decomposition, shown in Figure 5.43b, transforms this design to the final form of the Design-Specific ERD.



**FIGURE 5.43** A two-step decomposition of the ERD shown in Figure 5.42a

Figure 5.44 is yet another alternative decomposition of the design presented in Figure 5.42a. Here, the modeling variation portrays CONTRACT as the weak entity child, with two identifying parents, COMPANY and CONSULTANT, instead of the weak entity child of the relationship between COMPANY and CONSULTANT—that is, AGREEMENT. Please note that the semantics conveyed by the decomposition shown in Figures 5.43b and 5.44 are not the same. The former model suggests that in order for a contract to exist between a company and a consultant, an agreement with or without a retainer value is required. Likewise, the presence of an agreement does mandate a contract between the same pair. The latter permits a company and a consultant to sign a contract without a formal agreement and also to have a retainer-based agreement without a contract.



**FIGURE 5.44** An alternative decomposition for the ERD in Figure 5.42a

Given the original source ERD in Figure 5.42a, which of the two decompositions—the one in Figure 5.43b or the one in Figure 5.44—more appropriately captures the import of the intended semantics of Figure 5.42a? The reader is encouraged to investigate this question.

### 5.5.3 Decomposing a Cluster Entity Type

A cluster entity type is essentially a virtual depiction sometimes requiring no further decomposition if the only purpose served is simply to express the cluster as a single collective semantic construct in a conceptual data model to enhance understanding. However, no matter how simple (Figure 5.10) or how involved (Figure 5.13 or Figure 5.22) a cluster entity type may be, if it is related to an entity type outside the cluster, then the cluster entity type will have to be decomposed, because a cluster entity type cannot be

expressed “as is” in the logical schema. The solution is to configure the cluster entity type either as a weak entity type or as a gerund entity type.

Invariably, there is a nucleus relationship within a cluster from which the cluster entity type emerges. Often, this relationship type gets distilled to an entity type due to previous decompositions, such as a gerund entity type resulting from the decomposition of an m:n binary or recursive relationship type or any other relationship type of higher degree. Otherwise, at this time, this nucleus relationship from which the cluster entity type emerges can be condensed to a gerund or weak entity type. Observe that the cluster entity type AGREEMENT in Figure 5.43a gets decomposed to the gerund entity type AGREEMENT, as seen in Figure 5.43b. Also, notice in Figure 5.14 that the relationship type *Offering* can be expressed as a gerund entity type to represent a cluster entity type called SECTION. However, in Figure 5.17, SECTION will decompose to a weak entity child of a relationship among INSTRUCTOR, COURSE, and QUARTER. The reader may work these out as exercises.

### 5.5.4 Decomposing Recursive Relationship Types

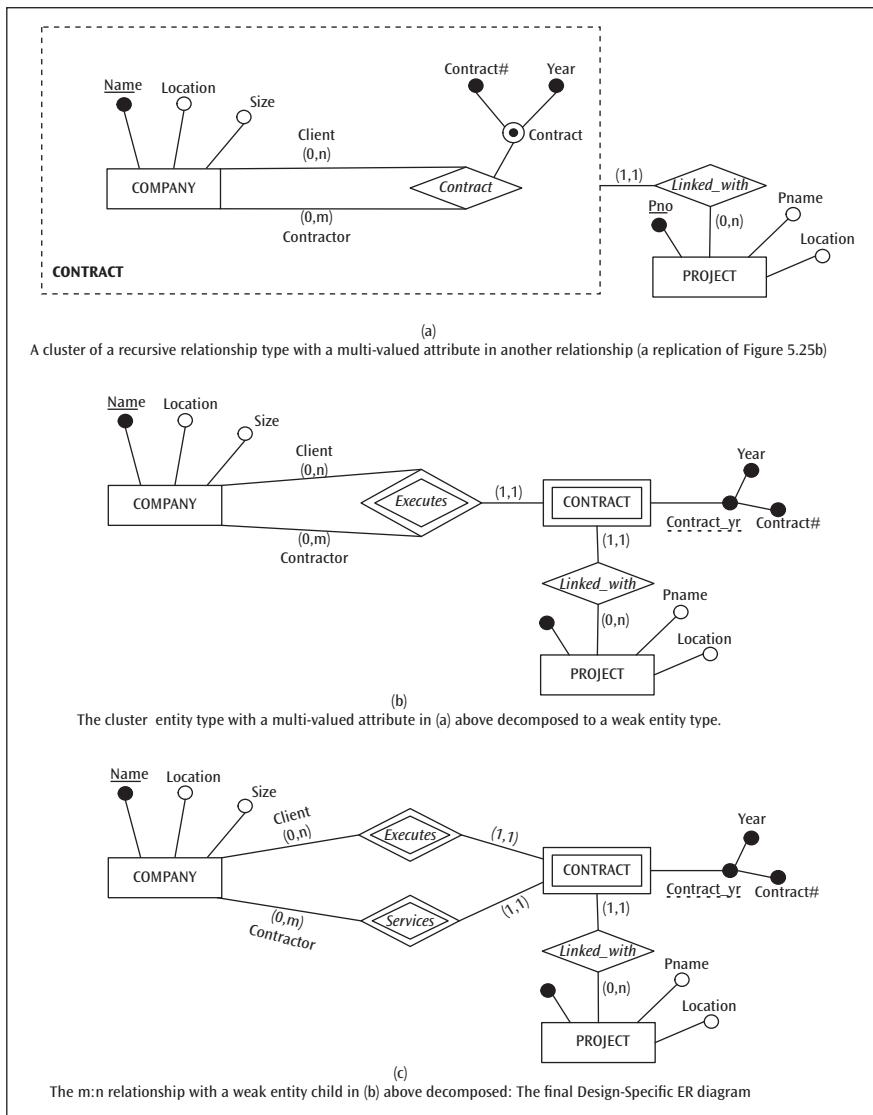
Decomposition of a recursive relationship is no different from the other examples discussed above except for some minor subtleties. An example ought to clarify this. Figure 5.45a is a direct replication of Figure 5.25b. The three-stage decomposition to the final form of the Design-Specific ERD displayed in 5.45 ought to be evident.

As a second example, let us consider the decomposition of the ERDs depicting a scenario of connection between two flights. While discussing this scenario earlier in the section (see Figures 5.26a and 5.26b), it was proposed that the difference between the two ERDs will get further clarified after the final decomposition of the two models.

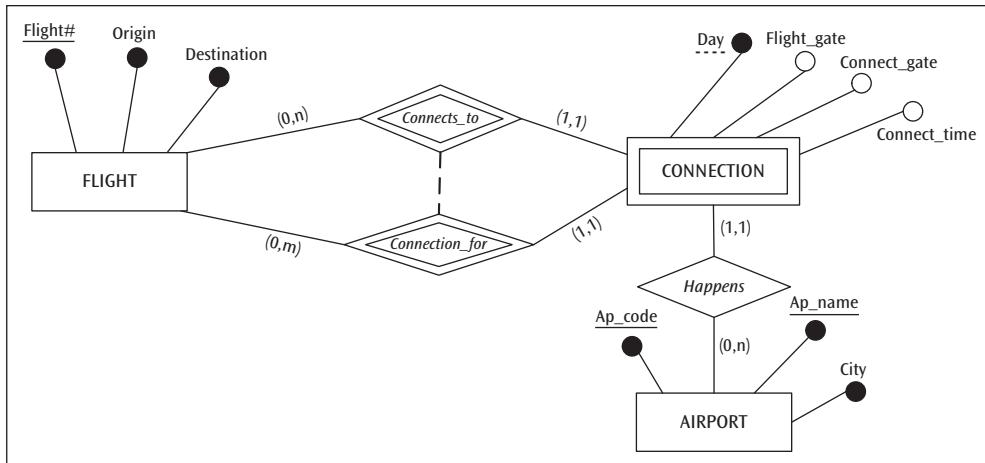
Figure 5.26a depicts the cluster entity type FLIGHT\_CONNECTION as a temporal event; that is, a flight connection is defined as an event on a given day between two flights. The decomposition of Figure 5.26a to the final form of a Design-Specific ERD is shown in Figure 5.46a, where CONNECTION is modeled as a weak entity type with two identifying parents, both being entities of FLIGHT; the partial key **Day** serves as the temporal discriminator. While the structural constraints of the relationship between CONNECTION and AIRPORT ensure that a connection happens only in one airport, that one airport need not be the same airport on different days. This is due to the temporal property modeled as a characteristic of the entity type CONNECTION in Figures 5.26a and 5.46a.

The ERD in Figure 5.26b, on the other hand, models the cluster entity type FLIGHT\_CONNECTION as a non-temporal occurrence. This property gets well clarified in the decomposition shown in Figure 5.46b. The gerund entity type CONNECTION simply depicts a relationship between two flights, irrespective of the day(s) on which these connections happen. Thus, a connection between any two flights occurs not just in one airport but in the same airport on all days when the connections occur. The gerund entity type also serves as the identifying parent of the weak entity type SCHEDULE, which captures the temporal aspect of the FLIGHT\_CONNECTION independent of the non-temporal aspect of the relationship with the AIRPORT.

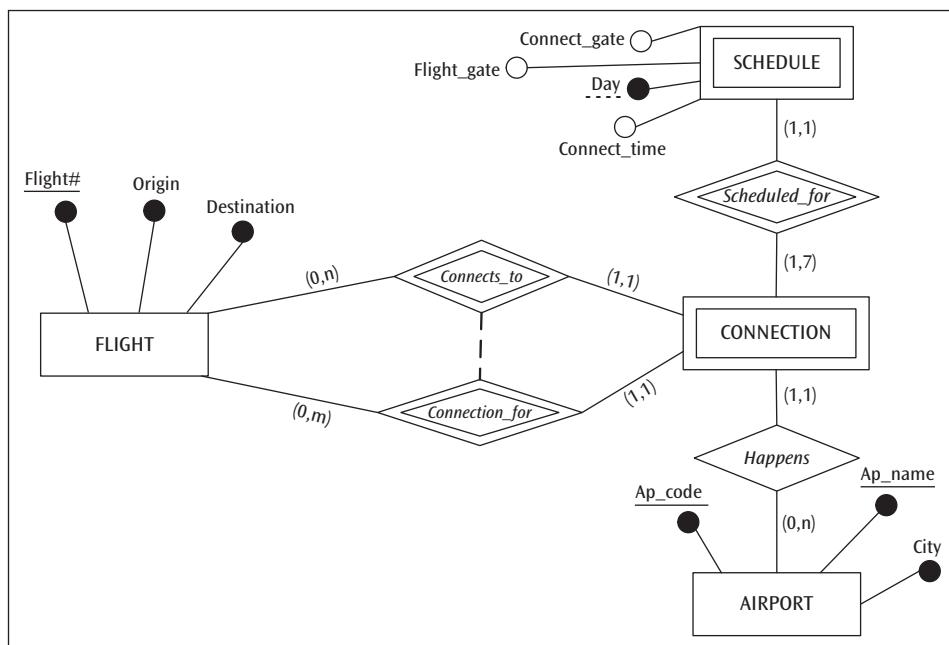
How does one model in Figures 5.46a and 5.46b the business rule that a flight cannot connect to itself? The exclusion dependency between the relationship types *Connects\_to* and *Connection\_for* imposes this constraint. Observe that the two identifying relationship types *Connects\_to* and *Connection\_for* also serve the role of weak relationship type for the inter-relationship constraint in these ERDs.



**FIGURE 5.45** Decomposition of a recursive relationship to Design-Specific ERD



**FIGURE 5.46a** CONNECTION between two flights modeled as a temporal event



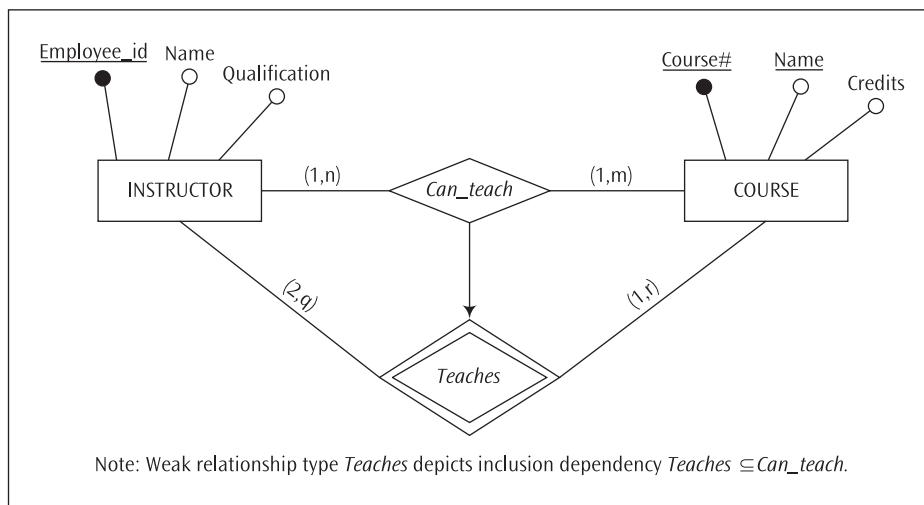
**FIGURE 5.46b** CONNECTION between two flights modeled as a non-temporal event

### 5.5.5 Decomposing a Weak Relationship Type

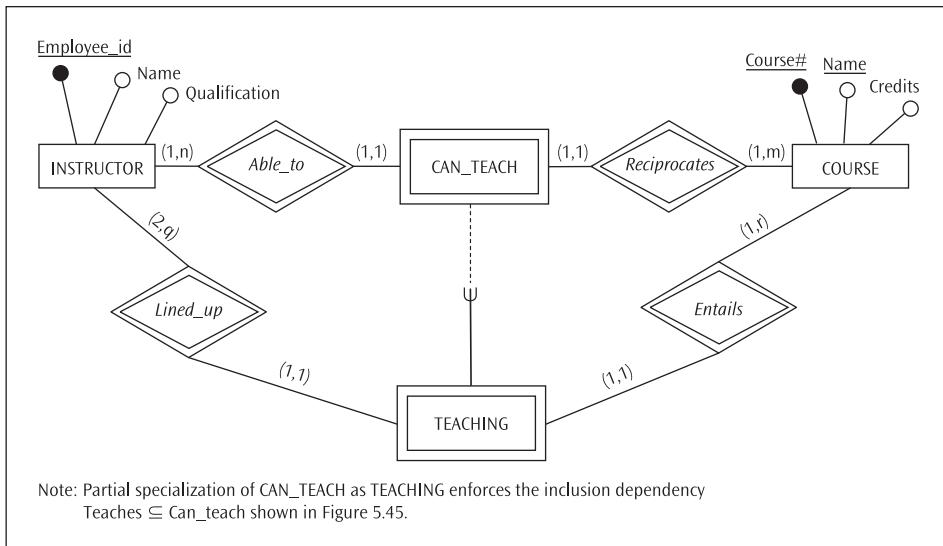
Business rules expressed through weak relationship types sometimes do not require any decomposition before mapping to a logical schema. For instance, in Figure 5.28, since the cardinality constraint of either one of the relationship types, *Works\_in* or *Managed\_by*, is of the form m:n, it is possible to map this ERD to the logical tier without any additional decomposition of the inter-relationship constraint expressed through the weak relationship type.

At other times, decomposition is necessary and possible. There are also times when some decomposition coupled with specification of semantic integrity constraints is required to fully capture a business rule. Occasionally, a business rule expressed through a weak relationship type may not be amenable to any decomposition and so will have to be carried forward as a semantic integrity constraint. Since the various cases in this chapter are highly context sensitive, they are not covered with an exhaustive set of illustrations.

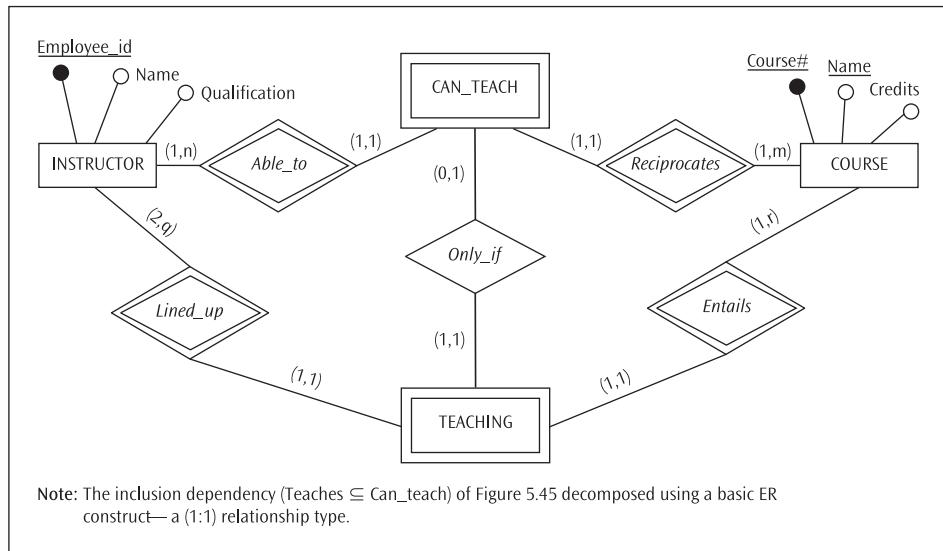
However, in order to get a general idea about how to decompose a weak relationship type to render it ready for transformation to a logical schema, let us review a situation where a decomposition is necessary using the example that appeared in Figure 5.29. For convenience, this figure is reproduced as Figure 5.47a. The two binary relationships, *Can\_teach* and *Teaches*, between INSTRUCTOR and COURSE require no further elaboration. However, how does one interpret the business rule *In order to teach a course, an instructor should be capable of teaching that course*, implemented by the inclusion dependency  $\text{Teaches} \subseteq \text{Can\_teach}$ , in the ERD? The answer is that this rule translates to the gerund entity type TEACHING having existence dependency on the gerund entity type CAN\_TEACH. This can be implemented in the design via a partial specialization of CAN\_TEACH with TEACHING as its subclass, as shown in Figure 5.47b. Figure 5.47c shows a less desirable alternative decomposition that is equivalent to the design appearing in Figure 5.47b.



**FIGURE 5.47a** *Teaches* as a condition-precedent weak relationship type



**FIGURE 5.47b** Decomposition of the weak relationship type *Teaches* using the EER construct Specialization



**FIGURE 5.47c** Decomposition of the weak relationship type *Teaches*: an alternative design

## 5.6 VALIDATION OF THE CONCEPTUAL DESIGN

Up to this point in this book, we have gone through a detailed requirements-to-model process using the ER modeling grammar and have implicitly assumed that this process takes care of or explains to the reader the outcome—namely, the conceptual model. A formal or informal investigation of how well the developed conceptual model answers the explicit and/or implicit questions present in the requirements specification has not been undertaken. In other words, the ER model has not undergone a critical inspection in light of the specified requirements and been validated accordingly. Having identified the entity types, their respective attributes, and the various types of relationships among the entity types, it is also necessary to verify that the conceptual model thus developed is a true representation of the “universe of interest” being modeled.

A religious adherence to the rules of the modeling grammar ensures that the conceptual model is syntactically correct. A clear understanding of the business rules implied in the requirements specification often leads to a semantically correct conceptual model. Occasionally, misinterpretations of the semantics conveyed by the requirements specification lead to a semantically incomplete conceptual model. This pitfall can be avoided, however, through a systematic validation of the developed conceptual model. Since this book uses the ER modeling grammar for conceptual modeling, this section is devoted to validation of ER models. The textual component of the ER model—namely, the semantic integrity constraints—lists the business rules not captured in the ERD. Thus, the focus of this section is on the validation of the semantics captured in the ERD—in particular, the semantics captured by the relationship types modeled.

In his 1989 book *Data Analysis for Data Base Design*, David Howe refers to errors caused in an ERD by the misinterpretation of relationships as **connection traps**. Two basic types of connection traps are common: the **fan trap** and the **chasm trap**. Since connection traps emanate from the structural aspects of the ER modeling grammar, an ERD may contain several potential connection traps. Many of these, however, may be of no significance in the context of the requirements specification, whereas others can be eliminated by restructuring the ERD appropriately. The idea is to evaluate each potential connection trap for possible obstruction to semantic completeness.

Misinterpretation of the meaning of any relationship implied by the requirements specification is fundamental to connection traps in general. In order to avoid errors of misinterpretation, it is imperative that the analyst carefully define the relationship semantics and that the designer thoroughly understand them.

### 5.6.1 Fan Trap

A **relationship fan** emerges when two or more relationship types “fan out” from (cardinality constraint of 1:n) or “fan in” to (cardinality constraint of n:1) a particular entity type. In other words, when an entity type serves as the focal point (parent or child) in more than one relationship type on which it is not identification dependent, a relationship fan

occurs.<sup>10</sup> A fan trap results when the pathway between certain entities in the relationship fan becomes ambiguous. The following example illustrates a fan trap:

*Suppose a library has a large membership and each patron is a member of exactly one library. Every library also stocks a lot of books, while a specific book can only be in one library. Every patron borrows at least one book, and every book is borrowed by a patron. A patron may borrow books only from the library in which he or she is a member. Since we are considering the database environment at a given point in time, a book is borrowed by only one patron.*

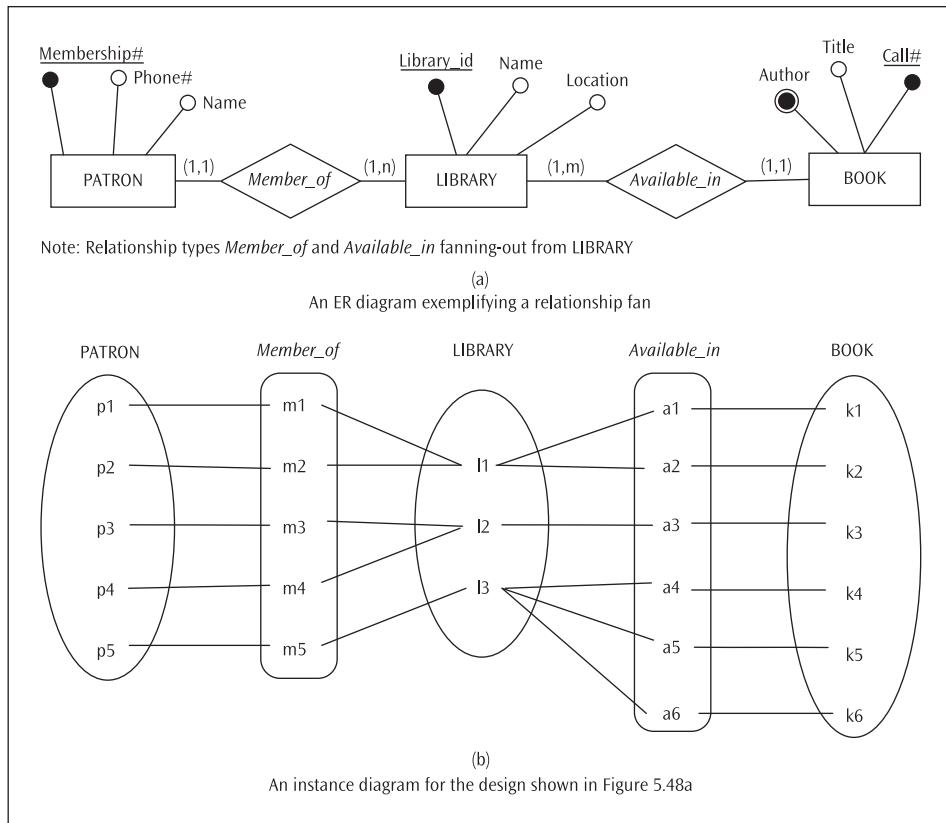
The ERD in Figure 5.48a models this scenario. The relevant entity types and their attributes are arbitrarily assigned. At the outset, it may appear that the connection of PATRON to BOOK via LIBRARY will facilitate deduction of which book(s) is/are available for borrowing to which patron. But a closer scrutiny of the ERD reveals this not to be the case. For instance, from the description of the scenario, one may reasonably expect questions about the following to be answered by the design shown in Figure 5.48a:

1. Number of members in a given library
2. Library in which a particular patron has membership
3. Library in which a particular book is present
4. Number of books in a given library
5. Number of books borrowed by a patron
6. The patron who has borrowed a particular book

Observe that it is impossible to answer questions about items 5 and 6 from the current design. The design then, while semantically correct, is not semantically complete. The cause of this error could be a fan trap present in the design. Clearly, relationship types *Member\_of* and *Available\_in* are “fanning out” from LIBRARY. Thus, a relationship fan does exist. What we need to investigate is whether the relationship fan, in this case, creates ambiguity in the pathway between patrons and books, resulting in a fan trap.

The instance diagram (see Figure 5.48b), reflecting a legal state of the relationships indicated in the ERD, facilitates the investigation. The pathway in the relationship fan clearly shows that p5 can borrow three books (k4, k5, and k6); no one else can borrow these three books. But can we know the number of books borrowed by patron p1 or p2? The answer is “No.” Also, can we know who borrowed the books k1 and k2? The answer, again, is “No.” We do know that p1 and p2 are members of the library l1, and could have borrowed a book only from l1. We also know that the books k1 and k2 are available for borrowing only from l1. From these two facts, it is impossible to infer who between p1 and p2 borrowed which of the two books, k1 and k2. Likewise, it is impossible to infer who between p3 and p4 could have borrowed the book k3. That is, the pathway connecting patrons and books through the relationships has ambiguity, and it is not possible to answer

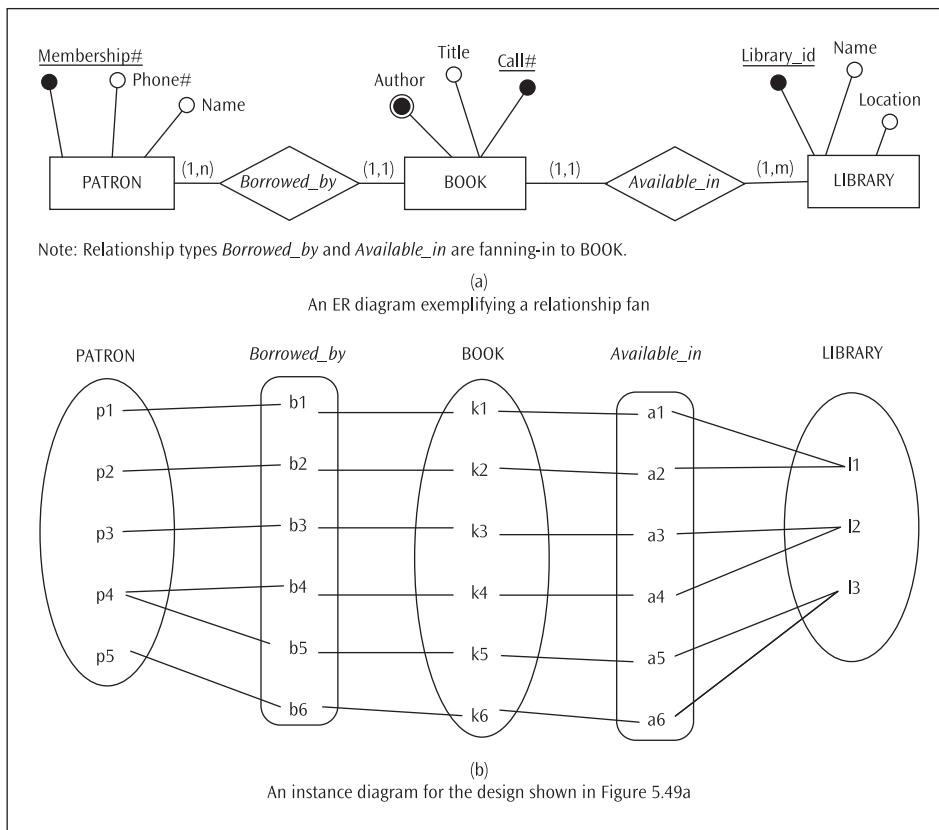
<sup>10</sup>Decomposition of an m:n relationship type results in a gerund entity type that, when viewed as the focal point (in this case, child) into which two relationship types fan in, should not be misconstrued as a relationship fan because the gerund entity type does not have an independent existence as an entity type—that is, it is identification-dependent on the entity types constituting the original m:n relationship type. Thus, the apparent fan structure emanating from the gerund entity type is at best a trivial relationship fan incapable of a potential fan trap.

**FIGURE 5.48** An example of a fan trap

questions about items 5 and 6 in the list of questions from this design. This is because the relationship fan here is causing this ambiguity and so is a fan trap. The reason this is a “trap” is because, superficially, the design appears to provide an unambiguous pathway between PATRON and BOOK, although in reality it doesn’t.

Figure 5.49 depicts an alternative design. This design is also syntactically and semantically correct. Furthermore, the design answers questions about items 5 and 6 that are not answerable using the design presented in Figure 5.48. So, is this design the correct solution? Is it semantically complete in the context of the stated scenario and the list of anticipated questions? Let us investigate. The instance diagram in Figure 5.49b reflects a legal state of the relationships indicated in the ERD that appears in Figure 5.49a. As per the design shown in the ERD (Figure 5.49a), a patron may borrow several books. So, the patron p4 has borrowed the books k4 and k5. Likewise, a library may have many books. Observe that libraries l1, l2, and l3 have two books each. A book, however, can be in only one library. Accordingly, the book k4 is in library l2 and book k5 is in l3. In short, the instance diagram does not violate any relationship constraint defined in the ERD. If we navigate through the available pathway in the design from p4 to the library entities, it is seen that p4 is linked to libraries l2 and l3. Incidentally, the only link available from the entity type PATRON to the entity type LIBRARY is through the entity type BOOK. So, the

inevitable inference about membership from this design is that p4 is a member of two libraries. Thus, the design violates a business rule of the stated scenario that *each patron is a member of exactly one library*. Consequently, the design also yields wrong answers to questions about items 1 and 2 in the list. The cause of this error could be a fan trap present in the design. Clearly, relationship types *Borrowed\_by* and *Available\_in* are “fanning in” to *BOOK*. So, a relationship fan does exist. Our investigation reveals that the relationship fan, in this case, does create ambiguity in the pathway between patrons and libraries resulting in a fan trap. The design then, while syntactically correct, is not semantically correct, but far less complete.

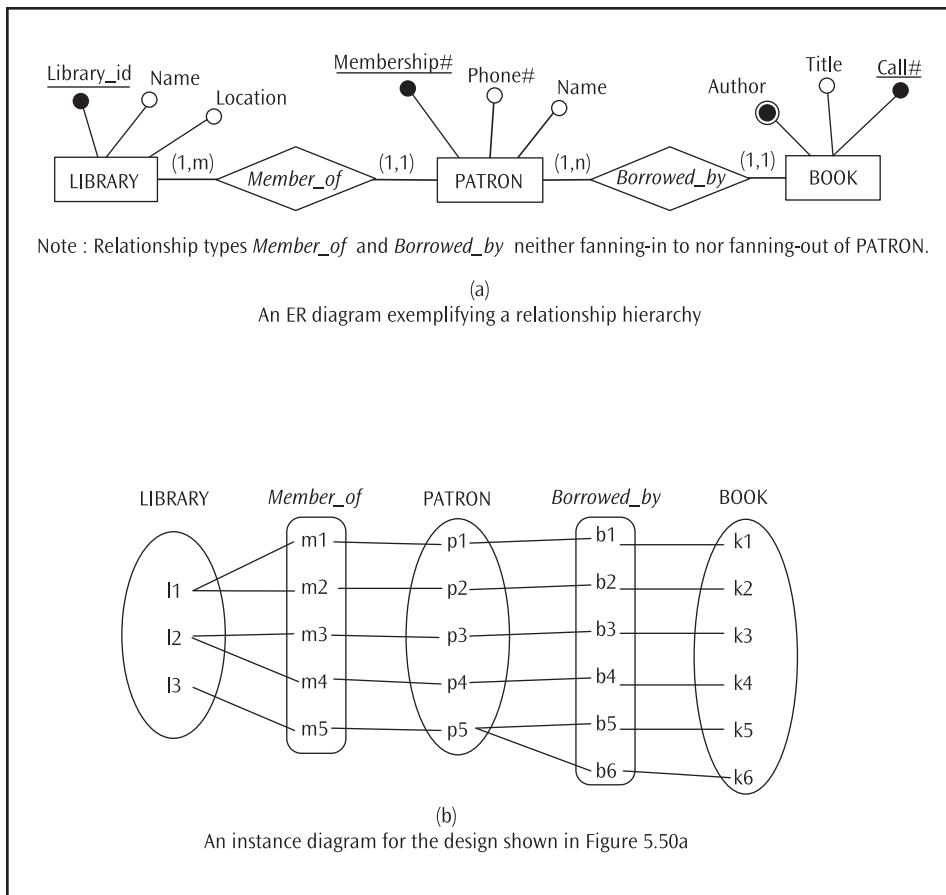


**FIGURE 5.49** An alternative solution for the design shown in Figure 5.48

One way to prove that the relationship fans in the designs shown in Figures 5.48 and 5.49 are indeed fan traps is to demonstrate the absence of ambiguous pathways among the entities in a design that is free of fan traps (proof by contradiction). Figure 5.50 is a design of the same scenario restructured to eliminate relationship fans. In fact, the ERD depicts a relationship hierarchy. The instance diagram of Figure 5.50b demonstrates that questions

pertaining to all six items about the scenario are answered using the design shown in Figure 5.50a. The ERD that appears in Figure 5.50a is:

- syntactically correct, as are the ERDs in Figure 5.48a and 5.49a
- semantically correct, as is the ERD in Figure 5.48a, in that both portray the scenario specified equally accurate (whereas Figure 5.49a has been shown to be semantically incorrect)
- semantically complete because it does not have a fan trap, while the designs in Figures 5.48a and 5.49a are plagued by fan traps and therefore are semantically incomplete

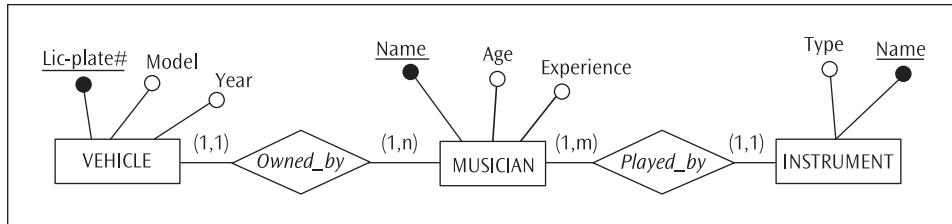


**FIGURE 5.50** Resolution of the fan trap present in Figures 5.48 and 5.49

That said, it is important to understand that all relationship fans are not necessarily fan traps. If structurally apparent fan traps are not of any semantic significance in the context of the requirements specification prevailing over its scenario, then those relationship fans are not fan traps. Thus, unconditional avoidance of relationship fans in

ERDs limits the richness of the ER modeling grammar and is not recommended. For example, the scenario depicted by the ERD in Figure 5.51 expresses the following story line:

*A music group has several musicians. Every musician owns one or more vehicles, and a given vehicle is owned by only one musician. Likewise, a musician can play several instruments, but in this group an instrument is played by only one musician.*



**FIGURE 5.51** A relationship fan semantically irrelevant as a fan trap

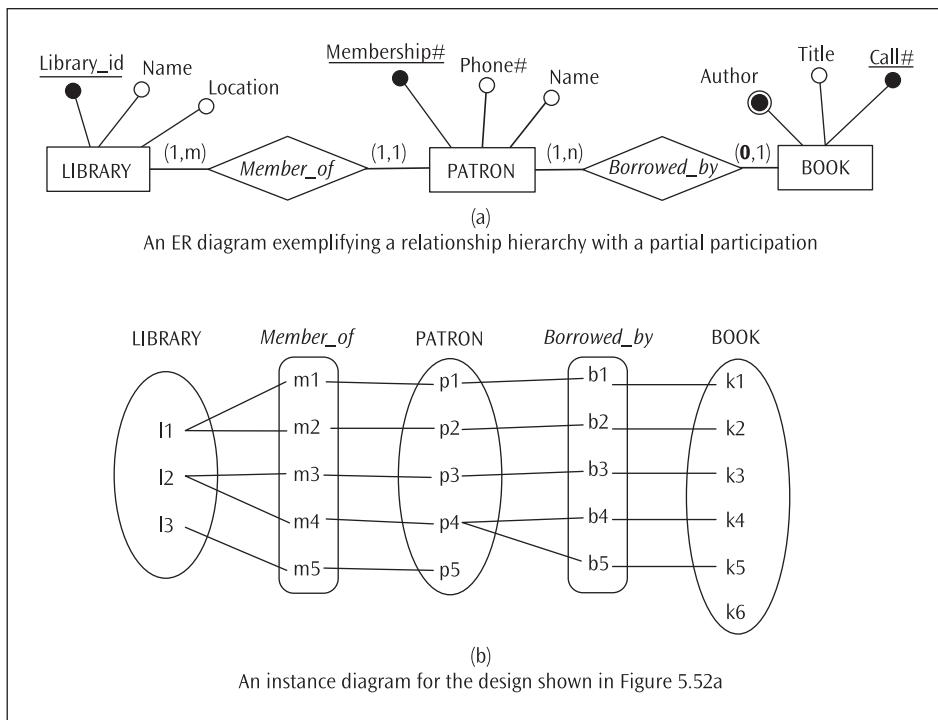
To begin with, the relevant entity types and their attributes in the ERD (Figure 5.51) are arbitrarily assigned. Observe that a relationship fan exists in this design—two distinct relationship types, *Owned\_by* and *Played\_by*, fan out of the entity type **MUSICIAN**. Is there a fan trap inherent in the design? It is true that there can be ambiguities in the pathway between **VEHICLE** and **INSTRUMENT**. If questions like “what vehicle does a musician own while playing a guitar” or “how many instruments does a musician play while owning an SUV” are semantically relevant, then this relationship fan indeed constitutes a fan trap. In other words, since a pathway between **VEHICLE** and **INSTRUMENT** is not semantically relevant in the story line, any ambiguity in the pathway caused by the structural arrangement in the design is irrelevant. Therefore, the relationship fan, in this case, does not cause a fan trap.

## 5.6.2 Chasm Trap

A chasm trap occurs when a design models certain relationship types but a pathway does not exist between certain entities through the defined relationships. A chasm trap may occur when there is at least one relationship type in the pathway with optional participation of the referencing (child) entity type.

Suppose we impose a new business rule in the library scenario we discussed in the previous section: *Some books are not borrowed by any patron*. Since the design presented in Figure 5.50 is free of any fan traps, let us impose this new business rule on the ERD in Figure 5.50a. In fact, this design also models a direct relationship between patrons and books. The business rule is incorporated in the design by rendering optional the participation of **BOOK** in the relationship type *Borrowed\_by*. The revised ERD and a corresponding instance diagram are presented in Figure 5.52. Observe that book k6 is not borrowed by any patron. This is reflected in the ERD by the min = 0 in the participation of **BOOK** in *Borrowed\_by*. Suppose one asks, “Which library holds the book k6?” The current design fails to answer this question. In fact, the current design fails to answer items 3 and 4 in the

list provided in Section 5.6.1. The original designs shown in Figures 5.48 and 5.49 do indeed answer this particular question but are unacceptable solutions because the presence of fan traps in these designs raises other semantic issues relevant to the scenario. So, what is the solution?



**FIGURE 5.52** An example of a chasm trap

An ERD with an additional independent relationship type *Available\_in* connecting LIBRARY and BOOK directly, as shown in Figure 5.53, results in a design free of connection traps in the context of the scenario and the associated business rules being modeled here. It is important to understand that the addition of the relationship type *Available\_in* is intended to supplant the effect of the chasm trap induced by the optional participation ( $\min = 0$ ) of BOOK in the *Borrowed\_by* relationship type. In other words, mandatory participation ( $\min = 1$ ) of BOOK in the *Borrowed\_by* relationship type preempts the presence of a chasm trap, rendering redundant the addition of the relationship type *Available\_in*. However, it is not the prerogative of the designer to make up business rules to resolve connection traps. The purpose of this discussion is to sensitize the designer to the nuances in the available design options so that he or she can carry on informed interaction with the user community during the development of the requirements specification for the application domain and avoid deceptive pitfalls in the design.

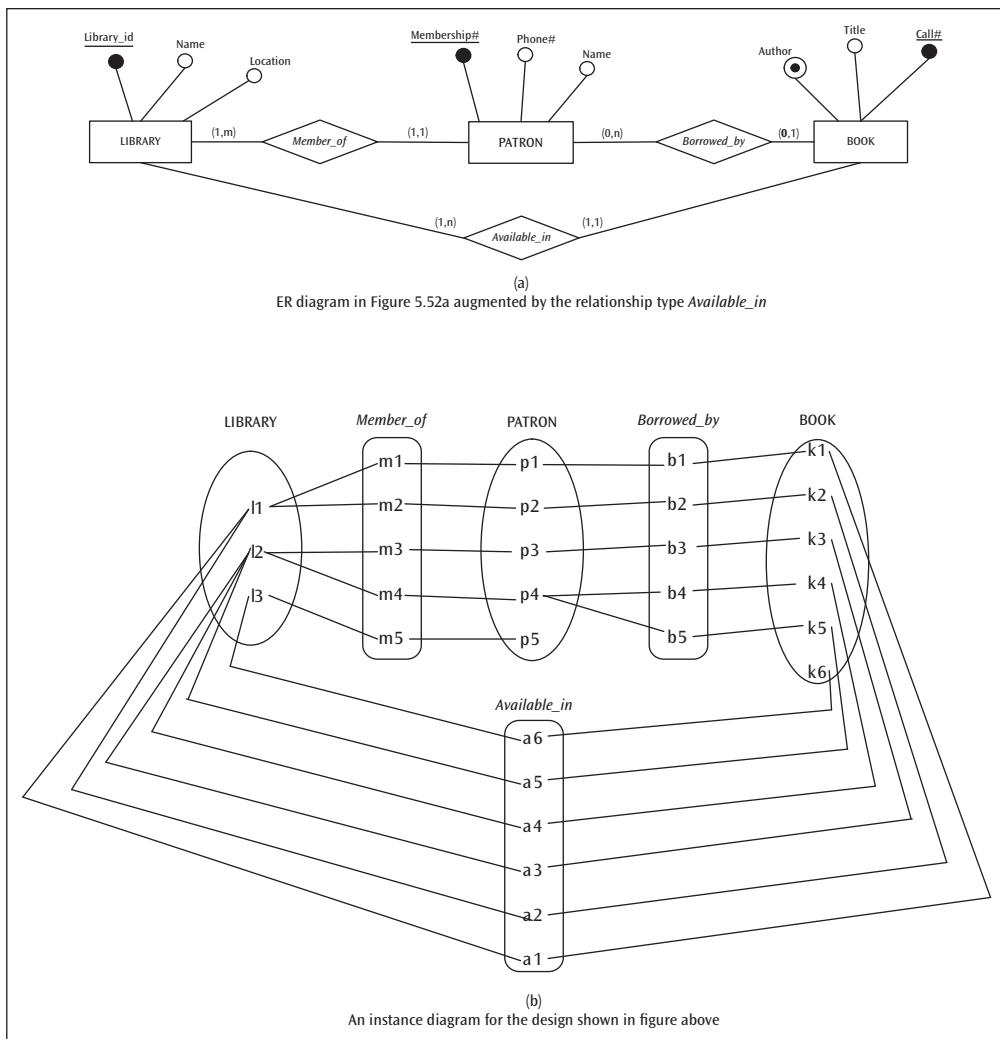


FIGURE 5.53 Final design free of connection traps for the scenario specified

### 5.6.3 Miscellaneous Semantic Traps

While fan traps and chasm traps are common occurrences in a conceptual design and are also easily recognizable due to their defined structural patterns, there are other semantic traps that are more difficult to identify because they do not necessarily conform to predefined structural patterns. Presumably, such semantic traps are also less common occurrences in simple conceptual designs. In this section, we review a couple of such complex connection traps.

### 5.6.3.1 Vignette 5

Suppose vendors supply products to projects. A critical thing to know is which vendor supplies what product to which project and the frequency of the supplies. In addition, say there is another business rule: A project can get a specific product only from one vendor. This does not preclude a project from getting other products from the same and other vendors. It only restricts a project from getting the same product from several vendors.

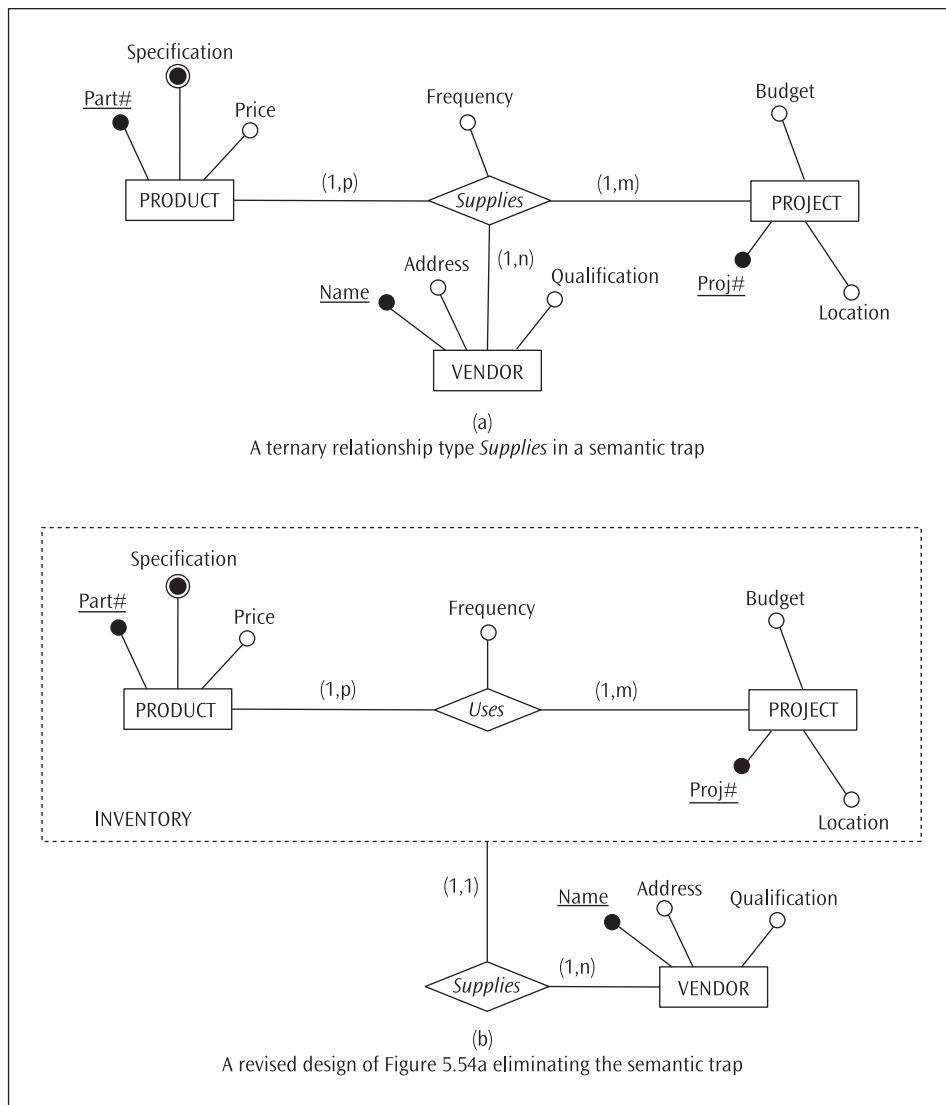
Based on the several examples provided at the beginning of this chapter (see Section 5.1), it appears that a ternary relationship among entity types VENDOR, PRODUCT, and PROJECT, with **Frequency** as the attribute of the relationship type, captures this scenario. This design is presented in the ERD shown in Figure 5.54a. Does this ERD capture the business rule *A project can get a specific product only from one vendor*? The answer is “No.” From a modeling perspective, what needs to be accomplished is that a {product, project} pair must be restricted to a relationship with just one vendor. The instinctive reaction to this constraint specification is to change the structural constraints of VENDOR in the *Supplies* relationship from (1, n) to (1, 1). This is a semantic trap in that the change does more than what the business rule specifies. That is, not only can a project get a specific product from just one vendor, as required by the business rule, but a vendor can supply no more than one product, and that product can be supplied to no more than one project. This unexpected side effect amounts to a semantic trap.

The solution lies in restricting the relationship of a {product, project} pair to just one vendor while permitting a vendor to relate to multiple {product, project} pairs. This cannot be done by manipulating the structural constraints of a ternary relationship type among PRODUCT, PROJECT, and VENDOR. This is accomplished by restructuring the ternary relationship type to two binary relationships:

- Rendering an m:n relationship type connecting PRODUCT and PROJECT to a cluster entity type, and
- Specifying a 1:n relationship type between VENDOR and the cluster entity type

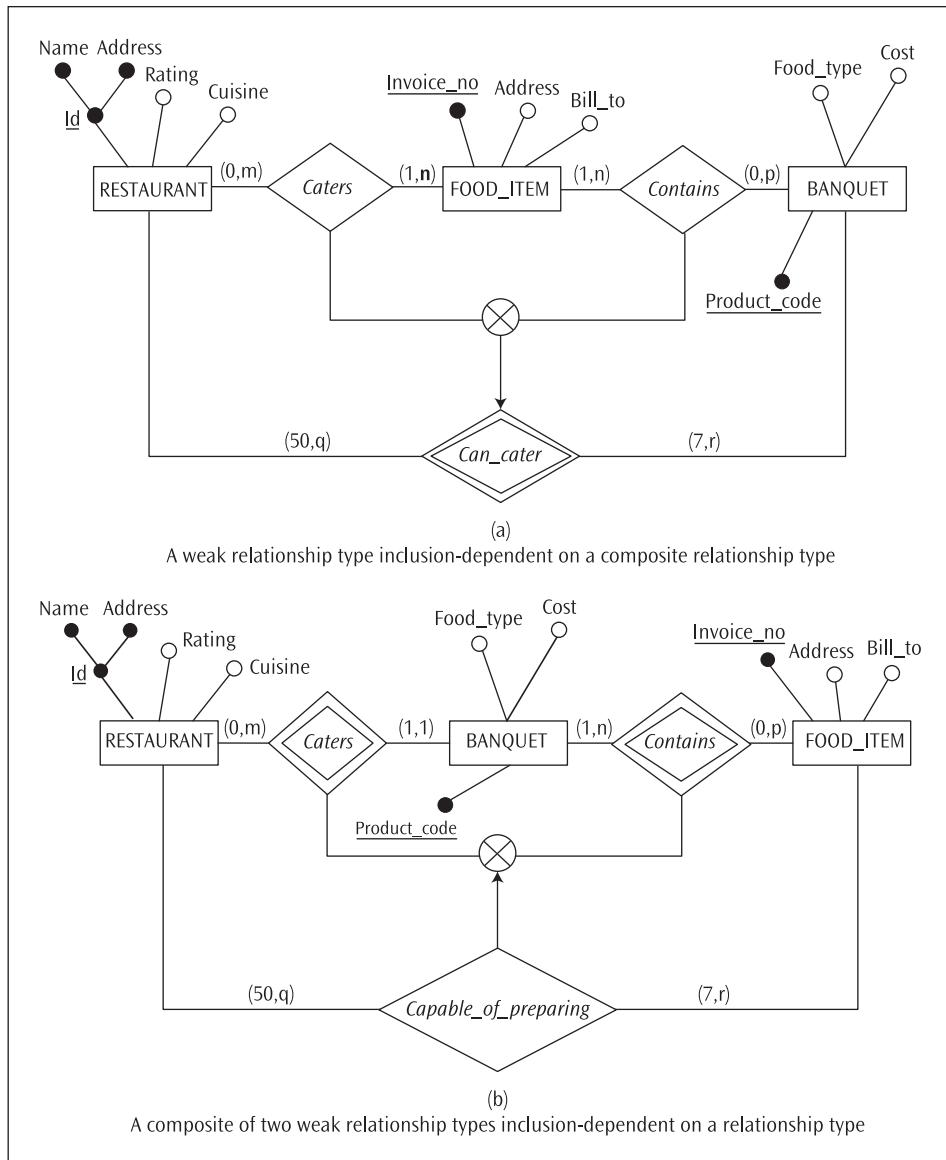
The ERD for this revised design is depicted in Figure 5.54b. Observe that **Frequency** is now the attribute of the binary relationship *Uses*. Also, the cluster entity type has been arbitrarily named INVENTORY. In essence, what appeared to be a possible ternary relationship at first glance is not the correct model to capture the complete scenario portended in the story line. Does the revised model (Figure 5.54b) continue to preserve the requirement as to which vendor supplies what product to which project and the frequency of supplies? It certainly does capture the frequency of use of a certain product by a certain vendor in the *Uses* relationship in the cluster entity type INVENTORY. Since there is only one vendor related to this {product, project} pair, as indicated by the (1, 1) structural constraint of INVENTORY in the *Supplies* relationship, the requirement is intact in the revised design.

A couple of other examples are available in Section 5.2.2, where the cluster entity type as an ER modeling construct is first introduced.

**FIGURE 5.54** Demonstration of a semantic trap**5.6.3.2 Vignette 6**

Let us revisit another example from earlier in the chapter to examine a concealed semantic trap. The second example in Section 5.4.1 involves a scenario of restaurants catering to banquets. For convenience, the scenario is reproduced here along with the ER model (from Figure 5.35), which is shown in Figure 5.55a:

*Restaurants cater to banquets. A banquet has a menu of food items and a restaurant caters various food items. Unless a restaurant is capable of preparing the set of food items contained in a banquet's menu, the restaurant cannot cater that particular banquet.*



**FIGURE 5.55** Inclusion dependency in composite relationship types: Two contrasting examples

Possible questions pertaining to this scenario include:

1. Which banquets does a restaurant cater set?
2. How many restaurants does a particular banquet use?
3. What is the menu of a particular banquet?
4. What items from a banquet's menu can a given restaurant not prepare?
5. What are the banquets whose menus no single restaurant is capable of preparing alone?

A quick scrutiny of the ERD in Figure 5.55a reveals that it is not possible to answer the first two questions using this ERD, even though the ERD is syntactically correct and can in fact be claimed as semantically correct, too. From the ERD, it is possible to list the banquets a particular restaurant is capable of catering, but that does not indicate whether the restaurant actually catered any or all of these banquets (Question 1). Likewise, it is possible to find out the number of restaurants a banquet can use given its menu. However, one cannot identify the number of restaurants that actually catered a given banquet (Question 2). Thus, the ERD is certainly not semantically complete. In other words, a semantic trap is present in the design. It is a trap simply because its presence is concealed and the design was completed without recognizing the presence of a semantic trap, a case in point for the importance of data model validation.

An alternative design for the same scenario is portrayed in Figure 5.55b. Structurally, the entity types BANQUET and FOOD\_ITEM are rearranged so that a direct relationship between RESTAURANT and BANQUET is enabled. This should facilitate answering Questions 1 and 2. This rearrangement triggers another structural change in order to preserve the other business rules prevailing over the scenario. A composite of *Caters* and *Contains* becomes inclusion-dependent on *Capable\_of\_preparing*. As a consequence, pursuant to ER modeling grammar rules, *Caters* and *Contains* become weak relationship types (double diamond), and the direction of the solid arrow in the ERD is accordingly reversed. Is this design superior to the one developed earlier (Figure 5.55a)? The answer to this question is context dependent. Given the scenario and the list of probable questions pertaining to the scenario, the alternative design just developed (Figure 5.55b) is indeed superior since it fully captures all the specified semantics and specifically eliminates the semantic trap identified in the original ERD.

The more important question is: What is the approach used to solve this problem? Unfortunately, semantic traps often do not fall in a pattern like connection traps, where a structural configuration of the ERD capable of generating a specific connection trap is known; and the approach(es) to resolve the particular connection trap is(are) known as well. The two examples presented here bear no similarity to each other regarding the identification or resolution of the semantic trap. Trial and error and experimentation based on possible hints discerned in the scenario and its business rules seem to be the only available approach. The lesson to be learned here is simply that validation of the conceptual design is a crucial step in the data modeling process. Inadvertent misinterpretation of the semantics embedded in a requirements specification is an unavoidable aspect of conceptual modeling. Being aware of the possibility of connection traps and other semantic traps sensitizes a designer to pay close attention to the requirements specification during the conceptual modeling process, and including a formal step of model validation in the conceptual modeling process enhances the quality of the conceptual modeling script (e.g., ER model).

## 5.7 COUGAR MEDICAL ASSOCIATES

The rest of this chapter shows how the advanced modeling techniques introduced thus far can be used to model complex real-world scenarios.

Cougar Medical Associates (CMA) is a clinic located in Kemah, Texas, owned by a group of medical corporations and individual physicians. Clinic personnel include physicians, surgeons, nurses, and support staff. All clinic personnel except the surgeons are on an annual

salary. Surgeons do not receive a salary but work for Cougar Medical Associates on a contract basis. It is possible for a physician to have an ownership position in the clinic.

Since surgeons perform surgery on patients as needed, it is required that a surgery schedule keep track of the operation theater where a surgeon performs a certain surgery type on a particular patient and when that surgery type is performed. Some patients need surgeries and others don't. Surgeons perform surgeries in the clinic; some do a lot, others just a few. Some surgery types are so rare that they may not yet have been performed in the clinic, but there are others that are performed numerous times. In addition, there is the need to keep track of nurses who can be assigned to a specific surgery type since all nurses cannot be assigned to assist in all types of surgeries. A nurse cannot be assigned to more than one surgery type. It is the policy of the clinic that all types of surgery have at least two nurses. The clinic maintains a list of surgery skills. A surgery type requires at least one but often many surgery skills. However, all surgery skills are not utilized in the clinic, whereas some surgery skills are utilized in numerous surgery types. Nurses possess one or more of these surgery skills. There are certain surgery skills for which no nurse in the clinic qualifies; at the same time, there are other surgery skills that have several qualified nurses. In order to assign a nurse to a surgery type, a nurse should possess one or more of the skills required for the surgery type.

Depending on the illness, some patients may stay in the clinic for a few days, but most require no hospitalization. In-patients are assigned a room and a bed. A nurse attends to several in-patients but must have at least five. No more than one nurse attends to an in-patient, but some in-patients may not have any nurse attending to them. If a nurse leaves the clinic, the association of all in-patients who were previously attended to by that nurse should be temporarily removed in order to allow these patients to be transferred to another nurse at a later time. Every physician serves as a primary care physician for at least seven patients; however, no more than 20 patients are allotted to a physician. If a physician leaves the clinic, that physician's patients should be temporarily assigned to the clinic's chief of staff. Clinic personnel can also become ill and be treated in the clinic. A patient is assigned one physician for primary care.

Physicians prescribe medications to patients; thus, it is necessary to capture which physician(s) prescribe(s) what medication(s) to which patient(s), along with dosage and frequency. In addition, no two physicians can prescribe the same medication to the same patient. If a physician leaves the clinic, all prescriptions prescribed by that physician should be removed because this information is also retained in the archives. A person affiliated with the clinic as a surgeon cannot be deleted as long as a record of all surgeries performed by the surgeon is retained.

A patient may be taking several medications, and a particular medication may be taken by several patients. However, in order for a patient to take a medicine, the medicine must be prescribed to that patient. As a medicine may interact with several other medicines, the severity of such interactions must be recorded in the system. Possible interactions include S = Severe interaction, M = Moderate interaction, L = Little interaction, and N = No interaction.

A patient may have several illnesses, and several patients may have the same illness. In order to qualify as a patient, a patient must have at least one illness. Also, a patient may have several allergies.

All clinic personnel have an employee number, name, gender (male or female), address, and telephone number. With the exception of surgeons, all clinic personnel also have a salary (which can range from \$25,000 to \$300,000), but salaries of some can be missing. Each person who works in the clinic can be identified by an employee number.

For each physician, his or her specialty is captured; whereas, for each surgeon, data pertaining to his or her specialty and contract are captured. Contract data for surgeons include the type of contract and the length of the contract (in years). Grade and years of experience represent the specific data requirements for nurses.

A surgery code is used to identify each type of surgery. In addition, the name, category, anatomical location, and special needs are captured for each surgery type. There are two surgery categories: those that require hospitalization (category = H) and those that can be performed on an outpatient basis (category = O). A surgery skill is identified by its description and a unique skill code. Data for patients consists of personal data and medical data. Personal data includes patient number (the unique identifier of a patient), name, gender (male or female), date of birth, address, and telephone number. Medical data includes the patient's blood type, cholesterol (consisting of HDL, LDL, and triglyceride), blood sugar, and the code and name of all the patient's allergies.

For both clinic personnel and patients, a Social Security number is collected. For each illness, a code and description are recorded. Additional data for each in-patient consists of a required date of admission along with the patient's location (nursing unit, room number, and bed number). Nursing units are numbered 1 through 7, rooms are located in either the Blue or Green wing, and the bed numbers in a room are labeled A or B. Medications are identified by their unique medication code, and medication data also includes name, quantity on hand, quantity on order, unit cost, and year-to-date usage. For medical corporations with ownership interest in the clinic, the corporation name and headquarters are obtained. Corporation name uniquely identifies a medical corporation. The percentage ownership of each clinic owner is also recorded.

The physicians who work in the clinic have recently embarked on a program to monitor the cholesterol level of its patients because cholesterol contributes to heart disease. Risk of heart disease is classified as N (None), L (Low), M (Moderate), and H (High). The ratio of a person's total cholesterol divided by HDL is used in the field of medicine as one indicator of heart risk. Total cholesterol is calculated as the sum of the HDL, LDL, and one-fifth of triglycerides. A total cholesterol/HDL ratio less than 4 suggests no risk of heart disease due to cholesterol; a ratio between 4 and 5 reflects a low risk; and a ratio greater than 5 is a moderate risk. The high-risk category is not coded as a function of cholesterol.

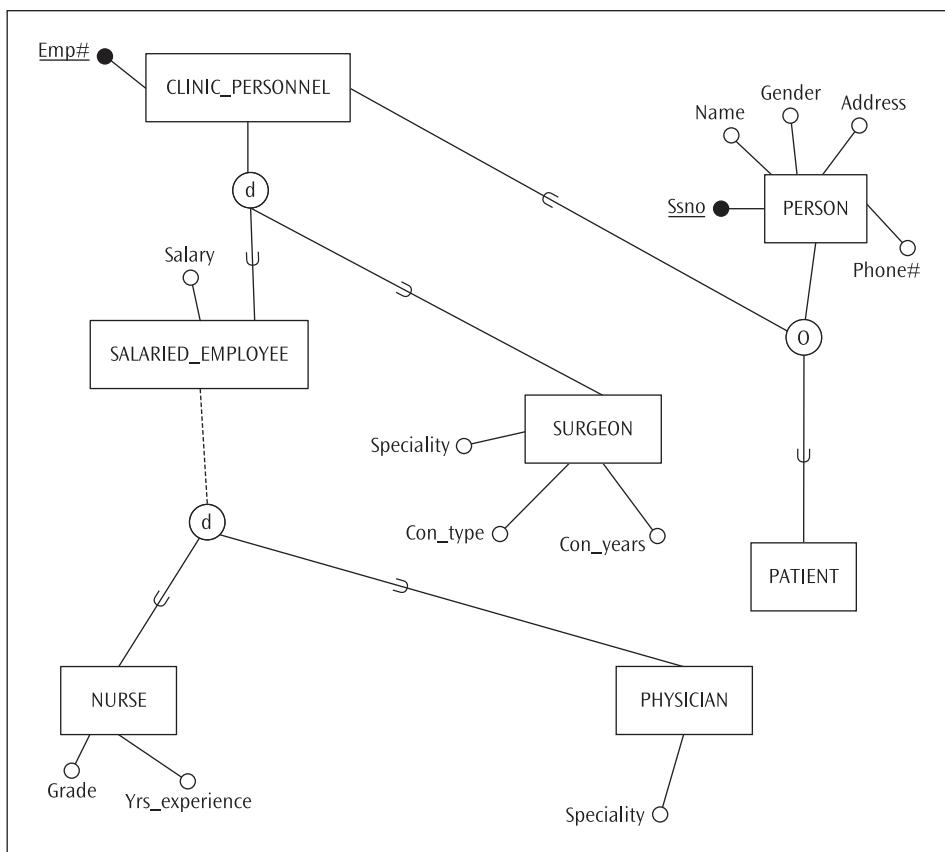
### 5.7.1 Conceptual Model for CMA: The Genesis

Recall that conceptual modeling is a heuristic as opposed to a scientific process. Therefore, the analyst must draw on intuition and expect to iteratively enhance the data model through several states of punctuated equilibrium before arriving at a final conceptual schema. Then, additional iterative enhancement of the conceptual schema may become necessary during logical and sometimes even physical design. With this in mind, let us begin modeling Cougar Medical Associates (CMA).

After several focused readings of the narrative, one may discern that nurses, surgeons, physicians, support staff, and patients emerge as the "human" entities in this scenario. One can also see that the first four in this list can be grouped under "clinic personnel." Having learned about Superclass/subclass (SC/se) relationships in Chapter 4, knowing from the CMA story that clinic personnel may also be patients from time to time, and noticing several attributes shared between clinic personnel and patients (Social Security number, name, gender, etc.), one would instinctively generalize these common attributes to form a base entity type to serve as the root of a

specialization/generalization hierarchy. Sometimes, at this stage, it may be useful simply to list all the attributes involved in this group of entities and examine the list for obvious entity types and relationship types. Such a process, in this case, provides an opportunity to create two entity types called PATIENT and CLINIC\_PERSONNEL and enlist them as subclasses in an overlapping generalization where an entity type labeled PERSON can serve as the superclass.

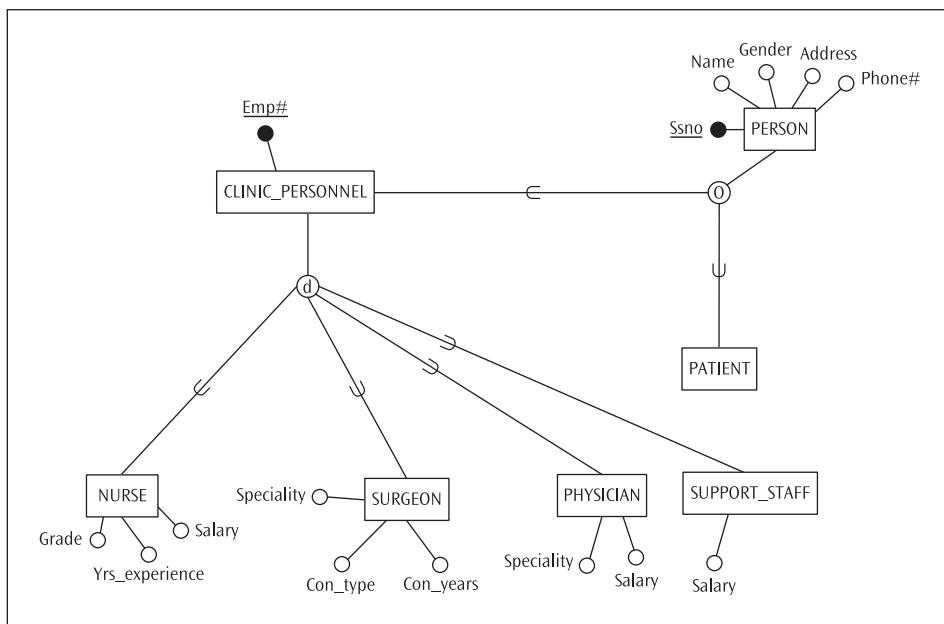
All the clinic personnel except the surgeons happen to be salaried employees of the clinic. So, we can group all the salaried employees as the entity type SALARIED\_EMPLOYEE, which can be a subclass (with SURGEON as the other subclass) in a disjoint specialization of CLINIC\_PERSONNEL. At first glance, SUPPORT\_STAFF appears to be a valid subclass, along with NURSE and PHYSICIAN, in a disjoint specialization of SALARIED\_EMPLOYEE. A closer inspection reveals that there are no specific attributes for SUPPORT\_STAFF beyond the only attribute of SALARIED\_EMPLOYEE, nor do any specific relationships exist for SUPPORT\_STAFF. Thus, SUPPORT\_STAFF can be absorbed in SALARIED\_EMPLOYEE by simply making the specialization a partial one (i.e., the completeness constraint is partial). Figure 5.56 shows this conceptualization as an ERD. Observe that this ERD is a multi-tier specialization hierarchy.



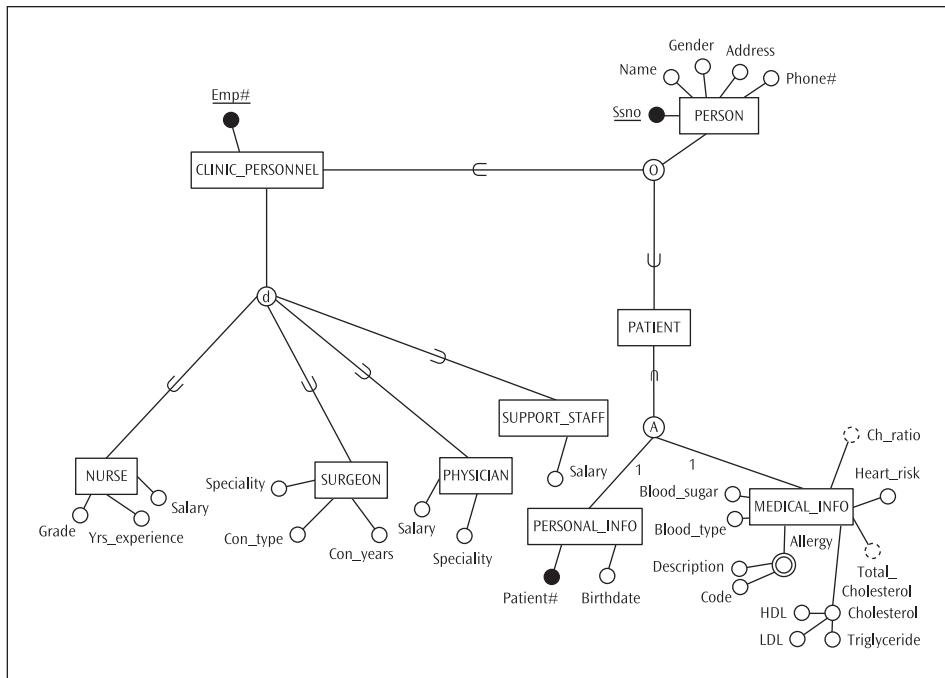
**FIGURE 5.56** Presentation Layer ERD for Cougar Medical Associates—Stage 1

A slightly simplified alternative design can model NURSE, PHYSICIAN, SURGEON, and SUPPORT\_STAFF as subclasses of a disjoint specialization of CLINIC\_PERSONNEL. Observe that SUPPORT\_STAFF is modeled as a subclass instead of subsumed in CLINIC\_PERSONNEL because **Salary** cannot be included as an attribute of CLINIC\_PERSONNEL—that is, surgeons are not salaried members of clinic personnel. Therefore, in this design, **Salary** is included as an attribute of NURSE, PHYSICIAN, and SUPPORT\_STAFF. The ERD depicting this design appears in Figure 5.57. This design has the same number of entity types as the one in Figure 5.56 and actually has one less tier in the specialization hierarchy. Therefore, this design is used for the ER model of CMA.

The list of attributes for the entity type PATIENT is relatively large and appears clearly demarcated as personal and medical data about a patient. Although all these attributes can certainly be recorded under PATIENT, it may be worthwhile to model PERSONAL\_INFO and MEDICAL\_INFO as separate entity types, especially if CMA expects to treat a large number of patients and the use of personal and medical information is clearly divided between administrative and medical personnel of the clinic. Since personal and medical information together make up patient information, the aggregation construct seems an appropriate way to depict this relationship. Figure 5.58 captures this aggregation construct.



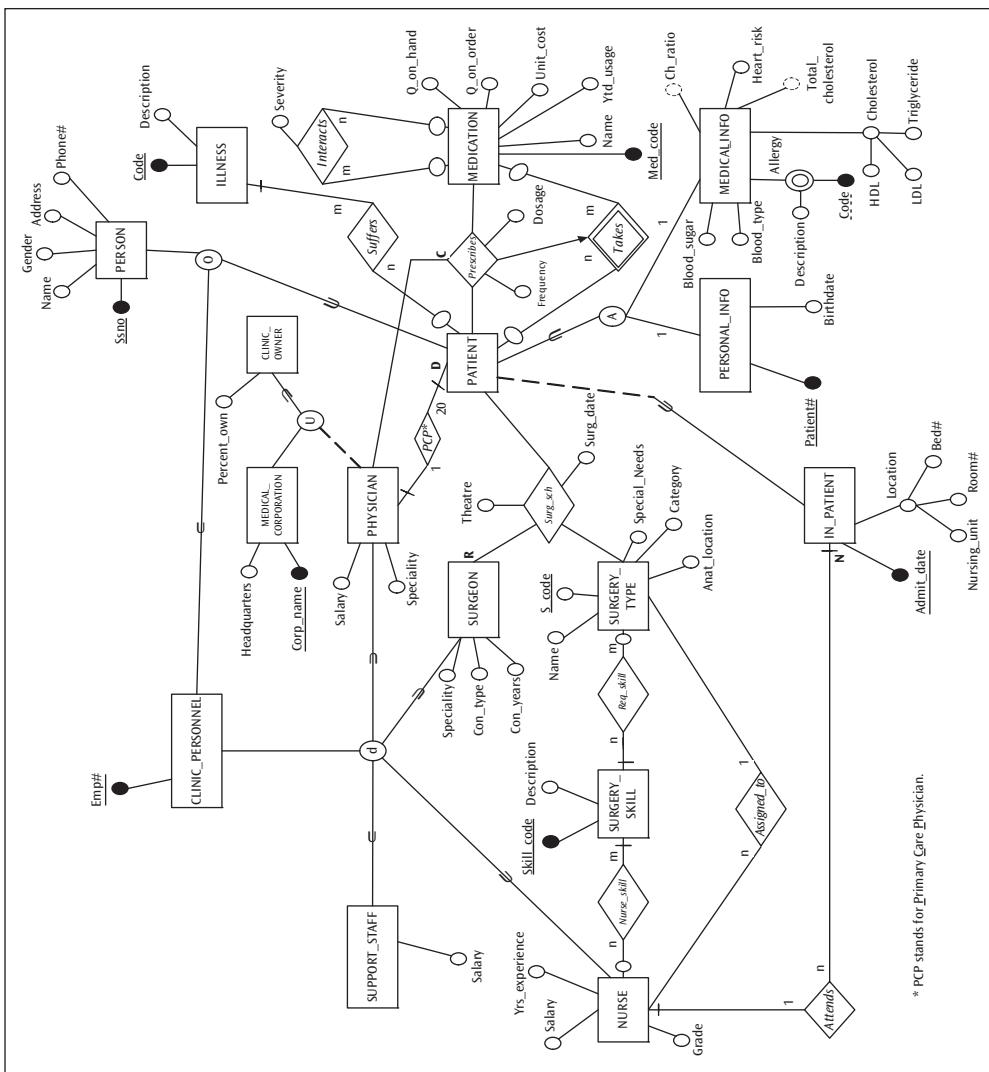
**FIGURE 5.57** Presentation Layer ERD for Cougar Medical Associates—Stage 1 (an alternative design)



**FIGURE 5.58** Presentation Layer ERD for Cougar Medical Associates—Stage 2

Next, a patient having several illnesses as well as certain illnesses afflicting many patients can be captured as an **m:n** relationship between a new entity type called **ILLNESS** and the entity type **PATIENT**. But then, observe that **Allergy** is modeled as a multi-valued attribute of a patient's **MEDICAL\_INFO** instead of as an entity type **ALLERGY**. Why? The requirements specified in the narrative simply state that "*a patient may have several allergies*" and nothing more. That **Allergy** can be a multi-valued attribute of **PATIENT** is quite clear from this statement. Attempting to specify an entity type **ALLERGY** amounts to speculating beyond the requirements specification and is incorrect in the context of the CMA scenario.

A relationship between **PATIENT** and **PHYSICIAN** also seems obvious from the story. **MEDICATION** appears to be another candidate for being modeled as an entity type. Once again, an **m:n** relationship between **PATIENT** and **MEDICATION** appears imminent. Which physician(s) prescribe(s) what medication(s) to which patient(s) appears to fit the mold of a ternary relationship type, with **Dosage** and **Frequency** as the attributes of the relationship type. Likewise, medicines interacting with other medicines convey an ideal recursive relationship type. Finally, the fact that a medicine must be prescribed in order for a patient to take that medicine can be handled by an inter-relationship constraint. These, as well as the relationships that follow, are incorporated in the Presentation Layer ERD shown in Figure 5.59.

**FIGURE 5.59** Presentation Layer ERD for Cougar Medical Associates—The genesis

Another ternary relationship looms in the story line “Surgeons perform surgery on patients as needed.” For this, the creation of an entity type called SURGERY\_TYPE is necessary. Incidences of surgery performed by a surgeon on a patient indicating the time and operation theater for these events are captured by this ternary relationship type. Note that the structural constraints of the ternary relationship types cannot be accurately reflected in the presentation layer because of the use of “look-across” notation in the grammar. It is important to note that nurses are assigned to surgery types (heart surgery, knee surgery, etc.), not to scheduled surgery events. Since nurses also attend to in-patients and since other attributes specific to in-patients do not apply to patients in general, specializing IN\_PATIENT as a subclass of PATIENT and relating it to NURSE makes sense.

At first glance, it is conceivable to think of a nurse’s surgery skills and skills required for a surgery type as multi-valued attribute(s). However, this will not be adequate to model the kinds of associations among nurses, surgery types, and surgery skills specified in the requirements. A closer perusal of the story will justify the creation of an entity type for SURGERY\_SKILL and relate it to SURGERY\_TYPE as well as to NURSE.

Finally, the fact that the clinic can have multiple owners and since the owners can be medical corporations and individual physicians (belonging to two different entity classes) leads to the modeling of CLINIC\_OWNER as a category arising from a subset of the union between PHYSICIAN and another new entity type, MEDICAL CORPORATION. Alternatively, when the system is being developed for just one clinic, the ownership information may be captured in the entity types PHYSICIAN and MEDICAL CORPORATION. MEDICAL CORPORATION, in this case, will be a stand-alone entity type in the ERD which, while technically acceptable, may not be considered an elegant design.

This gives an initial version of the ERD, which will serve as the input for further refinement and specification of other semantic integrity constraints that cannot be incorporated in the Presentation Layer ERD shown in Figure 5.59. Notice that the cardinality ratios and participation constraints culled from the narrative are shown in the ERD. The deletion rules embedded in the narrative are:

1. If a nurse leaves the clinic, temporarily remove the association of all patients previously attended to by that nurse in order to allow these patients to be transferred to another nurse sooner or later.
2. If a physician leaves the clinic, temporarily assign the physician’s patients to the clinic’s chief of staff.
3. If a physician leaves the clinic, all prescriptions prescribed by that physician should be removed because this information will be retained in the archives.
4. A person affiliated with the clinic as a surgeon cannot be deleted as long as a record of surgeries performed by that surgeon is retained.

These deletion rules have also been incorporated in the ERD as deletion constraints. Observe that the narrative hasn’t provided a comprehensive set of deletion rules for the scenario. As we know, at the time of database implementation, missing deletion constraints will, by default, translate to the Restrict (**R**) option, which can sometimes create

specification conflicts. It is the responsibility of the data modeler to make sure that such a default specification for the deletion constraint does not conflict with the participation constraint prevailing in the relationship type or the specified deletion constraint on an associated relationship type.

The business rules that cannot be expressed in the ERD are listed as semantic integrity constraints in Table 5.3.

#### Attribute-Level Business Rules

1. The gender of a person (i.e., a person affiliated with the clinic or a patient) is either male or female.
2. Nursing unit numbers range from 1 to 7.
3. Salaries of clinic personnel range from \$25,000 to \$300,000.
4. A surgery type can be either H — require hospitalization or O — be performed on an outpatient basis.
5. Rooms in the clinic are located in either the B = Blue wing or G = Green wing.
6. Bed numbers in a room are labeled either A or B.
7. Severity of medication interaction can be N = No interaction; L = Little interaction; M = Moderate interaction; and S = Severe interaction.
8. Heart risk can be N = No risk; L = Low risk; M = Moderate risk; and H = High risk.

#### Entity-Level Business Rules

1. A patient's heart risk is (a) "N" when the cholesterol ratio is below 4; (b) "L" when the cholesterol ratio is between 4 and 5; and (c) "M" when the cholesterol ratio is greater than 5.

#### Miscellaneous Business Rules

1. A physician serves as a primary care physician for at least seven but no more than 20 patients.
2. Each nurse is assigned a minimum of five patients.
3. All types of surgery require at least two nurses.

**TABLE 5.3** Semantic integrity constraints for the Presentation Layer ER model for Cougar Medical Associates

### **5.7.2 Conceptual Model for CMA: The Next Generation**

A reappraisal of the requirements specification reveals that the following business rules have not been incorporated into the initial Presentation Layer ERD shown in Figure 5.59: (1) *No two physicians can prescribe the same medication to the same patient*, and (2) *In order to assign a nurse to a surgery type, a nurse should possess one or more of the skills required for the surgery type*.

To uphold the principle of information preservation, either these rules should be reflected in the Presentation Layer ERD or they should become part of the semantic integrity constraints that accompany the Presentation Layer ERD. Let us explore if and how these rules can be captured in the ERD.

With regards to business rule (1), a similar condition was discussed earlier in this chapter (see Section 5.1.2). The initial “gut reaction” to this business rule is to make the maximum cardinality of PHYSICIAN in the relationship type *Prescribes* a 1. This will certainly accomplish the goal of no two physicians prescribing the same medication to the same patient; unfortunately, it will also prevent a physician from prescribing more than one medication to the same patient and also the same medication to another patient. That is, a

maximum cardinality of 1 for the PHYSICIAN in *Prescribes* will limit the relationship of a physician to one {patient, medication} pair. The chances are that this side effect is unintended and unacceptable. If so, then this is a case of a semantic trap like the one discussed in vignette 5 (Section 5.6.3.1). The solution lies in restricting the relationship of a {patient, medication} pair to just one physician while permitting a physician to relate to multiple {patient, medication} pairs. This cannot be done by manipulating the structural constraints of a ternary relationship type among PATIENT, PHYSICIAN, and MEDICATION. This is accomplished by restructuring the ternary relationship type to two binary relationships:

- Rendering an m:n relationship type connecting PATIENT and MEDICATION (*Prescribes*) as a cluster entity type (PRESCRIPTION)
- Specifying a 1:n relationship type (*Writes*) between PHYSICIAN and the cluster entity type, PRESCRIPTION

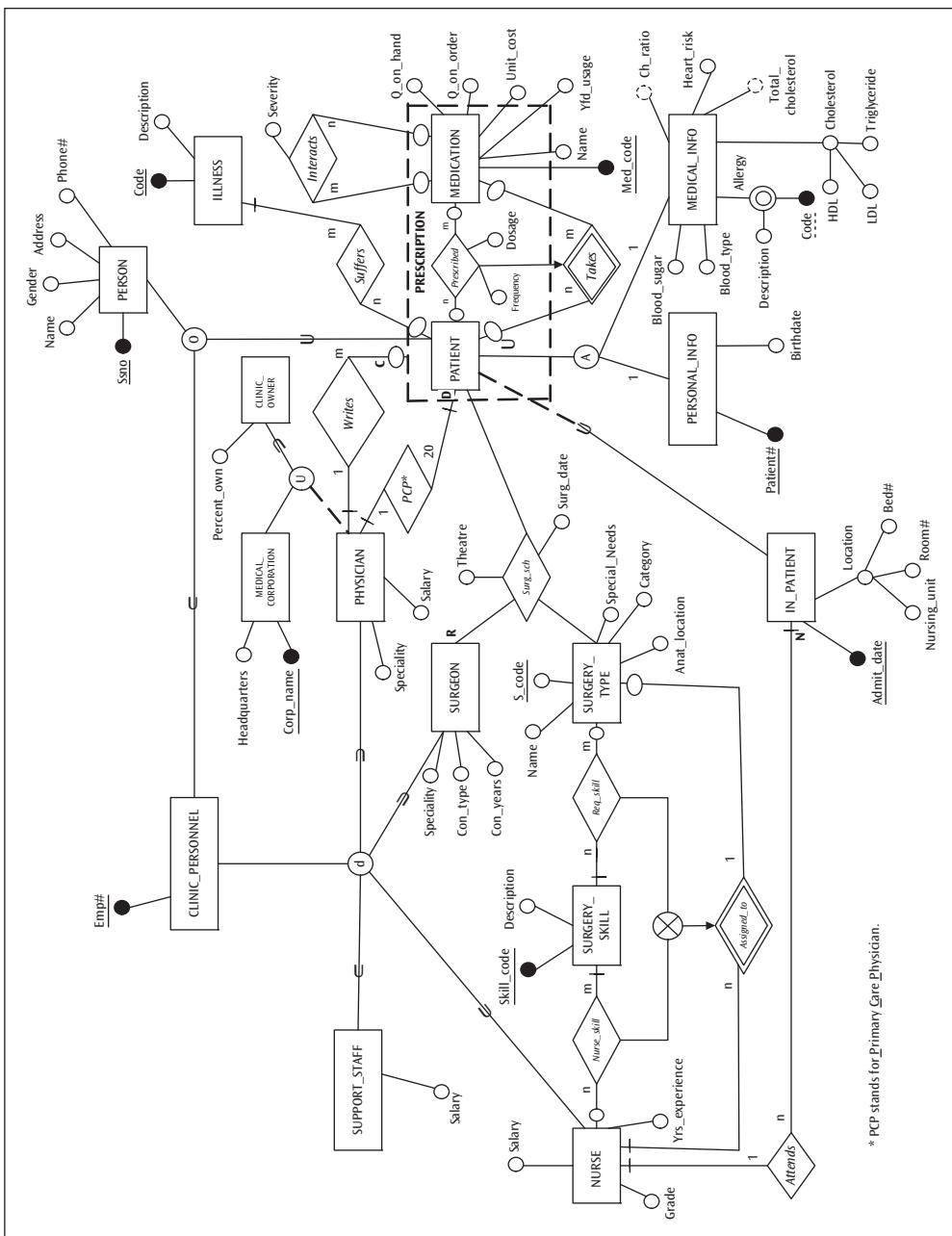
This solution implements the stated business rule without causing any unexpected side effects. This modification to the design is shown in Figure 5.60. An interesting question that may arise at this point is: Is there a need for a cluster entity type? How about specifying a base entity type called PRESCRIPTION, which conventional wisdom would suggest? Strict adherence to the story line of the CMA scenario reveals that a base entity type called PRESCRIPTION is not feasible because, given the information available in the requirements specification, a unique identifier for such an entity type cannot be culled out. Furthermore, a closer observation of the design reveals that the cluster entity type PRESCRIPTION will get decomposed to a gerund entity type PRESCRIPTION at the design-specific level.

Business rule (2) can be incorporated into the ERD by transforming the *Assigned\_to* relationship type to a weak relationship type with an inclusion dependency on the composite of *Skill\_set* and *Req\_skill*. This particular design was discussed earlier in this chapter (see Section 5.4.1).

At this point the ER model consists of the ERD in Figure 5.60 and the list of semantic integrity constraints in Table 5.3.

### 5.7.3 The Design-Specific ER Model for CMA: The Final Frontier

As we learned in Chapter 3, the user-oriented Presentation Layer ERD is at a high level of abstraction. It does not capture attribute characteristics and includes constructs that cannot be directly mapped to a logical schema. While it serves as an effective vehicle for the analyst to interact with the user community, it is not adequately equipped for migration to the next tier of data modeling: the logical tier. The Design-Specific ER model is aimed at preparing the Presentation Layer ER model for direct mapping to the logical schema. The first step in the transformation of the Presentation Layer ER model to the design-specific tier is to capture the domain specifications for the attributes (data type, size, and other domain constraints, if any, for the attributes). This information, collected from the users at this time, is presented in Table 5.4.

**FIGURE 5.60** Presentation Layer ERD for Cougar Medical Associates—Final

Entity/Relationship Type Name	Attribute Name	Data Type	Size	Domain Constraint
PERSON	Ssno	Alphanumeric	9	
PERSON	Name	Alphabetic	30	
PERSON	Gender	Alphabetic	1	M or F
PERSON	Address	Alphanumeric	50	
PERSON	Phone#	Alphanumeric	10	
CLINIC_PERSONNEL	Emp#	Alphanumeric	6	
PHYSICIAN	Speciality	Alphabetic	20	
PHYSICIAN	Salary	Numeric	6	Salaries range from 25000 to 350000
SURGEON	Speciality	Alphabetic	20	
SURGEON	Con_type	Alphabetic	20	
SURGEON	Con_years	Numeric	(1.1)*	
NURSE	Grade	Alphabetic	20	
NURSE	Yrs_experience	Numeric	2	
NURSE	Salary	Numeric	6	Salaries range from 25000 to 350000
SUPPORT_STAFF	Salary	Numeric	6	Salaries range from 25000 to 350000
SURGERY_TYPE	S_code	Alphanumeric	3	
SURGERY_TYPE	Name	Alphabetic	30	
SURGERY_TYPE	Category	Alphabetic	1	O = Outpatient; H = Hospitalization
SURGERY_TYPE	Anat_location	Alphabetic	20	
SURGERY_TYPE	Special_needs	Alphanumeric	5000	
SURGERY_SKILL	Skill_code	Alphabetic	3	
SURGERY_SKILL	Description	Alphabetic	20	
PERSONAL_INFO	Patient#	Alphanumeric	10	
PERSONAL_INFO	Birthdate	Date	8	
MEDICAL_INFO	Blood_sugar	Numeric	3	
MEDICAL_INFO	Blood_type	Alphanumeric	2	
MEDICAL_INFO	Allergy_code	Alphabetic	3	
MEDICAL_INFO	Allergy_description	Alphabetic	20	
MEDICAL_INFO	HDL	Numeric	3	
MEDICAL_INFO	LDL	Numeric	3	
MEDICAL_INFO	Triglyceride	Numeric	3	
MEDICAL_INFO	Total_cholesterol**	Numeric	3	Computed as HDL + LDL + (0.20*Triglyceride)

\*(n<sub>1</sub>,n<sub>2</sub>) is used to indicate n<sub>1</sub> places to the left of the decimal point and n<sub>2</sub> places to the right of the decimal point

\*\*Derived attribute (not intended to be stored in the database)

**TABLE 5.4** Domain specifications for the attributes of the ER model for Cougar Medical Associates

Entity/Relationship Type Name	Attribute Name	Data Type	Size	Domain Constraint
MEDICAL_INFO	Ch_ratio**	Numeric	(1.2)*	Computed as Total_cholesterol/HDL
MEDICAL_INFO	Heart_risk	Alphabetic	1	N = No Risk; L = Low Risk; M = Moderate Risk; H = High Risk
IN_PATIENT	Admit_date	Date	8	
IN_PATIENT	Nursing_unit	Numeric	1	Integer values from 1 to 7
IN_PATIENT	Room#	Alphanumeric	3	ANN, where A = B or G (for the Blue or Green wing); NN is a two-digit room number
IN_PATIENT	Bed#	Alphabetic	1	A or B
MEDICATION	Med_code	Alphabetic	3	
MEDICATION	Name	Alphanumeric	30	
MEDICATION	Q_on_hand	Numeric	4	
MEDICATION	Q_on_order	Numeric	4	
MEDICATION	Unit_cost	Alphanumeric	30	
ILLNESS	Code	Alphabetic	3	
ILLNESS	Description	Alphabetic	20	
MEDICAL_CORPORATION	Corp_name	Alphabetic	30	
MEDICAL_CORPORATION	Headquarters	Alphabetic	15	
CLINIC_OWNER	Percent_own	Numeric	(2.1)*	
Interacts	Severity	Alphabetic	1	N = No; L = Little; M = Moderate; S = Severe
Prescribes	Frequency	Numeric	2	
Prescribes	Dosage	Numeric	4	
Schedule	Theatre	Alphabetic	15	
Schedule	Surg_date	Date	8	

\*(n<sub>1</sub>.n<sub>2</sub>) is used to indicate n<sub>1</sub> places to the left of the decimal point and n<sub>2</sub> places to the right of the decimal point

\*\*Derived attribute (not intended to be stored in the database)

#### Entity Level Domain Constraints

1. A patient's heart risk is (a) N when the cholesterol ratio is below 4; (b) L when the cholesterol ratio is between 4 and 5; and (c) M when the cholesterol ratio is greater than 5.

**TABLE 5.4** Domain specifications for the attributes of the ER model for Cougar Medical Associates (continued)

Three major tasks in this transformation process for the ERD are:

1. Map the structural constraints of relationship types from the “look across” (Chen’s) notation to the “look near” [(min, max)] notation.
2. Decompose m:n relationship types to gerund entity types.
3. Transform multi-valued attributes to single-valued attributes.

Since each of these three tasks is discussed in detail in Chapters 3 and 4, suffice it to say here that there is only one multi-valued attribute (**Allergy**) requiring evaluation; and the m:n binary relationships requiring decomposition to gerund entity types are: *Nurse\_skill*, *Req\_skill*, *Prescribes*, *Takes*, *Interacts*, and *Suffers*. In addition, the ternary relationship type *Surg\_sch* requires decomposition in preparation for mapping to the logical tier. How to decompose a ternary relationship was illustrated in Section 5.5.1. Following that procedure, the decomposition of *Surg\_sch* results in a gerund entity type *SURG\_SCH* with three identifying parents, *SURGEON*, *SURGERY\_TYPE*, and *PATIENT*, as shown in Figure 5.61.

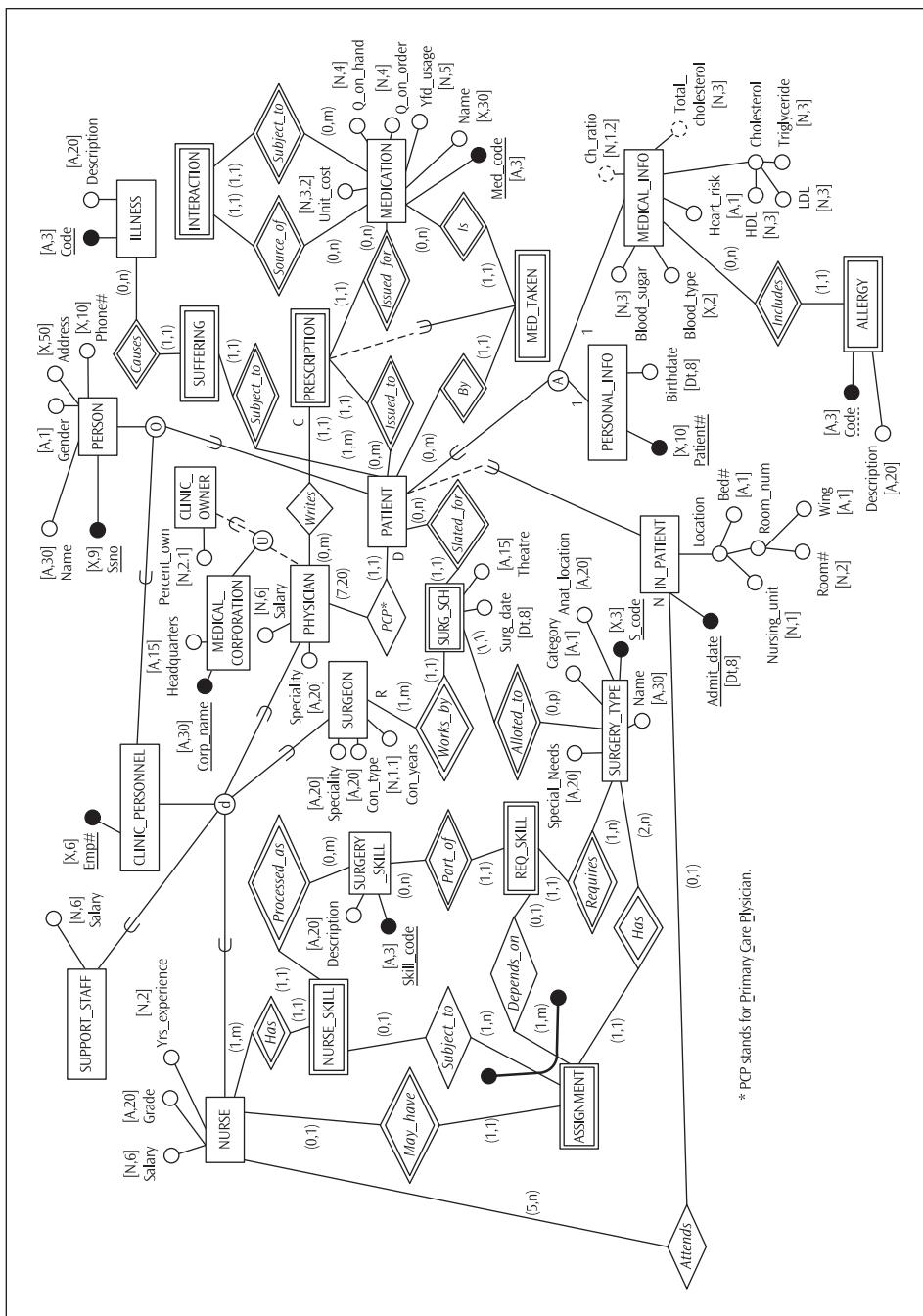
Next, the weak relationship type *Takes*, inclusion-dependent on the regular relationship type *Prescribes*, needs attention. Note that both *Prescribes* and *Takes*, being relationship types depicting the m:n cardinality ratio, will first be decomposed to the gerund entity types *PRESCRIPTION* and *MED\_TAKEN*, respectively. Then, following the procedure prescribed in Section 5.5.4 for transforming a weak relationship type to the design-specific state, a partial specialization involving *PRESCRIPTION* and *MED\_TAKEN* is modeled as superclass and subclass respectively (see Figure 5.61).

The situation with the weak relationship type *Assigned\_to* in the Presentation Layer (Figure 5.60) is somewhat different because it has inclusion dependency on a composite of the two relationship types *Nurse\_skill* and *Req\_skill*. Remember, the story line states that a nurse must have some of the skills required for the surgery type in order to get assigned to that surgery type. Our first task is to recognize that *Nurse\_skill* and *Req\_skill* first get translated to the gerund entity types *NURSE\_SKILL* and *REQ\_SKILL*, respectively, because these two are relationship types bearing an m:n cardinality ratio. The task here is two-fold: (1) rendering the inclusion dependency  $\text{Assigned\_to} \subseteq (\text{Nurse\_skill} \otimes \text{Req\_skill})$  implementable, and (2) keeping the structural constraints of the relationship type *Assigned\_to* intact.

As shown in Figure 5.61, using a cross-referencing design for mapping *Assigned\_to*, the weak entity type *ASSIGNMENT* is identification-dependent on *NURSE*. This meets the specification in item (2). The relationship type *Subject\_to* between *ASSIGNMENT* and *NURSE\_SKILL* and the relationship type *Depends\_on* between *ASSIGNMENT* and *REQ\_SKILL* followed by an inclusive arc across these two relationship types partially captures the specification in item (1). In addition, a constraint specifying that the {*Nurse*, *Surgery\_type*} projection from *ASSIGNMENT* should be a subset of the {*Nurse*, *Surgery\_type*} set resulting from the intersection of *NURSE\_SKILL* and *REQ\_SKILL* must be included in the list of semantic integrity constraints.<sup>11</sup>

The final version of the Design-Specific ERD ready for conversion to a logical schema appears in Figure 5.61. Note that in addition to all the necessary decompositions, all miscellaneous business rules from Table 5.3 and attribute characteristics from Table 5.4 have been incorporated in this ERD. Table 5.5 records the remaining semantic integrity constraints to be carried forward to the logical modeling phase, thus rendering the Design-Specific ER model fully information-preserving.

<sup>11</sup>Projection is a vertical subset of attributes from an entity set. Intersection of two entity sets results in a third entity set containing entities common to the first two. A precise definition and detailed explanation of “projection” and “intersection” are included in Chapters 6 and 11.

**FIGURE 5.61** The final form of the Design-Specific ERD for Cougar Medical Associates

> Constraint	Gender	IN ('M', 'F')
> Constraint	Salary	IN (25000 through 300000)
> Constraint	Category	IN ('O', 'H')
> Constraint	Heart_risk	IN ('N', 'L', 'M', 'H')
> Constraint	Nursing_unit	BETWEEN (1 and 7)
> Constraint	Wing	IN ('B', 'G')
> Constraint	Bed#	IN ('A', 'B')
> Constraint	Severity	IN ('N', 'L', 'M', 'S')
> Constraint	Ch_ratio	(<4 and Heart_risk = 'N') OR ((BETWEEN 4 and 5) and Heart_risk = 'L') OR (>5 and Heart_risk = 'M') OR Heart_risk = 'H'

Constraints Carried Forward to Logical Design

1. (Nurse, Surgery\_type) projection from ASSIGNMENT should be a subset of the (Nurse, Surgery\_type) set resulting from the intersection of NURSE\_SKILL and REQ\_SKILL

**TABLE 5.5** Semantic integrity constraints for the final Design-Specific ER model for Cougar Medical Associates

## Chapter Summary

---

This chapter focuses on modeling relationships beyond binary relationships, characteristics of weak relationship types, the decomposition of complex relationships, and model validation.

How to handle ternary (degree 3), quaternary (degree 4), and quintary (degree 5) relationships is shown through a series of application scenarios and vignettes. The cluster entity type is a way to represent entity types that naturally emerge from a higher-order relationship type and/or a group of entity types and associations among them.

273

The relationship construct known as the weak relationship type was originally defined by Dey, Storey, and Barron (1999). A weak relationship type occurs when two relationship types are linked by either an event-precedent sequence or a condition-precedent sequence. An event-precedent weak relationship type occurs when an event associated with the occurrence of one relationship type must precede an occurrence of the weak relationship type. A condition-precedent weak relationship type occurs when a condition that triggers the occurrence of one relationship type must precede the occurrence of the weak relationship type.

The decomposition of ternary and higher-order relationship types is very similar to the decomposition of binary relationship types with an m:n cardinality ratio. It involves converting the relationship type to a gerund entity type such that the resulting transformation contains nothing but a set of binary relationships with cardinality ratios of 1:m. Binary relationship types with multi-valued attributes and design alternatives in their decomposition are also introduced. A brief discussion about the alteration of weak relationship types in preparation for logical model mapping follows.

It is important that a conceptual model be an accurate representation of the “universe of interest.” Such an objective can only be achieved through the careful evaluation of how well the developed conceptual model addresses the explicit and/or implicit questions associated with the requirements specification. This is accomplished through a formal design validation step in the conceptual modeling process.

The Cougar Medical Associates case represents a real-world scenario and provides an opportunity to employ several complex relationship constructs described in the chapter. This case is developed all the way to a Design-Specific ER model ready for transformation to the logical tier.

## Exercises

---

1. What is a cluster entity type?
2. What is a weak relationship type? Contrast a condition-precedent weak relationship type with an event-precedent weak relationship type.
3. What is a composite relationship type?
4. What is required in order to decompose a ternary and higher-order relationship type in preparation for mapping to a logical schema? What is the cardinality ratio of each of the resulting set of binary relationships?
5. What is required to decompose a cluster entity type in preparation for mapping to a logical schema?

6. Figures 5.4, 5.7, and 5.39 show the decomposition of a ternary relationship. Explain why the entity types SCHEDULE and PRESCRIPTION in Figures 5.4 and 5.7 are shown as base entity types while the entity type USE in Figure 5.39 is shown as a weak entity type.
7. Consider the ERD in Figure 5.5. State meaningful semantics for additional binary relationships among the entity types in the diagram and update the ERD accordingly with full specification of the structural constraints of the relationship types you have proposed. Can the three binary relationships among the three entity types present collectively capture the semantics conveyed by the ternary relationship type? State the condition under which the answer is “yes” and the condition under which the answer is “no.”
8. The Design-Specific ERD in Figure 5.13a requires further decomposition before it can be mapped to the logical tier. Specify the final form of the Design-Specific ERD that will render this design ready for mapping to a logical schema.
9. Decompose the Design-Specific ERDs in Figures 5.17 and 5.18 to the final Design-Specific stage and explain the differences.
10. Convert the ERD in Figure 5.22 to the final form of Design-Specific ERD.
11. Transform the Design-Specific ERD in Figure 5.28 to a final-form Design-Specific ERD.
12. The weak relationship type shown in Figure 5.32 requires further decomposition preparatory to mapping to a logical schema. Develop the final form of the Design-Specific ERD.
13. Business Process, Inc. (BPI), a consulting company offering business process reengineering and application system development expertise, wants to develop a prototype of a simple University Registration System (UNIVREG) to handle student/faculty information, course/section schedules, and co-op and lab information. Many small universities are in need of such a system. BPI believes that the profit potential from economies of scale alone in custom-fitting such an IS application to small universities that primarily offer a small number of programs is an attractive business opportunity. You have recently been hired by BPI and assigned to develop the conceptual design for this application. Here are the data specifications:

A university has several departments and these departments employ faculty (professors) for purposes of teaching, research, and administration. A department may have many professors but has to employ at least five. A professor, however, belongs to only one department at any time. In addition to teaching, some of the professors may work as department heads. Each department has a department head, but no more than one. Every department should continue to exist as long as it has at least one professor associated with it or it offers at least one course. If a faculty member serving as a department head leaves the university, some other professor (often, the most senior faculty member of the department) assumes the role by default.

The departments may offer several courses as part of their academic missions. However, any particular course is offered by only one department. Not all courses are offered all the time, but every course is offered sometime. When a course is offered, multiple sections of that course may be offered during a specific quarter of the year. If a particular course is no longer offered, all offerings (sections) of that course should be deleted unless there are students enrolled in the course sections. If, however, a student leaves, that student's

enrollment in all associated course sections should be removed. A course may be a prerequisite for several other courses, but a course may have no more than one prerequisite. A course cannot be removed from the database as long as it is a prerequisite for other course(s); however, if it only has prerequisites, then its deletion should be accompanied by the removal of its links to all of its prerequisites.

Some professors in the university also write textbooks. Sometimes, more than one professor may be co-authoring a book, but all textbooks used by the university need not have one of its professors as an author. Some professors are also authors of multiple textbooks. There is no plan for this system to record the authorship of professors who are not working at this university. When a professor leaves the university, the university no longer keeps track of books written by that professor. Likewise, if a textbook is no longer in use, the authorship of the textbook is not preserved. The system also needs to record which professor uses what book in which course. Removal of a textbook from the database is prohibited if it is used by a professor in a course. However, if a course is removed from the catalogue, its link to a professor using a particular textbook is also removed. Likewise, if a faculty member leaves the university, the link to the textbook used by the professor in a specific course is deleted. All professors teach, and a professor may teach several course sections. A course section, however, is taught by just one professor and must have some professor assigned to teach it. Some of the course sections may have multiple lab sessions in a quarter. Each lab session caters to only one course section—that is, there are no joint lab sessions. If a course section has an associated lab session, cancellation of the course section is not permitted.

Students enroll in course sections. In fact, to remain a student, one has to take at least one course (section), but university rules forbid a student from taking more than six courses (sections) in a quarter. Each section has to have at least 10 students enrolled; otherwise, it will be cancelled. If a student has registered for a section, the section should continue to exist. Also, when a professor is assigned to teach a section, deletion of the professor's record is prohibited. The university admits mostly graduate and undergraduate students, but a few non-matriculating students are also admitted. The undergraduate students may, as part of their academic programs, enroll for professional practice (co-op) sessions with companies. Several students may be enrolled in the same co-op session, and a co-op session has at least one undergraduate student enrolled in it. An undergraduate student can co-op more than once. When a graduate student leaves the university for whatever reason, the associated graduate student record is deleted from the database; if an undergraduate student leaves, the associated student record is not deleted so that the co-op status of the student can be properly verified. If, having verified the co-op status, the decision is made to drop the undergraduate student information from the system, all co-op enrollments for that student should also be erased. Cancellation of a co-op session is prohibited if there is/are student(s) enrolled in it. As part of their academic experience some of the graduate students are assigned to conduct one or more lab sessions. A lab session can be conducted by at most one graduate student, but some lab sessions are not assigned to any graduate student. When a graduate student graduates, the lab sessions assigned to him/her cannot be cancelled; instead, the capability should exist to indicate that, for the present, the lab session is not handled by a graduate student.

Students borrow books from a single (main) library on the campus. A student may borrow a lot of books, and a book may be borrowed by several students, when available. Book-returns by students are also recorded in this system. The return pattern is the same as that of borrowing. Deletion of a student record is not allowed if he or she has any borrowed books outstanding. If a book is removed from the library catalogue, all borrow and return links for that book are removed. When a student leaves, the book-return links for that student are also discarded. It is important to note that a book should have been borrowed in order for it to be returned.

The registration system should capture student information like the name [o], address, and a unique student ID for each student. *Please note that optional attributes are marked by an [o]; so, the rest of the attributes are mandatory.* In addition, the status of the student should be recorded. For undergraduate students, data on the student's concentrations should be available; all undergraduate students have multiple (at least two) concentrations. Thesis option [o] and the undergraduate major of each graduate student should be captured by this system. A co-op session is identified by year and quarter, and each co-op session has a session manager [o]. A particular student during a particular co-op session works in a company, and the database should record the name of the company and co-op assessment [o] for the student for each co-op session. Every professor has a name, employee ID, office [o], and phone [o]. Both professor name and employee ID have unique values. Data gathered about a department are: department name [o], department code [o], location, and phone# [o]. For a department, the name and code are both unique. The courses offered have data on course name, credit hours, college [o], and course#. The course# is used to distinguish between courses. Each course may have multiple course sections, with data including the classroom [o], class time, class size [o], section number, quarter, and year. There is no unique identifier for course section because the course section has existence dependency on course; section number, quarter, and year together in conjunction with course# can uniquely identify course sections. The grade a student makes in a particular course should be available through the system. The lab sessions have information about the topic [o], time, lab location, and the lab session number for a given course section. Attributes of textbooks include ISBN, the unique identifier, year [o], title, and publisher [o]. The library books, on the other hand, are identified by a call#. The ISBN# and copy# together also identify a copy of the book. The name of the book [o] and author [o] are also recorded.

- a. Develop a Presentation Layer ER model for the UNIVREG. The ERD should be fully specified, with the unique identifiers, other attributes for each entity type, and the relationship types that exist among the various entity types. All business rules that can be captured in the ERD must be present in the ERD. Any business rule that cannot be captured in the ERD should be specified as part of a list of semantic integrity constraints.
- b. Incorporate the following business rule into the Presentation Layer ER model: *No two courses can be taught by the same professor using the same textbook.*
- c. Transform the Presentation Layer ER model developed in Exercise 13a to a Design-Specific ER model. Note that attribute characteristics are not provided and thus need not appear in the ERD.

# PART

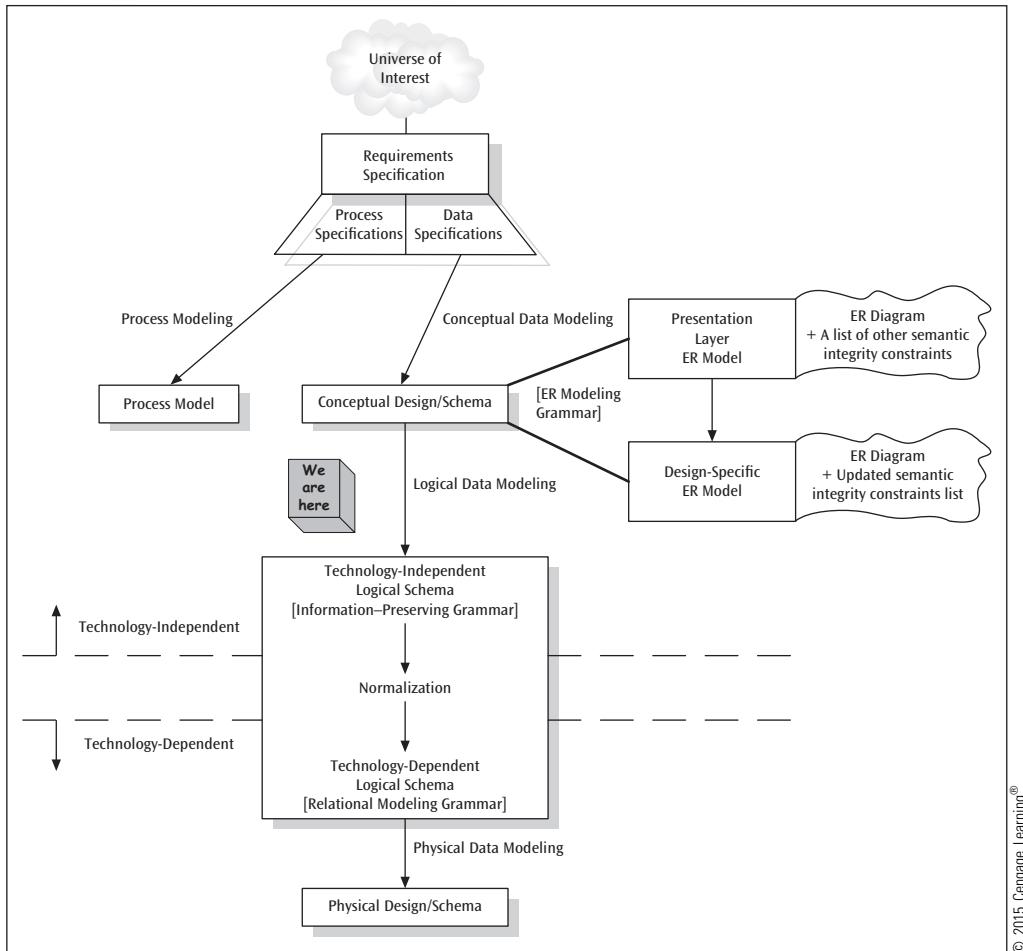


## LOGICAL DATA MODELING

### INTRODUCTION

At the completion of the conceptual modeling phase, the systems analyst/application developer usually has a reasonably clear understanding of the data requirements for the application system at a high level of abstraction. It is important to note that a conceptual data model is technology independent. During conceptual modeling, the analysis and design activities are not constrained by the technology that will be used for implementation. A conceptual schema may contain constructs not directly compatible with the technology intended for implementation. In addition, some of the design may require refinement to eliminate data redundancy. The next step after conceptual modeling, then, is to transform the conceptual schema into a logical schema that is more compatible with the implementation technology of choice.

Part II introduces logical data modeling, which serves as the transition between the conceptual schema and the physical design. Logical data modeling begins with the creation of a technology-independent logical schema, proceeds through the normalization process,<sup>1</sup> and concludes with a technology-dependent logical schema expressed using the relational modeling grammar. Figure II.1 points to where we are now in the database development process.



**FIGURE II.1** Roadmap for data modeling and database design

<sup>1</sup>Normalization is the subject of Chapters 7, 8, and 9 in Part III.

Part II discusses techniques and procedures for mapping a conceptual schema to a logical schema in the relational data model. Because database management systems are based on logical data models, and because the relational data model forms the basis for the data modeling and database design discussed in this textbook, Chapter 6 shows how logical data modeling is performed in the context of the entity-relationship (ER) and enhanced entity-relationship (EER) conceptual data models. As part of the discussion, we will also see that the process of mapping a conceptual schema to its logical counterpart can be either information reducing or information preserving, depending upon the approach used.<sup>2</sup>

---

<sup>2</sup>Two basic logical data structures—inverted tree and network—underlie conventional database design; and three basic data model architectures—relational, hierarchical, and CODASYL (Conference on Data Systems Languages)—employ one or both of the logical data structures. A brief discussion of the inverted tree and network data structures and an overview of how the hierarchical and CODASYL data model architectures express the inverted tree and network data structures respectively are presented in Appendix A. Object-oriented concepts have drawn considerable attention among researchers and practitioners since the late 1980s and have significantly influenced efforts to incorporate in the DBMS the ability to process complex data types beyond just storage and retrieval. Appendix B briefly introduces the reader to object-oriented concepts exclusively from a database—or, to be more precise, from a data modeling perspective.

# CHAPTER 6

# THE RELATIONAL DATA MODEL

This chapter introduces the relational data model and the process of mapping a conceptual schema that is technology independent to a logical schema (in this case, a relational data model) that provides the transition to a technology-dependent database design. Recall that in Part I of this book, the ER modeling grammar was used to develop a conceptual data model in the form of a Design-Specific ER model. It is this model that serves as the input for the logical data modeling activity.

The chapter flows as follows. Section 6.1 formally defines a relation, and Section 6.2 gives an informal description of a relation. Section 6.3 discusses the data integrity constraints pertaining to a relational data model. This is followed by a brief introduction in Section 6.4 to relational algebra as a means of specifying the logic for data retrieval from a series of relations. Section 6.5 introduces the concepts of views and materialized views as different ways of looking at data stored in relations. The rest of the chapter discusses mapping a conceptual schema to its logical counterpart using several examples. Section 6.6 introduces the idea of information preservation in data model mapping. Sections 6.7.1.1 and 6.7.1.2 present a detailed discussion of fundamental methods for transforming basic ER constructs (entity types, relationship types) to the logical tier. Section 6.7.1.3 demonstrates mapping techniques using Bearcat Incorporated's Design-Specific ERD as the source (conceptual) schema. The solution highlights the information-reducing nature of the transformation process. Then, an information-preserving grammar for the logical schema is presented in Section 6.7.2. A discussion of the heuristics for mapping EER constructs to the logical schema and the metadata lost in the transformation process follows in Section 6.8.1. Next, Section 6.8.2 presents the information-preserving grammar for modeling EER constructs at the logical tier. Finally, mapping complex ER modeling constructs to the logical tier is covered in Section 6.9.

## **6.1 DEFINITION**

---

In *Foundation for Object/Relational Databases: The Third Manifesto*, C. J. Date and Hugh Darwen wrote the following: “The foundation of modern database technology is without question the relational model; it is that foundation that makes the field a science. Thus, any book on the fundamentals of database technology that does not include a thorough coverage of the relational model is by definition shallow. Likewise, any claim to expertise

in the database field can hardly be justified if the claimant does not understand the relational model in depth.”<sup>3</sup>

E. F. Codd proposed the relational data model in 1970 as a logically sound basis for describing the structure of data as well as data manipulation operations.<sup>4</sup> The model uses the concept of mathematical relations as its foundation and is based on set theory. The relational data model includes a group of basic data manipulation operations called **relational algebra**. This chapter discusses the structural aspect of the relational data model and contains a brief introduction to relational algebra. Relational algebra is covered in more detail in Chapter 11.

The simplicity of the concept and its sound theoretical basis are two reasons why the relational data model has gained popularity as a logical data model for database design. As the name implies, the **relational data model** represents a database as a collection of relation values (“relations,” for short), where a relation resembles a two-dimensional table of values presented as rows and columns. A row in the table represents a set of *related* data values and is called a **tuple**. All values in a column are of the same data type. A column is formally referred to as an **attribute**. The set of all tuples in the table goes by the name **relation**. A relation consists of two parts: (a) an empty shell called the **heading**, which is a tuple of attribute names, and (b) a **body** of data that inhabits the shell; this body of data is a set of tuples all having the same heading. The heading of a relation is also referred to in the literature as a **relation schema**, **schema**, **scheme**, or **intension**. When the heading is called intension, then the body of the relation is referred to as **extension**.

Recall from Chapter 2 that the domain of an attribute is the set of possible values for the attribute. In a relational data model, a **domain** is defined as a set of atomic values for an attribute, and an attribute is the name of a role played by a domain in the relation.

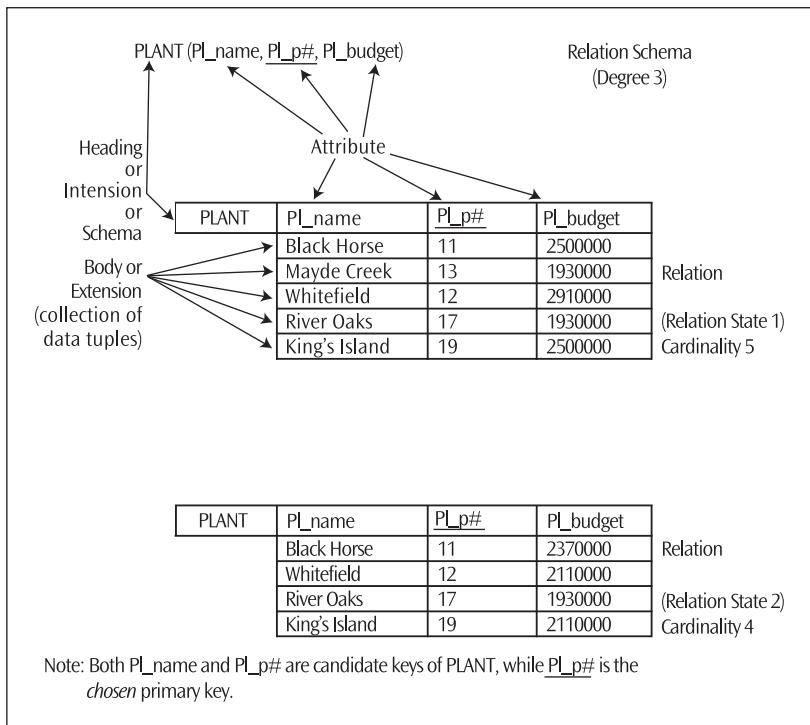
In formal terms, an attribute,  $A$ , is an ordered pair  $(N, D)$  in which  $N$  is the name of the attribute and  $D$  is the domain that the named attribute represents. If  $r$  is a relation whose structure is defined by a set of attributes  $A_1, A_2 \dots, A_n$ , then  $R(A_1, A_2 \dots, A_n)$  is called the **relation schema**<sup>5</sup> of the relation  $r$ . In other words, a relation schema,  $R$ , is a named collection of attributes  $(R, C)$  in which  $R$  is the name of the relation schema and  $C$  is the set  $\{(N_1, D_1), (N_2, D_2), \dots, (N_n, D_n)\}$ , where  $N_1, N_2 \dots, N_n$  are *distinct* names.  $r$  is the relation (or relation state) over the schema  $R$ . The domain of  $A_i$  ( $i = 1, 2, \dots, n$ ) is often denoted as **Dom** ( $A_i$ ). The number of attributes ( $n$ ) in  $R$  is called the **degree** (or **arity**) of  $R$ . A relation state  $r$  of the relation schema  $R(A_1, A_2 \dots, A_n)$ , also denoted as  $r(R)$ , is a set of  $n$ -tuples  $\{t_1, t_2 \dots, t_m\}$ . Each  $n$ -tuple  $t_j$  ( $j = 1, 2, \dots, m$ ) in  $r(R)$  is an ordered list of  $n$  values  $\langle v_{1j}, v_{2j}, \dots, v_{nj} \rangle$  where each  $v_{ij}$  ( $i = 1, 2, \dots, n$ ;  $j = 1, 2, \dots, m$ ) is an element of **Dom** ( $A_i$ ) ( $i = 1, 2, \dots, n$ ) or, when allowed, a missing value represented by a special value called **null**. The number of tuples,  $m$ , in the relation state is called the

<sup>3</sup>Date, C. J., and Hugh Darwen. *Foundation for Object/Relational Databases*, Addison-Wesley, 1998.

<sup>4</sup>Codd, E. F. “A Relational Model for Large Shared Data Banks,” *Communications of the ACM*, 13, 6 (June, 1970) 377–387.

<sup>5</sup>A relation schema is sometimes loosely referred to as a relation. C. J. Date (2004) has coined the term “relvar” (for relation variable) to distinguish a relation schema from a relation. It is important to notice the difference between a relation and a relation schema as well as between a relation schema and a *relational* schema. A **relational schema** defines a set of relation schemas in a relational data model.

**cardinality** of the relation. Figure 6.1 shows an example of a relation schema of degree 3 and two relation states of cardinality 5 and 4, respectively, for that relation schema.



**FIGURE 6.1** An example of a relation schema and its relation states

## 6.2 CHARACTERISTICS OF A RELATION

The definitions in Section 6.1 ascribe certain characteristics to a relation schema and its relation states. These characteristics are as follows:

- A relation is a mathematical term approximately equivalent to a two-dimensional table.
- A relation has a heading—that is, a tuple of attribute names (also known as intension, relation schema, relvar (relation variable), or shell) and a body, which is a set of tuples all having the same heading (also known as extension or a relation state).
- Attributes of a relation schema have unique names.<sup>6</sup>
- Values of an attribute in a relation come from the same domain.

<sup>6</sup>An implicit assumption of the relational database theory is that attributes have unique names over the entire *relational* schema. Additional discussion of this issue follows in Section 6.6.2.

- The order of arrangement of tuples is immaterial.
- Each attribute value in a tuple is atomic; hence, composite and multi-valued attributes are not allowed in a relation schema.<sup>7</sup>
- The order of arrangement of attributes in a relation schema is immaterial as long as the correspondence between attributes and their values in the relation is maintained.
- Derived attributes are not captured in a relation schema.
- All tuples in a relation must be distinct (that is, a relation schema must have a unique identifier).

Since the relational database theory stipulates that attribute names must be unique over the entire relational schema, the following guidelines represent the approach used in this chapter for developing attribute names:

- Each attribute name begins with a prefix of up to three letters that represents a meaningful abbreviation of the name of the relation schema to which the attribute belongs. This prefix is followed by an underscore character.
- Only the first letter of the prefix is capitalized.
- Following the underscore character is a suffix that corresponds to the attribute name itself. This suffix may contain only lowercase letters, the pound sign (#), and underscore characters; it corresponds to the name of the attribute in the conceptual data model.

These guidelines were used in developing the attribute names for the PLANT relation schema in Figure 6.1.

### **6.3 DATA INTEGRITY CONSTRAINTS**

---

Data integrity constraints, generally referred to as just **integrity constraints**, are rules that govern the behavior of data at all times in a database. Data integrity constraints are technical expressions of the business rules that emerge from the user requirements specification for a database application system. To that extent, these integrity constraints prevail across all tiers of data modeling—conceptual, logical, and physical.

In general, integrity constraints are considered to be part of the schema in that they are declared along with the structural design of the data model (conceptual, logical, and physical) and hold for all valid states of a database that correctly model an application (Ullman and Widom, 1997). Although it is possible to specify all the integrity constraints as a part of the modeling process, some cannot be expressed explicitly or implicitly in the schema of the data model. For instance, an ERD (conceptual schema) is not capable of expressing domain constraints of attributes. Likewise, constraints that are not declarative in nature and therefore require procedural intervention (e.g., “no more than two projects can be in one location at given time”) cannot be directly specified in a relational schema. As a consequence, such constraints are carried forward through the data modeling tiers in

---

<sup>7</sup>A relation maps to a flat file at the physical level. This is supported by the assumption of the theory behind the relational model called the First Normal Form assumption. A discussion of normal forms begins in Chapter 8.

textual form and are often referred to as **semantic integrity constraints**,<sup>8</sup> whereas the constraints directly expressed in the schema of the data model are labeled as **schema-based or declarative constraints**.<sup>9</sup> Domain constraints, key constraints, entity integrity constraints, referential integrity constraints, and functional dependency constraints are part of the declarative constraints of a relational data model. (Entity integrity constraints and referential integrity constraints are discussed in the following sections. Functional dependency constraints are discussed in Chapter 7.) Uniqueness constraints and structural constraints of a relationship type are declarative constraints specified at the conceptual level in an ER model. Semantic integrity constraints that require procedural intervention can always be specified and enforced through application programming code and are called **application-based constraints**. Procedural enforcement of integrity constraints is also often possible via mechanisms incorporated in the DBMS that use general purpose constraint specification languages, such as triggers and assertions.

Since every valid state of a database must satisfy the declarative and procedural forms of the integrity constraints noted earlier, these constraints are collectively referred to as **state constraints**. Sometimes, integrity constraints may have to be specified to define legal transitions of state. For example, the value of the attribute **Marital\_status** can change from Married to Divorced or Widowed, but not to Single. These types of constraints are referred to as **transition constraints** and invariably require procedural language support either through application programs or general-purpose constraint specification languages of the DBMS.

### 6.3.1 The Concept of Unique Identifiers

An attribute or a collection of attributes can serve as a unique identifier of a relation schema. A set of one or more attributes that, taken collectively, uniquely identifies a tuple of a relation is referred to as the **superkey** of the relation schema. Accordingly, no two distinct tuples in any state  $r$  of a relation schema  $R$  can have the same value for the superkey. For example, consider the data in the PRESCRIPTION-A relation shown in Table 6.1. There are 10 superkeys in PRESCRIPTION-A. They are:

**Rx\_rx#**  
**(Rx\_rx#, Rx\_pat#)**  
**(Rx\_rx#, Rx\_medcode)**  
**(Rx\_rx#, Rx\_dosage)**  
**(Rx\_pat#, Rx\_medcode)**  
**(Rx\_rx#, Rx\_pat#, Rx\_medcode)**  
**(Rx\_rx#, Rx\_pat#, Rx\_dosage)**  
**(Rx\_rx#, Rx\_medcode, Rx\_dosage)**  
**(Rx\_pat#, Rx\_medcode, Rx\_dosage)**  
**(Rx\_rx#, Rx\_pat#, Rx\_medcode, Rx\_dosage)**

<sup>8</sup>Note, however, that all integrity constraints pertain to the “semantics” of the database application.

<sup>9</sup>A third category of constraints is inherent to the data model; these are called inherent model-based constraints and do not necessarily emerge from the semantics of the application. For example, the characteristics of a relation stated in Section 6.2 are inherent to a relational schema. Similarly, an inherent constraint of an ERD is that a relationship type can link only entity types.

**PRESCRIPTION-A**

Rx_rx#	Rx_pat#	Rx_medcode	Rx_dosage
A100	7642	PCN	3
A103	4678	TYL	2
A102	4772	CLR	2
A101	6742	ASP	2
A104	4772	ZAN	3
A105	7456	CLR	2
A107	2222	TYL	2
A106	4772	VAL	2
A108	7384	CLR	3
A109	7384	ZAN	2
A110	7642	VAL	2

**PRESCRIPTION-B**

Rx_rx#	Rx_pat#	Rx_medcode	Rx_dosage
B100	7642	PCN	3
B103	4678	TYL	2
B102	4772	CLR	2
B101	6742	ASP	2
B102	4772	ZAN	<b>2</b>
B105	7456	CLR	2
B107	2222	TYL	2
B106	4772	VAL	2
B108	7384	CLR	3
B109	7384	ZAN	2
B100	7642	VAL	2

**TABLE 6.1** The PRESCRIPTION-A and PRESCRIPTION-B relations

In general, if K is a superkey, any **superset**<sup>10</sup> of K (i.e., any collection of attributes that includes K) is also a superkey.

Another type of unique identifier is called a **candidate key**. A candidate key is defined as a superkey with no **proper subsets**<sup>11</sup> that are superkeys. In short, a candidate key has two properties:

- **Uniqueness**—Two tuples of a relation schema cannot have identical values for the collection of attributes (or single attribute) that constitute(s) the candidate key.
- **Irreducibility**—No proper subset of the candidate key has the uniqueness property.

The uniqueness property is common to both a superkey and a candidate key, while the irreducibility property is present only in a candidate key. This indicates that a superkey may contain superfluous attributes while a candidate key does not.

As shown in Table 6.2, only **Rx\_rx#** and **(Rx\_pa#, Rx\_medcode)** are candidate keys of PRESCRIPTION-A. For example, **(Rx\_rx#, Rx\_pat#)**, **(Rx\_rx#, Rx\_medcode)**, and **(Rx\_rx#, Rx\_dosage)** are not candidate keys of PRESCRIPTION-A because a proper subset of each, **(Rx\_rx#)**, is also a superkey of PRESCRIPTION-A. **(Rx\_pat#, Rx\_medcode)** is a candidate key of PRESCRIPTION-A, because neither of its proper subsets **(Rx\_pat#)** and **(Rx\_medcode)** is a superkey of PRESCRIPTION-A. In essence, any superset of attributes that contains either **(Rx\_rx#)** or **(Rx\_pat#, Rx\_medcode)**, while being a superkey, is not a candidate key of PRESCRIPTION-A.

As noted earlier, **Rx\_rx#** is a candidate key of PRESCRIPTION-A. However, **Rx\_rx#** does not uniquely identify tuples of PRESCRIPTION-B, as seen in Table 6.1. That is, **Rx\_rx#** is not a superkey of PRESCRIPTION-B. But, as can be seen in Table 6.2, six superkeys exist in PRESCRIPTION-B. They are:

- (Rx\_rx#, Rx\_medcode)**
- (Rx\_pat#, Rx\_medcode)**
- (Rx\_rx#, Rx\_pat#, Rx\_medcode)**
- (Rx\_rx#, Rx\_medcode, Rx\_dosage)**
- (Rx\_pat#, Rx\_medcode, Rx\_dosage)**
- (Rx\_rx#, Rx\_pat#, Rx\_medcode, Rx\_dosage)**

---

<sup>10</sup>A set  $S_1$  is a *superset* of another set  $S_2$  if every element in  $S_2$  is in  $S_1$ .  $S_1$  *may* have elements that are not in  $S_2$ .

<sup>11</sup>A set  $S_2$  is a *subset* of another set  $S_1$  if every element in  $S_2$  is in  $S_1$ .  $S_1$  *may* have exactly the same elements as  $S_2$ . If  $S_2$  is a subset of  $S_1$ ,  $S_1$  is a superset of  $S_2$ . A set  $S_2$  is a *proper subset* of another set  $S_1$  if every element in  $S_2$  is in  $S_1$  and  $S_1$  has some elements which are not in  $S_2$ .

PREScription (Rx\_rx#, Rx\_pat#, Rx\_medcode, Rx\_dosage)

	PRESCRIPTION-A Superkey	Candidate Key	PRESCRIPTION-B Superkey	Candidate Key
Rx_rx#	Yes	Yes	No	No
Rx_pat#	No	No	No	No
Rx_medcode	No	No	No	No
Rx_dosage	No	No	No	No
Rx_rx#, Rx_pat#	Yes	No	No	No
Rx_rx#, Rx_medcode	Yes	No	Yes	Yes
Rx_rx#, Rx_dosage	Yes	No	No	No
Rx_pat#, Rx_medcode	Yes	Yes	Yes	Yes
Rx_pat#, Rx_dosage	No	No	No	No
Rx_medcode, Rx_dosage	No	No	No	No
Rx_rx#, Rx_pat#, Rx_medcode	Yes	No	Yes	No
Rx_rx#, Rx_pat#, Rx_dosage	Yes	No	No	No
Rx_rx#, Rx_medcode, Rx_dosage	Yes	No	Yes	No
Rx_pat#, Rx_medcode, Rx_dosage	Yes	No	Yes	No
Rx_rx#, Rx_pat#, Rx_medcode, Rx_dosage	Yes	No	Yes	No

**TABLE 6.2** Superkeys and candidate keys in the PRESCRIPTION-A and PRESCRIPTION-B relations

The superkey (**Rx\_rx#, Rx\_medcode**) is a candidate key of PRESCRIPTION-B because neither of its proper subsets—(**Rx\_rx#**) or (**Rx\_medcode**)—is a superkey of PRESCRIPTION-B. Similarly (**Rx\_pat#, Rx\_medcode**) is a candidate key of PRESCRIPTION-B because neither of its proper subsets is a superkey of PRESCRIPTION-B. Next, let's evaluate whether the superkey (**Rx\_rx#, Rx\_pat#, Rx\_medcode**) is a candidate key of PRESCRIPTION-B. This can be done by investigating if any of its proper subsets is a superkey of PRESCRIPTION-B. The proper subsets of (**Rx\_rx#, Rx\_pat#, Rx\_medcode**) are:

- (Rx\_rx#)
- (Rx\_pat#)
- (Rx\_medcode)
- (Rx\_rx#, Rx\_pat#)
- (Rx\_rx#, Rx\_medcode)
- (Rx\_pat#, Rx\_medcode)

While (**Rx\_rx#**), (**Rx\_pat#**), (**Rx\_medcode**), and (**Rx\_rx#, Rx\_pat#**) are not superkeys of PRESCRIPTION-B (**Rx\_rx#, Rx\_medcode**) and (**Rx\_pat#, Rx\_medcode**) are indeed superkeys of PRESCRIPTION-B. Therefore, (**Rx\_rx#, Rx\_pat#, Rx\_medcode**) is not a candidate key of PRESCRIPTION-B because two of its proper subsets are superkeys of PRESCRIPTION-B.

The same procedure can be used to evaluate if any other superkey of PRESCRIPTION-B is also a candidate key of PRESCRIPTION-B. A review of the data in Table 6.1 indicates that none of the other superkeys is also a candidate key of PRESCRIPTION-B.

Every attribute, whether atomic or composite, plays one of three roles in a relation schema. It is a key attribute, a non-key attribute, or a candidate key. Any attribute that is a proper subset of a candidate key is a **key attribute**. An attribute that is not a subset of a candidate key is a **non-key attribute**. For example, in PRESCRIPTION-A, because **(Rx\_pat#, Rx\_medcode)** is a candidate key, **Rx\_pat#** and **Rx\_medcode** are key attributes. Because **Rx\_rx#**, an atomic attribute, is a candidate key of PRESCRIPTION-A, it is, by definition, *not* a key attribute in PRESCRIPTION-A because **Rx\_rx#** is not a proper subset of **Rx\_rx#**. However, **Rx\_rx#** is not a non-key attribute either, because it is a candidate key of PRESCRIPTION-A. In short, *a candidate key in itself is neither a key attribute nor a non-key attribute in R*. Further discussion of this appears in Chapter 7.

A **primary key** serves the role of uniquely identifying tuples of a relation. A primary key is a candidate key (an irreducible unique identifier) with one additional property. This additional property results from what is known as the **entity integrity constraint**, which specifies that the primary key of a relation schema cannot have a “missing” value (i.e., a null value), essentially assuring identification of every tuple in a relation. Given a set of candidate keys for a relation schema, exactly one is chosen as the primary key. Since the entity integrity constraint applies exclusively to a primary key, by implication the rest of the candidate keys<sup>12</sup> not chosen as the primary key apparently tolerate “missing” values; otherwise, the entity integrity rule will apply to all candidate keys (Date, 2004). In short, when a candidate key of a relation schema is chosen to be the primary key of that relation schema, it is bound by the entity integrity constraint, and from that point forward, the alternate keys (other candidate keys) may entertain “missing” values.<sup>13</sup> What happens when an alternate key is chosen as the primary key at a later time? The answer is inherent in the definition of the primary key—that is, the entity integrity constraint must be enforced in order for the alternate key to become the primary key of the relation schema.

Since the primary key value is used to identify individual tuples in a relation, if null values are allowed for the primary key, some tuples cannot be identified; hence, the entity integrity constraint, which disallows null values for a primary key. A primary key of a relation schema is denoted by underlining the atomic attributes (or single attribute) that constitute a primary key. In Table 6.3, **PI\_p#** is the primary key of PLANT. Observe that both **PI\_name** and **PI\_p#** are candidate keys of PLANT as modeled in the conceptual schema (refer to Figure 3.11) and that **PI\_p#** has been chosen by the systems analyst/database designer as the primary key. This implies that the entity integrity constraint is imposed only on **PI\_p#** and not on **PI\_name**.

---

<sup>12</sup>Once the primary key is chosen from among the set of candidate keys, the remaining candidate keys are referred to as “alternate keys.” The choice of a primary key of a relation schema from among its candidate keys is essentially arbitrary.

<sup>13</sup>This indeed is in contradiction with the strict definition of a relation—i.e., a relation cannot have missing value for any of its attributes. Nonetheless, relaxation of this constraint is very common in practice.

PLANT (Pl\_name, Pl\_p#, Pl\_budget)

PROJECT (Prj\_name, Prj\_p#, Prj\_location, *Prj\_pl\_p#*)

Version 1

OR

PROJECT (Prj\_name, Prj\_p#, Prj\_location, *Prj\_pl\_name*)

Version 2

Note: The Foreign keys are italicized.

PLANT	<u>Pl_name</u>	<u>Pl_p#</u>	<u>Pl_budget</u>
	Black Horse	11	1230000
	Mayde Creek	13	1930000
	Whitefield	12	2910000
	River Oaks	17	1930000
	King's Island	19	2500000
	Ashton	15	2500000

PROJECT	<u>Prj_name</u>	<u>Prj_p#</u>	<u>Prj_location</u>	<i>Prj_pl_p#</i>
	Solar Heating	41	Sealy	11
	Lunar Cooling	17	Yoakum	17
	Synthetic Fuel	29	Salem	17
	Nitro-Cooling	23	Parthi	12
	Robot Sweeping	31	Ponca City	11
	Robot Painting	37	Yoakum	19
	Ozone Control	13	Parthi	19

Version 1

Note: PROJECT.*Prj\_pl\_p#* is the foreign key referencing PLANT.Pl\_p#, the primary key of PLANT.

PROJECT	<u>Prj_name</u>	<u>Prj_p#</u>	<u>Prj_location</u>	<i>Prj_pl_name</i>
	Solar Heating	05	Sealy	Black Horse
	Lunar Cooling	17	Yoakum	River Oaks
	Synthetic Fuel	29	Salem	River Oaks
	Nitro-Cooling	23	Parthi	Whitefield
	Robot Sweeping	31	Ponca City	Black Horse
	Robot Painting	37	Yoakum	King's Island
	Ozone Control	13	Parthi	King's Island

Version 2

Note: PROJECT.*Prj\_pl\_name* is the foreign key referencing PLANT.Pl\_name, a candidate key of PLANT.

**TABLE 6.3** Enforcement of referential integrity constraint

### 6.3.2 Referential Integrity Constraint in the Relational Data Model

While the key constraints (superkey and candidate key) and entity integrity constraint (primary key) pertain to individual relation schemas, a **referential integrity constraint** is specified between two relation schemas, R1 and R2. It is common for tuples in one relation to reference tuples in the same or other relations. A referential integrity constraint stipulates that a tuple in a relation, **r2** (i.e., **r(R2)**), that refers to another relation, **r1** (i.e., **r(R1)**), refers to a tuple that exists in **r1**. R2 is called the *referencing* relation schema, and R1 is known as the *referenced* relation schema. An important type of referential integrity specification is known as the **foreign key constraint**. A foreign key constraint (a) establishes an explicit association between the two relation schemas and (b) maintains the integrity of such an association (i.e., maintains consistency among tuples of the two relations). In order to establish a foreign key constraint between two relation schemas, R1 and R2, it is necessary that every value of an attribute  $A_2$ , atomic or composite, in relation **r2** must either occur as the value of a candidate key  $A_1$  in some tuple of the relation **r1** that **r2** refers to, or, if allowed, be null.<sup>14</sup> The attribute in R2 that meets this condition is called a **foreign key** of R2.<sup>15</sup> In other words, the definition of a foreign key requires that the set of referenced attributes be a candidate key in the referenced relation schema. Incidentally, **r1** and **r2** need not be distinct relations. A referential integrity constraint, on the other hand, is defined more generally in that it *does not* require the referenced set of attributes to be any sort of key in the referenced relation. In relational database theory, this constraint is referred to as an **inclusion dependency** and is algebraically expressed as follows:

$$R2.\{A_2\} \subseteq R1.\{A_1\}$$

A foreign key constraint is just a special kind of inclusion dependency. In terms of the foreign key constraint, the foreign key value in **r2** represents a *reference* to the tuple containing the matching candidate key value in the referenced tuple in **r1**. When a relation schema includes a foreign key that references some candidate key in the same relation schema, then the relation schema is said to be *self-referencing* (equivalent to a recursive relationship type in the ERD).

As an example, consider a scenario in which manufacturing plants undertake projects. All plants need not undertake projects, but any plant may undertake several projects. Likewise, a project is controlled by only one plant and not all projects are controlled by plants. Version 1 of the PROJECT relation in Table 6.3 uses **Prj\_pl\_p#** as the foreign key referencing the primary key of PLANT (**Pl\_p#**). Observe that the name of the foreign key attribute begins with the prefix **Prj** that represents an abbreviation

---

<sup>14</sup>If the participation of R2 in this relationship with R1 is partial, the foreign key attribute,  $A_2$ , can have null values.

<sup>15</sup>The relational model originally required that foreign keys reference, very specifically, the primary key, not just candidate keys. This limitation is unnecessary and undesirable in general, although it might often constitute good discipline in practice (Date, 2004, p. 274).

of the referencing relation schema name; the suffix is the name of the referenced attribute **PI\_p#** from the referenced relation schema. The constraint is expressed as follows:

$$\text{PROJECT}.\{\text{Prj\_pl\_p}\} \subseteq \text{PLANT}.\{\text{Pl\_p}\} \text{ or } \emptyset$$

Here,  $\emptyset$  indicates “null,” meaning that a project need not be controlled by a plant.

In Version 2 of the PROJECT relation, **Prj\_pl\_name** in PROJECT references **Pl\_name** in PLANT, a candidate key of PLANT, not the primary key of PLANT.

The constraint is expressed as follows:

$$\text{PROJECT}.\{\text{Prj\_pl\_name}\} \subseteq \text{PLANT}.\{\text{Pl\_name}\} \text{ or } \emptyset$$

The naming convention applied here to name a foreign key attribute in the referencing relation schema consists of (a) the prefix used in conjunction with the attribute names in the referencing relation schema, (b) an underscore, and (c) the referenced attribute name. The example in Table 6.3 demonstrates enforcement of a referential integrity constraint along with this naming convention. Observe that in its current state, all projects in the PROJECT relation are controlled by some plant.

## 6.4 A BRIEF INTRODUCTION TO RELATIONAL ALGEBRA

---

The relational data model includes a group of basic data manipulation operations. On the basis of its theoretical foundation in set theory, these data manipulation operations include Union, Difference, and Intersection. Five other operations also exist: Selection, Projection, Cartesian product, Join, and Division. Collectively, these eight operations comprise what is known as relational algebra.

This section contains a brief introduction to six of these operations and provides examples of their use in the context of the AW\_PLANT, TX\_PLANT, and PROJECT relations that appear in Table 6.4.<sup>16</sup> The AW\_PLANT and PROJECT relations are the same as those that appear in Table 6.3. The AW\_PLANT relation contains data on award-winning plants, whereas the TX\_PLANT relation contains data on plants located in the state of Texas.

---

<sup>16</sup>A formal discussion of relational algebra, including the Cartesian product and Division operations, appears in Chapter 11.

AW\_PLANT (Aw\_pl\_name, Aw\_pl\_p#, Aw\_pl\_budget)

TX\_PLANT (Tx\_pl\_name, Tx\_pl\_p#, Tx\_pl\_budget)

PROJECT (Prj\_name, Prj\_p#, Prj\_location, *Prj\_aw\_pl\_p#*)

Note: The foreign keys are italicized.

AW_PLANT	<u>Aw_pl_name</u>	<u>Aw_pl_p#</u>	Aw_pl_budget
	Black Horse	11	1230000
	Mayde Creek	13	1930000
	Whitefield	12	2910000
	River Oaks	17	1930000
	King's Island	19	2500000
	Ashton	15	2500000

TX_PLANT	<u>Tx_pl_name</u>	<u>Tx_pl_p#</u>	Tx_pl_budget
	Southern Oaks	16	1230000
	River Oaks	17	1930000
	Kingwood	18	2910000

PROJECT	<u>Prj_name</u>	Prj_n#	Prj_location	<i>Prj_aw_pl_p#</i>
	Solar Heating	41	Sealy	11
	Lunar Cooling	17	Yoakum	17
	Synthetic Fuel	29	Salem	17
	Nitro-Cooling	23	Parthi	12
	Robot Sweeping	31	Ponca City	11
	Robot Painting	37	Yoakum	19
	Ozone Control	13	Parthi	19

TABLE 6.4 Sample relations for relational algebra examples

#### 6.4.1 Unary Operations: Selection ( $\sigma$ ) and Projection ( $\pi$ )

The Selection operation is used to create a second relation by extracting a horizontal subset of tuples from a relation that matches specified search criteria. On the other hand, the Projection operation creates a second relation by extracting a vertical subset of columns

from a relation. Selection and Projection are referred to as unary operations because they produce a new relation by manipulating only a single relation.

**Example of a Selection operation:** Which award-winning plants have a budget that exceeds \$2,000,000?

**Result:**<sup>17</sup>

R_aw_pl_name	R_aw_pl_p#	R_aw_pl_budget
Whitefield	12	2910000
King's Island	19	2500000
Ashton	15	2500000

293

**Example of a Projection operation:** What is the plant number and budget of each award-winning plant?

**Result:**

R_aw_pl_p#	R_aw_pl_budget
11	1230000
13	1930000
12	2910000
17	1930000
19	2500000
15	2500000

If each attribute involved in a Projection operation is not unique, it is possible for the new relation produced to have duplicate tuples. If this occurs, these duplicate tuples are deleted.

#### 6.4.2 Binary Operations: Union ( $\cup$ ), Difference ( $-$ ), and Intersection ( $\cap$ )

Each of the three binary operations involves the manipulation of two union-compatible relations in order to produce a third relation. **Union compatibility** requires that each relation be of the same degree and that corresponding attributes in the two relations come from (or share) the same domain. The union of two relations is formed by adding the tuples from one relation to those from a second relation to produce a third relation consisting of tuples that are in *either* the first relation or the second relation. The difference of two relations is a third relation that contains the tuples from the first relation that are not in the second relation. The intersection of two relations is a third relation containing tuples common to the two relations.

---

<sup>17</sup>The result obtained in this and all other examples in this section produces the new relation RESULTS, which contains its own unique attribute names.

**Example of the Union operation:** *What plants are either located in Texas or are award-winning plants?*

**Result:**

R_aw_pl_name	R_aw_pl_p#	R_aw_pl_budget
Black Horse	11	1230000
Mayde Creek	13	1930000
Whitefield	12	2910000
River Oaks	17	1930000
King's Island	19	2500000
Ashton	15	2500000
Southern Oaks	16	1930000
Kingwood	18	1930000

Observe that duplicate tuples are omitted (i.e., the River Oaks plant, an award-winning plant located in Texas, appears only once in the result). In addition, note that AW\_PLANT and TX\_PLANT are union compatible.

**Example of the Difference operation:** *Which Texas plants are not award-winning plants?*

**Result:**

R_aw_pl_name	R_aw_pl_p#	R_aw_pl_budget
Southern Oaks	16	1930000
Kingwood	18	1930000

Note the difference between this result and the following result, to the question “Which award-winning plants are not located in Texas?”

**Result:**

R_aw_pl_name	R_aw_pl_p#	R_aw_pl_budget
Black Horse	11	1230000
Mayde Creek	13	1930000
Whitefield	12	2910000
King's Island	19	2500000
Ashton	15	2500000

**Example of the Intersection operation:** Which award-winning plants are located in Texas?

Result:

R_aw_pl_name	R_aw_pl_p#	R_aw_pl_budget
River Oaks	17	1930000

### 6.4.3 The Natural Join (\*) Operation

The Join operation combines two relations into a third relation by matching values for attributes in the two relations that come from the same domain. The tuples in the new relation consist of the tuples extracted from the first relation concatenated with each tuple in the second relation where there is a match on the joining attributes. When the new relation contains all the attributes from the first relation plus all the attributes from the second relation but does not redundantly carry the joining attributes, the result is called a Natural Join.<sup>18</sup>

**Example of a Natural Join operation:** Perform a natural join of the award-winning plant and project relations.

Result:

R_prj_name	R_prj_n#	R_prj_location	R_aw_pl_p#	R_aw_pl_p#
Solar Heating	41	Sealy	11	Black Horse
Lunar Cooling	17	Yoakum	17	River Oaks
Synthetic Fuel	29	Salem	17	River Oaks
Nitro-Cooling	23	Parthi	12	Whitefield
Robot Sweeping	31	Ponea City	11	Black Horse
Robot Painting	37	Yoakum	19	King's Island
Ozone Control	13	Parthi	19	King's Island

Note that the joining attributes here are **Aw\_pl\_p#** and **Prj\_aw\_pl\_p#**. Relational algebra operations can be combined to form more complex expressions. Using the Natural Join operation (as shown here) followed by a Selection operation on plant number 11 and a Projection operation on the project name yields a relation that contains the names of the projects controlled by plant number 11 (i.e., Solar Heating and Robot Sweeping).

---

<sup>18</sup>Three other types of joins exist: Equijoin, Theta Join, and Outer Join. These are discussed in Chapter 11.

## 6.5 VIEWS AND MATERIALIZED VIEWS IN THE RELATIONAL DATA MODEL

A view is a named “virtual” relation schema constructed from one or more relation schemas through the use of one or more relational algebra operations. Unlike a relation schema, a view does not store actual data but is just a logical window to view selected data (attributes and tuples) from one or a set of relations. The value of a view at any given time is a derived relation *state* and results from the evaluation of a specified relational expression at that time. In a database environment, views play a number of roles:

- They allow the same data to be seen by different users in different ways at the same time. This makes it possible for different users with different requirements to “view” only the portions of the database that are relevant to them.
- They provide security by restricting user access to a predetermined set of tuples and attributes from predetermined relations.
- They hide data complexity from the user because data selected from several relations can be made to appear to the user as a single object.

Implicit in this description is the concept of logical data independence. Recall that logical data independence is the *immunity of external schema* (the top tier of the three-schema architecture outlined in Chapter 1) to changes in the logical structure of the conceptual schema (the middle tier of the three-schema architecture). Views are the means by which logical data independence is implemented in a relational database system.

Changes to the logical structure of the conceptual schema consist of two aspects: **growth** and **restructuring**.<sup>19</sup>

- Growth is expansion of existing relation schema in the relational data model by adding new attributes. Addition of a new relation schema to the existing relational data model is another dimension of growth. Both imply incorporation of new information in the conceptual schema.
- Restructuring is about changes to the logical structure of the conceptual schema other than growth. The idea is about restructuring the schema that is *information-equivalent* to the current conceptual schema—that is, the restructuring should be reversible. For instance, replacement of a relation schema with multiple relation schemas for some reason (e.g., normalization)<sup>20</sup> amounts to restructuring that is information-equivalent.<sup>21</sup>

Observe that growth does not have any impact on the existing external schema—that is, the existing user views and programs. The impact of restructuring can be compensated

<sup>19</sup>Date, C. J., and Hugh Darwen. *Foundation for Object/Relational Databases*, Addison-Wesley, 1998.

<sup>20</sup>Normalization is covered in Part III.

<sup>21</sup>Deletion of an attribute or a relation schema usually does not yield an information-equivalent logical structure; if it does, this implies that the original conceptual schema had information redundancy.

for by re-creating the original structure to serve as the source schema(s) for the construction of the external schema (e.g., a denormalized view of the normalized relation schemas that simulates the original logical structure). Views are an effective means to achieve logical data independence as long as any restructuring of the conceptual schema creates a version that is information-equivalent to the original conceptual schema.

A **materialized view** (also known as a **snapshot**), despite the similarity in name, is not a view. Like a view, it is constructed from one or more relation schemas; unlike a view, a materialized view is stored in the database and refreshed when updates occur to the relation schemas from which the materialized view is generated. Materialized views are often used to freeze data as of a certain moment without preventing updates to continue on the data in the relation schemas on which they are based. A materialized view is often deleted when it is not used for a period of time and then reconstructed from scratch as future needs dictate.

## **6.6 THE ISSUE OF INFORMATION PRESERVATION**

---

In Part I, Conceptual Data Modeling, it was argued that a database design is primarily driven by the business rules specified by the user community, and that a comprehensive set of business rules must be captured during the requirements analysis. While the resulting conceptual schema (ER model) adds significant value to the database design activity, it also has the burden of preserving the information content of the business rules through the remainder of the design steps. The goal of any schema transformation method ought to be the preservation of the information capacity of the source data model, such as an ER model in the target data model (in this case, the relational data model).

Many data modeling scholars (Batini, Ceri, and Navathe, 1992; Navathe, 1992; Fahrner and Vossen, 1995) have noted that representations of logical schema (e.g., relational schema) continue to suffer from the limitations of information-reducing transformations from the conceptual model. According to Fahrner and Vossen:

The major difficulty of logical database design—i.e., of transforming an ER schema into a schema in the language of some logical model—is the *information preservation issue*. Indeed, assuring a complete mapping of all modeling constructs and constraints which are inherent, implicit, or explicit in the source schema is problematic since constraints of the source model often cannot be represented directly in terms of the structures and constraints of the target model. In such cases they must be realized through application programs; or, alternatively, an information-reducing transformation must be accepted.<sup>22,23</sup>

---

<sup>22</sup>Formally, a transformation in which all possible database instances that can be represented in a source schema can be represented in a target schema implies information preservation (Fahrner and Vossen, 1995). To that extent, the spirit of the discussion here pertains to *design* information preservation.

<sup>23</sup>Fahrner, C., and Vossen, G. "A Survey of Database Design Transformations Based on the Entity-Relationship Model." *Data & Knowledge Engineering*, 15, 3, 1995: 213–250.

Thus, it is crucial that a logical schema captures as much of the inherent, implicit, and explicit constructs and constraints conveyed by the conceptual schema as possible through the logical modeling grammar, and that it carry forward the remainder of the semantic integrity constraints to the next step in the design process, which is physical data modeling.

However, popular mapping techniques presently in vogue that map directly to a relational schema are information-reducing in nature. Section 6.7 discusses these techniques and points out their information-reducing aspects. Then, Section 6.8 presents a new information-preserving logical modeling grammar<sup>24</sup> for transforming ER and EER models to their logical counterparts.

## 6.7 MAPPING AN ER MODEL TO A LOGICAL SCHEMA

The major task in this step is to convert the Design-Specific ERD to a target logical schema. Constructs and constraints that cannot be captured in the target logical schema cannot be ignored but must be carried forward to the next step in the design process. In addition, the list of semantic integrity constraints that supplements the Design-Specific ERD needs to be evaluated for mapping to the target logical schema. The product of this activity is a logical data model consisting of: (a) a logical schema expressed using a logical modeling grammar, and (b) a list of semantic integrity constraints not incorporated into the logical schema to be carried forward to the next step in the database design activity. The logical modeling grammar presented here is a variation of a relational schema that captures additional constructs and constraints typically not present in a relational schema. The reason for this choice stems from the fact that the technology-dependent version is intended for deployment in a relational database management system (RDBMS).

### 6.7.1 Information-Reducing Mapping of ER Constructs

A popular technique for representing the logical counterpart of an ERD is described in this section and evaluated for how well it preserves, in the logical schema, the information available from the ER model. To illustrate the transformation process, different variations of subsets from the Design-Specific ERD developed in Chapter 3 (see Figure 3.13) for Bearcat Incorporated will be used.

#### 6.7.1.1 Mapping Entity Types

Review the ERD in Figure 6.2. The first step in logical data modeling is to create a relation schema for each base entity type in the ERD. Only the stored attributes are translated to the logical level. In the case of composite attributes, only their constituent atomic components are recorded. A primary key is chosen from among the candidate keys for each relation schema. The atomic attribute or attributes that make up the primary key of the relation schema are underlined. In the ERD in Figure 6.2, either the composite attribute [**Emp\_a, Emp\_n**] or the composite attribute [**Fname, Minit, Lname, Name\_tag**] can serve as

<sup>24</sup>An early version of this grammar was presented in the Workshop on Information Technology and Systems (WITS) in December 2000 at Brisbane, Australia (Umanath and Chiang, 2000).

the primary key of EMPLOYEE because both are defined as candidate keys of EMPLOYEE in the ERD. Likewise, either **Pl\_name** or **Pnumber** becomes the primary key of PLANT. Observe in Figure 6.3 that [**Emp\_e#a**, **Emp\_e#n**] and **Pl\_p#** have been specified as the primary key of the relation schema (EMPLOYEE and PLANT, respectively), as indicated by the underlining of these attributes.

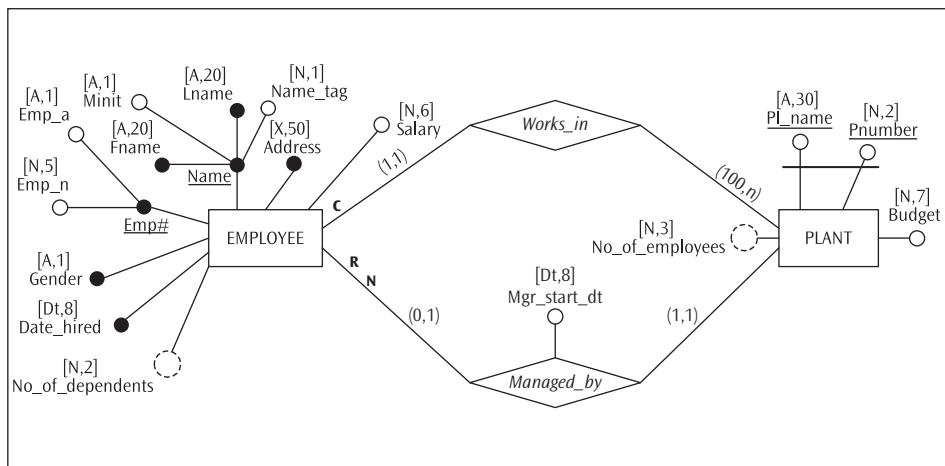


FIGURE 6.2 Excerpt from Design-Specific ERD of Figure 3.13

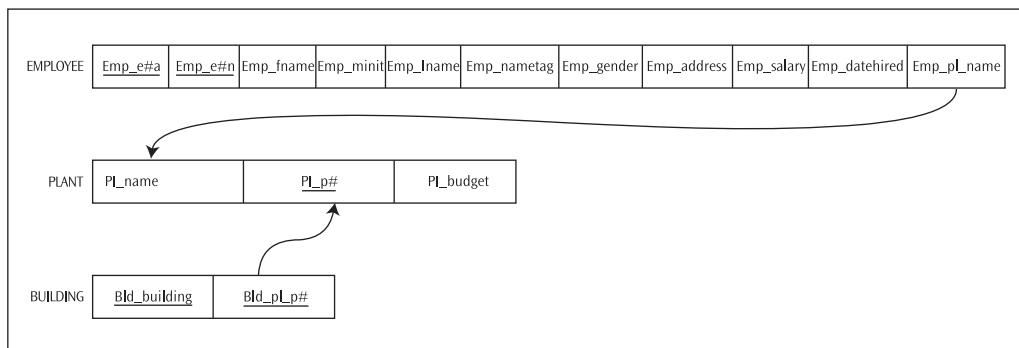


FIGURE 6.3 Logical schema for the ERD in Figure 6.2: Foreign key design

A weak entity type in the ERD is mapped in a similar manner except that the primary key of each identifying parent of the weak entity type is added to the relation schema. The attributes thus added, along with the partial key of the weak entity type, form the primary key of the relation schema representing the weak entity type. In other words, there is no such thing called a “weak” relation schema. All relation schemas are “strong”—meaning they have a primary key. Observe the mapping of the weak entity type BUILDING in Figure 6.3. The primary key of PLANT, the only identifying parent of BUILDING, has been concatenated to the partial key, **Bld\_building**, to form the primary key of the relation schema BUILDING.

### 6.7.1.2 Mapping Relationship Types

As was the case with the entity types, each relationship type (diamond symbol) in the ERD is individually accounted for in the logical model. Remember that considerable effort has been expended to prepare the Design-Specific ER model to be suitable for direct mapping to the logical tier. As a result, notice that all relationship types in the Design-Specific ERD are binary (or recursive) in nature and exhibit a cardinality ratio of **1:1** or **1:n**; there are no complex relationship types and no **m:n** cardinality ratios. In addition, there are no multi-valued attributes. The mapping of a relationship type is accomplished by simply imposing a referential integrity constraint between the relation schemas over the relationship type being mapped (see Section 6.3.2).

**Mapping the Cardinality Ratio of 1:n: Foreign Key Design** When the cardinality ratio of the relationship type under consideration is **1:n**, the entity type on the *n-side* of the relationship type is the child in the PCR (parent-child relationship), and hence it becomes the **referencing relation schema** in the logical tier. Because each tuple in the child relation is related to at most one tuple in the referenced (parent) relation, the placement of foreign key attributes (or attributes) in the *referencing relation schema* maps the relationship type specified in the ERD. Of course, the foreign key placed in the referencing schema shares the same domain with a candidate key (invariably and preferably, but not necessarily the primary key) of the **referenced relation schema**. This is expressed diagrammatically by drawing a *directed arc* originating from the foreign key attribute(or attributes) to the relation schema it references, with the arrow head terminating at the referenced candidate (or primary) key. In addition, a rule requiring that all foreign key values match some value of the referenced candidate key except, of course, when the foreign key value is null (i.e., referential integrity constraint) is implied. This technique is often labeled as the **foreign key technique/design**.

In the example in Figure 6.2, the *Works\_in* relationship type represents a PCR where PLANT is the parent and EMPLOYEE is the child. Therefore, the mapping of *Works\_in* is implemented in the logical schema (Figure 6.3) by adding the attribute **Emp\_pl\_name** to EMPLOYEE, which fulfills the role of a foreign key by referencing **Pl\_name**, an alternate key of PLANT.<sup>25</sup> Also, in Figure 6.3, notice the directed arc originating from **EMPLOYEE.Emp\_pl\_name** and pointing at **PLANT.PI\_name**. The attributes of a relationship type in the ERD, if any, are also added to the referencing (child) relation schema along with the foreign key attribute(s). Next, while the primary key of BUILDING is **[Bld\_building, Bld\_pl\_p#]**, **BUILDING.Bld\_pl\_p#** also serves as the foreign key referencing **PLANT.PI\_p#**, the primary key of the (identifying) parent, PLANT, thus accurately portraying the presence of the identifying relationship type, *Houses*.

The clarity of the foreign key design deteriorates very quickly as the number of relation schemas in the relational data model increases, because the spaghetti of directed arcs becomes difficult to trace. Alternatively, instead of the directed arcs, it is possible to express the relationship types via the specification of *inclusion dependencies*. This method of expressing a referential integrity constraint is shown in Figure 6.4. Both meth-

---

<sup>25</sup>Instead, the foreign key added to EMPLOYEE could have been an attribute that references **PI\_p#**, the primary key of PLANT.

ods fail to map the participation constraints available in the ERD to the logical tier—a case in point for information reduction in mapping.

```

EMPLOYEE (Emp_e#a, Emp_e#n, Emp_fname, Emp_minit, Emp_lname, Emp_nametag, Emp_gender, Emp_address, Emp.salary, Emp_datehired, Emp_pl_name)
# EMPLOYEE.{Emp_pl_name} ⊆ PLANT.{PL_name}
PLANT (PL_pname, PL_p#, PL_budget)
BUILDING (Bld_building, Bld_pl_p#)
# BUILDING.{Bld_pl_p#} ⊆ PLANT.{PL_p#}

```

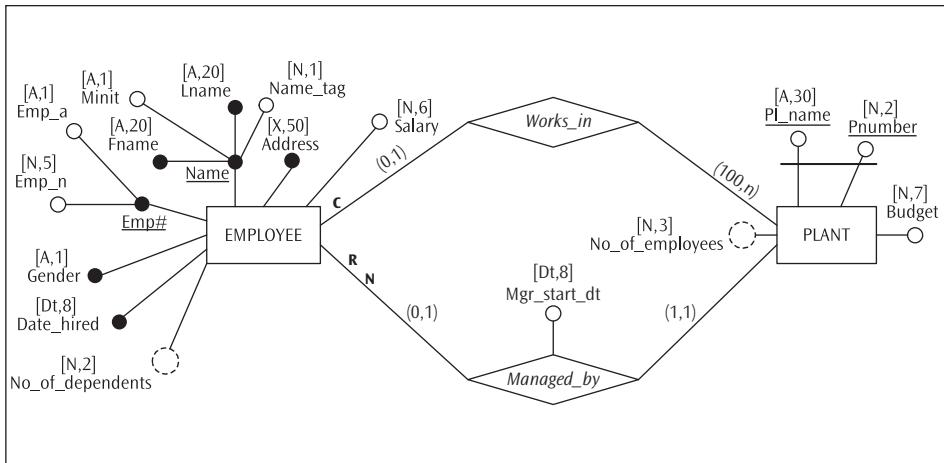
301

**FIGURE 6.4** Referential integrity constraints in Figure 6.3 expressed as inclusion dependencies: Foreign key design

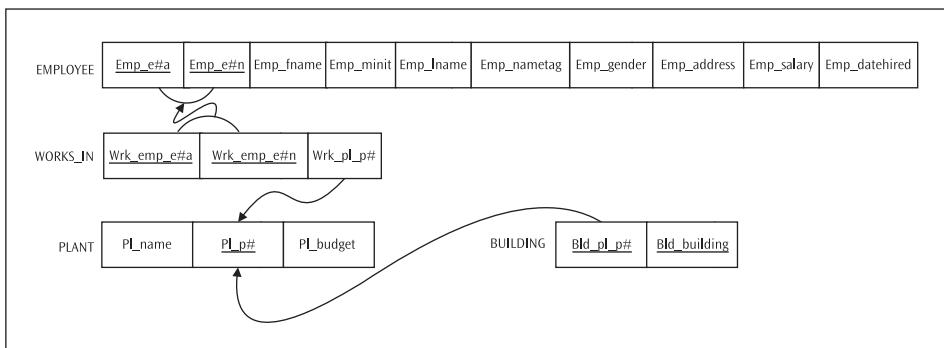
When using the foreign key design, it is important to note what happens should the foreign key attribute inadvertently be placed in the parent relation (what should be the referenced relation schema) instead of in the child relation (what should be the referencing relation schema). For example, mapping the *Works\_in* relationship type by placing the primary key of EMPLOYEE [**PI\_emp\_e#a, PI\_emp\_e#n**] as a foreign key attribute in PLANT amounts to a reversal of the cardinality constraint and results in a serious semantic error.

**Mapping the Cardinality Ratio of 1:n: Cross-Referencing Design** Suppose an employee does not necessarily have to work in a plant. This is modeled by changing the participation of EMPLOYEE in *Works\_in* from a 1 to a 0. This condition is handled at the logical tier by allowing the foreign key, **EMPLOYEE.Emp\_pl\_name**, to have null values (see Figures 6.3 and 6.4). There is an alternative modeling approach that guarantees the absence of null values in the foreign key.<sup>26</sup> This **cross-referencing design** entails the creation of a relation schema to represent the relationship type, and it is illustrated in Figures 6.5 through 6.7. Figure 6.6 portrays this design where WORKS\_IN is a relation schema. No relation state in this design need have null values for the foreign key(s). Any relation state of the relation schema, WORKS\_IN, will contain tuples for only those employees who work in one of the plants. In other words, employees who do not work in any of the plants may still be employees of the corporation, and tuples in the EMPLOYEE relation will capture this information. Because these particular employees do not work in any of the plants, though, WORKS\_IN will not have tuples for any of these employees.

<sup>26</sup>There is a sense of tentativeness associated with the status of “null” values in data. Therefore, a database design that avoids presence of null values is expected to be relatively more robust than the ones that freely allow null values in the data.



**FIGURE 6.5** Reproduction of Figure 6.2 with a change in participation constraints of *Works\_in*



**FIGURE 6.6** Logical schema for the ERD in Figure 6.5: Cross-referencing design

```

EMPLOYEE (Emp_e#a, Emp_e#n, Emp_fname, Emp_minit, Emp_lname, Emp_nametag, Emp_gender, Emp_address, Emp.salary, Emp.datehired)
WORKS_IN (Wrk_emp_e#, Wrk_emp_e#n, Wrk_pl_p#)
# WORKS_IN.{Wrk_emp_e#, Wrk_emp_e#n} ⊆ EMPLOYEE.{Emp_e#a, Emp_e#n}
# WORKS_IN.{Wrk_pl_p#} ⊆ PLANT.{Pl_p#}
PLANT (Pl_pname, Pl_p#, Pl_budget)
BUILDING (Bld_building, Bld_pl_p#)
# BUILDING.{Bld_pl_p#} ⊆ PLANT.{Pl_p#}

```

**FIGURE 6.7** Referential integrity constraints in Figure 6.6 expressed as inclusion dependencies: Cross-referencing design

In the previous approach (the foreign key design illustrated in Figures 6.2 through 6.4), if there is a need to add an employee who does not work for any plant, a

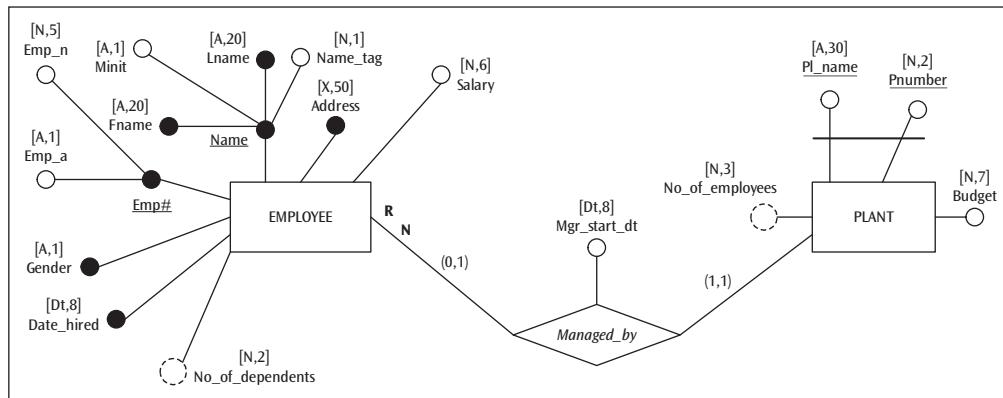
tuple is added to the EMPLOYEE relation with a null value for the foreign key

**EMPLOYEE.Emp\_pl\_name**. In the current cross-referencing design, however, an employee can be added without any concern about whether he or she works in a plant or not, and yet not use a null value in EMPLOYEE to portray this. Figure 6.7 is a representation of the same cross-referencing design using inclusion dependencies.

The cross-referencing design is not as compact as the foreign key design and can proliferate the logical data model with numerous relation schemas in a hurry. Even when the participation of EMPLOYEE in *Works\_in* is optional, it is probably practical to use the foreign key design and allow null values for the foreign key **EMPLOYEE.Emp\_pl\_name** in the interest of a somewhat more efficient design. However, the price paid is that, by definition, we don't have a 'relational schema'; plus, the null value possible for foreign key in some of the EMPLOYEE tuples is also an issue to be considered. We will encounter another such condition in the following discussion about mapping 1:1 relationship types to the logical tier.

**Mapping the Cardinality Ratio of 1:1** When the cardinality ratio of a relationship type is 1:1, mapping such a relationship type to the logical tier becomes somewhat complicated because either one of the entity types engaged in this relationship type can be the parent or the child. Three solutions are possible, and each is conducive to specific situations—the situations occasioned by particular participation constraints in the relationship.

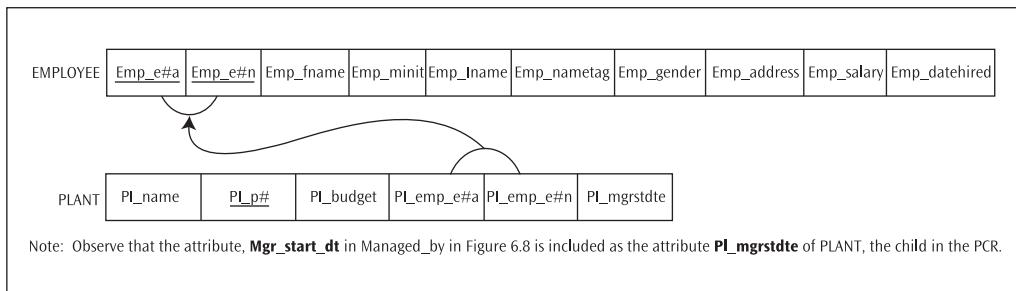
**Case 1:** The participation constraint of *one* of the entity types participating in the relationship type is *total*, as shown in Figure 6.8.



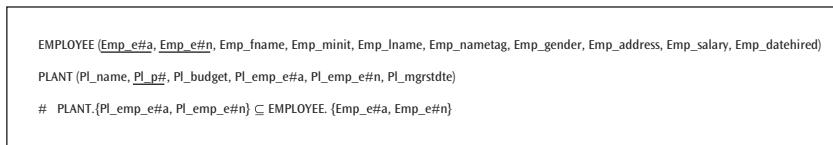
**FIGURE 6.8** A 1:1 relationship type with total participation of PLANT in *Managed\_by*

The best way to handle this case is to choose the entity type with total participation in the relationship type to assume the role of child in the PCR. Then the foreign key design described at the beginning of this section for mapping a 1:n cardinality ratio can be directly applied here as well, as shown in Figures 6.9 and 6.10. As the foreign key design amounts to an implicit specification of 1:n cardinality ratio, an additional constraint explicitly specifying that the foreign key value must be unique is necessary to convey

the **1:1** cardinality ratio, which incidentally renders the foreign key an alternate key (candidate key) of the child entity type.



**FIGURE 6.9** Logical schema for the ERD in Figure 6.8: Foreign key design



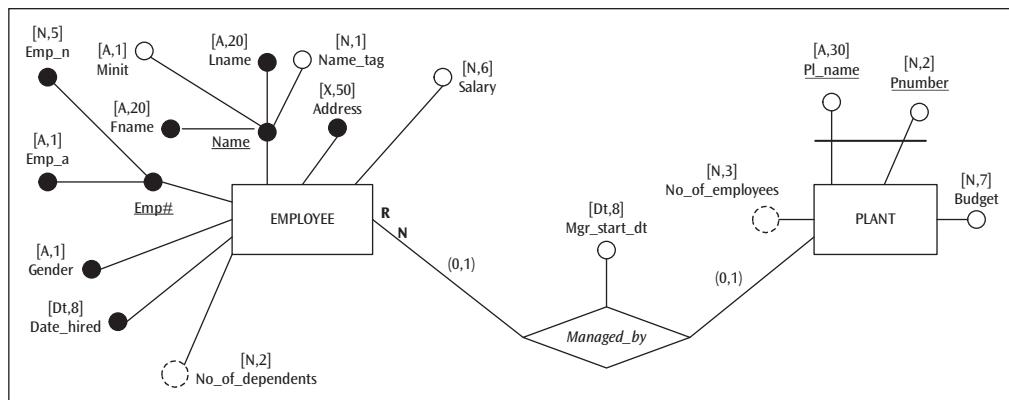
**FIGURE 6.10** Referential integrity constraint in Figure 6.9 expressed as an inclusion dependency: Foreign key design

Equally important, the total participation of the child in the relationship type is incorporated in the design via an explicit specification of “no missing value” for the foreign key. For instance, consider the *Managed\_by* relationship type in Figure 6.8. The participation of PLANT in this relationship type is total, as indicated by the (min) value of **1**. Therefore, if an attribute (or attributes) representing the foreign key is (are) added to the relation schema PLANT and this foreign key shares the same domain with the primary key [**Emp\_e#a**, **Emp\_e#n**] or any other candidate key (for example, [**Emp\_fname**, **Emp\_minit**, **Emp\_Iname**, **Emp\_nametag**]) of the relation schema EMPLOYEE (see Figure 6.9), then with additional constraint specifications of uniqueness and “not null” on the foreign key in PLANT, the *Managed\_by* relationship type will be fully implemented in the relational data model. *These two constraints can be specified declaratively.*

On the other hand, suppose EMPLOYEE is chosen as the child in this 1:1 PCR. Then, under the foreign key design, either **Emp\_pl\_name** or **Emp\_pl\_p#** will be added to the relation schema EMPLOYEE as the foreign key to depict the *Managed\_by* relationship type. However, since the participation of EMPLOYEE in the *Managed\_by* relationship is only partial, the corresponding foreign key values can legitimately have null values in some of the EMPLOYEE tuples. As a consequence, addition of a tuple in PLANT will require a procedural intervention in EMPLOYEE that ensures, at the least, a concurrent assignment of an employee to manage a plant, because every plant must have a manager (total participation of PLANT in the *Managed\_by* relationship type). In other words, a plant added to the PLANT relation must reference some employee tuple in the EMPLOYEE relation.

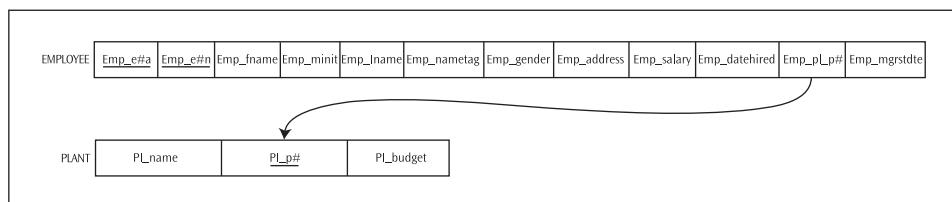
Thus, it is fairly obvious that including the foreign key in the relation schema that has mandatory participation in the relationship type is the better solution.

**Case 2:** The participation constraints of *both* entity types in the relationship type are *partial*, as shown in Figure 6.11.

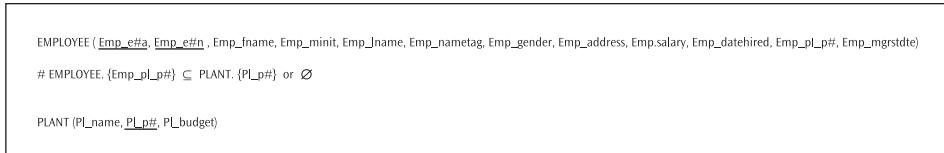


**FIGURE 6.11** A 1:1 relationship type with partial participation of both EMPLOYEE and PLANT in *Managed\_by*

In this case, from a strictly design perspective, addition of a foreign key in either one of the relation schemas involved in the 1:1 relationship type is sufficient. Figures 6.11, 6.12, and 6.13 display an example for Case 2. The ERD in Figure 6.11 is a simple variation of the earlier example (Figure 6.8) in that the participation of PLANT in *Managed\_by* is also partial, as reflected by the (min) value of 0. Since the participation of both EMPLOYEE and PLANT in the *Managed\_by* relationship type is partial, either PLANT or EMPLOYEE can assume the role of the child in this relationship. Figure 6.12 shows the foreign key design using a directed arc in which EMPLOYEE, as the child, carries the foreign key. The same design using inclusion dependency appears in Figure 6.13.



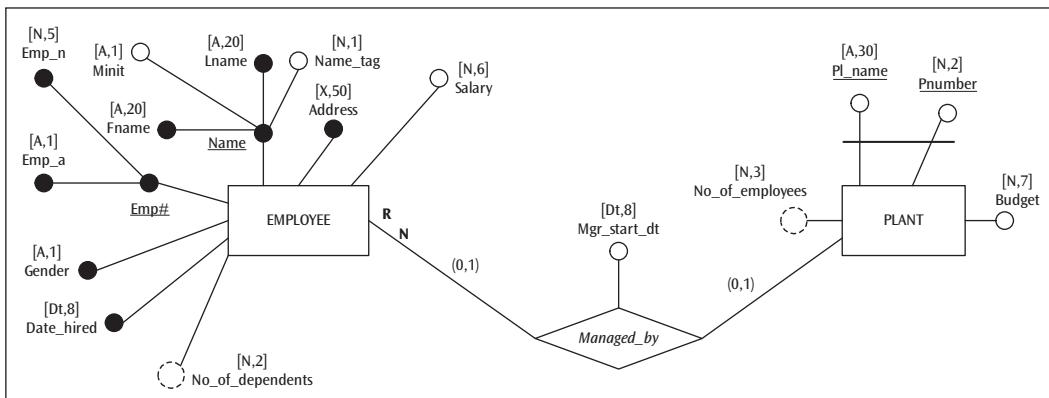
**FIGURE 6.12** Logical schema for the ERD in Figure 6.11: Foreign key design



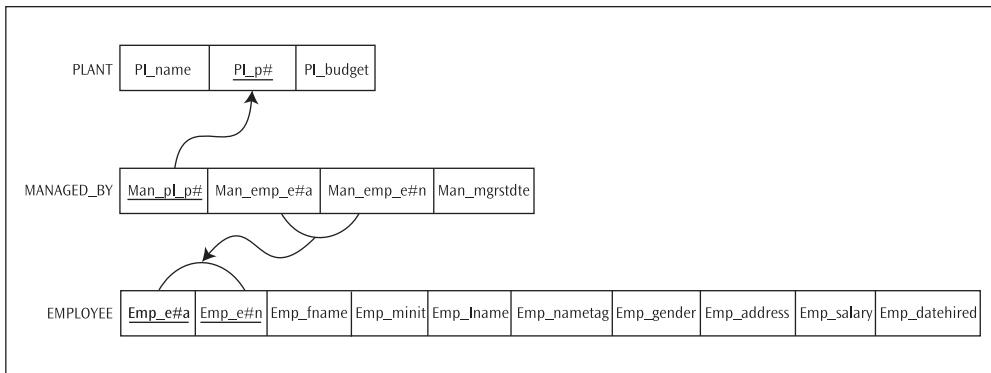
**FIGURE 6.13** Referential integrity constraint in Figure 6.12 expressed as an inclusion dependency: Foreign key design

306

Other semantic or operational considerations may sometimes suggest inclusion of the foreign key in a specific relation schema. For instance, the user may have a predisposition towards the semantics of the relationship. Sometimes, one of the entity types will have a small entity set relative to the other, in which case it is operationally efficient to designate it as the child in the PCR. In certain cases, optimal data access is facilitated by *mutual-referencing*—that is, when the two relation schemas directly reference each other by placing foreign keys in both. In this situation, cross-referencing ought to be considered instead of mutual-referencing, because mutual-referencing between two relation schemas entails specification of *additional* constraints to ensure consistency maintenance (i.e., reference to the correct tuple). Such constraints can only be implemented procedurally, and the ramifications are further clarified in the upcoming discussion of Case 3. The cross-referencing design eliminates imposition of such a constraint, however, at the expense of adding a relation schema to portray the relationship type. Sometimes, such an alternative may be worth considering, such as if the expected size of the intervening relation is small relative to the two base relations in the relationship type. The cross-referencing designs (using directed arcs and inclusion dependencies) for the ERD that was shown in Figure 6.11 and reproduced in Figure 6.14 are portrayed in Figures 6.15 and 6.16, respectively.



**FIGURE 6.14** A 1:1 relationship type with partial participation of both EMPLOYEE and PLANT in *Managed\_by*



**FIGURE 6.15** Logical schema for the ERD in Figure 6.14: Cross-referencing design

```

PLANT (PL_name, PL_p#, PL_budget)

MANAGED_BY (Man_pl_p#, Man_emp_e#, Man_emp_e#n, Man_mgrstdte)

#  MANAGED_BY {Man_pl_p#} ⊆ PLANT. {PL_p#}
#  MANAGED_BY {Man_emp_e#, Man_emp_e#n} ⊆ EMPLOYEE. {Emp_e#, Emp_e##n}

EMPLOYEE (Emp_e#, Emp_e##n, Emp_fname, Emp_mininit, Emp_lname, Emp_nametag, Emp_gender, Emp_address, Emp.salary, Emp_datehired)

```

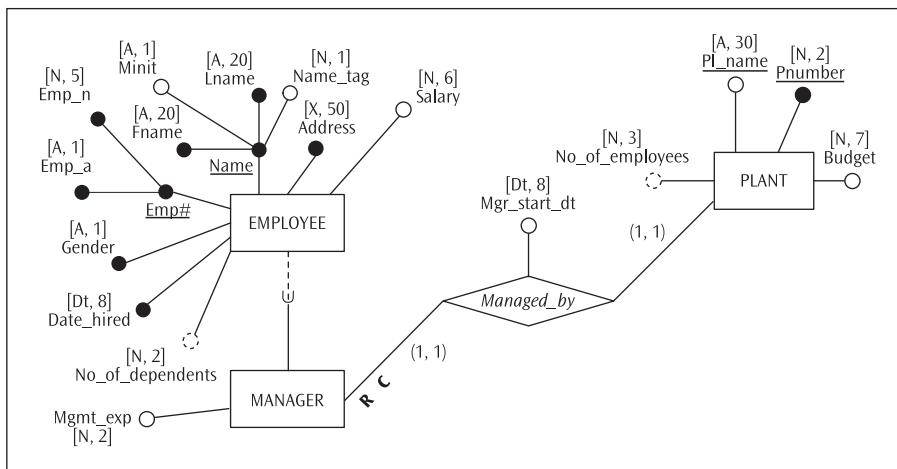
**FIGURE 6.16** Referential integrity constraints in Figure 6.15 expressed as inclusion dependencies: Cross-referencing design

**Case 3:** The participation constraints of *both* entity types in the relationship type are total.

Here, it is first necessary to add a foreign key in both relation schemas engaged in the relationship—in other words, mutual-referencing. Only by constraining both foreign keys to be unique can it be ascertained that the cardinality ratio is **1:1**. By virtue of this constraint, the defined foreign keys also become alternate keys of the respective relation schemas. Total participation of both entity types in the relationship type is incorporated in the design by not allowing null values for the two foreign keys. The presence of foreign keys in both relation schemas referencing each other creates two problems. The first problem is that it becomes necessary to make sure that the [primary/candidate key, foreign key] pairs in the two relations match. This cannot be done using declarative constraints; procedural intervention is necessary to accomplish this. For instance, if employee A12357 manages plant 19, since mutually referencing foreign keys are present in both relations, plant 19 must be managed by employee A12357 and nobody else. This is an additional constraint and can only be implemented via procedural intervention. Notice that in a cross-referencing design (as shown in Figure 6.15), the fact that A12357 manages plant 19 and *vice versa* is captured in the MANAGED\_BY relation; this eliminates the need for any procedural intervention. The second problem is that the two relation schemas referencing each other create a *cycle*. Therefore, enforcement of at least one of the two referential integrity constraints must be deferred to run time. An alternative solution

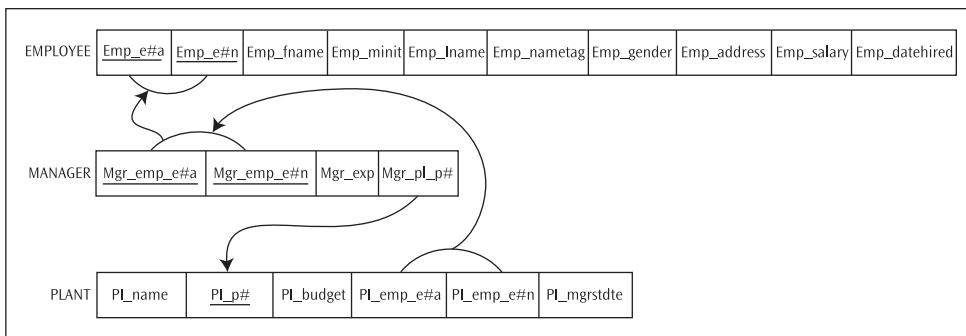
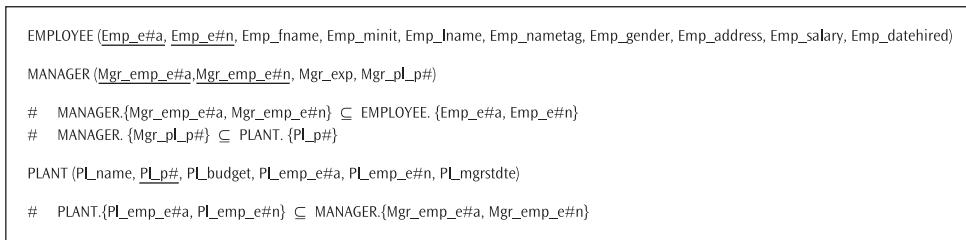
to preempt these problems is to merge the two relation schemas into a *single-schema design*. However, it is not always possible to adopt a single-schema design, especially if the relationship involves two distinct entity types and/or the entity types also participate independently in other relationship types.

A variation of the *Managed\_by* relationship type (the variation is intended only for lending better semantic sense) appears in Figure 6.17. MANAGER is a partial specialization of EMPLOYEE, a given plant is managed by exactly one manager, and each manager manages exactly one plant. The total participation of both MANAGER and PLANT in *Managed\_by* implies mutual-referencing between PLANT and MANAGER.



**FIGURE 6.17** A 1:1 relationship type with total participation of both MANAGER and PLANT in *Managed\_by*

The mutual-referencing designs in Figures 6.18 and 6.19 reflect this relationship. Clearly, a procedural constraint is required to verify that the pairs (**[Mgr\_emp\_e#a, Mgr\_emp\_e#n, Mgr\_pl\_p#]**, **[Pl\_emp\_e#a, Pl\_emp\_e#n, Pl\_p#]**) of values from MANAGER and PLANT match. In addition, deferred enforcement of at least one of the two referential integrity constraints (Figure 6.19) is also necessary. Additional procedures may be required to manage other constraints—for example, the deletion rule that restricts deletion of a tuple in MANAGER if a matching tuple in PLANT exists. Incidentally, the example here includes a partial specialization of EMPLOYEE as MANAGER. Mapping of enhanced ER model constructs is discussed in Section 6.8.

**FIGURE 6.18** Logical schema for the ERD in Figure 6.17: Mutual-referencing design**FIGURE 6.19** Referential integrity constraints in Figure 6.18 expressed as inclusion dependencies: Mutual-referencing design

Note that the redundant inclusion of foreign keys in both relation schemas referencing each other (mutual-referencing) can be done in all three cases as long as one is willing to incur the penalty of maintaining consistency. Similarly, in any relationship type that has a 1:1 cardinality ratio, combining the two entity types into a single relation schema requires evaluation because a single-schema design, when feasible, minimizes complex integrity constraints and attribute redundancies.

Also, the self-referencing property inherent to a recursive relationship type does not pose any special problems beyond what has been considered in the discussions in this section.

### 6.7.1.3 A Comprehensive Example

At this point, let us go through the process of mapping the Design-Specific conceptual model to a relational schema using a more comprehensive example. This process is summarized in Table 6.5.

- Create a relation schema for each base entity type in the ER diagram.
- Only the stored attributes are translated to the logical level – derived attributes are not mapped.
- In the case of composite attributes, only their constituent atomic components are recorded.
- A primary key is chosen from among the candidate keys for each relation schema. The atomic attribute(s) that make(s) up the primary key of the relation schema is/are underlined.
- When a weak entity type in the ER diagram is mapped, the primary key of each identifying parent of the weak entity type is added to the relation schema. The attributes thus added plus the partial key of the weak entity type together form the primary key of the relation schema representing the weak entity type. *There is no such thing as a “weak” relation schema.*
- Based on the cardinality ratio and participation constraints associated with a relationship type, choose either the foreign key design, the cross-referencing design, or the mutual-referencing design.
- The placement of foreign key attribute(s) in the *referencing* relation schema (child in the PCR) maps the relationship type specified (1:n or 1:1) and facilitates enforcement of the *referential integrity constraint*.
 

Note: *A candidate key (invariably and preferably, but not necessarily the primary key) of the parent (referenced relation schema) is the referenced attribute(s).*
- Attribute(s) of a 1:n or a 1:1 relationship type is (are) added to the referencing relation schema (child in the PCR).

**TABLE 6.5** A stepwise guide for mapping a Design-Specific ERD to a relational schema using the foreign key design

The Design-Specific ERD for Bearcat Incorporated, the domain constraints on the attributes, and a few other semantic integrity constraints not recorded in the ERD, which were provided in Chapter 3, are reproduced in Figure 6.20 and Table 6.6 for convenience.<sup>27</sup> The objective of this section is to develop the corresponding logical data model. To this end, the section first focuses on mapping the ERD to a logical schema.

---

<sup>27</sup>Recall that an ER model constitutes the ERD and the semantic integrity constraints not specified in the ERD.

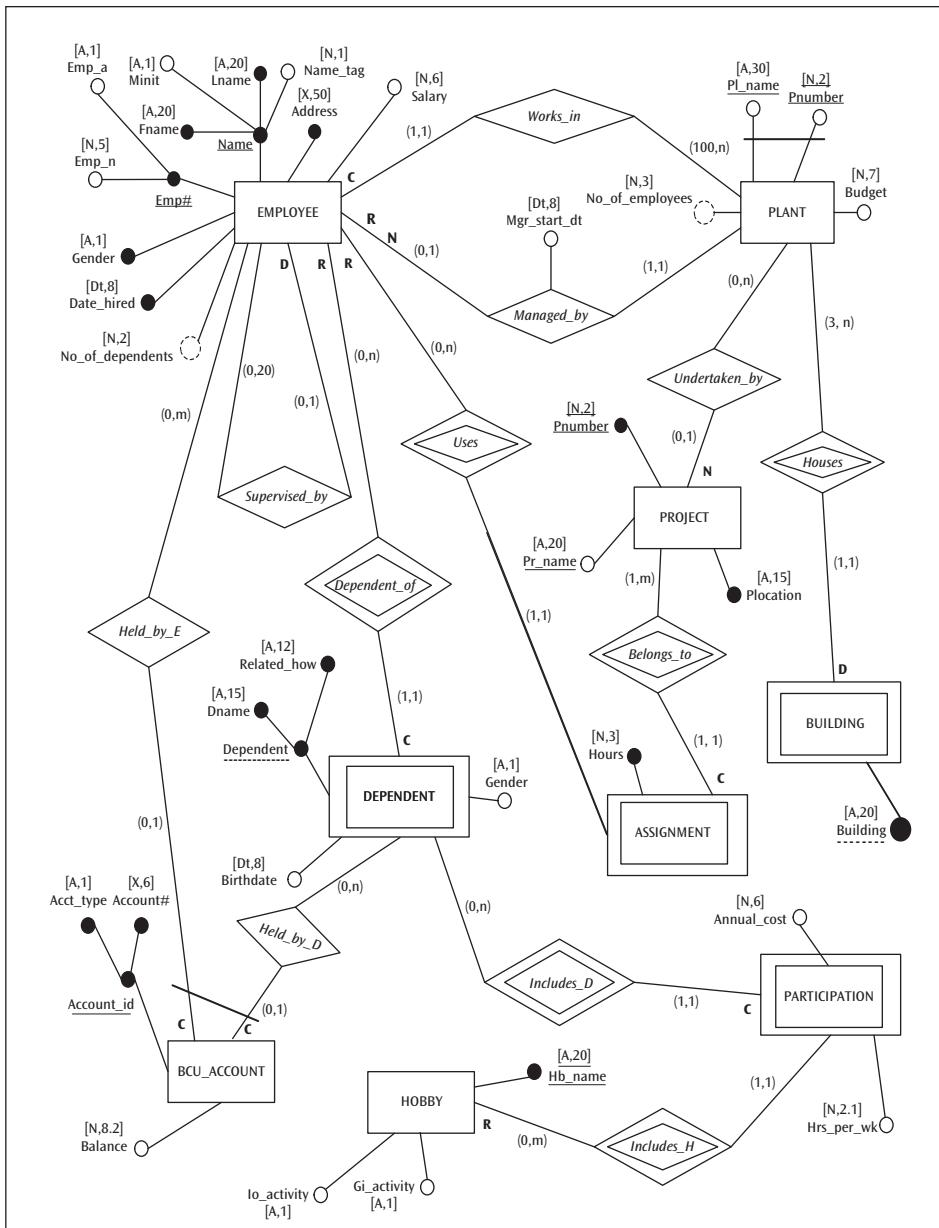


FIGURE 6.20 Design-Specific ERD for Bearcat Incorporated

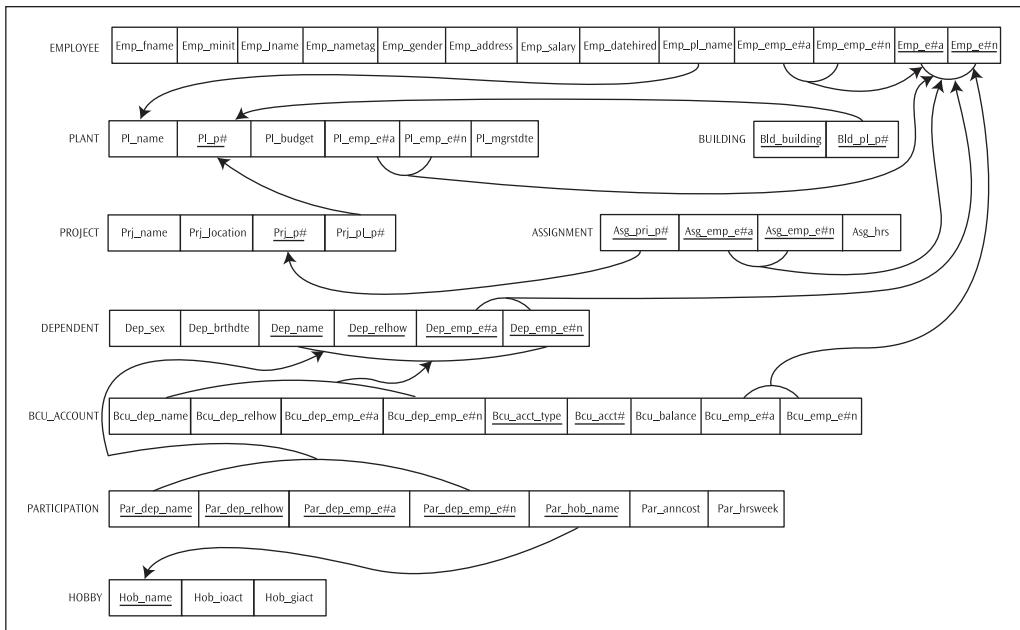
> Constraint	PLANT.Pnumber	IN (10 through 20)
> Constraint	Nametag	IN (1 through 9)
> Constraint	Gender	IN ("M," "F")
> Constraint	Salary	IN (35000 through 90000)
> Constraint	PROJECT.Pnumber	IN (1 through 40)
> Constraint	Plocation	IN ("Bellaire," "Blue Ash," "Mason," "Stafford," "Sugarland")
> Constraint	Aect_type	IN ("C," "S," "I")
> Constraint	Io_activity	IN ("I," "O")
> Constraint	Gi_activity	IN ("G," "I")
> Constraint	Related_how	IN ("Spouse") OR IN (( "Mother," "Daughter") AND Gender IN ("F")) OR IN (( "Father," "Son") AND Gender IN ("M"))

**Constraints Carried Forward to Logical Design**

1. An employee cannot be his or her own supervisor.
2. A dependent can have a joint account only with an employee of Bearcat Incorporated with whom he or she is related.
3. The salary of an employee cannot exceed the salary of the employee's supervisor.
4. Every plant is managed by an employee who works in the same plant.

**TABLE 6.6** Semantic integrity constraints for the final Design-Specific ER model

Using the mapping guide in Table 6.5 and applying the foreign key design discussed thus far in this chapter, the logical schema shown in Figure 6.21 and Table 6.7 can be obtained (the reader may wish to do this as an exercise and verify the result against Figure 6.21). The corresponding relational schema using inclusion dependencies is shown in Figure 6.22.



**FIGURE 6.21** Logical schema for Bearcat Incorporated: Foreign key design using directed arcs

Although the foreign key, cross-referencing, and mutual-referencing designs using directed arcs are good visual tools to understand mapping of relationship constructs from an ERD to a logical schema in isolated examples, and although replacing directed arcs with inclusion dependencies may add more precision to the expression of the relationships per se, both methods give up a significant amount of information (metadata) in the process of transforming a Design-Specific ERD to a relational schema. The lost information includes the following:

- Both methods are incapable of distinguishing between **1:1** and **1:n** cardinality ratios.
- Neither method maps the participation constraints of a relationship type.
- The optional/mandatory property of an attribute is not retained in the transformation.
- Alternate keys (candidate keys not chosen as the primary key) can no longer be identified.
- The composite nature of some collection of atomic attributes is ignored in the mapping process.
- Derived attributes specified in the ERD are not carried forward.
- Deletion rules are not mapped to the logical schema.
- Attribute type and size specified in the ERD are not carried forward.

> Constraint	Pl_p#	IN (10 through 20)
> Constraint	Nametag	IN (1 through 9)
> Constraint	Gender	IN ('M', 'F')
> Constraint	Salary	IN (35000 through 90000)
> Constraint	Prj_p#	IN (1 through 40)
> Constraint	Plocation	IN ('Bellaire', 'Blue Ash', 'Mason', 'Stafford', 'Sugarland')
> Constraint	Acct_type	IN ('C', 'S', 'I')
> Constraint	Io_activity	IN ('I', 'O')
> Constraint	Gi_activity	IN ('G', 'I')
> Constraint	Related_how	IN ('Spouse') OR
	Related_how	IN ('Mother', 'Daughter') AND Gender IN ('F')) OR
	Related_how	IN ('Father', 'Son') AND Gender IN ('M'))
> Constraint	Building	COUNT (not < 3)
> Constraint	No_of_employees	NOT < 100

#### Constraints Carried Forward to Physical Design

1. An employee cannot be his or her own supervisor.
2. A dependent can have a joint account only with an employee of Bearcat Incorporated with whom he or she is related.
3. The salary of an employee cannot exceed the salary of the employee's supervisor.
4. Every plant is managed by an employee who works in the same plant.

**TABLE 6.7** Semantic integrity constraints for the logical schema

None of the lost information is trivial. These items convey the business rules specified by the user community and design characteristics explicitly expressed in the Design-Specific ERD that serves as the source schema for the mapping process. Most of this metadata is needed to implement the physical data model correctly. One alternative is to include all the lost information in a list of semantic integrity constraints at the logical tier. Alternatively, the conceptual data model (the Design-Specific ERD and the semantic integrity constraints) can be used to supplement the underdeveloped logical schema. In that case, the very utility of a logical schema in the systematic development of a database design becomes questionable. The next section presents a logical modeling grammar that can produce an information-preserving script (logical schema).

```

L1: EMPLOYEE (Emp_fname, Emp_minit, Emp_lname, Emp_nametag, Emp_emp_e#a, Emp_emp_e#n, Emp_address, Emp_salary, Emp_pl_name, Emp_gender,
    Emp_datedhired, Emp_e#a, Emp_e#n)

# EMPLOYEE.{Emp_emp_e#a , Emp_emp_e# n} ⊆ EMPLOYEE.{Emp_ e#a, Emp_ e#n} or Ø
EMPLOYEE.{Emp_pl_name} ⊆ PLANT.{Pl_name}

L2: PLANT (Pl_p#, Pl_budget, Pl_name, Pl_emp_e#a, Pl_emp_e#n, Pl_mgrstdte)
# PLANT.{Pl_emp_e#a, Pl_emp_e#n} ⊆ EMPLOYEE.{Emp_ e#a, Emp_ e#n}

L3: BUILDING (Bld_building, Bld_pl_p#)
# BUILDING.{Bld_pl_p#} ⊆ PLANT.{Pl_p#}

L4: PROJECT (Prj_name, Prj_location, Prj_p#, Prj_pl_p#)
# PROJECT.{Prj_pl_p#} ⊆ PLANT.{Pl_p#} or Ø

L5: ASSIGNMENT (Asg_prj_p#, Asg_e_mp_e#a, Asg_emp_e#n, Asg_hrs)
# ASSIGNMENT.{Asg_prj_p# } ⊆ PROJECT.{Prj_p#}
ASSIGNMENT.{Asg_emp_e#a, Asg_emp_e#n} ⊆ EMPLOYEE.{Emp_ e#a, Emp_ e#n}

L6: DEPENDENT (Dep_sex , Dep_birthday , Dep_name, Dep_relhow , Dep_e_mp_e#a, Dep_emp_e#n)
# DEPENDENT.{Dep_emp_e#a, Dep_emp_e#n} ⊆ EMPLOYEE.{Emp_ e#a, Emp_ e#n}

L7: BCU_ACCOUNT (Bcu_dep_name, Bcu_dep_relhow, Bcu_dep_emp_e#a, Bcu_dep_emp_e#n, Bcu_acct_type, Bcu_acct#, Bcu_balance, Bk_emp_e#a,
    Bcu_emp_e#n)
# BCU_ACCOUNT.{Bcu_emp_e#a, Bcu_emp_e#n} ⊆ EMPLOYEE.{Emp_e#a, Emp_e#n} or Ø
BCU_ACCOUNT.{Bcu_dep_name, Bcu_dep_relhow, Bcu_dep_emp_e#a, Bcu_dep_emp_e#n} ⊆ DEPENDENT.{Dep_name, Dep_relhow, Dep_emp_e#a, Dep_emp_e#n} or Ø

L8: PARTICIPATION (Par_dep_name, Par_dep_relhow, Par_dep_emp_e#a, Par_dep_emp_e#n, Par_hob_name, Par_anncost, Par_hrsweek)
# PARTICIPATION.{Par_hob_name} ⊆ HOBBY.{Hob_name}
PARTICIPATION.{Par_dep_name, Par_dep_relhow, Par_dep_emp_e#a, Par_dep_emp_e#n} ⊆ DEPENDENT.{Dep_name, Dep_relhow, Dep_emp_e#a, Dep_emp_e#n}

L9: HOBBY (Hob_name, Hob_joact, Hob_giact)

```

**FIGURE 6.22** Logical schema for Bearcat Incorporated: Foreign key design using inclusion dependencies

### 6.7.2 An Information-Preserving Mapping

The steps involved in the information-preserving mapping are the same as those discussed in Section 6.7.1 except that the grammar used for expressing the logical schema is different. Constraints that define a relation schema limit expression of several conceptual modeling constructs in the logical tier, such as alternate keys, participation constraints, optional/mandatory attributes, composite attributes, derived attributes, and deletion rules. In this book, we use the term **logical scheme** instead of “relation schema” for an entity type transformed to the logical tier in order to relax these constraints. A set of logical schemes becomes a logical schema. The logical schema thus developed can certainly be implemented in a relational database environment. The systematic steps to map a Design-Specific ERD to a logical schema are as follows:

Lx: SCHEME	Att1 (t,s)	Q[Att2] (t,s)	Att3 <sup>*</sup> (t,s)	Att4 (t,s)	[Att5] (t,s)	Att6 (t,s)	Att7. (t,s)	FKAtt1 (t,s)	FKAtt2 (t,s)	... (t,s)	AttZ (t,s)
min Ly max min Lz max											
<b>or</b>											
Lx: SCHEME	(Att1, Q[Att2], Att3 <sup>*</sup> , Att4, [Att5, Att6], Att7, ... FKAtt1, FKAtt2, ..., AttZ) (t,s) (t,s)	min Ly max min Lz max									
min D max min D max											

**Step 1:** Specify a logical scheme for each base and weak entity type in the ERD following the grammar described here:

Where x = (1, 2, 3, 4, ..., N)

- SCHEME is the name of the entity type being mapped. (Use all capital letters.)
- Lx is a label for the SCHEME. (Or it could be an abbreviated short name of the SCHEME.)
- N is the number of SCHEMES in the logical schema.
- Att1, ..., AttZ are the names of the atomic attributes from the entity type. (Capitalize first letter only.)<sup>28</sup>
- The primary key is underlined—the constituent attributes *need not* be recorded successively.
- Attributes specified as mandatory in the ERD (•) are marked by an • following the attribute.
- The data type (t) and size (s) of an attribute mapped from the ERD are recorded immediately below the attribute.
- Composite (molecular) attributes are enclosed by square brackets [ ..., ...]; for this reason, the constituent atomic attributes are recorded next to each other.
- Alternate keys are enclosed in square brackets and marked by a Q (meaning unique) preceding the alternate key attribute(s).
- Derived attributes not stored in the database are denoted by a dotted underline (— — —).

<sup>28</sup>The *Universal Relation Schema (URS)* assumption dictates that every attribute name must be unique because attributes have a global meaning in a database schema. Therefore, if an attribute name appears in several relation schemas, all of these denote the same meaning—that is, the attributes are semantically join-compatible. In an ER model, however, the same attribute name is allowed to appear in different entity types since they imply different roles for the attribute name. We have adopted the URS assumption for the logical schema presented here. Thus, mapping of attributes from an ER model to a logical schema requires careful attention in order to ensure unique attribute names in the logical schema. Note that a referencing foreign key and corresponding referenced primary (or alternate) key having the same attribute name in a logical schema do not violate the URS assumption.

**Step 2:** Map each relationship type in the ERD using either the foreign key design, cross-referencing design, or mutual-referencing design, as discussed in Section 6.7.1.2:

- Each relationship type (diamond in the ERD) is accounted for by adding a foreign key (say, *FKAtt1*) attribute or attributes to the child scheme in the PCR. Obviously, the foreign key should share the same domain with the primary key of the parent scheme in the relationship type (the highlighting of the foreign key with italics is a convention just intended to draw attention). *While the reference implied in general is to the primary key of the parent scheme, if the design intention is to refer, in some cases, to an alternate key of the parent scheme, it is accomplished by coding the foreign key name to exactly match the name of the alternate key that is the target of the reference. This does not violate the universal relation schema assumption (see Footnote 28).*
- The label (e.g., Ly, Lz) of the parent relation is coded on top of the foreign key box that depicts the relationship. The explicit reference connoted by the foreign key is supplemented via this notation for ease of locating the referenced scheme. Alternatively, the label can be prefixed to the foreign key attribute's name.

**Step 3:** Incorporate the structural constraints of the relationships (i.e., cardinality ratio and participation constraints) using (min, max), as described here:

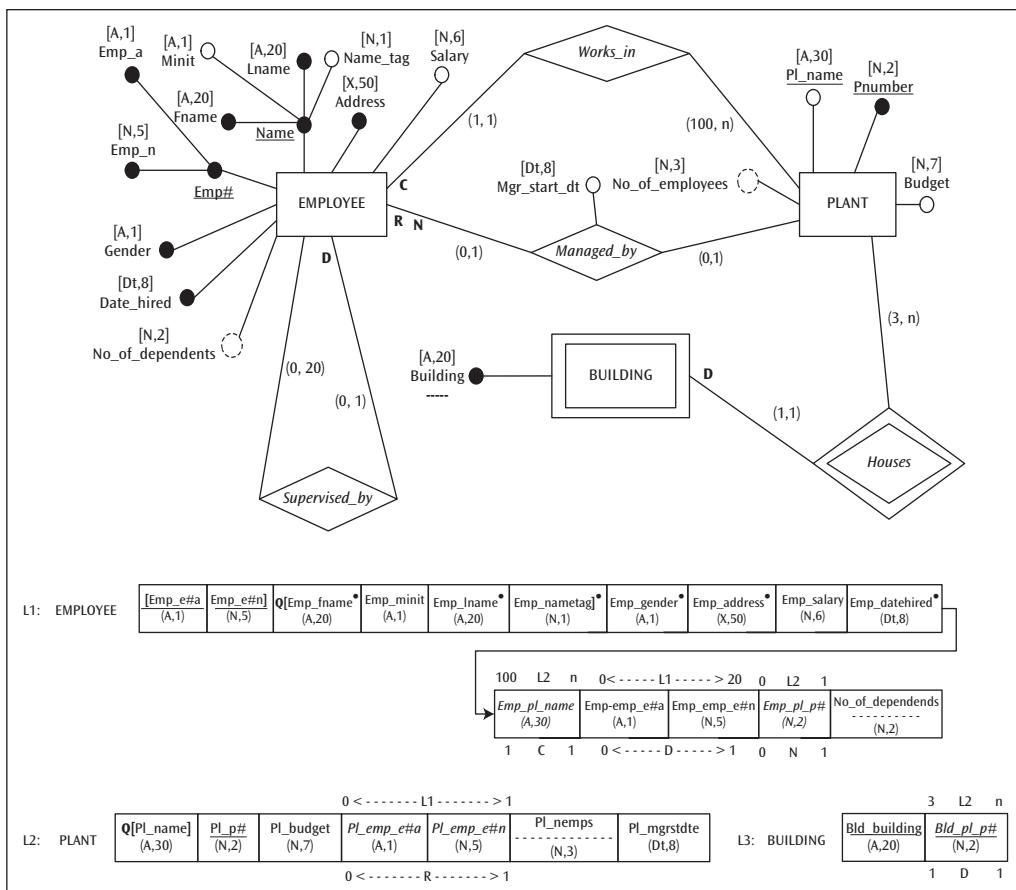
- The structural constraints of the relationship (i.e., cardinality ratio and participation constraint) are expressed using the (min, max) notation as follows: *The (min, max) expression coded on the parent edge of the PCR in the ERD is shown on the top of the foreign key, and the (min, max) expression coded on the child edge of the PCR is shown on the bottom of the foreign key.*
- min: participation constraint ( $\text{min} = 0$ ) indicates partial participation; ( $\text{min} \geq 0$ ) indicates total participation.
- max: cardinality ratio ( $\text{max} \geq \text{min}$ ).

**Step 4:** Indicate the deletion rule parameter (C, N, D, or R).

- A deletion constraint parameter (C, N, D, or R) between min and max recorded below the foreign key specifies the deletion rules—that is, action to be taken when a tuple from the parent scheme in the relationship type (PCR) is deleted. Four options are possible: C = Cascade; N = Set null; D = Set default value provided; R = Restrict.

Despite its significant capacity to preserve design information, the grammar presented here is not capable of preserving all metadata (i.e., expressed in the ERD and semantic integrity constraints) declaratively. For instance, the domain constraints on the attributes often listed as semantic integrity constraints are not captured by this grammar, and so will have to be carried forward via a list of semantic integrity constraints prepared at the logical tier. Also, the specific names and the roles of the relationship types are not preserved in the mapping process even though the relationship types themselves are fully captured.

Following the grammar just discussed, an example for an information-preserving mapping of a Design-Specific ERD to a logical schema is explicated in Figure 6.23. The source schema (at the top of the figure) is a combination of the ERDs shown in Figures 6.2 and 6.11 along with a recursive relationship type, *Supervised\_by*. First, the two base entity types EMPLOYEE and PLANT and the weak entity type BUILDING are converted to logical schemes with corresponding names; the three logical schemes are labeled L1, L2, and L3, respectively, and are shown at the bottom of Figure 6.23.



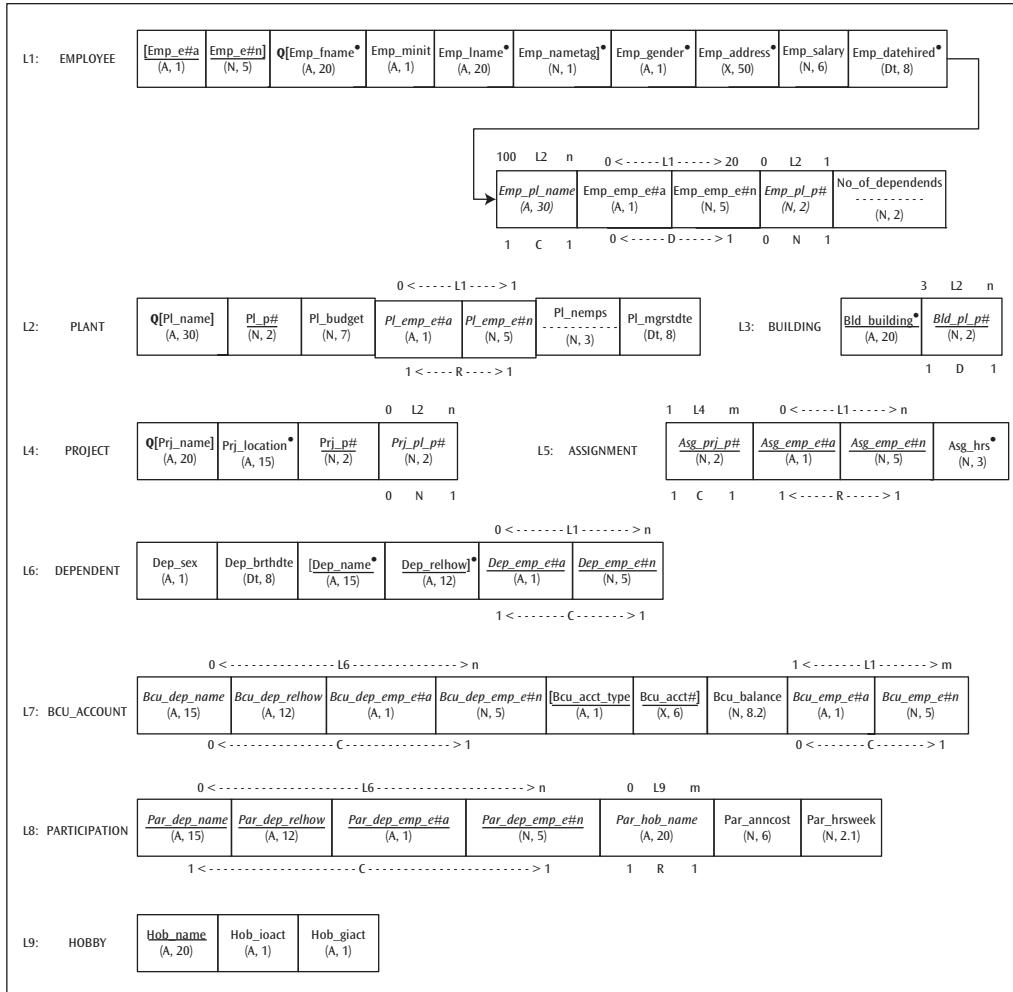
**FIGURE 6.23** A Design-Specific ERD and its associated information-preserving logical schema

The attribute type and size for each attribute are shown right below the attribute name. The mandatory property of an attribute is marked by • above the attribute. The primary key for each logical scheme is denoted by the underline. Observe that when a primary key is a composite attribute, then all atomic attributes constituting the primary key are underlined (for example, **[Bld\_building, Bld\_pl\_p#]**).<sup>29</sup> An alternate key (or keys) of a logical scheme is identified by the letter Q preceding the bracketed attribute(s); for example **Q[Pl\_name]**. **Q[Emp\_fname, Emp\_minit, Emp\_lname, Emp\_nametag]** means that this composite attribute is unique—not the constituent atomic attributes. Foreign keys are highlighted by italics (**Emp\_pl\_name** in EMPLOYEE, **Bld\_pl\_p#** in BUILDING). The derived attribute of PLANT (**Pl\_nemps**) is also mapped to the logical tier, the dotted underline indicating the derived nature of the attribute. The label of the parent in a PCR and the (min, max) on the parent edge of the relationship type are stated on top of the foreign key representing the relationship type. For example, **100 L2 n** above **Emp\_pl\_name** in EMPLOYEE signifies that L2 (i.e., PLANT) is the parent in the PCR *Works\_in*, and *Works\_in* is represented in the logical schema by the foreign key **Emp\_pl\_name** in EMPLOYEE. The (min, max) comes from the edge connecting PLANT to *Works\_in*. Likewise, the (min, max) on the edge connecting the child in the PCR (i.e., EMPLOYEE) to *Works\_in* as well as the deletion rule for *Works\_in* are stated right below the foreign key representing that relationship type (i.e., **1 C 1** below **Emp\_pl\_name** in EMPLOYEE). Notice that the attribute pair **[Emp\_emp\_e#a, Emp\_emp\_e#n]** in EMPLOYEE references L1, which is EMPLOYEE itself, thus capturing the recursive relationship type *Supervised\_by*. In the same fashion, *Managed\_by* and *Houses* are captured by **PLANT.[Pl\_emp\_e#a, Pl\_emp\_e#n]** and **BUILDING.Bld\_pl\_p#** respectively. In essence, the logical schema in Figure 6.23 preserves all the design information portrayed in the ERD above it.

The information-preserving logical schema for the Design-Specific ERD of Bearcat Incorporated shown in Figure 6.20 is given in Figure 6.24. In addition to incorporating all metadata conveyed by the ERD in the logical schema, item 4 of the semantic integrity constraints for the Design-Specific ER model (Table 6.6) is also implicitly captured in the logical schema. The rest of the semantic integrity constraints in Table 6.7 along with the logical schema (Figure 6.24) complete the logical data model, which then becomes fully information-preserving.

---

<sup>29</sup>It is not mandatory that atomic attributes comprising the primary key of a logical scheme be listed contiguously, although it is a good practice to do so.



**FIGURE 6.24** Information-preserving logical schema for the Design-Specific ERD for Bearcat Incorporated in Figure 6.20

## 6.8 MAPPING ENHANCED ER MODEL CONSTRUCTS TO A LOGICAL SCHEMA

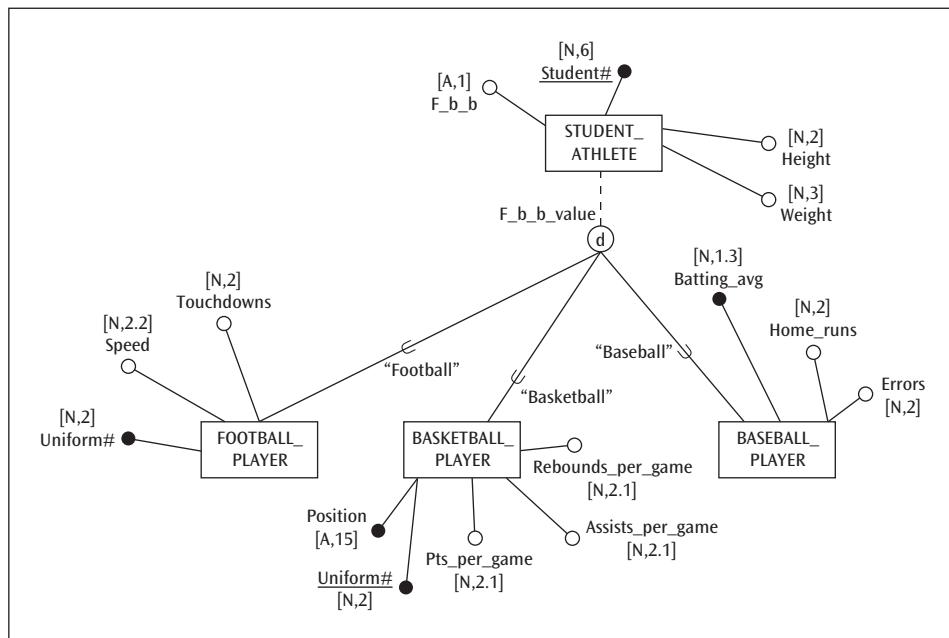
Chapter 4 discussed four different constructs of Superclass/subclass (SC/sc) relationships: specialization/generalization hierarchy, specialization/generalization lattice, categorization, and aggregation. In all these constructs, the cardinality ratio of an SC/sc relationship is **1:1**. In addition, the participation of a subclass in the relationship is always total. Thus, the mapping of these constructs to the logical tier can follow the same strategy as Case 1 and Case 3, discussed in Section 6.7.1.2.3, depending on the type of participation of the superclass (partial or total) in the relationship.

## 6.8.1 Information-Reducing Mapping of EER Constructs

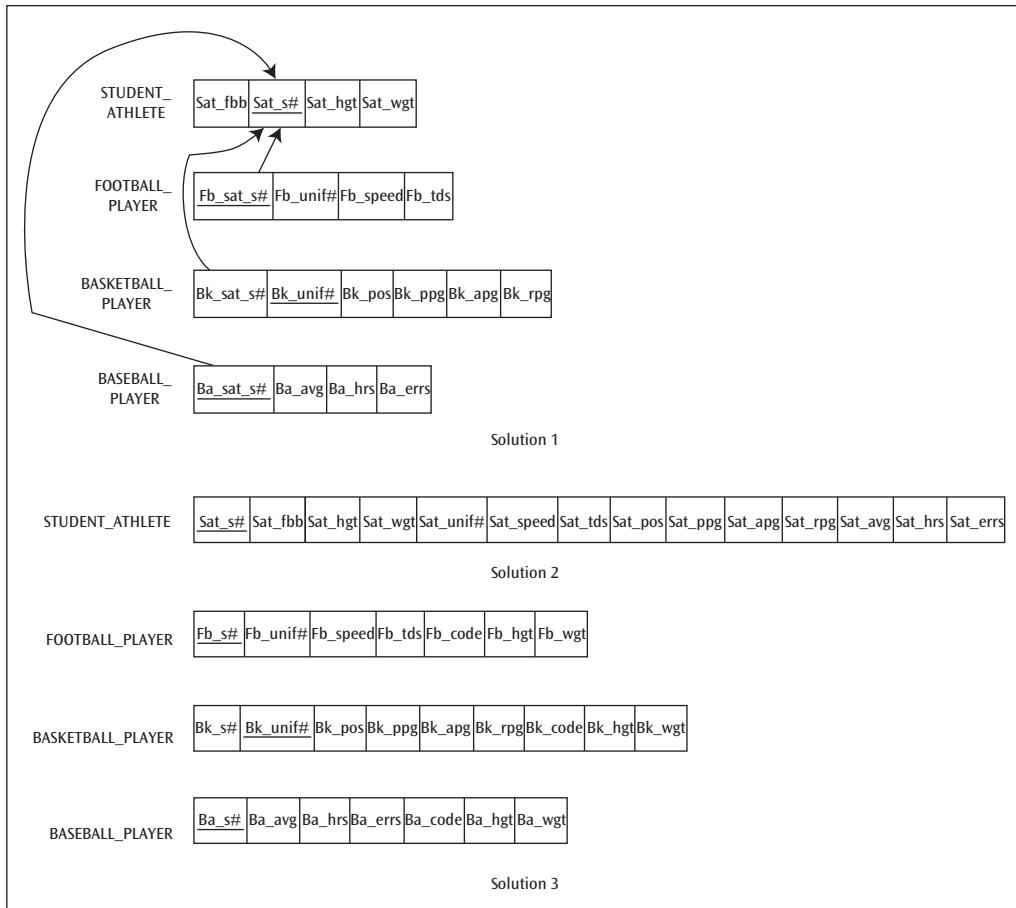
This section presents the foreign key design using directed arcs to depict the SC/sc relationships of the various EER modeling constructs. EER diagrams from earlier chapters are used to illustrate the mapping process.

### 6.8.1.1 Mapping a Specialization

Figure 6.25 is a variation of the ERD that appears in Figure 4.6 depicting a simple specialization. Three different logical schema solutions are possible (Figure 6.26). The most general form of solution that supports all four combinations of disjointness and completeness constraints (disjoint/partial, disjoint/total, overlapping/partial, overlapping/total) is shown in Figure 6.26, Solution 1. Here the inheritance property of the specialization is implicitly defined. The second solution merges all subclasses into the superclass yielding just one relation schema, shown in Figure 6.26, Solution 2. Thus, what are otherwise subclasses are inherent in the single schema. Figure 6.26, Solution 3, implements the inheritance property of a specialization explicitly.



**FIGURE 6.25** An example of a specialization



**FIGURE 6.26** Three possible logical schemas for the specialization in Figure 6.25

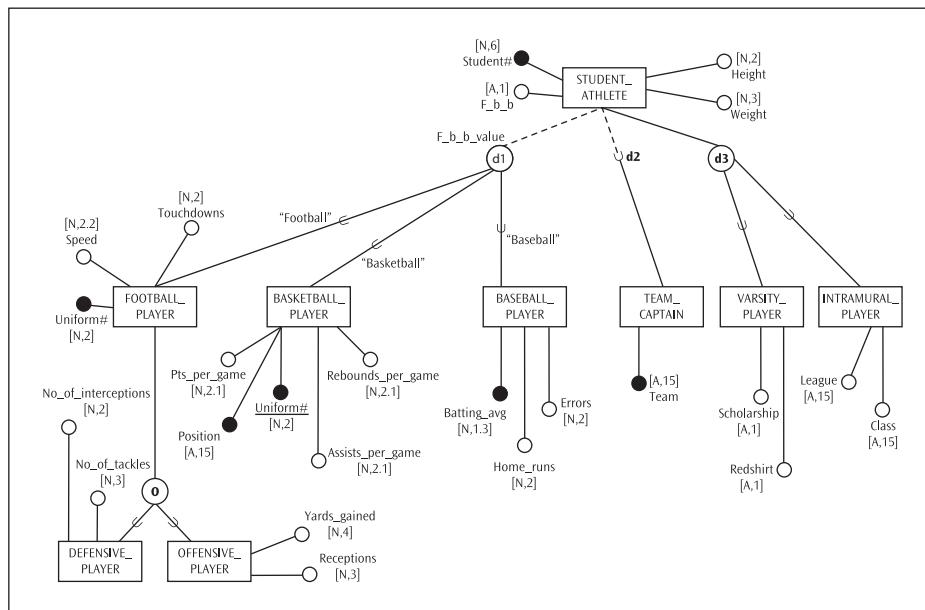
The first solution yields one logical scheme each for the subclasses and the superclass in the specialization. Observe that the inheritance property of the specialization yields a candidate key for every subclass (sc) in the specialization; since the cardinality ratio between a SC and sc is **1:1**, this candidate key in the sc also serves as the foreign key referencing the superclass (SC) in the specialization. Sometimes, when the number of attributes in the subclasses is very small, the efficiency of this design, with all its referential integrity constraints, becomes questionable. The single-schema design (Solution 2) also supports all four combinations of disjointness and completeness constraints by essentially eliminating the need for these constraints. The absence of referential integrity constraints and a single-schema design certainly enhance operational efficiency. Unless the specialization is overlapping (as opposed to disjoint), the database will have null values for several attributes in each tuple. If subclasses contain lots of attributes, this may be an

issue. In addition, any specific relationship types in which one or more of the subclasses independently or collectively participate cannot always be optimally implemented. The third solution will have to be rejected if the completeness constraint is partial, as in the example under illustration. The information that there are student-athletes who are neither football players, nor basketball players, nor baseball players will be lost in this design. Assuming that the completeness constraint is total, this solution can be an optimal middle ground among the three, especially when the number of attributes in the superclass is minimal. However, if the specialization is overlapping, some data redundancy is to be expected. The alternative foreign key design using inclusion dependencies in place of the directed arcs is left as an exercise for the reader.

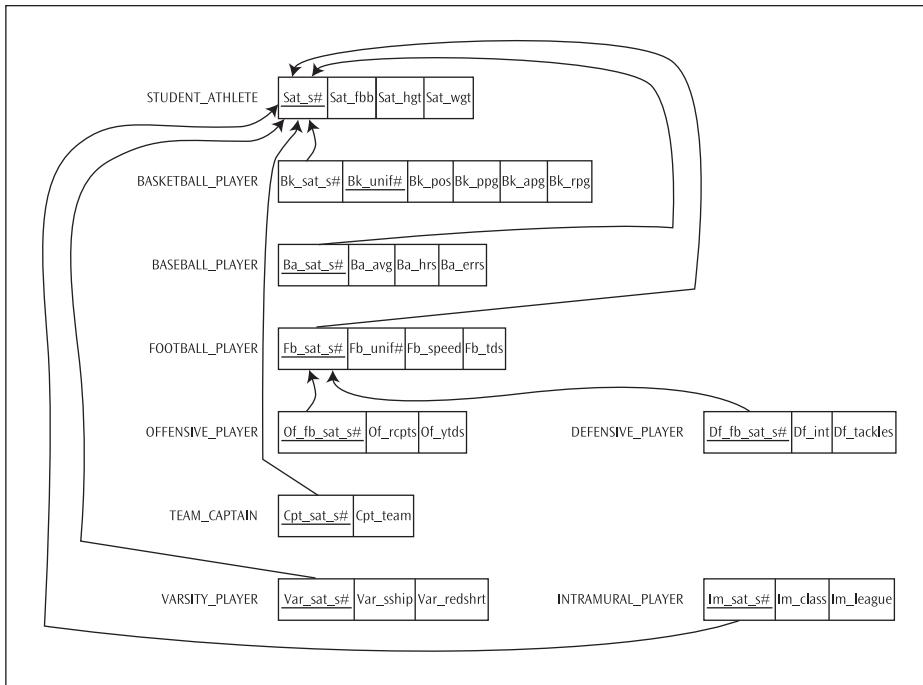
The remainder of the discussion in this section demonstrates only the foreign key design using the directed arc notation shown in Figure 6.26, Solution 1.

### 6.8.1.2 Mapping a Specialization Hierarchy

The ER model and the corresponding logical schema (foreign key design using directed arcs) displayed in Figures 6.27 and 6.28 represent a specialization hierarchy. Notice that STUDENT\_ATHLETE has been specialized in three different ways. In all three cases, it is a disjoint specialization and all but one are partial specializations. The general solution in Figure 6.28 allows for a team to have more than one captain but for a student-athlete to serve as captain of at most one team. Once again, the reader may translate the solution to depict the directed arcs by inclusion dependencies.



**FIGURE 6.27** An example of a specialization hierarchy

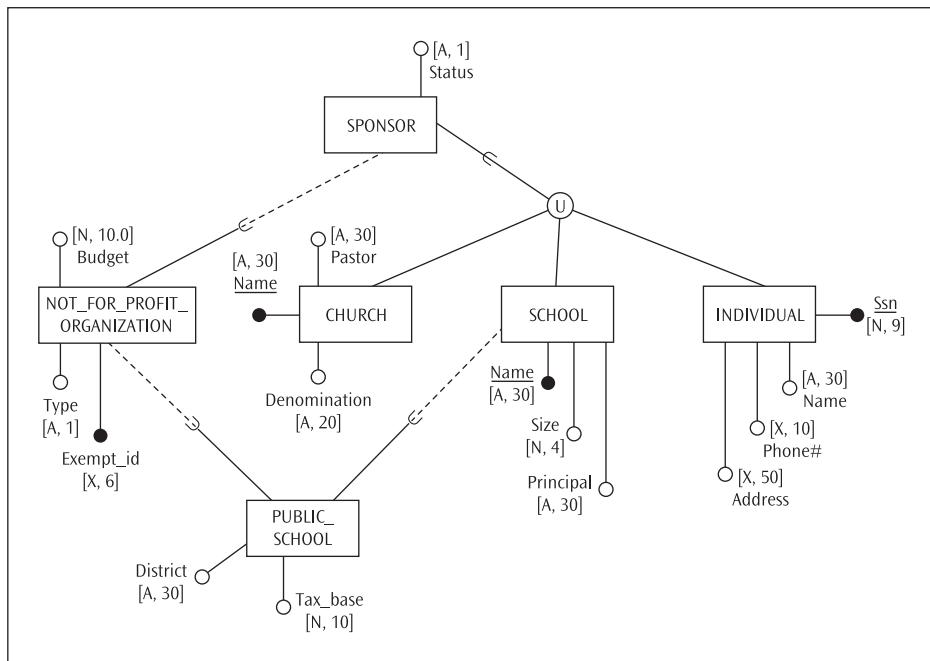


**FIGURE 6.28** Logical schema for the specialization hierarchy in Figure 6.27: Foreign key design using directed arcs

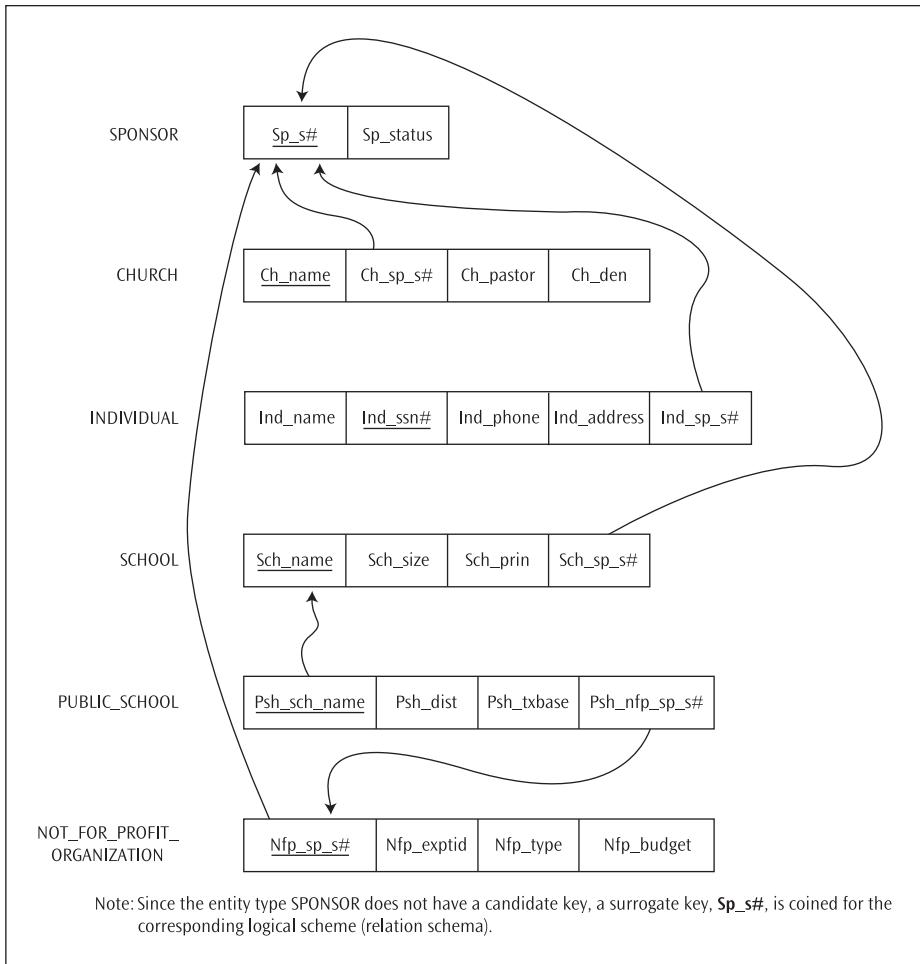
### 6.8.1.3 Mapping a Specialization Lattice and a Categorization

The example shown in Figure 6.29 is an excerpt from the Design-Specific ERD of Bearcat Incorporated (see Figure 4.22). The example includes: a total category SPONSOR as a union of CHURCH, SCHOOL, and INDIVIDUAL; a partial specialization of SPONSOR as NOT\_FOR\_PROFIT\_ORGANIZATION; and a specialization lattice in which PUBLIC\_SCHOOL is a shared subclass participating in two partial specializations, one with NOT\_FOR\_PROFIT\_ORGANIZATION as the superclass and the other with SCHOOL as the superclass. Before embarking on the logical schema mapping task, note that SPONSOR does not have a candidate key. Therefore, a surrogate key called, say, **Sp\_s#** (N,5) needs to be created. Further, while the same attribute name reflecting different roles in different entity types is permissible in the ERD, pursuant to the universal relation schema assumption, attribute names in the logical schema must be unique. Therefore, let us rename **CHURCH.Name** as **Ch\_name** and **SCHOOL.Name** as **Sch\_name** in the corresponding logical

schemes. Figure 6.30 portrays the logical schema for the ERD in Figure 6.29 using the foreign key design with directed arcs. It must be noted that in a categorization a candidate key of the subclass (i.e., SPONSOR) is added to each superclass (i.e., CHURCH, SCHOOL, and INDIVIDUAL) as the foreign key instead of the other way around as is done in a specialization.



**FIGURE 6.29** An example of a specialization lattice and a categorization

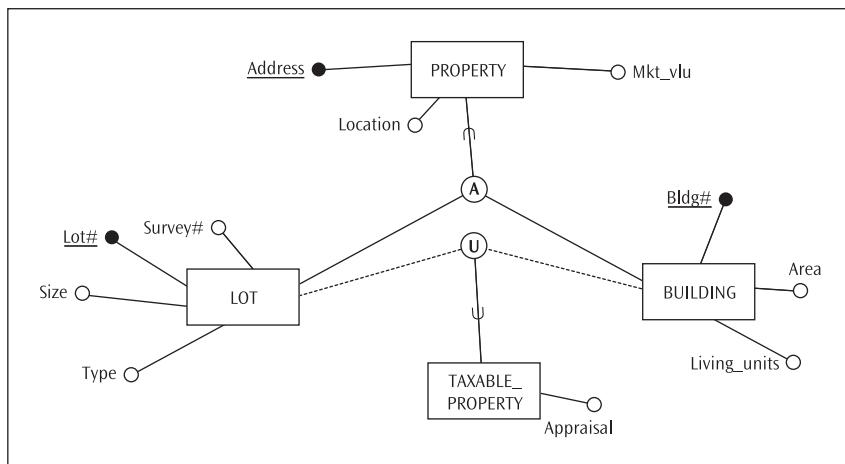


**FIGURE 6.30** Logical schema for the specialization lattice and categorization in Figure 6.29:  
Foreign key design using directed arcs

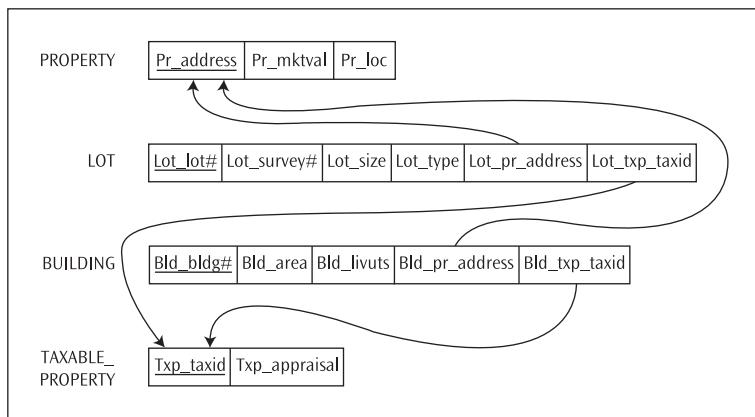
#### 6.8.1.4 Mapping an Aggregation

Aggregation along with a categorization was demonstrated in Figure 4.19. For the convenience of the reader, the ERD is reproduced in Figure 6.31. The basic difference between PROPERTY and TAXABLE\_PROPERTY is that a lot and a building together constituting a “property” is identified by a single address, while every individual lot and building that is a “taxable property” is identified by a unique **Txp\_taxid**. The logical schema reflecting the foreign key design with directed arcs is shown in Figure 6.32. Incidentally, **Txp\_taxid** is a surrogate key of TAXABLE\_PROPERTY that is “manufactured” since TAXABLE\_PROPERTY does not have a candidate key. Also, **Lot\_txp\_taxid** and **Bld\_txp\_taxid** are

alternate keys for LOT and BUILDING respectively. Likewise, **Lot\_pr\_address** and **Bld\_pr\_address** are also alternate keys for LOT and BUILDING, respectively. As in a categorization, a candidate key of an aggregate (subclass) is mapped as a foreign key in every superclass that participates in that aggregation. The reader may wish to develop the logical schema using inclusion dependencies as an exercise.



**FIGURE 6.31** A category and an aggregate contrasted



**FIGURE 6.32** Logical schema for the category and aggregate in Figure 6.31:  
Foreign key design using directed arcs

### 6.8.1.5 Information Lost While Mapping EER Constructs to the Logical Tier

Once again, the techniques discussed in Section 6.8.1 are information-reducing. Two kinds of metadata information are lost: user-specified business rules and design features.

In addition to all but the first two information loss items listed in Section 6.7.1.3, the following information is lost:

- The type of relationship (e.g., specialization/generalization, categorization, aggregation) is not carried forward to the logical schema.
- SC/sc (i.e., intra-entity class) relationships become indistinguishable from the regular (inter-entity class) relationships.
- The disjointness constraint of a specialization/generalization is lost during the conversion process.
- Multiple specializations of the same superclass are not captured.
- Specialization lattices are not discernible.
- The number of subclasses participating in a specialization and the number of superclasses participating in a categorization and/or aggregation are lost in the mapping.
- The completeness constraint of an SC/sc relationship is not present in the logical schema.

Are these losses of information trivial? If so, why are they collected in the conceptual data model to begin with? The argument that the ERD and the list of semantic integrity constraints can be used to supplement the logical schema during physical design defeats the purpose of even attempting to develop a logical schema. Furthermore, there is nothing wrong in attempting to develop a self-sufficient logical data model. *Simplicity of design as the rationale for an underdeveloped logical schema is not a worthy compromise.* The next section describes an extension to the information-preserving logical modeling grammar presented in Section 6.7.2, as applied to EER modeling.

### 6.8.2 Information-Preserving Grammar for Enhanced ER Modeling Constructs

Once again, the steps involved in the mapping process for EER constructs are the same as in Section 6.7.2 except that the grammar used for expressing the logical schema is different and is shown here:

	<b>min L<sub>s</sub> #</b>
L <sub>u</sub> : SCHEME	<u>FKAtt1</u> (t, s)   Att1 (t, s)   Att2 (t, s)   ...   AttZ (t, s)
	<b>min D Jx</b>
Or	
	<b>min L<sub>s</sub> #</b>
L <sub>u</sub> : SCHEME	( <u>FKAtt1</u> , Att1, Att2, ..., AttZ) (t, s) (t, s) (t, s) (t, s)
	<b>min D Jx</b>

Where u = (1, 2, ...N)

- min: completeness constraint  
(min = 0) indicates partial completeness; (min = 1) indicates total completeness.

**Note:** Since the cardinality ratio in an SC/sc relationship is always (1:1) the max part of (min,max) is always a 1. Therefore, the max is inherently preserved.

- $L_S$  is the label of the parent scheme coded on top of the foreign key that depicts the SC/sc relationship.
- # denotes the number of sc's in the specialization/generalization or the number of SCs in the categorization, specialization lattice, and aggregation.
- $Jx$  is the SC/sc label where:  
 $J$  is the disjointness constraint value: d = disjoint; o = overlapping; u = union; a = aggregate; L = specialization lattice; - = none.  
\*  $x$  is the marker for the specialization/categorization/aggregation/specialization lattice (e.g., 1, 2, 3, ... )
- The D between min and  $Jx$  under the foreign key specifies action to be taken when a tuple from the parent entity in a relationship is deleted. C = Cascade; N = Set null; D = Set default value provided; R = Restrict.

While the grammar notation is almost the same as that of the one employed for the ER constructs, two syntactical markers are specific to intra-entity class (SC/sc) relationships: **Jx** and **#**. **J** in the **Jx** reflects the value of the disjointness constraint in the cases of specialization/generalization (**o** for overlapping and **d** for disjoint), the union property (**u**) in the case of categorization, and the aggregate property (**a**) in the case of aggregation. The **x** in the **Jx** marks the specific specialization type occurrence in which the scheme participates. The same is true for categorization and aggregation as well. The **#** marker above the foreign key is an integer that indicates the number of subclasses in the specialization flagged by **Jx**. In the cases of categorization, aggregation, and specialization lattice, this will be the number of superclasses in the particular SC/sc construct.

The general form of the specialization lattice is shown here:

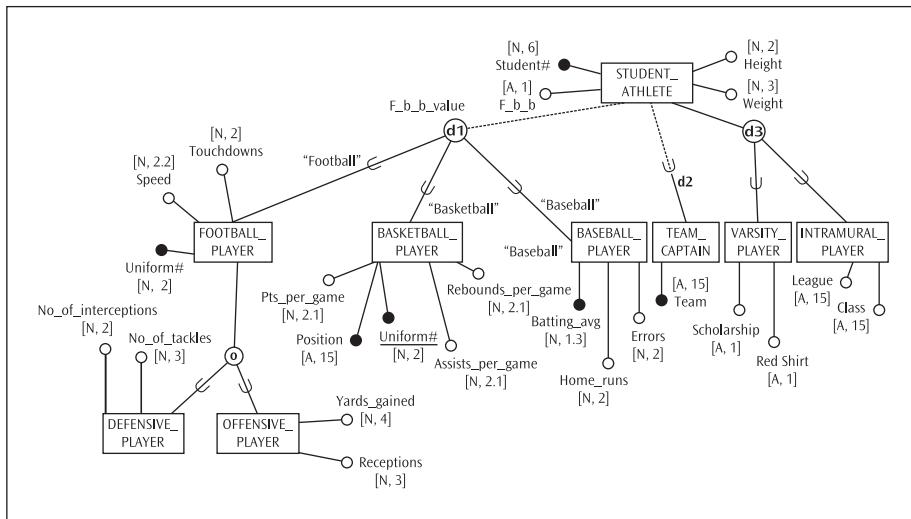
	<b>min <math>L_{S1}</math> # min <math>L_{S2}</math> # min <math>L_{S3}</math> #</b>												
$L_L$ : SCHEME	<table border="1"> <tbody> <tr> <td><u>FKAtt1</u> (t, s)</td><td>FKAtt2 (t, s)</td><td>FKAtt3 (t, s)</td><td>Att1 (t, s)</td><td>Att2 (t, s)</td><td>...</td><td>AttZ (t, s)</td></tr> </tbody> </table>						<u>FKAtt1</u> (t, s)	FKAtt2 (t, s)	FKAtt3 (t, s)	Att1 (t, s)	Att2 (t, s)	...	AttZ (t, s)
<u>FKAtt1</u> (t, s)	FKAtt2 (t, s)	FKAtt3 (t, s)	Att1 (t, s)	Att2 (t, s)	...	AttZ (t, s)							
	<b>min D Jx min D Jx min D Jx</b>												
Or													
$L_L$ : SCHEME	<table border="1"> <tbody> <tr> <td><u>(FKAtt1, ..., FKAtt2, ..., FKAtt3, ..., Att1, Att2, ..., AttZ)</u> (t, s) (t, s) (t, s) (t, s) (t, s) (t, s)</td><td colspan="5"></td></tr> </tbody> </table>						<u>(FKAtt1, ..., FKAtt2, ..., FKAtt3, ..., Att1, Att2, ..., AttZ)</u> (t, s) (t, s) (t, s) (t, s) (t, s) (t, s)						
<u>(FKAtt1, ..., FKAtt2, ..., FKAtt3, ..., Att1, Att2, ..., AttZ)</u> (t, s) (t, s) (t, s) (t, s) (t, s) (t, s)													
	<b>min D Jx min D Jx min D Jx</b>												

Note:  $L_L$  is the shared subclass in the lattice and  $L_{S1}$ ,  $L_{S2}$ , ... point to parents in the SC/sc relationships.

Note that each specialization in which the shared subclass is a member is captured by an independent foreign key.  $L_L$  is the shared subclass in the lattice, and  $L_{S1}$ ,  $L_{S2}$ , ... point to parents in the SC/sc relationships.

The following examples illustrate the extension to the information-preserving grammar prescribed earlier for mapping EER model constructs to the logical schema. To begin with, let us revisit the specialization hierarchy depicted in Figure 6.27, which is

reproduced in Figure 6.33. As pointed out in Section 6.8.1.5, both the foreign key design using directed arcs approach for logical model mapping (Figure 6.28) and its inclusion dependency alternative are information-reducing.



**FIGURE 6.33** An example of a specialization hierarchy

Following the information-preserving grammar rules described in this section produces the logical schema shown in Figure 6.34. The primary key, data type and size, optional properties of attributes (mandatory or optional value), alternate key, and foreign key are mapped following the information-preserving grammar described in Section 6.7.2. As for the structural constraints of the specialization constructs, the **min** component of the (min, max)—that is, the participation constraint—is captured as is, and the **max** component is *not* mapped because the two max values depicting the cardinality ratio are always **1** in a specialization/generalization relationship. Instead, in the *max slot* on top of the foreign key (#) is utilized to record the number of subclasses in the specialization and the *max slot* below the foreign key (**Jx**) is used to show the specialization label.

For example, the value **3** for the # in FOOTBALL\_PLAYER, BASKETBALL\_PLAYER, and BASEBALL\_PLAYER each denotes that there are three subclasses in the specialization **d1** (**Jx** value in all three logical schemes) of STUDENT\_ATHLETE marked by **L1**. The value **d1** for **Jx** in all three indicates that the three logical schemes are subclasses of the *same* specialization of STUDENT\_ATHLETE marked **d1**. Specialization **d3**, on the other hand, has only two subclasses. This is reflected by the value for the # in both VARSITY\_PLAYER and INTRAMURAL\_PLAYER. The specialization label **d2** may appear superfluous at first, because with just one subclass in this specialization the disjoint property itself has no meaning. Nonetheless, the label is needed to record the fact that the relationship type between TEAM\_CAPTAIN and STUDENT\_ATHLETE is indeed a specialization/generalization. Thus,

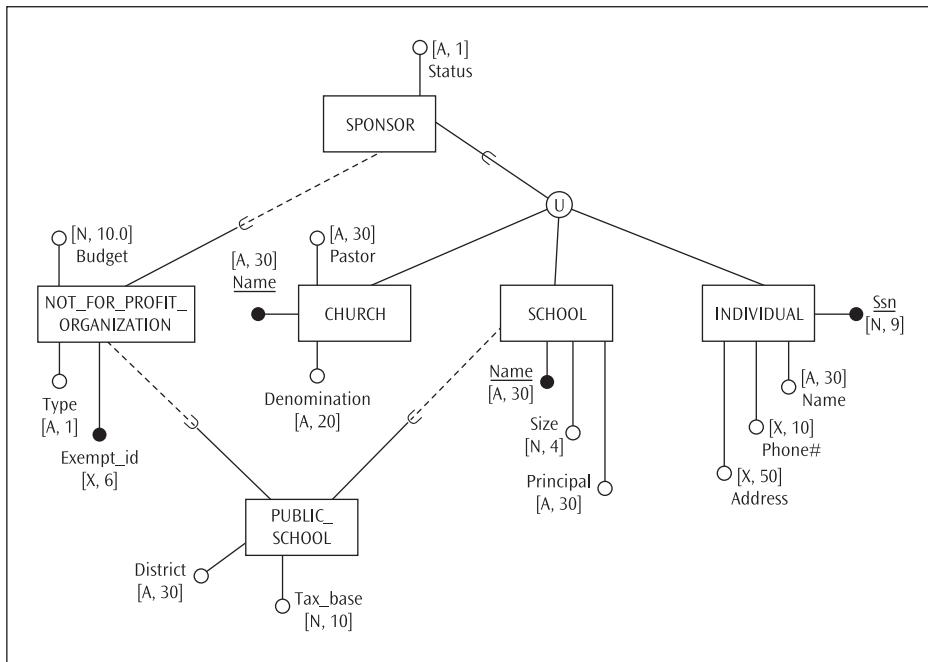
the fact that STUDENT\_ATHLETE participates as the superclass (SC) in three distinct specializations, and the related metadata are fully captured in the logical schema with no loss of information in the transformation process. The overlapping specialization of the superclass FOOTBALL\_PLAYER as {OFFENSIVE\_PLAYER, DEFENSIVE\_PLAYER} is coded by the value **o** for **Jx** (instead of **o1** or **o2**) because there is only one specialization of FOOTBALL\_PLAYER.

L1: STUDENT_ATHLETE	<u>Sat_s#</u> (N, 6)	Sat_fbb (A, 1)	Sat_hgt (N, 2)	Sat_wgt (N, 3)			
L2: FOOTBALL_PLAYER	0 L1 3	<u>Fb_sat_s#</u> (N, 6)	Fb_unif#*	Fb_speed (N, 2.2)	Fb_tds (N, 2)		
L3: OFFENSIVE_PLAYER	1 d1	<u>Of_fb_sat_s#</u> (N, 6)	Of_rcpts (N, 3)	Of_yds (N, 4)		L4: DEFENSIVE_PLAYER	1 L2 2
L5: BASKETBALL_PLAYER	1 L2 2	<u>Bk_sat_s#</u> (N, 6)	Bk_unif# (N, 2)	Bk_pos* (A, 15)	Bk_ppg (N, 2.1)	Df_fb_sat_s# (N, 6)	Df_int (N, 2)
L6: BASEBALL_PLAYER	1 0	<u>Ba_sat_s#</u> (N, 6)	Ba_avg* (N, 1.3)	Ba_hrs (N, 2)	Ba_errs (N, 2)	Df_tackles (N, 3)	
L7: TEAM_CAPTAIN	0 L1 3	<u>Cpt_sat_s#</u> (N, 6)	Cpt_team (A, 15)				
L8: VARSITY_PLAYER	1 d1	<u>Var_sat_s#</u> (N, 6)	Var_sship (A, 1)	Var_redshrt (A, 1)		L9: INTRAMURAL_PLAYER	1 L1 2
	0 L1 1					<u>Im_sat_s#</u> (N, 6)	Im_class (A, 15)
	1 d2						Im_league (A, 15)
	1 L1 2					1 d3	

**FIGURE 6.34** Information-preserving logical schema for the specialization hierarchy in Figure 6.33

Let us turn our attention back to Figure 6.29, in which a categorization and a specialization lattice are portrayed. In fact, while the relational schema in Figure 6.30 indicates the presence of these relationships, the fact that these are SC/sc relationships and represent a categorization and a specialization lattice is completely lost in Figure 6.30. It is true that the logical schema in Figure 6.30 is correct in that the physical implementation of the design will work, but the mapping is not complete, thus the implemented database system will not be robust. Indeed, some of the metadata lost in the transformation may not matter in implementations in certain relational DBMSs. The lost information, however, is integrity constraints arising from the business rules of the application—that is, metadata of the data model. While declarative implementation of some of these constraints may not be possible, procedural implementation methods can be used to make up for it. In sum, during database design, losing metadata through the data modeling tiers cannot be casually accepted and should be strictly avoided.

In this spirit, let us examine an information-preserving logical schema for this ER model (reproduced as Figure 6.35) that appears in Figure 6.36. Notice once again that the logical mapping of a categorization construct is somewhat opposite to that of a specialization. Here, despite the fact that a subclass inherits attributes and relationship types from the superclasses (selective type inheritance in the case of a categorization; see Chapter 4), the primary key of the category (subclass) is carried to the superclasses as foreign keys in order to establish the relationships. This is so because if the primary key of the respective superclasses is mapped as a foreign key in the subclass as in a specialization, not only will the category include several foreign keys (one corresponding to each superclass in the categorization), all but one foreign key will have null values in each tuple of the category. This is obviously not a desirable modeling practice.



**FIGURE 6.35** An example of a specialization lattice and a categorization

L1: SPONSOR	$\frac{\text{Sp\_s\#}}{(N,5)}$	Sp_status (A,1)	
L2: CHURCH	$\frac{\text{Ch\_name}}{(A,30)}$	$\frac{\text{Ch\_sp\_s\#}}{(N,5)}$	$\frac{\text{Ch\_pastor}}{(A,30)}$
	1	L1	3
	1	u	
L3: INDIVIDUAL	$\frac{\text{Ind\_name}}{(A,30)}$	$\frac{\text{Ind\_ssn\#}}{(N,9)}$	$\frac{\text{Ind\_phone}}{(X,10)}$
		Ind_address (X,50)	$\frac{\text{Ind\_sp\_s\#}}{(N,5)}$
	1	u	
L4: SCHOOL	$\frac{\text{Sch\_name}}{(A,30)}$	Sch_size (N,4)	$\frac{\text{Sch\_prin}}{(A,30)}$
			$\frac{\text{Sch\_sp\_s\#}}{(N,5)}$
	1	u	
L5: PUBLIC_SCHOOL	$\frac{\text{Psh\_sch\_name}}{(A,30)}$	Psh_dist (A,30)	$\frac{\text{Psh\_txbase}}{(N,10)}$
	0	L4	1
	1	L	
	0	L6	1
L6: NOT_FOR_PROFIT_ORGANIZATION	$\frac{\text{Nfp\_sp\_s\#}}{(N,5)}$	Nfp_exptid (X,6)	$\frac{\text{Nfp\_type}}{(A,1)}$
	0	L1	1
	1	d	
Note: While the entity type SPONSOR need not have a candidate key, every logical scheme, by definition, must have a primary key. Therefore, the surrogate key, Sp_s# is "manufactured" for the relation scheme.			

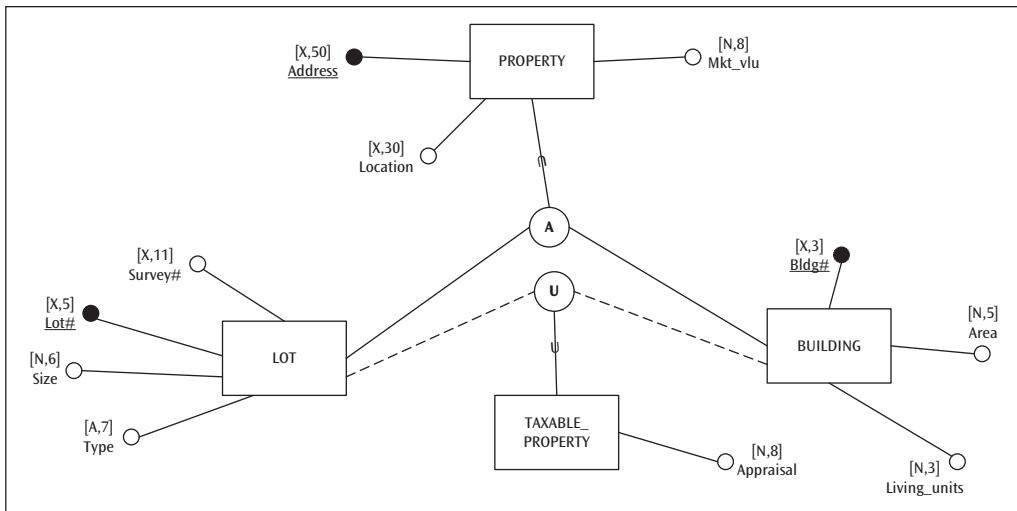
**FIGURE 6.36** Information-preserving logical schema for the specialization lattice and categorization in Figure 6.35

In Figure 6.35, observe that the category SPONSOR does not have a candidate key. While this is okay in the conceptual tier, absence of a candidate key in a logical scheme is, by definition, unacceptable. Therefore, a surrogate key, **Sp\_s#**, has been coined for the logical scheme of SPONSOR in Figure 6.36 and mapped as the foreign key in each of the three superclasses in the categorization: CHURCH, SCHOOL, and INDIVIDUAL. Also, conforming to the universal relation schema assumption, some of the attributes in the logical schema have been renamed so that the attribute names are globally unique in the logical schema—for example, **Ch\_name** in CHURCH and **Sch\_name** in SCHOOL. Note that these foreign keys also end up as candidate keys of the respective logical schemes due to the cardinality ratio of **1:1** implicitly present in the design.<sup>30</sup>

In the specialization lattice, however, PUBLIC\_SCHOOL is a shared subclass in two specializations. It inherits attributes and relationship types from both SCHOOL and NOT\_FOR\_PROFIT\_ORGANIZATION. Since this is a lattice of “specialization,” the shared subclass, PUBLIC\_SCHOOL, assumes the role of child in the two PCRs of specialization. Therefore, there are two foreign keys, **Psh\_sch\_name** and **Psh\_nfp\_sp\_s#**, in PUBLIC\_SCHOOL that capture the relationships with SCHOOL and NOT\_FOR\_PROFIT\_ORGANIZATION, respectively. Observe that the two foreign keys are also candidate keys of PUBLIC\_SCHOOL because of the **1:1** cardinality ratios inherent in specialization relationships. Since **Psh\_sch\_name** is chosen as the primary key, **Psh\_nfp\_sp\_s#** automatically becomes an alternate key of the logical schema.

How do you convert an aggregation construct to the logical tier? This was discussed earlier in this chapter and illustrated in Figures 6.31 and 6.32 (see Section 6.8.1.4). The information-preserving mapping of aggregation is similar to that of categorization except that the aggregate is denoted by an **a** for the **Jx** value instead of a **u**. Recall that the **#** above the foreign key indicates the value of the number of superclasses participating in the aggregation. Therefore, when the cardinality ratio of an SC/sc relationship in an aggregation is not **1:1**, the associated metadata is not captured in the logical schema; instead, it is recorded in the list of semantic integrity constraints that accompanies the logical schema. Figure 6.37 shows the design-specific rendition of the ERD (a repeat of Figure 6.31, for the reader’s convenience). The information-preserving logical schema for the aggregation and categorization shown in Figures 6.31 and 6.37 appears in Figure 6.38.

<sup>30</sup>Selective type inheritance as a distinguishing property of categorization is inherent in the relationship construct. This property is not explicitly seen in the logical schema either. The constraint means that the same value of **Sponsor#** cannot occur in more than one of the relations: CHURCH, SCHOOL, and INDIVIDUAL. The subtle nature of this constraint deems it necessary to be explicitly specified in the list of semantic integrity constraints that accompanies the local schema.



**FIGURE 6.37** A Design-Specific ERD contrasting a category and an aggregate

L1:	PROPERTY	<u>Pr_address</u> (X,50)	Pr_mktval (N,8)	Pr_loc (X,30)				
L2:	LOT	<u>Lot_lot#</u> (X,5)	Lot_survey# (X,11)	Lot_size (N,6)	Lot_type (A,7)	<u>Lot_pr_address</u> (X,50)	$Q[Lot\_txp\_taxid]$ (X,7)	1 L1 2 1 L4 2
L3:	BUILDING	<u>Bld_bldg#</u> (X,3)	Bld_area (N,5)	Bld_livuts (N,3)	<u>Bld_pr_address</u> (X,50)	$Q[Bld\_txp\_taxid]$ (X,7)	1 L1 2 1 L4 2	1 a 1 u
L4:	TAXABLE_PROPERTY	<u>Txp_taxid</u> (X,7)	Txp_appraisal (N,8)					

Note: **Txp\_taxid** is a surrogate key of TAXABLE\_PROPERTY; the cardinality ratio of the aggregation will be included in the list of semantic integrity constraints.

**FIGURE 6.38** Information-preserving logical schema for the category and aggregate in Figure 6.37

In Chapter 4, the Bearcat Incorporated story was further enriched to incorporate elements that would readily translate to EER constructs. The associated Design-Specific EER model (the ERD and the semantic integrity constraints) is collectively depicted in Figure 4.22 and Table 4.5. The reader is encouraged to develop the corresponding logical

data model, first using the foreign key design with both the directed arcs and inclusion dependencies alternatives, and then using the information-preserving grammar. Such an exercise will not only reinforce understanding of the information-preserving grammar, it will also clarify the information-preserving nature of this new grammar. The logical schemas in Figures 6.21, 6.22, and 6.24 may be used as springboards to embark on this adventure.

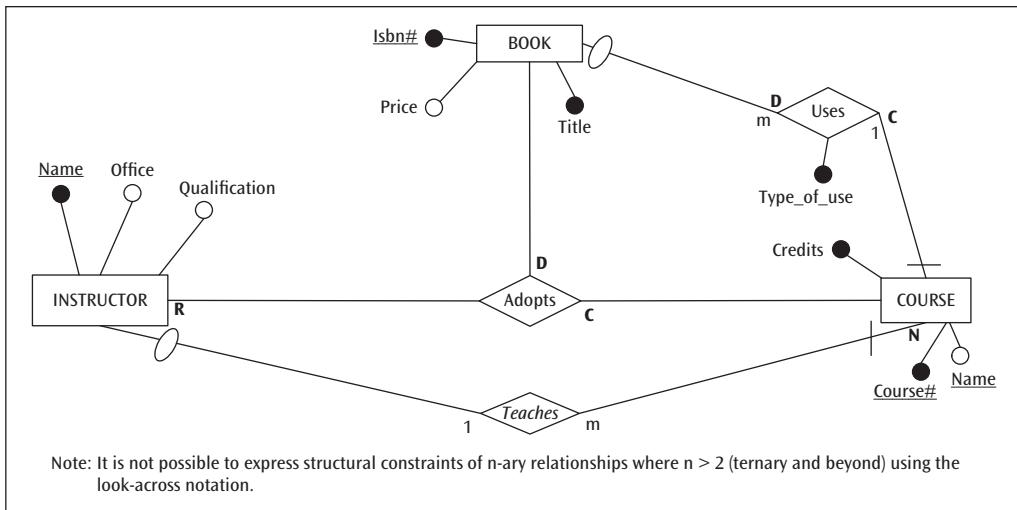
## 6.9 MAPPING COMPLEX ER MODEL CONSTRUCTS TO A LOGICAL SCHEMA

Chapter 5 is focused on complex conceptual modeling techniques using the ER modeling grammar. In fact, a few additional ER modeling constructs are introduced in Chapter 5. This section is dedicated to the discussion of mapping such complex ER models to the logical tier.

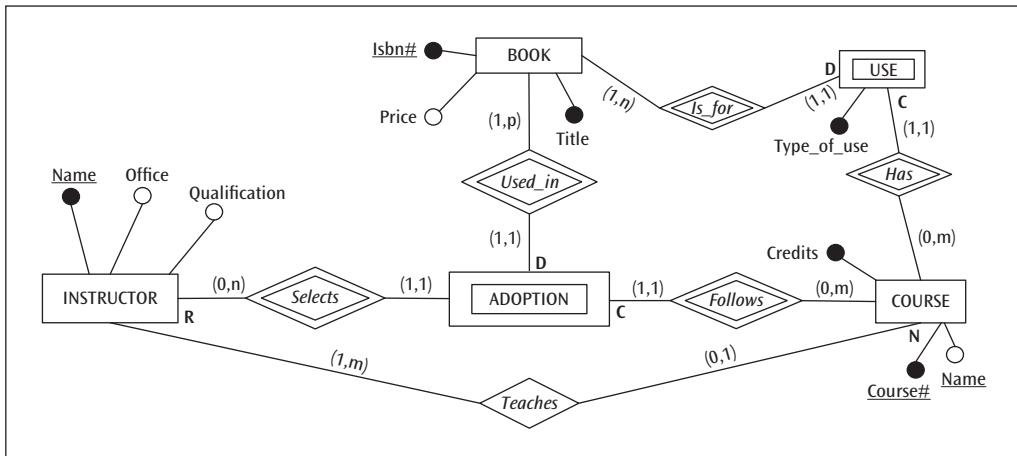
As we should know, the input to the logical mapping process is the final Design-Specific ER model. Some of the examples in this section will also refresh your memory on the transformation of an ER model to the final design-specific state. Earlier in this chapter, you learned an information-reducing as well as an information-preserving logical modeling specification and method. An information-reducing ER model does not imply permanent loss of information; it simply means that the information lost in the mapping of the ER model to a logical schema will have to be added to the list of semantic integrity constraints, rendering the effort expended in the conceptual modeling process a waste. The fact remains that all specifications contained in the ER model will have to reach the physical database design stage intact, one way or another.

The examples in this section present both the information-reducing and the information-preserving logical schema. The reader at this point should be able to observe the specifics of the lost information in the information-reducing logical schema. However, an information-reducing logical schema is closely compatible with the relational data model architecture. Preservation of information is accomplished in the information-preserving logical schema by willfully violating the relational modeling grammar rules. Nonetheless, an information-preserving logical schema can be implemented in the relational modeling architecture with ease.

Figure 6.39 depicts a progressive development of the mapping of an ERD to the logical tier in four steps. It must be noted that the look-across technique is not capable of expressing the specification of the structural constraints of a relationship type—namely, the cardinality constraint and the participation constraint—for relationship types beyond binary. This can be observed in the Presentation Layer ERD shown in Figure 6.39a. The ERD transformed to the design-specific tier appears in Figure 6.39b. Observe that the decomposition of a [m:n:p] ternary relationship type to a gerund entity type is no different from that of a [m:n] binary relationship type. The gerund entity type resulting from the decomposition of a ternary relationship type ends up with three identifying parents. Also, the “look-near” [min, max] notation employed in the Design-Specific ERD facilitates expression of the structural constraints of the ternary relationship type.

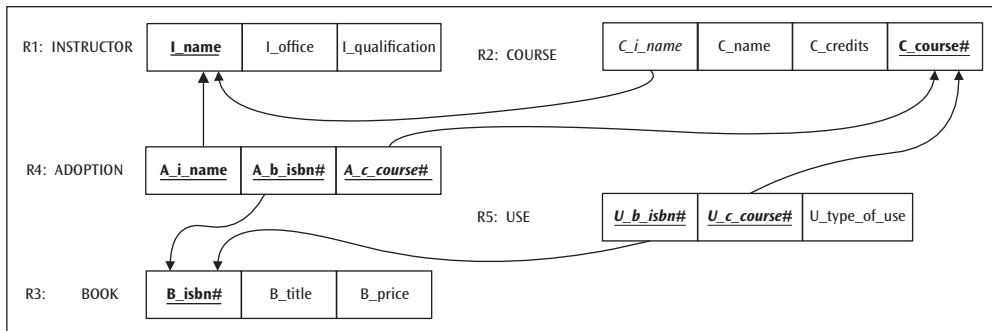
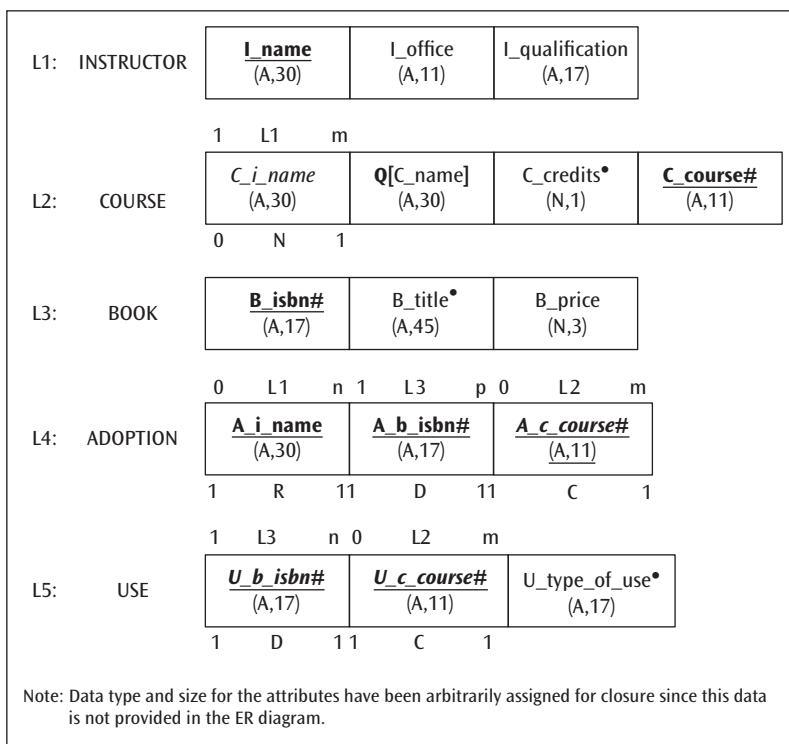


**FIGURE 6.39a** Presentation Layer ERD for a ternary relationship type *Adopts*

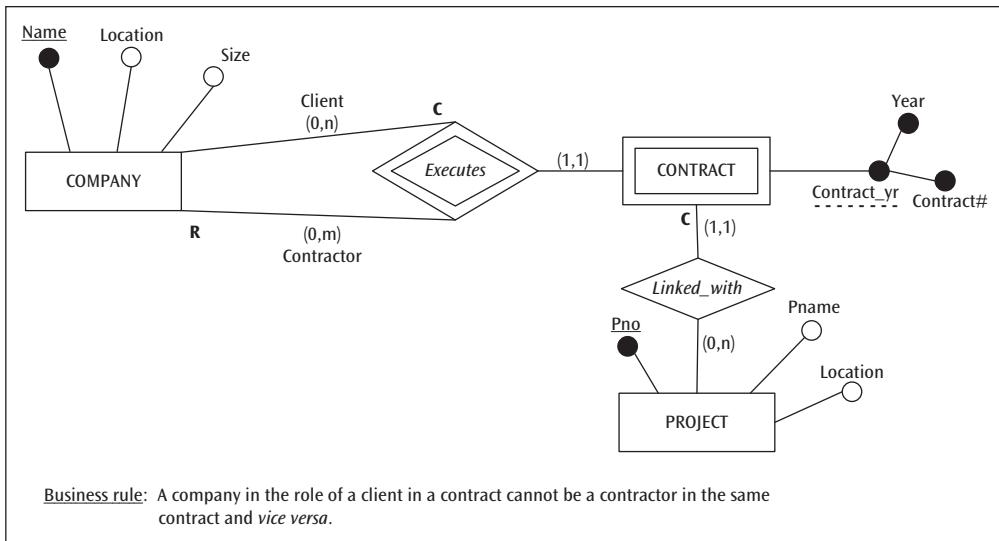


**FIGURE 6.39b** A Design-specific ERD for the Presentation Layer ERD shown in Figure 6.39a

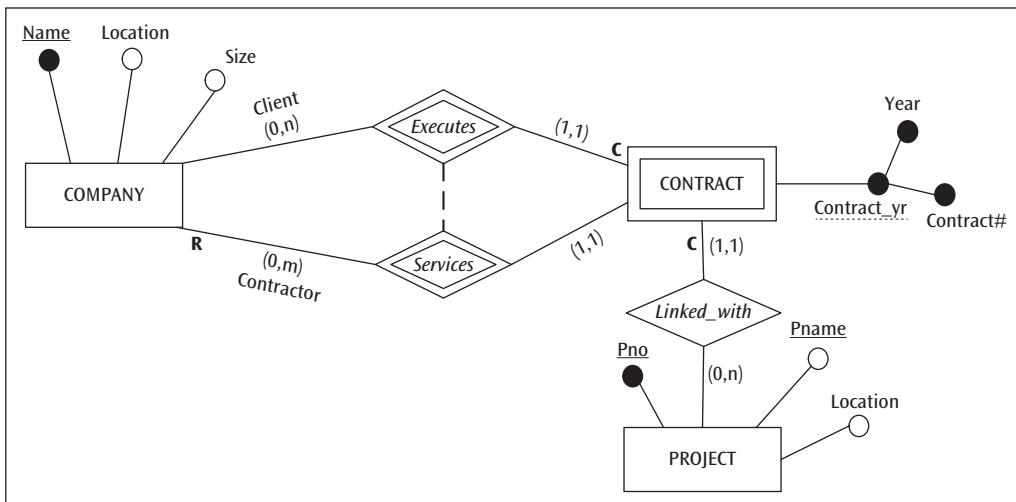
The mapping of a Design-Specific ERD to the logical tier has been discussed extensively earlier in this chapter. Thus, the information-reducing logical schema (Figure 6.39c) and the information-preserving logical schema (Figure 6.39d) require no additional clarification except to draw the attention of the reader to the logical scheme ADOPTION in both the figures.

**FIGURE 6.39c** Logical schema (information-reducing) for the Design-Specific ERD in Figure 6.39b**FIGURE 6.39d** Logical schema (information-preserving) for the Design-Specific ERD in Figure 6.39b

A weak entity type child in a recursive relationship type is modeled in Figure 6.40a. The decomposition of this relationship type appears in the final Design-Specific ERD in Figure 6.40b. Also, the business rule stated as a semantic integrity constraint in Figure 6.40a is incorporated as the inter-relationship constraint “exclusion dependency” in Figure 6.40b using weak relationships.

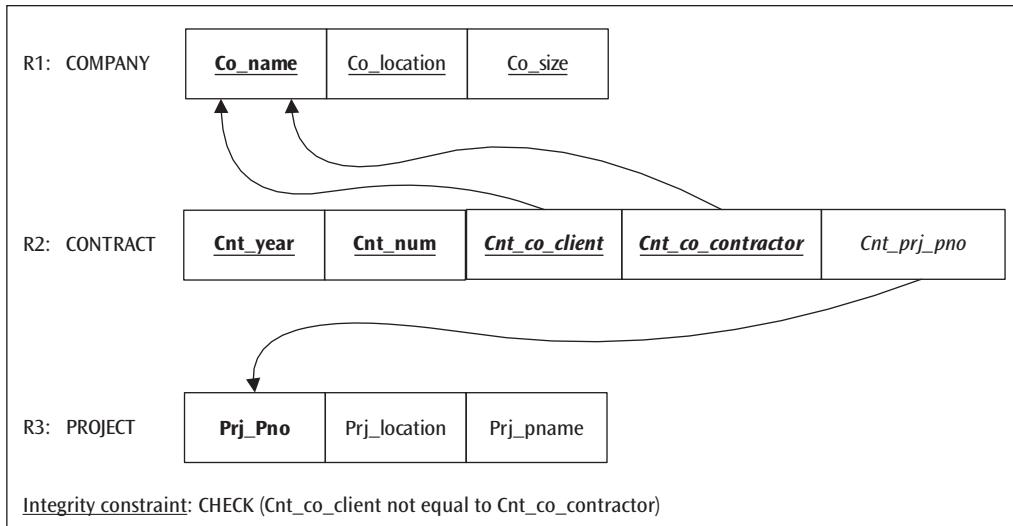
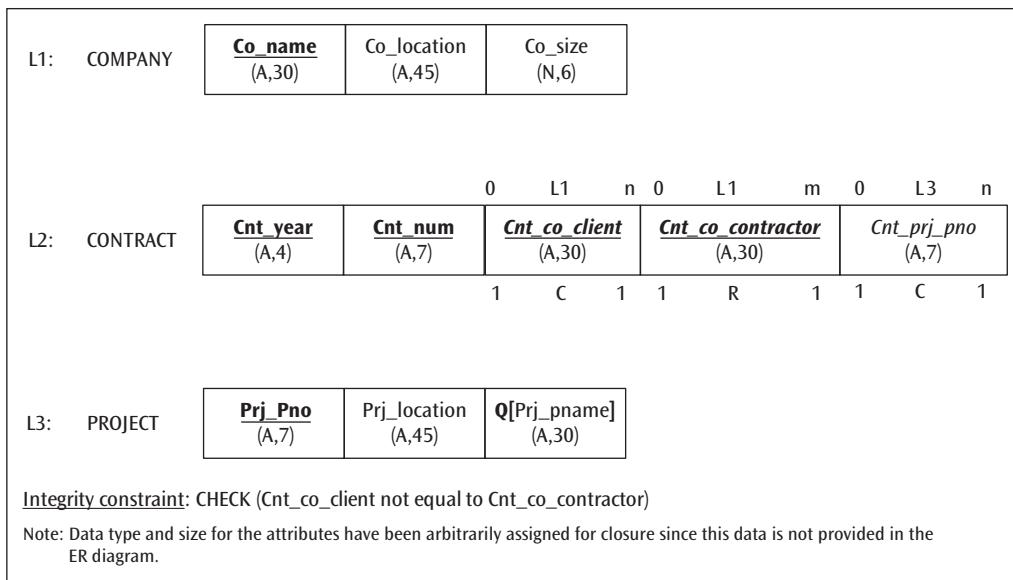


**FIGURE 6.40a** A copy of the ERD presented in Figure 5.45b with added deletion constraints



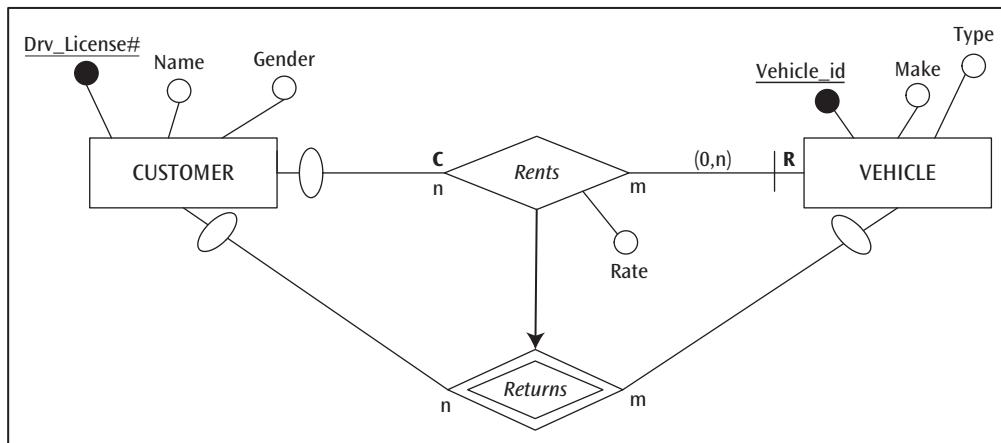
**FIGURE 6.40b** The m:n relationship with a weak entity child in Figure 6.40(a) decomposed:  
The final Design-Specific ERD

Observe that although the mapping to a logical schema is straightforward (see Figures 6.40c and 6.40d), the logical schema does not have a construct to capture the exclusion dependency. However, the stated integrity constraint expresses the business rule in precise technical terms so that it can be easily implemented in the physical database design.

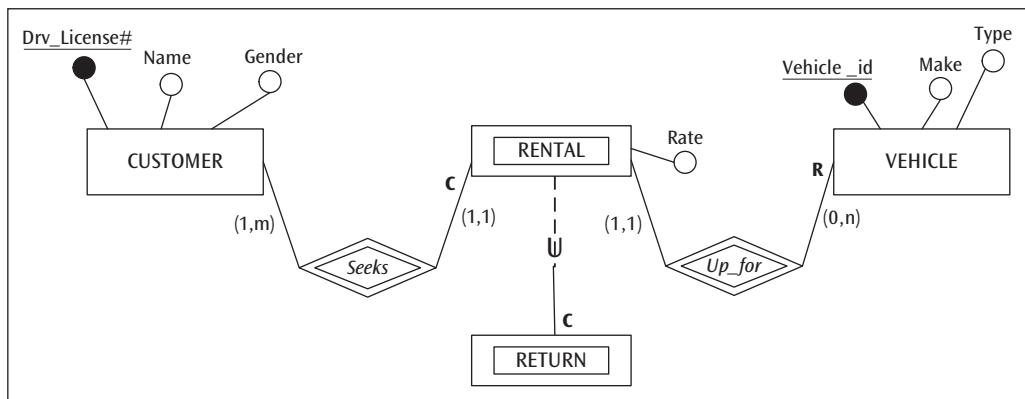
**FIGURE 6.40c** Logical schema (information-reducing) for the Design-Specific ERD in Figure 6.40b**FIGURE 6.40d** Logical schema (information-preserving) for the Design-Specific ERD in Figure 6.40b

The simplest case of the inter-relationship constraint “inclusion dependency” is expressed in Figure 6.41a using a weak relationship type. The Design-Specific ERD transformed from this Presentation Layer ERD is shown in Figure 6.41b. The point to

remember is that the standard decomposition procedure will establish CUSTOMER and VEHICLE as the parents of RETURN through the appropriate identification relationship type constructs. However, the partial specialization of RENTAL as RETURN renders these identifying relationships redundant; that is why they are absent in Figure 6.41b. Once this modeling insight is clear, the mapping to the logical tier becomes a simple mechanical process.

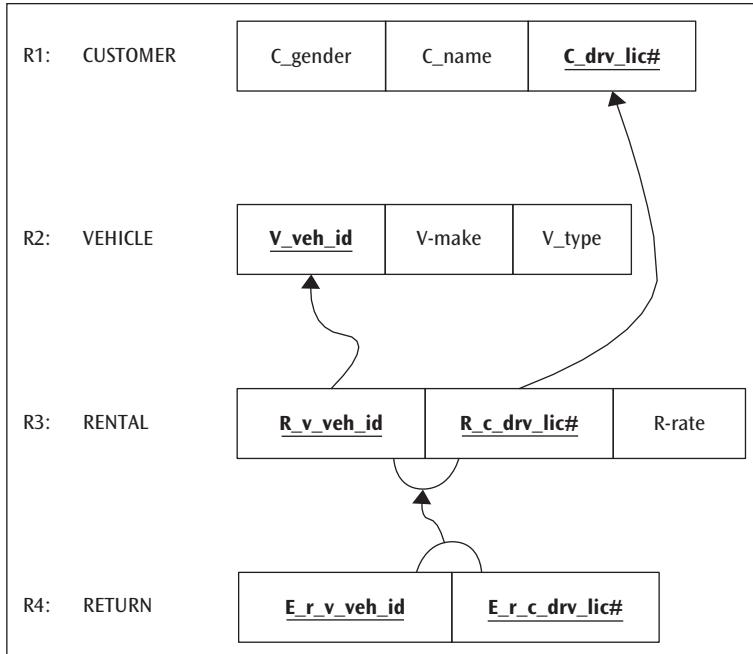


**FIGURE 6.41a** Presentation Layer ERD with a weak relationship expressing inclusion dependency

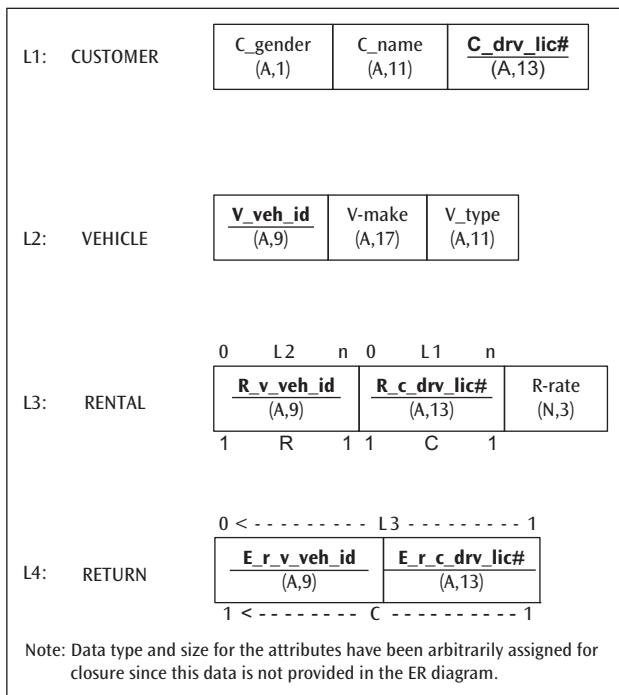


**FIGURE 6.41b** Design-specific transformation of the ERD in Figure 6.41a

Figures 6.41c and 6.41d display the information-reducing and the information-preserving logical schema respectively for the design-specific ERD portrayed in Figure 6.41b. Please note that the foreign key reference from RETURN to RENTAL implies reference based on the two attributes **Vehicle\_id** and **Drv\_license#** together—not individually. Individual foreign key references here will be an error.

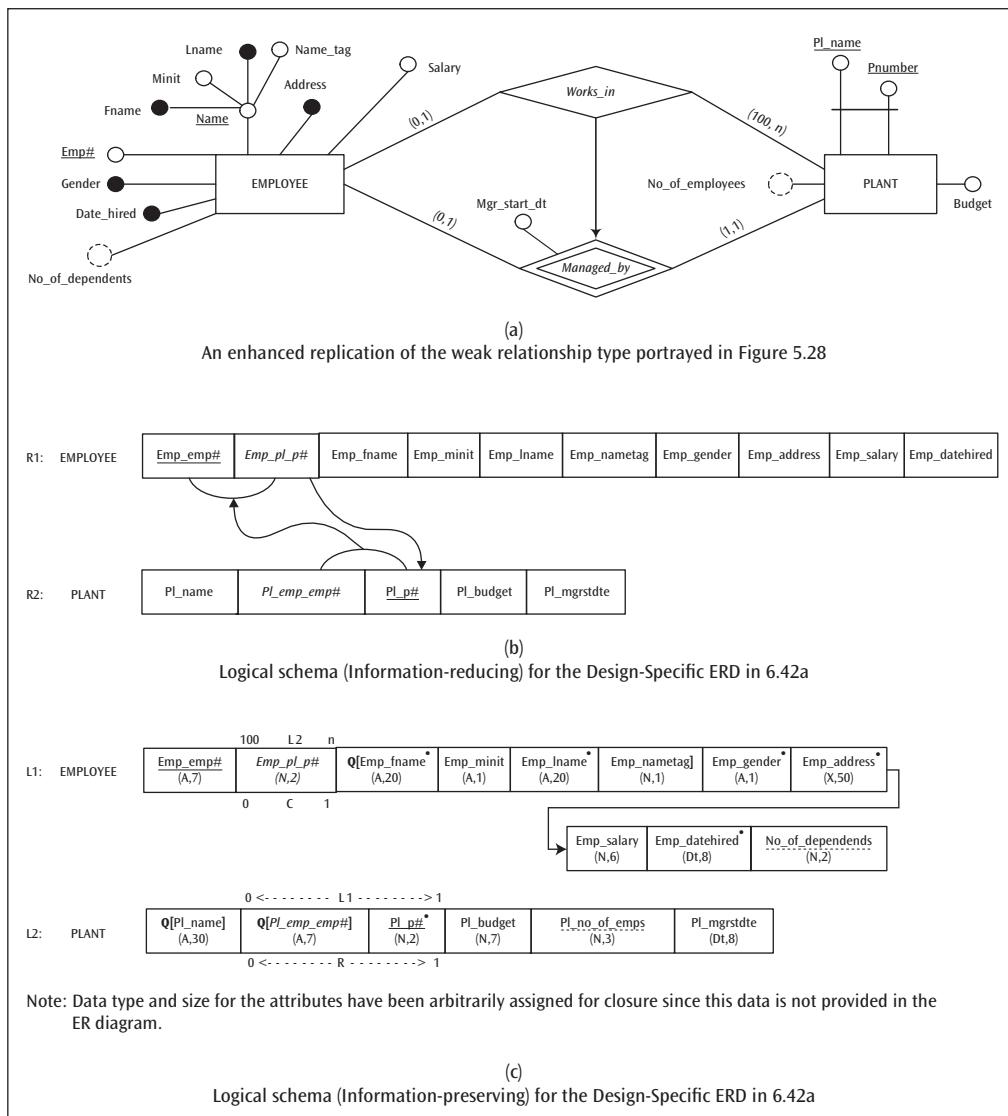


**FIGURE 6.41c** Logical schema (information-reducing) for the Design-Specific ERD in Figure 6.41b



**FIGURE 6.41d** Logical schema (information-preserving) for the Design-Specific ERD in Figure 6.41b

A second example for capturing an inter-relationship constraint through “inclusion dependency” (weak relationship type) is presented in Figure 6.42. The scenario here differs from that of the previous example (please see Figure 6.41a) in that the cardinality constraint for the *Works\_in* relationship type is (1,n) and the one for the *Managed\_by* relationship type is (1,1). Thus, no decomposition of these two relationships is necessary<sup>31</sup>.



**FIGURE 6.42a,b,c** Mapping a weak relationship type to the logical tier—A second example

<sup>31</sup>A strict relational schema would in this case require a cross-referencing design (for example, see Figure 6.15) because EMPLOYEE, the child entity type, sports a partial participation in the *Works\_in* relationship type and thus will permit null value for the foreign key, which is prohibited in a strict relational schema.

Here, mapping the *Works\_in* relationship type to the logical tier is rather straightforward, as reflected in Figures 6.42b and 6.42c. The intricate aspect of the design is the ability to capture the *Managed\_by* relationship type and the inclusion dependency of *Managed\_by* on *Works\_in* using a single foreign key reference, as depicted in Figures 6.42b and 6.42c. The reader is encouraged to study this mapping closely.

## Chapter Summary

---

The relational data model was first outlined in a paper published by E. F. Codd in 1970. The model uses mathematical relations as its foundation and is based on set theory. The simplicity of the concept and the sound theoretical basis are two reasons why the relational data model has become the model on which most commercial database systems are based. The relational data model represents a database as a collection of relations, where a relation resembles a two-dimensional table of values presented as rows and columns. A row in the table represents a set of related data values and is called a tuple. All values in a column are of the same data type. A column is formally referred to as an attribute. The set of all tuples in the table goes by the name “relation.” A relationship (association) between two relations in the relational data model takes the form of a referential integrity constraint.

An attribute or collection of attributes can serve as a unique identifier of a relation. A superkey is a set of one or more attributes, which, taken collectively, uniquely identifies a tuple. A second type of unique identifier is called a candidate key. A candidate key is defined as a superkey with no proper subsets that are superkeys. A candidate key has two properties: uniqueness and irreducibility. The uniqueness property is common to both a superkey and a candidate key, whereas the irreducibility property is present only in a candidate key. Every attribute plays only one of three roles in a relation schema: It is a candidate key, a key attribute, or a non-key attribute of the relation schema. Any attribute that is a constituent part (proper subset) of a candidate key of the relation schema is a key attribute. An attribute that is not a candidate key is a non-key attribute. A primary key serves the role of uniquely identifying tuples of a relation. In addition to possessing the uniqueness and irreducibility properties, a primary key is not allowed to have a missing (i.e., null) value (this property is known as the entity integrity constraint).

The relational data model includes a group of basic manipulation operations that involves relations. Collectively, these operations comprise what is known as relational algebra. Section 6.4 discussed six (selection, projection, union, minus, intersection, and natural join) of the eight basic relational algebra operations. A more in-depth treatment of relational algebra appears in Chapter 11. The relational data model also includes views and materialized views. A view is defined as a named “virtual” relation schema constructed from one or more relation schemas through the use of one or more relational algebra operations. In a database environment, views (a) allow the same data to be seen by different users in different ways at the same time, (b) provide security by restricting user access to predetermined data, and (c) hide complexity from the user. Unlike a view, a materialized view is real and contains its own separate data. Materialized views are used to freeze data at a certain point in time without preventing updates to continue on the data in the relation schemas on which they are based.

Sections 6.6 through 6.8 described ways to map conceptual schemas (both ER and EER models) to logical schemas. Approaches for mapping ER constructs to a logical schema begin by creating a relation schema for each base and weak entity type present in the Design-Specific ERD. Only the stored attributes of the entity type become attributes of the relation schema. In the case of composite attributes, only their constituent atomic components are recorded. For each relation schema based on a base entity type, the atomic attribute(s) serving as the primary key is (are) underlined. The primary key of a relation schema for a weak entity type includes the partial key of the weak entity type plus the primary key of each identifying parent of the weak entity type.

A Design-Specific ERD contains binary and recursive relationship types that exhibit a cardinality ratio of **1:n** and **1:1**. In cases where the cardinality ratio is **1:n**, the entity type on the *n-side*

of the relationship type is the child in the parent-child relationship and the child (or referencing relation schema) contains the foreign key attribute(s). This approach, where the foreign key in the child shares the same domain with a candidate key (most of the time, the primary key) of the parent, is known as the foreign key design. The foreign key design can be expressed diagrammatically via the use of directed arcs or by the specification of inclusion dependencies. An alternative to the foreign key design is the cross-referencing design (which can also be expressed using directed arcs or inclusion dependencies), which entails the creation of a relation schema to represent the relationship type. This approach can be used if the absence of null values in the foreign key is an important consideration.

In situations where the cardinality ratio of the relationship type is **1:1**, either one of the entity types can be the parent or child. Approaches for handling a **1:1** cardinality ratio depend on the nature of the participation constraint that characterizes the relationship. In cases where the participation constraint of only one of the entity types participating in the relationship type is total, the entity type with total participation in the relationship type assumes the role of the child in the parent-child relationship and the foreign key design is applied. When the participation constraints of both entity types in the relationship type are partial, a variety of approaches can be considered: (a) a foreign key design with the addition of a foreign key in either one of the relation schemas involved in the **1:1** relationship type, (b) mutual-referencing, in which the two relation schemas directly reference each other via foreign keys included in both, and (c) a cross-referencing design. A third case is when the participation constraints of both entity types in the relationship type are total. Situations of this type are handled by using mutual-referencing. Mutual-referencing must be accompanied by the imposition of several constraints, some of which can be established declaratively and some of which require procedural intervention. Merging the entity types involved in this or any other type of **1:1** relationship into a single relation schema is always a possibility but often not employed if the distinct nature of the entity types is lost or the entity types also participate independently in other relationship types.

The information-reducing nature of design approaches that make use of directed arcs and inclusion dependencies for mapping ER constructs (i.e., entity types and relationship types) to a logical schema is illustrated via their application to the mapping of the Design-Specific ER Model for Bearcat Incorporated to a logical schema. Information lost (i.e., ignored) in the transformation process includes: the nature of the cardinality ratios, the participation constraints of each relationship type, the optional/mandatory property of an attribute, the identification of alternate keys, the composite nature of certain atomic attributes, the existence of derived attributes, deletion rules, and attribute type and size. A logical modeling grammar capable of producing a logical schema that is information-preserving is described and then applied to the mapping of the Design-Specific ER Model for Bearcat Incorporated.

The next section started with a discussion of the application of the foreign key design approach using directed arcs to map the EER constructs of (a) the specialization/generalization hierarchy, (b) the specialization/generalization lattice, (c) categorization, and (d) aggregation. With the exception of aggregation, given the presence of only **1:1** cardinality ratios, the mapping of these constructs can make use of strategies that are similar to those used to map relationship types with **1:1** cardinality ratios, depending on the type of participation of the superclass (partial or total) in the relationship. Since aggregation permits the relaxation of the inherent property of the **1:1** cardinality ratio in an SC/sc relationship (an aggregate is a subclass that is a subset of the aggregation of the superclasses in the relationship), the logical mapping of an aggregation is

similar to that of the foreign key design in a **1:n** relationship type. As was the case when discussing the application of the foreign key design using directed arcs and inclusion dependencies in the context of ER constructs, use of these techniques in the context of EER constructs to develop a logical schema was shown as information-reducing and thus amenable to an extension of the information-preserving logical modeling grammar described previously. Accordingly, the information-preserving grammar for EER constructs was described at this point and application of this grammar was demonstrated using a few examples.

Section 6.9 explicated mapping techniques for some of the advanced ER modeling grammar constructs as well as a few complex ER models. Both information-reducing and information-preserving logical modeling grammars were used for the mapping process.

## Exercises

---

1. Define the terms “tuple,” “attribute,” and “relation.”
2. What is a relation schema? What is the difference between a relation, a relation schema, and a relational schema?
3. What is the difference between a derived attribute and a stored attribute in terms of their representation in a relation schema?
4. What is a null value? What gives rise to null values in a relation?
5. What is the difference between a subset and a proper subset?
6. What is a candidate key? How does a candidate key differ from a superkey?
7. What is a primary key? How do the properties of a primary key differ from those of a candidate key?
8. Identify the superkeys, candidate key(s), and the primary key for the following relation instance of the STU-CLASS relation schema.

Student Number	Student Name	Student Major	Class Name	Class Time
0110	KHUMAWALA	ACCOUNTING	BA482	MW3
0110	KHUMAWALA	ACCOUNTING	BD445	TR2
0110	KHUMAWALA	ACCOUNTING	BA491	TR3
1000	STEDRY	ANTHROPOLOGY	AP150	MWF9
1000	STEDRY	ANTHROPOLOGY	BD445	TR2
2000	KHUMAWALA	STATISTICS	BA491	TR3
2000	KHUMAWALA	STATISTICS	BD445	TR2
3000	GAMBLE	ACCOUNTING	BA482	MW3
3000	GAMBLE	ACCOUNTING	BP490	MW4

9. Define the term “referential integrity constraint.” Why is referential integrity important? How is the term “foreign key” used in the context of referential integrity?

10. Consider the following relations R1 and R2:

Relation R1		
R1.a	R1.b	R1.c
30	A	20
45	B	32
75	A	24

Relation R2		
R2.x	R2.y	R2.z
30	B	24
75	C	12
30	B	20

Show the relations created as a result of the following relational algebra operations:

- a. The union of R1 and R2
  - b. The difference of R1 and R2
  - c. The difference of R2 and R1
  - d. The intersection of R1 and R2
  - e. The natural join of R1 and R2. [Assume that R1.a and R2.x are the joining attributes.]
11. Consider the following relations of DRIVER, TICKET\_TYPE, and TICKET:

**DRIVER (Dr\_license\_no, Dr\_name, Dr\_city, Dr\_state)**

**TICKET\_TYPE (Ttp\_offense, Ttp\_fine)**

**TICKET (Tic\_ticket\_no, Tic\_ticket\_date, Tic\_dr\_license, Tic\_ttp\_offense)**

An instance of each of these relations appears here:

Dr_license_no	Dr_name	Dr_city	Dr_state
MVX 322	E. Mills	Waller	TX
RVX 287	R. Brooks	Bellaire	TX
TGY 832	L. Silva	Sugarland	TX
KEC 654	R. Lence	Houston	TX
MQA 823	E. Blair	Houston	TX
GRE 720	H. Newman	Pearland	TX

Ttp_offense	Ttp_fine
Parking	15
Red Light	50
Speeding	65
Failure To Stop	30

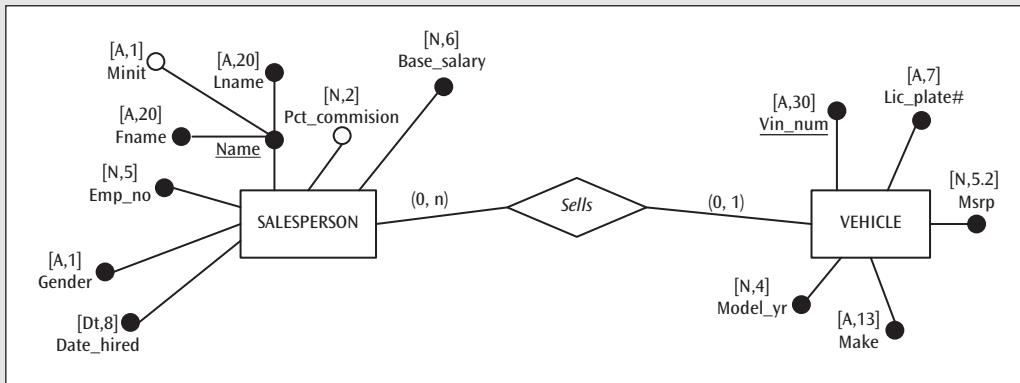
349

Tic_Ticket_no	Tic_ticket_date	Tic_dr_license_no	Tic_ttp_offense
1023	20-Dec-2007	MVX 322	Parking
1025	21-Dec-2007	RVX 287	Red Light
1397	03-Dec-2007	MVX 322	Parking
1027	22-Dec-2007	TGY 832	Parking
1225	22-Dec-2007	KEC 654	Speeding
1212	06-Dec-2007	MVX 322	Speeding
1024	21-Dec-2007	RVX 287	Speeding
1037	23-Dec-2007	MVX 322	Red Light
1051	23-Dec-2007	MVX 322	Failure To Stop

Use the data to answer the following questions. Also, list the relational algebra operation(s) required to obtain the answer.

- What are the names of all drivers?
  - What are the license numbers of all drivers who have been issued a ticket?
  - What are the license numbers of those drivers who have never been issued a ticket?  
*Hint:* Consider the use of the minus operator along with one other relational algebra operator.
  - What are the names of all drivers who have been issued a ticket?
12. What would cause a relational schema for a database to contain more relation schemas than there are entity types?
13. Discuss the concept of information preservation in data model mapping.
14. What is required to map a base entity type to a relation schema? Describe how this approach differs for a weak entity type.
15. What is required to map a relationship type that exhibits a **1:n** cardinality ratio?
16. What is the difference between the referencing relation schema and the referenced relation schema? How are these terms incorporated into the foreign key design?
17. What is the purpose of the cross-referencing design?
18. What complicates the mapping of **1:1** cardinality ratios?
19. Describe mutual-referencing and the complexities that it introduces.

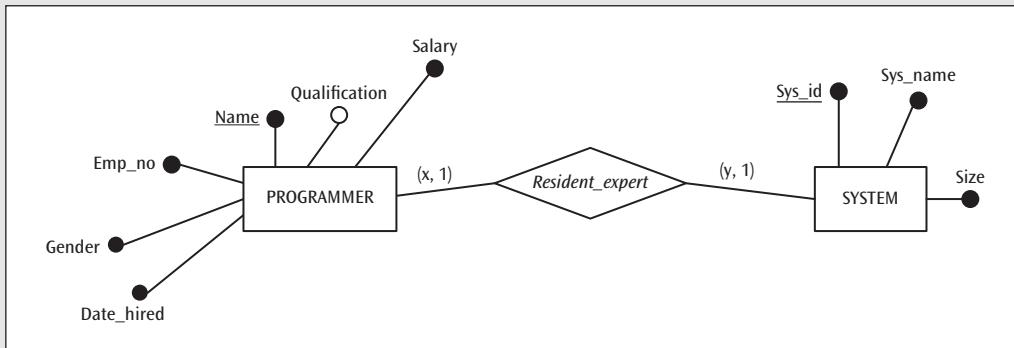
20. What information is lost by the use of the information-reducing grammar?
21. Describe how to map a specialization hierarchy, a specialization lattice, and a categorization.
22. What do you think are the ultimate consequences of failure to “preserve information” in the data model mapping process?
23. For the following excerpt from an ERD, specify the logical (relational) schema as per the foreign key design in the following ways:
  - using directed arcs
  - in terms of inclusion dependencies



24. List (tabulate) the metadata available in the ERD for Exercise 23 and indicate the ones captured in the logical schema of design 23(a) and design 23(b).
25. For the ERD for Exercise 23, specify the logical (relational) schema as per the cross-referencing design in the following ways:
  - using directed arcs
  - in terms of inclusion dependencies

Also, explain the merits and demerits of this cross-referencing design over the foreign key design solution.

26. Consider the following excerpt from an ERD:



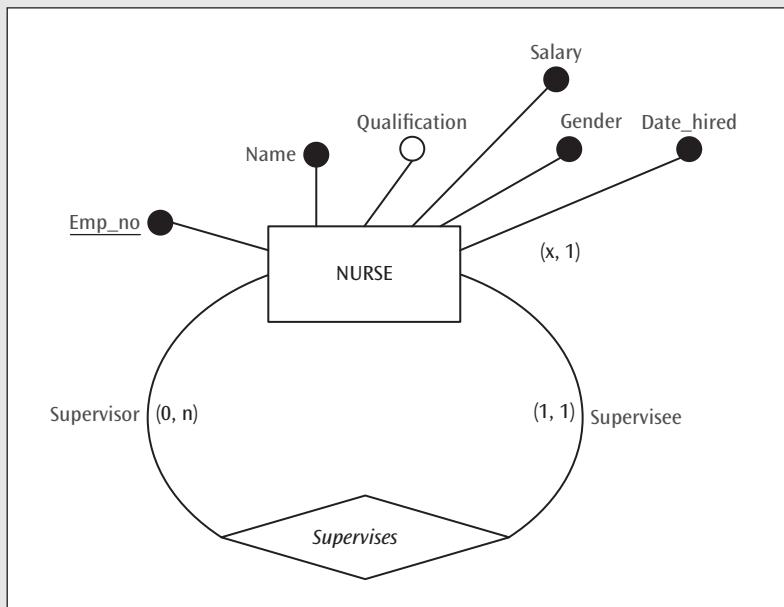
351

For the following three cases, specify the logical (relational) schema using either directed arcs or in terms of inclusion dependencies according to the foreign key design, cross-referencing design, and mutual-referencing design:

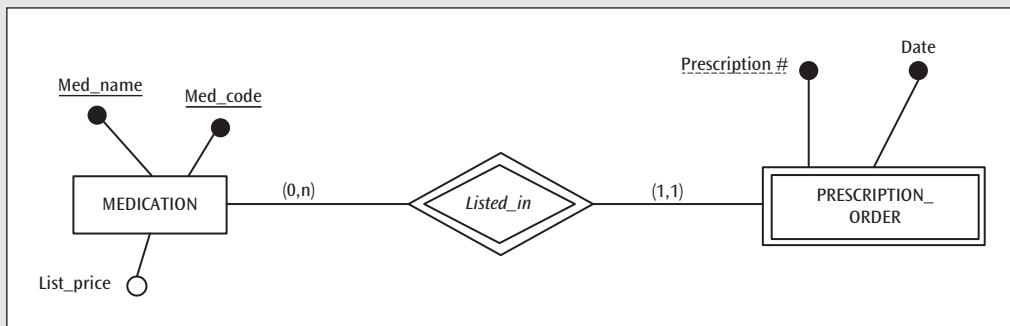
- when  $x = 0$  and  $y = 1$
- when  $x = 0$  and  $y = 0$
- when  $x = 1$  and  $y = 1$

In each case, offer a comparative discussion of the merits and demerits of the three design options.

27. Consider the following excerpt from an ERD:



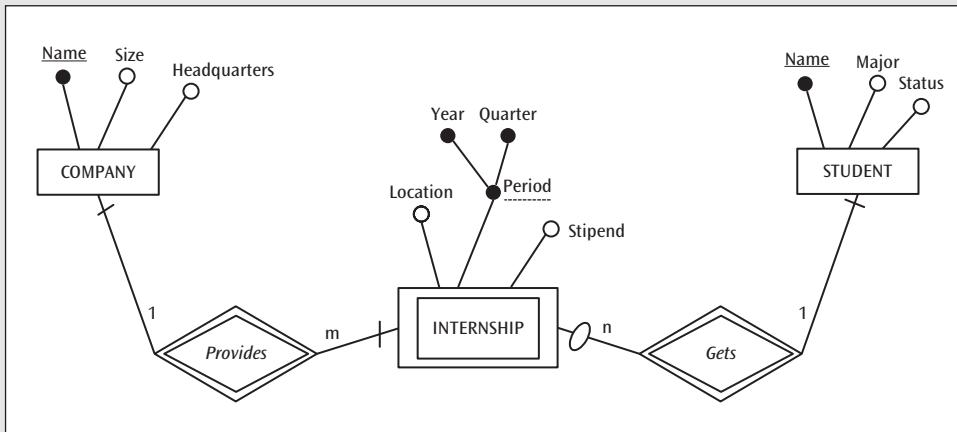
- Specify the logical (relational) schema (choosing a design without a need to use null values to indicate partial participation) in the following ways:
- using directed arcs
  - in terms of inclusion dependencies
28. How can the design requirement in Exercise 27 be satisfied if the “Supervisee” part of the relationship has the structural constraints (0, 1)? Again, show a design using both directed arcs and in terms of inclusion dependencies.
29. Specify the logical schema for the ERD in Exercise 23 using the information-preserving grammar, and indicate the metadata present in the ERD (Exercise 24) captured by this logical schema.
30. Specify the logical schema for the ERD for Exercise 27 using the information-preserving grammar.
31. Consider the following ERD:



Using the foreign key design, specify the logical schema for this diagram in the following ways:

- with a directed arc
- in terms of an inclusion dependency
- using the information-preserving grammar

32. Consider the following Presentation Layer ERD:



353

Using the foreign key design, specify the logical scheme in the following ways:

- with directed arcs
- in terms of inclusion dependencies
- using the information-preserving grammar

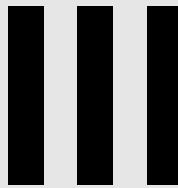
Also, list the metadata present in the ERD and indicate the ones captured by each of the three designs of the logical schema.

33. Specify the logical (relational) schema for the ERD shown in Figure 5.61 in Chapter 5 for the Cougar Medical Associates as per the foreign key design:

- using directed arcs
- in terms of inclusion dependencies



# PART



## NORMALIZATION

### INTRODUCTION

At this point in the database design life cycle, we are in the logical tier, and a logical data model comprising a logical schema and a list of semantic integrity constraints has been developed. The modeling process to this point has been more heuristic and intuitive than scientific, and in fact the source schema (ERD) for the logical modeling process was conceived intuitively.

Now that we are at the doorstep of implementing a database system using this design, a valid question to consider concerns the “goodness” of the design. What do we know about the quality of the data model we have in our hands? How do we vouch for the goodness of the initial conceptual model and the quality of the process of transforming the conceptual data model to its logical counterpart? The data models on hand at this point are probably “good” for user-analyst interaction purposes. But how can we make sure that the database design,

if implemented, will work without causing any problems? No matter what approach is taken,<sup>1</sup> grouping of attributes is an intuitive process and so requires validation for design quality. How do we go about doing this? The answer is normalization.

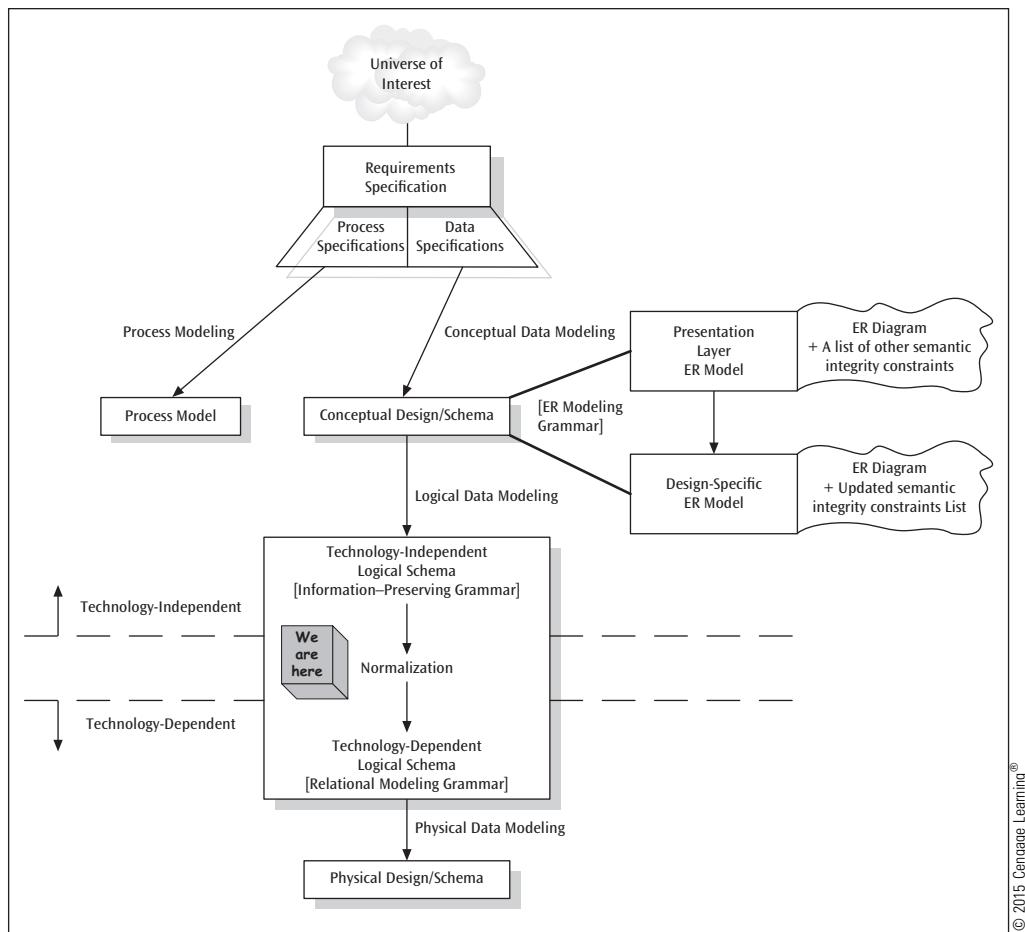
A major problem that often escapes attention during semantic considerations in data modeling is data redundancy.<sup>2</sup> Data redundancy creates the potential for inconsistencies in the stored data. Normalization is a technique that systematically eliminates data redundancies in a relational database. The principles of normalization have been developed as a part of relational database theory. While the dependency-preserving logical data model developed in Part II accommodates constructs beyond what is permissible in a relational data model, the issues and answers addressed by normalization principles apply equally to all data models in the logical tier. Since contemporary database systems are dominated by relational data models, we confine our attention to the relational data model and relational database systems. Figure III.1 points out our current location in the data modeling journey.

Chapter 7 looks at data redundancy in a relation schema and why it is a problem. The problem is then traced to its source, that is, undesirable functional dependencies. Functional dependencies are examined through inference rules called Armstrong's axioms. Next, we study techniques to derive the candidate keys of a universal relation schema for a given set of functional dependencies. Chapter 8 is dedicated to developing a solution to data redundancy problems triggered by undesirable functional dependencies; in other words, normalization. After discussing normal forms associated with functional dependencies in isolation, we examine the side effects of normalization—namely, the lossless-join property and dependency preservation property. Chapter 8 presents a comprehensive approach to resolving various normal form violations triggered by a set of functional dependencies in a universal relation schema. This is followed by a brief discussion of how to “reverse engineer” a normalized relational schema to the conceptual tier, which often forges a better understanding of the database design. Chapter 9 completes the discussion of normalization by examining the impact of multi-valued dependency and join-dependency on a relation schema.

---

<sup>1</sup>This book uses a top-down approach to database design (also known as design by analysis), as shown in Figure III.1 and the other Part-introductory figures. A bottom-up approach to database design, based on the early binary modeling work by Abrial (1974), is also possible. Somewhat less popular than the top-down approach, this design-by-synthesis approach is the basis for the NIAM model.

<sup>2</sup>Redundancy means “superfluous repetition” that does not add any new meaning.



© 2015 Cengage Learning®

**FIGURE III.1** Roadmap for data modeling and database design

# CHAPTER 7

# FUNCTIONAL DEPENDENCIES

When developing ER models, the entity types and relationships among them are intuitively distilled from the requirement specifications; then attributes are assigned to each entity type and sometimes to relationship types. Alternatively, all discernible data elements in the requirement specifications are treated as attributes, and these attributes are grouped based on apparent commonalities. The clusters of attributes are then labeled as entity types and related to each other as semantically obvious. Unfortunately, there is no objective means to validate the attribute allocation process during conceptual modeling.<sup>1</sup>

Normalization, the topic of Chapter 8, is a technique that facilitates systematic validation of participation of attributes in a relation schema from a perspective of data redundancy. The building block that enables a scientific analysis of data redundancy and the elimination of anomalies caused by data redundancy through the process of normalization is called functional dependency. This chapter introduces the concept of functional dependency and its role in the normalization process.

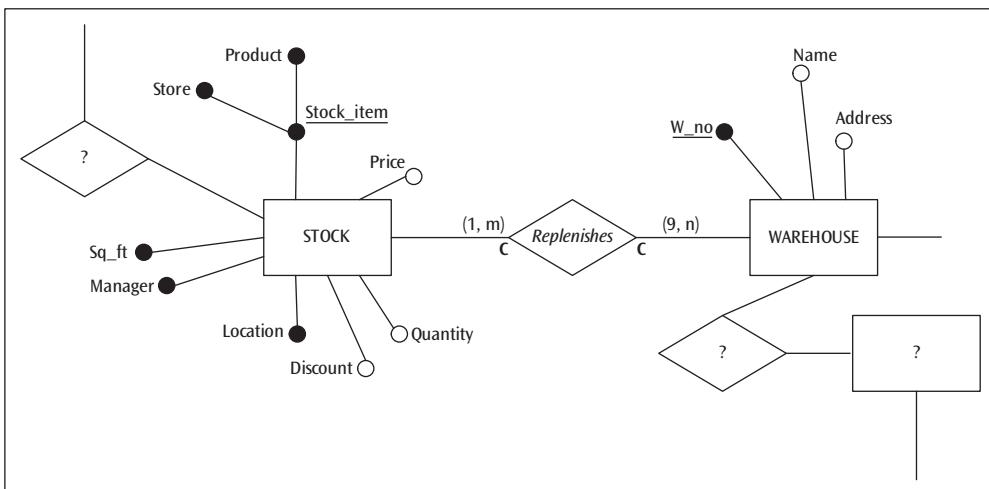
This chapter begins with a simple example in Section 7.1 that highlights the issues pertaining to “goodness” of design of a conceptual/logical data model. Section 7.2 introduces functional dependency and how this concept can be used to scientifically evaluate the “goodness” of a conceptual/logical design from the perspective of data redundancy. This section includes a definition of “functional dependency,” a discussion of inference rules that govern functional dependencies (called Armstrong’s axioms) and the idea of a minimal cover for a set of functional dependencies. Application of Armstrong’s axioms to systematically derive the candidate keys of a relation schema, given a set of functional dependencies that hold on the relation schema, is presented in Section 7.3.

---

<sup>1</sup>In fact, some people question the efficacy of the conceptual modeling step in a database design. Hypothetically speaking, it is possible to develop a relational data model directly from user requirement specifications by transforming all business rules to domain constraints and functional dependencies. However, we subscribe to the school that advocates conceptual modeling as a necessary and useful step in the database development process.

## 7.1 A MOTIVATING EXEMPLAR

Figure 7.1a is an excerpt from a larger ERD (ERD). This example focuses on the entity type STOCK that appears in Figure 7.1a. The relation schema for STOCK is shown in Figure 7.1b, and an instance of STOCK (i.e., a representative state of the Relation<sup>2</sup> hereafter referred to as a **relation instance**) appears in the data set in Figure 7.1c. Since we have chosen the relational database architecture for the implementation of the logical design, this data set is indeed a table.



**FIGURE 7.1a** An excerpt from an ERD

STOCK (Store, Product, Price, Quantity, Location, Discount, Sq_ft, Manager)
---

**FIGURE 7.1b** Relation schema for the entity type STOCK

<sup>2</sup>A representative state means that all characteristics of the real, complete relation can be inferred from the instance shown. That is, the tuples in the *relation instance* have been hand-picked to fully represent all the characteristics of the source relation. For instance, one can infer that each Product has exactly one Price from Figure 7.1c. It is incorrect to argue about the possibility of the **Price** of a **Product** varying from store to store on common sense grounds. After all, common sense varies from person to person! In other words, any inference about the properties (not data values) of this relation must be made from the instance of the relation presented—that is why the instance is made available.

STOCK							
Store	Product	Price	Quantity	Location	Discount	Sq_ft	Manager
15	Refrigerator	1850	120	Houston	5%	2300	Metzger
15	Dishwasher	600	150	Houston	5%	2300	Metzger
13	Dishwasher	600	180	Tulsa	10%	1700	Metzger
14	Refrigerator	1850	150	Tulsa	5%	1900	Schott
14	Television	1400	280	Tulsa	10%	1900	Schott
14	Humidifier	55	30	Tulsa		1900	Schott
17	Television	1400	10	Memphis		2300	Creech
17	Vacuum Cleaner	300	150	Memphis	5%	2300	Creech
17	Dishwasher	600	150	Memphis	5%	2300	Creech
11	Computer		180	Houston	10%	2300	Creech
11	Refrigerator	1850	120	Houston	5%	2300	Creech
11	Lawn Mower	300		Houston		2300	Creech

**FIGURE 7.1c** An instance of the relation schema STOCK

A quick review of the contents of Figure 7.1c confirms **{Store, Product}** to be a candidate key of STOCK, as denoted in the ERD. As a consequence, it can be the primary key of the relation schema STOCK, as indicated by the underlining here:

**STOCK (Store, Product, Price, Quantity, Location, Discount, Sq\_ft, Manager)**

A cursory look at the table in Figure 7.1c indicates all sorts of redundancy in its content—literally, every attribute value appears to be duplicated. A closer inspection reveals that there is some data redundancy in the table, but not all data that appear on the surface to be redundant are actually redundant. For instance, there are lots of duplicate values of **Quantity**. Does this mean there is data redundancy in the attribute **Quantity**? No, because there is no “superfluous repetition” of data values of **Quantity** in STOCK. It is true that a given **Product** has the same **Quantity** in more than one row of the table. This would be redundant only if this is the case irrespective of the store in which it is stocked. Since that is not the case, presence of duplicate values of **Quantity** in STOCK does not signify redundancy. On the other hand, the **Price** of a **Product** in STOCK is the same irrespective of any other fact in the table (e.g., any store). Therefore, duplication of the **Price** of a **Product** in multiple rows in the table amounts to redundant data. Based on similar reasoning, notice that there is redundancy in the data for **Location** as well as **Discount** in STOCK. It is a good exercise for the reader to reason this out.

The next issue to investigate is the “so what?” question—that is, why does the data redundancy matter? While the wasted storage space need not be a serious issue, there are more significant problems. Suppose we want to add Washing Machine to the stock with a **Price**. We cannot do this without knowing a **Store** where washing machines are stocked. This is because **{Store, Product}** is the primary key of this table STOCK, and the entity integrity constraint stipulates that neither **Store** nor **Product** can have “null” values in this table. This is what is called an **insertion anomaly**.<sup>3</sup> This is a serious problem because it may be an unreasonable imposition on the user community. Now, say that store 17 is closed. In order to remove store 17, not only do we need to remove several rows from the STOCK table, we

<sup>3</sup>An anomaly, according to the *Random House Dictionary*, means a “deviation from the rule, type, or form; an irregularity or abnormality.”

inadvertently lose the information that the vacuum cleaner is priced at \$300, since no other store presently stocks vacuum cleaners. This is a **deletion anomaly**. If, for instance, we want to change the **Location** of store 11 from Houston to Cincinnati, we need to update all rows in the STOCK table that are store 11. Failure to do so will result in store 11 being located in both Houston and Cincinnati. This is referred to as an **update anomaly**. In this and other chapters of the book, we use the umbrella term **modification anomalies** to collectively refer to insertion, deletion, and update anomalies. One way of addressing modification anomalies is to decompose the STOCK table into other relations, as shown in Figure 7.2. The three relations (tables) STORE, PRODUCT, and INVENTORY are decompositions that collectively replace the table STOCK shown in Figure 7.1.

STORE			
Store	Location	Sq_ft	Manager
15	Houston	2300	Metzger
13	Tulsa	1700	Metzger
14	Tulsa	1900	Schott
17	Memphis	2300	Creech
11	Houston	2300	Creech

PRODUCT	
Product	Price
Refrigerator	1850
Dishwasher	600
Television	1400
Humidifier	55
Vacuum Cleaner	300
Computer	
Lawn Mower	300
Washing Machine	750

INVENTORY			
Store	Product	Quantity	Discount
15	Refrigerator	120	5%
15	Dishwasher	150	5%
13	Dishwasher	180	10%
14	Refrigerator	150	5%
14	Television	280	10%
14	Humidifier	30	
17	Television	10	
17	Vacuum Cleaner	150	5%
17	Dishwasher	150	5%
11	Computer	180	10%
11	Refrigerator	120	5%
11	Lawn Mower		

**FIGURE 7.2** A decomposition of the STOCK instance in Figure 7.1c

Now, if we want to add Washing Machine and its price, we can add the information to the PRODUCT table; whenever a store begins to stock washing machines, we can add the necessary data to the INVENTORY table, which eliminates the insertion anomaly in STOCK. If store 17 is closed, a *single* row in the STORE table is deleted and there is no other loss of information (we still know that a vacuum cleaner costs \$300)—an example of removing the deletion anomaly. Changing the location for store 11 requires the update of a single row in the STORE table, as opposed to modifying several rows in the original STOCK table, removing the update anomaly. It is true that the decomposed design in Figure 7.2 may be less efficient for data retrieval; but then, that is the price for eliminating modification anomalies caused by redundant data.

STORE and PRODUCT now do not have any data redundancy. How about INVENTORY? Are **Store** and **Product** in this relation redundant because these attributes are already present in the other two relations? The answer is “No,” simply because the repetition of the attributes in INVENTORY is *not* superfluous. These attributes in INVENTORY convey more semantics than what is present in STORE and PRODUCT. However, it is obvious from the table instance INVENTORY as well as from the original table instance of STOCK that **Discount** values are redundantly stored. After all, for a given **Quantity**, there is only a single, specific **Discount** value. The solution is a simple further decomposition, shown in Figure 7.3.

STORE			
Store	Location	Sq_ft	Manager
15	Houston	2300	Metzger
13	Tulsa	1700	Metzger
14	Tulsa	1900	Schott
17	Memphis	2300	Creech
11	Houston	2300	Creech

PRODUCT	
Product	Price
Refrigerator	1850
Dishwasher	600
Television	1400
Humidifier	55
Vacuum Cleaner	300
Computer	
Lawn Mower	300
Washing Machine	750

INVENTORY		
Store	Product	Quantity
15	Refrigerator	120
15	Dishwasher	150
13	Dishwasher	180
14	Refrigerator	150
14	Television	280
14	Humidifier	30
17	Television	10
17	Vacuum Cleaner	150
17	Dishwasher	150
11	Computer	180
11	Refrigerator	120
11	Lawn Mower	

DISC_STRUCTURE	
Quantity	Discount
120	5%
150	5%
180	10%
280	10%
30	
10	

**FIGURE 7.3** A redundancy-free decomposition of the STOCK instance in Figure 7.1c

However, there are times when some data redundancy is willfully tolerated as a tradeoff for efficiency of querying (data retrieval). Suppose the discount structure, according to the user, is relatively stable (i.e., minimal changes). Then, the design in Figure 7.2 may be a more optimal design than that in Figure 7.3, despite the data redundancy in INVENTORY. Such a redundancy is sometimes referred to as **controlled redundancy**.

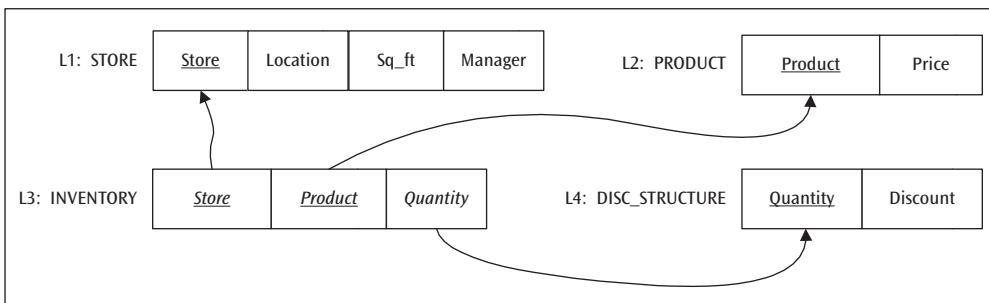
The table STOCK in Figure 7.1c is said to be “unnormalized”—that is, it has many data redundancies—whereas the set of tables in Figure 7.3 is said to be fully “normalized,” given that it has no data redundancies. The questions at this point ought to be:

- How do we systematically identify data redundancies?
- How do we know how to decompose the base relation schema under investigation?

- How do we know that the decomposition is correct and complete without looking at sample data?

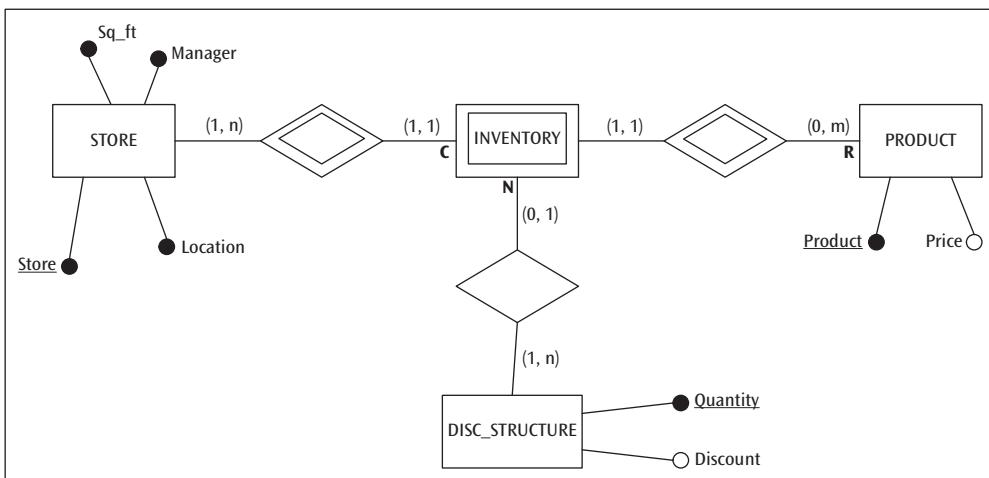
The remainder of this chapter is dedicated to answering these questions, and the process that emerges as a result is called *normalization*. But before embarking on this journey, let us explore the “normalized” design represented by the set of tables in Figure 7.3.

What is the relational schema that will yield this set of tables? Obviously, it will contain four relation schemas, one each for STORE, PRODUCT, INVENTORY, and DISC\_STRUCTURE. Figure 7.4a displays this relational schema.



**FIGURE 7.4a** A reverse-engineered logical schema for the set of tables in Figure 7.3

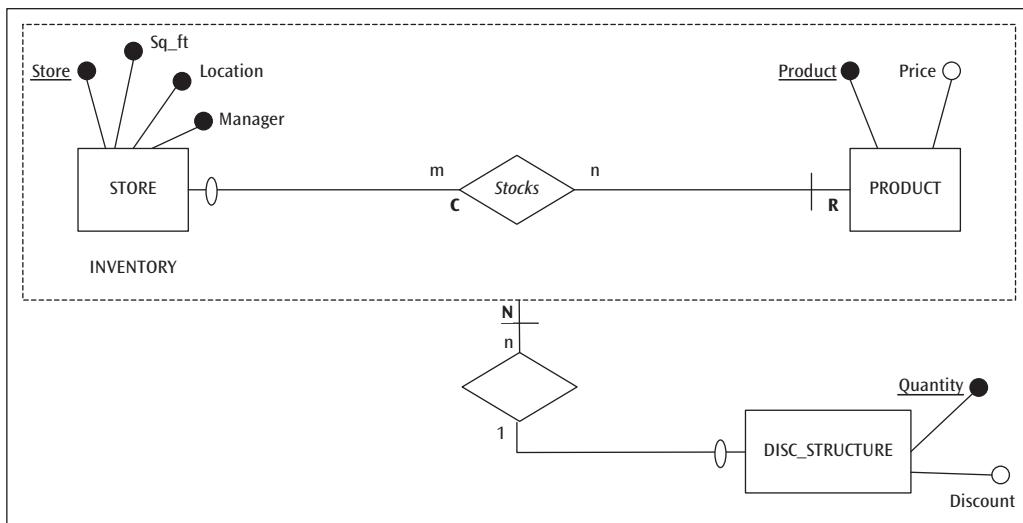
Note that the structural constraints of relationships emerge from the data in the tables. Since deletion rules are not discernible from the tables, the default value of restrict (R) is adopted. At this point, we are able to further “reverse engineer” the relational schema to a Design-Specific ERD, as shown in Figure 7.4b.



**FIGURE 7.4b** Design-Specific ERD reverse-engineered from the logical schema in Figure 7.4a

Observe that INVENTORY is a gerund entity type with two identifying parents, STORE and PRODUCT. Finally, we can also abstract the Design-Specific ERD to the Presentation

layer by unraveling the m:n relationship type implicit in the gerund entity type. Since ER modeling grammar does not allow a relationship type to be related to another relationship type, INVENTORY necessarily becomes a cluster entity type. The Presentation Layer ERD that will yield the Design-Specific ERD in Figure 7.4b is presented in Figure 7.4c, and we have just completed what is known in data modeling circles as “reverse engineering.” By comparing the ERDs in Figures 7.4c and 7.1, we ought to be able to appreciate the design problems due to data redundancy hidden in the original entity type STOCK.



**FIGURE 7.4c** Presentation Layer ERD reverse-engineered from Figure 7.4b

Similar problems may be present in WAREHOUSE (see Figure 7.1a) and every other relation schema in a relational data model. At this point in the database design life cycle, each relation schema in the relational data model must be independently scrutinized and “normalized” where needed. While conceptual modeling is not a scientific process, organized application of intuition in the development of the conceptual data model and careful mapping of the conceptual schema to the logical (relational) schema where most of the constituent relation schemas are fully normalized—that is, free from modification anomalies. Nonetheless, as shown in the example, a systematic verification of the “goodness” of the design at this stage of data modeling is imperative lest the implemented database system should fail to meet user expectations. To that end, let us proceed to investigate the “how-do” questions:

- How do we systematically identify data redundancies?
- How do we know how to decompose the base relation schema under investigation?
- How do we know that the decomposition is correct and complete without looking at sample data?

In order to engage in this enquiry, it is necessary to understand the concept of functional dependency.

## 7.2 FUNCTIONAL DEPENDENCIES

---

A crucially important aspect of relational database theory is the concept of **functional dependency (FD)**. Functional dependency is the building block of normalization principles. *Undesirable* Fds in a relation schema are the seeds of data redundancies that lead to modification anomalies. FDs, desirable or otherwise, are essentially technical translations of user-specified business rules expressed as constraints in a relation schema and so cannot be ignored or discarded when undesirable.

### 7.2.1 Definition of Functional Dependency

In simple terms, attribute A (atomic or composite) in a relation schema R functionally determines attribute B (atomic or composite) in R if, for a given value  $a_1$  of A, there is a *single, specific* value  $b_1$  of B in relation r of R.<sup>4</sup> Note that r represents all possible states (instances) of R. A symbolic expression of this FD is:

$$A \rightarrow B$$

where A (the left-hand side of the FD) is known as the **determinant** and B (the right-hand side of the FD) is known as the **dependent**.

In other words, if A functionally determines B (i.e., B is functionally dependent on A) in R, then it is invalid to have two (or more) tuples that have the same A value but different B values in R. That is, in order for the FD  $A \rightarrow B$  to hold in R, if two tuples of  $r(R)$  agree on their A values, they must necessarily agree on their B values. A corollary of this definition is as follows:

- If A is a candidate key of R (i.e., no two tuples in  $r(R)$  can have the same values for A), then  $A \rightarrow B$  for any subset of attributes, B of R.

Equally important,  $A \rightarrow B$  in R does not ever imply  $B \rightarrow A$  in R.

For example, from the instance of the relation schema STOCK (see Figure 7.1c), we can infer that **Product**  $\rightarrow$  **Price** because all tuples of STOCK with a given **Product** value also have the same **Price** value. Likewise, it can be inferred that **Store**  $\rightarrow$  **Location**.

Notice that **Location** does not determine **Store** in STOCK, because for a given value of **Location** in STOCK, there is more than one value of **Store**. This can be shown this way:

$$\text{Location} \not\rightarrow \text{Store}$$

When the determinant and/or dependent in an FD is a composite attribute, the constituent atomic attributes are enclosed by braces, as shown here:

$$\{\text{Store}, \text{Product}\} \rightarrow \text{Quantity}$$

Once again, an FD is a property of the semantics (meaning) of the relationship among attributes emerging from the business rules. The sample data shown in Figure 7.1c is intended to clarify these semantics. Any extrapolation of meaning beyond the facts conveyed in the relation instance is incorrect. Note that an FD is a property of the relation schema R, not of particular relation state r of R. Therefore, an FD cannot be automatically inferred from *any* relation state r of R. It must be explicitly specified as a constraint, and the source for this specification is the business rules of the application domain. However,

---

<sup>4</sup>As a matter of technicality, functional dependencies exist only when attributes or other elements involved have unique and singular identifiers (Kent, 1983, p. 121). Since a relation schema, by definition, has a candidate key, the functional dependency concept is applicable to a relation schema.

a *relation instance* is, by our definition, a specially prepared relation state  $r$  of  $R$  conforming to the set of FDs specified on  $R$  and can be used to infer the FDs in  $R$ .

### 7.2.2 Inference Rules for Functional Dependencies

Given a set of *semantically obvious* FDs for a relation schema  $R^5$  emerging from the business rules of an application domain, it is possible to deduce all the other FDs that hold in  $R$  but are not explicitly stated. The set of FDs that are explicitly specified on a relation schema is usually referred to as  $F$ . As an example, in the relation schema STOCK, if:

fd1:  $\{\text{Store, Product}\} \rightarrow \text{Quantity}$  and

fd2:  $\text{Quantity} \rightarrow \text{Discount}$

are specified in  $F$ , one can infer an FD:

fd3:  $\{\text{Store, Product}\} \rightarrow \text{Discount}$

Therefore, it is not necessary to explicitly state fd3. The notation:

$F \models \text{fd3 or } F \models \{\text{Store, Product}\} \rightarrow \text{Discount}$

is used to indicate that fd3 is inferred from the set of functional dependencies,  $F$ .

If a set of functional dependencies,  $F$ , prevails over the relation schema  $R$ ,  $F \models X \rightarrow Y$  also holds in  $R$ . That is, whenever a relation state  $r$  of  $R$  satisfies all FDs in  $F$ , then the FD  $X \rightarrow Y$ , if inferred from  $F$ , is also satisfied in that  $r$ . It is useful to define the idea of all possible FDs that hold in  $R$ . Referred to as *closure*, this includes all possible FDs that can be inferred from the set  $F$ . Denoted as  $F^+$ , the closure of  $F$  includes the set of FDs stated in  $F$ , plus all other FDs that can be inferred from  $F$ . While in practice there is little need to compute  $F^+$ , it is often necessary to know if a particular FD exists in  $F^+$ .

In a 1974 journal article, William Armstrong proposed a systematic approach to derive FDs that can be inferred from  $F$ .<sup>6</sup> Usually referred to as **Armstrong's axioms**, this set of inference rules can also be used to precisely derive the closure  $F^+$ .

Given  $R(X, Y, Z, W)$ , where  $X, Y, Z$ , and  $W$  are arbitrary subsets (atomic or composite) of the set of attributes of a universal relation schema  $R$ , three fundamental inference rules that can be directly proven from the definition of functional dependency are:

- **Reflexivity rule:** If  $Y$  is a subset of  $X$  (e.g., if  $X$  is  $\{A, B, C, D\}$  and  $Y$  is  $\{A, C\}$ ), then  $X \rightarrow Y$ .

*Note:* The reflexivity rule defines what is called a **trivial dependency**. A dependency is trivial if it is impossible to *not* satisfy it.<sup>7</sup>

- **Augmentation rule:** If  $X \rightarrow Y$ , then  $\{X, Z\} \rightarrow \{Y, Z\}$ ; also,  $\{X, Z\} \rightarrow Y$ .
- **Transitivity rule:** If  $X \rightarrow Y$ , and  $Y \rightarrow Z$ , then  $X \rightarrow Z$ .

---

<sup>5</sup>Here, it is useful to assume that the entire database for the application domain is a single universal relation schema.

<sup>6</sup>Armstrong, W. W., "Dependence Structures of Data Base Relationships." *Proc. IFIP Congress*. Stockholm, Sweden, 1974.

<sup>7</sup>An FD in  $R$  is trivial if and only if the dependent is a subset of the determinant. Since trivial dependencies do not provide any additional information (i.e., do not add any new constraints on  $R$ ), they are usually removed from  $F^+$ .

Using the illustration in Figure 7.1:

**{Store, Product} → Store** exemplifies the *reflexivity* rule.

The *augmentation* rule works as follows:

If **Store → Location**, then **{Store, Product} → {Location, Product}**

and:

**{Store, Product} → Location.**

The *transitivity* rule is demonstrated as follows:

If **{Store, Product} → Quantity** and **Quantity → Discount**, then transitively:

**{Store, Product} → Discount.**

Four more inference rules can be constructed from the these three, some of which help simplify the practical task of generating  $F^+$ . The four rules are:

- **Decomposition rule:** If  $X \rightarrow \{Y, Z\}$ , then  $X \rightarrow Y$  and  $X \rightarrow Z$ .
- **Union (or additive) rule:** If  $X \rightarrow Y$  and  $X \rightarrow Z$ , then  $X \rightarrow \{Y, Z\}$ .
- **Composition rule:** If  $X \rightarrow Y$  and  $Z \rightarrow W$ , then  $\{X, Z\} \rightarrow \{Y, W\}$ .
- **Pseudotransitivity rule:** If  $X \rightarrow Y$  and  $\{Y, W\} \rightarrow Z$ , then  $\{X, W\} \rightarrow Z$ .

Again, using the relation instance of STOCK in Figure 7.1,

If **Store → {Location, Sq\_ft, Manager}**, then per the *decomposition* rule:

**Store → Location** and **Store → Sq\_ft** and **Store → Manager**.

Given:

**Store → Location** and **Store → Sq\_ft**,

**Store → {Location, Sq\_ft}** exemplifies the *union* rule.

This example demonstrates the *composition* rule:

If **Store → Location** and **Product → Price**, then **{Store, Product} → {Location, Price}**.

The *pseudotransitivity* rule is a handy corollary of the transitivity rule and works the following way:

If **Manager → Store** and **{Store, Product} → Quantity**, then **{Manager, Product} → Quantity**.

The inference rules for FDs known as Armstrong's axioms are summarized in Table 7.1. Several of the inference rules discussed earlier can be derived from Darwen's *General Unification Theorem* (1992):

If  $X \rightarrow Y$  and  $Z \rightarrow W$ , then  $X \cup \{Z-Y\} \rightarrow \{Y, W\}$

In principle, the closure,  $F^+$ , of a given set of FDs,  $F$ , can be computed by the use of a rather inefficient algorithm: *Repeatedly apply Armstrong's axioms on F until it stops producing new FDs* (Date, 2004).

### 7.2.3 Minimal Cover for a Set of Functional Dependencies

If every FD in a set of FDs,  $F$ , can be inferred from another set of FDs,  $G$ , then  $G$  is said to *cover*  $F$ . In this case, every FD in  $F$  is also present in  $G^+$ —that is,  $F \subset G^+$ . Two sets of FDs,  $F$  and  $G$ , are considered *equivalent* if  $F^+ = G^+$ . This equivalence of the two sets of FDs is expressed as  $F \equiv G$ , implying that  $F \subset G^+$  and  $G \subset F^+$ ; in other words,  $G$  covers  $F$  and  $F$  covers  $G$ . From a practical perspective, when  $F \equiv G$ , one can choose to enforce either  $F$  or  $G$  and the valid database states will remain the same.

Rule	Definition
Reflexivity	If Y is a subset of X [i.e., if X is (A,B,C,D) and Y is (A,C)], then $X \rightarrow Y$ . (The reflexivity rule defines trivial dependency as a dependency that is impossible to <i>not</i> satisfy.)
Augmentation	If $X \rightarrow Y$ , then $\{X,Z\} \rightarrow \{Y,Z\}$ ; also, $\{X,Z\} \rightarrow Y$ .
Transitivity	If $X \rightarrow Y$ , and $Y \rightarrow Z$ , then $X \rightarrow Z$ .
Decomposition	If $X \rightarrow \{Y,Z\}$ , then $X \rightarrow Y$ and $X \rightarrow Z$ .
Union (or additive)	If $X \rightarrow Y$ , and $X \rightarrow Z$ , then $X \rightarrow \{Y,Z\}$ .
Composition	If $X \rightarrow Y$ , and $Z \rightarrow W$ , then $\{X,Z\} \rightarrow \{Y,W\}$ .
Pseudotransitivity	If $X \rightarrow Y$ , and $\{Y,W\} \rightarrow Z$ , then $\{X,W\} \rightarrow Z$ .

*References:* Armstrong, W. W. "Dependence Structures of Data Base Relationships." *Proc. IFIP Congress*. Stockholm, Sweden, 1974.; Darwen, H. "The Role of Functional Dependencies in Query Decomposition." In Date, C. J., and H. Darwen, *Relational Database Writings 1989–1991*. Addison-Wesley, 1992.

**TABLE 7.1** Inference rules for functional dependencies: Armstrong's axioms

It is likely that a set of FDs, F, translated from user-specified business rules has redundant (extraneous) attributes and sometimes redundant (extraneous) FDs<sup>8</sup> because, as narratives, requirement specifications are prone to be somewhat repetitive. For example, consider the attribute set **{Store, Product, Price}** and an associated set of FDs, F, culled from the requirement specification, where fd1: **{Store, Product} → Price** and fd2: **Product → Price**. Here, fd1 is redundant because  $(F - fd1)$  is equivalent to F, meaning removal of fd1 will not change  $F^+$ . Alternatively, **Store** in fd1 is a redundant attribute, and removal of **Store** from fd1 will not change  $F^+$ . Likewise, given an attribute set **{Store, Product, Price, Quantity}** and an associated set of FDs, G, where fd1: **{Store, Product} → {Quantity, Price}** and fd2: **Product → Price**, the attribute **Price** is redundant in fd1—that is, removal of **Price** from fd1 will not change  $G^+$ .

Suppose that a set of FDs, F, prevails over a relational schema. This means that whenever a user performs an update that entails changes to one or more relations in the database, the DBMS must ensure that the update does not violate any of the FDs in F. All FDs in F must hold in the updated database state; otherwise, the DBMS must roll back the updates and restore the database to the original state that prevailed before the update. It is always useful to identify a simplified set of FDs,  $G_c$ , equivalent to F—that is, having the same closure ( $F^+$ ) as F and not further reducible. This  $G_c$  is not only equivalent to F, but further reduction of  $G_c$  destroys the equivalence. The practical value of such a simplified set of FDs,  $G_c$ , is that the effort required to check for violations in the database is minimized because a database that satisfies  $G_c$  will also satisfy F and vice versa.  $G_c$  in this case is called the **canonical cover** or **minimal cover** of F. Formally, a set of FDs,  $G_c$  is a minimal cover for another set of FDs, F, if the following conditions are satisfied:

- $G_c \equiv F$  ( $G_c$  and F are equivalent.)
- The *dependent* (right-hand side) in every FD in  $G_c$  is a singleton attribute. This is known as the *standard* or *canonical form* of an FD and is intended to

<sup>8</sup>This is based on the application of the transitivity rule of Armstrong's axioms.

simplify the conditions and algorithms that ensure absence of redundancies in F.

- No FD in  $G_c$  is redundant—that is, no FD from  $G_c$  can be discarded without converting  $G_c$  into some set not equivalent to  $G_c$  and therefore not equivalent to F.
- The *determinant* (left-hand side) of every FD in  $G_c$  is irreducible—that is, no attribute can be discarded from the determinant of any FD in G without converting  $G_c$  into some set not equivalent to  $G_c$ .

The process of deducing minimal covers is illustrated in the following examples.

### 7.2.3.1 Example 1

Here is a simple example. Consider the set of attributes **{Tenant, Apartment, Rent}** and the set of FDs, F:

fd1: <b>Tenant</b> → {Apartment, Rent};	fd2: <b>Apartment</b> → Rent;
fd3: { <b>Tenant, Apartment</b> } → Rent;	fd4: { <b>Tenant, Rent</b> } → Apartment

Using Armstrong's axioms, we can deduce from F that **Tenant** → **Apartment** is in  $F^+$ . Note that the set of FDs, G:

fd1: <b>Tenant</b> → {Apartment, Rent};	fd2: <b>Apartment</b> → Rent;
fd3: { <b>Tenant, Apartment</b> } → Rent;	fd4: <b>Tenant</b> → Apartment

is a cover for F. However, is G a minimal cover for F? No. It will be a minimal cover only if there are no redundant attributes and redundant FDs in G. An examination of G reveals that, given fd2, the attribute **Tenant** is redundant in fd3. Removal of the redundant attribute from fd3 renders fd2 and fd3 identical; so one of these two FDs (say, fd3) is redundant and can be deleted. Next, given fd1, fd4 is redundant and can be removed without any consequence. Thus, we are left with  $G'$ : {fd1, fd2}, where

fd1: **Tenant** → {Apartment, Rent}; and fd2: **Apartment** → Rent

Is  $G'$  a minimal cover for F? The answer is still no, because **Tenant** → **Rent** in fd1 is still redundant. Removing this redundancy from fd1, we have  $G_x$ : {fd1, fd2}, where:

fd1: **Tenant** → **Apartment** and fd2: **Apartment** → Rent

which now is a minimal cover for F, G, and  $G'$ .

Is  $G_x$ : {fd1, fd2}, where

fd1: **Tenant** → **Apartment** and fd2: **Tenant** → **Rent**

a minimal cover for F?

The answer is no because  $G_x$  is not equivalent to F, in that  $G_x^+$  is not =  $F^+$ . For instance, the FD **Apartment** → **Rent** present in  $F^+$  is not present in  $G_x^+$ .

In short, a minimal cover  $G_c$  of a set of FDs, F, is not only equivalent to F (that is,  $F \equiv G_c$ , meaning  $F^+ = G_c^+$ ), it also contains neither redundant FDs nor redundant attributes. Every set of FDs, F, possesses a minimal cover. F can be its own minimal cover, too. In fact, careful construction of F from the business rules often yields F itself as a minimal cover of the set of FDs in it. Furthermore, there can be several minimal covers for F.

When several sets of FDs qualify as minimal covers of F, additional criteria are used to choose a minimal cover, such as the minimal cover with the least number of FDs or the minimal cover that most closely resembles F.

### 7.2.3.2 Example 2

Consider a set of attributes {A, B, C} and an associated set of FDs, F: {fd1, fd2, fd3, fd4}, where:

fd1: A → C;

fd2: (A, C) → B;

fd3: B → A;

fd4: C → (A, B)

370

fd4 can be rewritten in standard form as:

fd4a: C → A and fd4b: C → B

Based on fd4b, using Armstrong's axioms, we infer that A in fd2 is a redundant attribute. Removal of A from fd2 renders fd2 and fd4b identical. Therefore, one of these two FDs can be dropped with no consequence. fd4a is also a redundant FD since it is derived by the rule of transitivity applied on fd4b and fd3. The resulting set of FDs, G<sub>c</sub>: {fd1, fd2, fd3}, where:

fd1: A → C;

fd2: C → B;

fd3: B → A

is a minimal cover of F.

Targeting attributes and FDs for evaluation in a different sequence, it can be shown that: G<sub>m</sub>: {fd1, fd2, fd3}, where:

fd1: A → B;

fd2: B → C;

fd3: C → A

is also a minimal cover of F. The reader is encouraged to verify this.

An algorithm to compute the minimal cover for a given set of functional dependencies, F, is as follows:

- i. Set G to F.
- ii. Convert all FDs in G to *standard (canonical) form*—i.e., the right-hand side (dependent attribute) of every FD in G should be a singleton attribute.
- iii. Remove all redundant attributes from the left-hand side (determinant) of the FDs in G.
- iv. Remove all redundant FDs from G.

Two conditions in the algorithm are noteworthy:

- The execution of this algorithm may yield different results depending on the order in which the candidates for removal (both attributes and FDs) are evaluated, thus confirming the fact that it is possible to have multiple minimal covers for F.
- Steps iii and iv are not interchangeable—that is, executing step iv before step iii will not always return a minimal cover.

The next example applies this algorithm to derive the minimal cover of a set of FDs.

### 7.2.3.3 Example 3

Consider the set of attributes **{Student, Advisor, Subject, Grade}** and an associated set of FDs, F:

- fd1: **{Student, Advisor} → {Grade, Subject};**
- fd2: **Advisor → Subject;**
- fd3: **{Student, Subject} → {Grade, Advisor}**

G, the expression of F in standard form, can be written as follows:

fd1a: <b>{Student, Advisor} → Grade;</b>	fd1b: <b>{Student, Advisor} → Subject;</b>
fd2: <b>Advisor → Subject;</b>	
fd3a: <b>{Student, Subject} → Grade;</b>	fd3b: <b>{Student, Subject} → Advisor</b>

Given fd2, **Student** in fd1b is a redundant attribute. Elimination of this redundant attribute from fd1b renders fd1b entailed by fd2. Thus, fd1b becomes a redundant FD and can be removed. There are no other redundant attributes in the left-hand side (determinant) of any other FD in G. Next, fd1a is a redundant FD since it is entailed by the set of FDs {fd2, fd3a}. There are no other redundant FDs in G. Thus, we have  $G_c: \{fd2, fd3a, fd3b\}$ , where:

fd2: <b>Advisor → Subject;</b>	
fd3a: <b>{Student, Subject} → Grade;</b>	fd3b: <b>{Student, Subject} → Advisor;</b>

$G_c$  is a minimal cover of F and G.

### 7.2.3.4 Example 4

Consider the attribute set:

**{Product, Store, Vendor, Date, Quantity, Unit\_price, Discount, Size, Color}**

and the set of FDs, F:

- fd1: **Product → {Size, Color}**
- fd2: **{Vendor, Quantity} → {Unit\_price, Discount};**
- fd3: **{Product, Store, Date, Quantity} → {Vendor, Discount};**
- fd4: **{Product, Size, Color, Store, Date} → Vendor**

What is the minimal cover of F?

The first step is to express F in standard form—say, G {fd1a, fd1b, fd2a, fd2b, fd3a, fd3b, fd4}, where:

fd1a: <b>Product → Size;</b>	fd1b: <b>Product → Color;</b>
fd2a: <b>{Vendor, Quantity} → Unit_price;</b>	fd2b: <b>{Vendor, Quantity} → Discount;</b>
fd3a: <b>{Product, Store, Date, Quantity} → Vendor;</b>	fd3b: <b>{Product, Store, Date, Quantity} → Discount;</b>
fd4: <b>{Product, Size, Color, Store, Date} → Vendor</b>	

Now that G is in the standard form, the next step in the algorithm to deduce the minimal cover is to identify and remove redundant attributes from the left-hand side

(determinant) of the FDs constituting G. Accordingly, based on fd1a and fd1b, **Size** and **Color** in fd4 are redundant. Thus, fd4 reduces to fd4a: **{Product, Store, Date} → Vendor**. Based on fd4a, **Quantity** in fd3a becomes a redundant attribute, and elimination of this attribute from fd3a renders fd4a and fd3a identical. Thus, one of them (fd4a) entails the other (fd3a), which then becomes a redundant FD and so can be removed with no consequence. Next, fd3b, implied by {fd2b, fd4a}, becomes a redundant FD and so can be deleted. Thus, we have,  $G_c \{fd1a, fd1b, fd2a, fd2b, fd4a\}$ , where:

fd1a: <b>Product → Size;</b>	fd1b: <b>Product → Color;</b>
fd2a: <b>{Vendor, Quantity} → Unit_price;</b>	fd2b: <b>{Vendor, Quantity} → Discount;</b>
fd4a: <b>{Product, Store, Date} → Vendor</b>	

A close examination of F and  $G_c$  reveals that  $G_c$  is a cover for F since  $F \equiv G_c$ , meaning  $F^+ = G_c^+$ . Since  $G_c$  does not contain any redundant attributes or redundant FDs,  $G_c$  is then, by definition, a *minimal* cover for F. How do we know that  $G_c$  does not contain any redundant attributes or redundant FDs? This is tested by finding an attribute or an FD in  $G_c$ , the removal of which from  $G_c$  does not disturb the equivalence of  $G_c$  to F. The reader is encouraged to try this as an exercise.

Since **{Product, Store, Date} → Vendor** (fd4a), could we have retained fd3b in G instead of fd2b? In other words, is  $G_x \{fd1a, fd1b, fd2a, fd3b, fd4a\}$ , where:

fd1a: <b>Product → Size;</b>	fd1b: <b>Product → Color;</b>
fd2a: <b>{Vendor, Quantity} → Unit_price;</b>	
fd3b: <b>{Product, Store, Date, Quantity} → Discount;</b>	
fd4: <b>{Product, Store, Date} → Vendor</b>	

a minimal cover for F? The answer is no. In fact,  $G_x$  is not even a cover for F, let alone a minimal cover, because F is not  $\equiv G_x$ , meaning  $F^+$  is not  $= G_x^+$ . Note that **{Vendor, Quantity} → Discount** does not exist in  $G_x^+$ .

Once again, the practical value of a minimal cover,  $G_c$ , of a set of FDs, F, is that the effort required to check for violations in the database is minimized because a database that satisfies  $G_c$  will also satisfy F and vice versa.

#### 7.2.4 Closure of a Set of Attributes

A concept akin to the closure of a set of FDs is the idea of the closure of a set of attributes. Given a relation schema, R, a set of FDs, F, that hold in R, and a subset, Z, of attributes of R, the closure,  $Z^+$ , of Z under F is the set of all attributes of R that are functionally dependent on Z. Observe that a closure of a set of attributes is always subject to a set of specified FDs and is expressed as follows:  $Z^+ = \text{Closure}[Z \mid F]$ . As an illustration, suppose R (A, B, C, D, E, G, H) is a relation schema over which the following set of FDs, F, prevails:

fd1: <b>B → {G, H};</b>	fd2: <b>A → B;</b>	fd3: <b>C → D;</b>
-------------------------	--------------------	--------------------

What is the closure  $[A \mid F]$ ? What is being sought here is the set of attributes in R that are functionally dependent on A under the set of FDs, F. A quick perusal of F—that is, fd1, fd2, and fd3—indicates that  $A^+ = \{A, B, G, H\}$ .

Here is an algorithm to compute the closure of an attribute set, Z, under a set of FDs, F, in a relation schema R:

1. Set Closure  $[Z \mid F]$  to Z.
2. For each FD of the form  $X \rightarrow Y$  in F,  
if  $X \subseteq \text{Closure } [Z \mid F]$ , set Closure  $[Z \mid F]$  to  $(\text{Closure } [Z \mid F] \cup Y)$ .<sup>9</sup>
3. Iterate step 2 through F until no further change in the Closure  $[Z \mid F]$ .

Suppose we want to compute  $\{A,C\}^+$ , the Closure  $\{\{A,C\} \mid F\}$  in R using this algorithm.

Start:  $\{A,C\}^+ = \{A,C\}$

First iteration through F:

- In fd1, the determinant B is not a subset of  $\{A,C\}^+$ —so, no change in  $\{A,C\}^+$ .
- In fd2, the determinant A is a subset of  $\{A,C\}^+$ —so,  $\{A,C\}^+ = \{A,C\}^+ \cup B = \{A,C,B\}$ .
- In fd3, the determinant C is a subset of  $\{A,C\}^+$ —so,  $\{A,C\}^+ = \{A,C\}^+ \cup D = \{A,C,B,D\}$ .

Second iteration through F:

- In fd1, the determinant B is a subset of  $\{A,C\}^+$ —so,  $\{A,C\}^+ = \{A,C\}^+ \cup \{G,H\} = \{A,C,B,D,G,H\}$ .
- In fd2, the determinant A is a subset of  $\{A,C\}^+$ —so,  $\{A,C\}^+ = \{A,C\}^+ \cup B = \{A,C,B,D,G,H\}$ —no change.
- In fd3, the determinant C is a subset of  $\{A,C\}^+$ —so,  $\{A,C\}^+ = \{A,C\}^+ \cup D = \{A,C,B,D,G,H\}$ —no change.

Third iteration through F, as can be seen, is not necessary; so the algorithm terminates.  
End:  $\{A,C\}^+ = \{A,C,B,D,G,H\}$ .

Two useful corollaries are worthy of attention:

- Given F, it is possible to know if a specific FD  $X \rightarrow Y$  follows from F by computing the attribute closure  $X^+$ . If and only if Y is a subset of  $X^+$  can we infer that  $X \rightarrow Y$  follows from F. Note that we are able to determine whether the FD  $X \rightarrow Y$  follows from F without actually having to compute  $F^+$ .
- Given F, it is possible to know if a certain subset K of the attributes of R is a superkey of R by computing  $K^+$ , the closure  $[K \mid F]$ . K is a superkey of R if and only if the closure  $[K \mid F]$  is precisely the set of all attributes of R. If K happens to be an irreducible superkey of R under F, then K is a candidate key of R.

Note that  $\{A,C\}$  is not a superkey of R in the earlier example because  $\{A,C\}^+$  is not precisely the set of all attributes of R—that is,  $\{A,C\}^+ = \{A,C,B,D,G,H\}$  does not contain the attribute E of R.

---

<sup>9</sup>This is based on the application of the transitivity rule of Armstrong's axioms.

### 7.2.5 Whence Do FDs Arise?

Where do FDs come from? Why were they not considered during conceptual modeling? The answer is that FDs are technical expressions of user-specified business rules. They were considered implicitly during ER modeling. That is how attributes were collected as different entity types—that is, grouping a set of attributes that are independent of each other but functionally dependent on a specific attribute(s) (e.g., the candidate key). While an ER model is certainly useful in defining the scope of the database design, the ER modeling process itself is still more of an intuitive process. Now that the ER model has been mapped to the logical tier, it is necessary to *formally* examine each relation schema and make sure that the initial assignment of attributes to the entity types at the conceptual tier is correct. The ratification at this point is no longer an intuitive process. It is a scientific analysis and is important because this is the final opportunity to correct errors and establish a robust design before implementing the database system.

The logical analysis of attribute assignment to a relation schema may require further consultations with the user community to sharpen the business rules and lead to an explicit specification of inter-attribute constraints (i.e., FDs) that were intuitively considered during the development of the original entity types and their respective attribute sets. Note that the purpose here is the verification of the “goodness” of design. Although conceptual modeling may not be a scientific process, “organized” application of intuition in the development of the conceptual data model and careful mapping of the conceptual schema to the logical tier often yields a logical (relational) schema where most of the constituent relation schemas are fully normalized—that is, free from modification anomalies arising from undesirable FDs. Nonetheless, normalization is necessary to confirm that there are no modification anomalies, and to fine-tune the database design before embarking on implementation.

## 7.3 CANDIDATE KEYS REVISITED

The term “candidate key” is introduced in Chapter 6 along with the terms “superkey” and “primary key” as a part of the foundation concepts of a relational data model and is defined as an irreducible unique identifier of a relation schema. Now it is time to develop a more formal understanding of the idea of a candidate key and explore methods to derive the candidate keys of a relation schema. Remember that the identification of unique identifiers of an entity type during ER modeling is an intuitive process in spite of the fact that the identification is guided by the business rules of the application domain being modeled.

Also remember that a relation schema can have multiple candidate keys and that it is useful (sometimes, necessary) to know all the candidate keys of a relation schema. Two approaches are generally used to derive a candidate key of a relation schema. Given a set of FDs,  $F$ , it is possible to derive a candidate key for the Universal Relation Schema (URS) constructed from the set of attributes present in  $F$  through a process of synthesis. This method is based on the principle of the closure of an attribute set. In short, this method seeks to derive an irreducible set of attributes whose closure is precisely all attributes of the URS. Alternatively, given a URS and the set of FDs,  $F$ , that prevails over the URS, a candidate key can be derived through a process of decomposition of a superkey of the URS. After one candidate key is derived using either of these approaches, the method used to derive the rest of the candidate keys of the relation schema is the same. The synthesis approach is

somewhat more heuristic in nature, while the decomposition approach is more algorithmic. Each of these approaches is illustrated in the following sections.

### 7.3.1 Deriving Candidate Key(s) by Synthesis

The steps involved in the heuristic procedure to progressively synthesize a candidate key from a set of FDs are listed in Table 7.2 (see steps 1–6). The following example demonstrates the use of this procedure.

Derivation of First Candidate Key of URS	
Step 1	Derive the minimal cover $F_c$ for the set of functional dependencies (FDs), $F$ , that prevails over the URS.
Step 2	Select the FD with maximum number of attributes constituting the determinant as the starting point. Let us call this FD as the target FD, and the determinant of this first FD as the target determinant (TD1).
Note: If more than one such target FD exists, select one of them for now.	
Compute the attribute closure $[TD1 \mid F_c]$ .	
Step 3	If $TD1^+$ is precisely the set of all attributes of URS, then TD1 is a candidate key of URS. If so, skip to step 7.
Step 4	If $TD1^+$ is not precisely the set of all attributes of URS, select a functional dependency whose determinant is not a subset of $TD1^+$ as the next target FD—the determinant of this FD will be TD2.
Compute the attribute closure $[TD2 \mid F_c]$ .	
Step 5	If $TD2^+$ is precisely the set of all attributes of URS, then TD2 is a candidate key of URS. Otherwise, if $\{TD1^+ \cup TD2^+\}$ is precisely the set of all attributes of URS, then $\{TD1 \cup TD2\}$ is a candidate key of URS. If either one is true, skip to step 7.
Step 6	Otherwise, repeat steps 4 and 5 using the next target FD. Repeat steps 4 and 5 until an attribute set K is derived such that $K^+$ is precisely the set of all attributes of URS. Then, K is a candidate key of URS.
Derivation of Other Candidate Key(s) of URS	
Step 7	If $F_c$ contains an FD, $fdx$ , where a candidate key of URS is a dependent, then the determinant of $fdx$ is also a candidate key of URS.
Step 8	When a candidate key of URS is a composite attribute, for each key attribute (atomic or composite), evaluate if the key attribute is a dependent in an FD, $fdy$ , in $F_c$ . If so, then the determinant of $fdy$ , by the rule of pseudotransitivity, can replace the key attribute under consideration, thus yielding additional candidate key(s) of URS.
Step 9	Repetition of steps 7 and 8 for every candidate key of URS will systematically reveal all the other candidate key(s), if any, of URS.

**TABLE 7.2** A heuristic for the derivation of candidate key(s) by synthesis, given a URS and a set of functional dependencies,  $F$ , that prevails over it

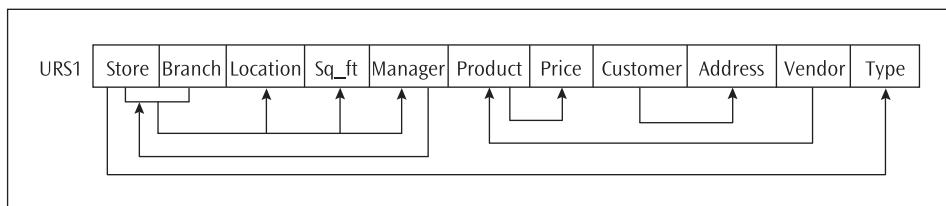
Suppose there is a set of eight functional dependencies, F, as follows:

fd1: {Store, Branch} → Location;	fd5: Product → Price;
fd2: Customer → Address;	fd6: {Store, Branch} → Manager;
fd3: Vendor → Product;	fd7: Manager → {Store, Branch};
fd4: {Store, Branch} → Sq_ft;	fd8: Store → Type;

Here is a universal relation schema that includes all these FDs:

**URS1 (Store, Branch, Location, Customer, Address, Vendor, Product, Sq\_ft, Price, Manager, Type)**

The set of FDs in F may also be expressed via a dependency diagram, as shown in Figure 7.5.



**FIGURE 7.5** Dependency diagram for the relation schema, URS1

### 7.3.1.1 Deriving the First Candidate Key by Synthesis

The following steps demonstrate the application of the synthesis approach to derive the first candidate key of URS1 under the set of constraints in F. Section 7.3.1.2 shows the steps to derive the remaining candidate keys of URS1.

**Step 1.** Derive the minimal cover,  $F_c$ , for the set of functional dependencies, F, that prevails over the URS.

There are no redundant attributes or redundant FDs in F. The reader can verify this using the algorithm given at the end of Section 7.2.3.2. Thus, in this example,  $F_c = F$ .

**Step 2.** Select the FD with maximum number of attributes constituting the determinant as the starting point. Let us call this FD the target FD, and let us call the determinant of this first FD the target determinant (TD1). Note: If more than one such target FD exists, select one of them for now.

Compute the attribute closure  $[TD1 \mid F_c]$ .

Target FD fd1: {Store, Branch} → Location;      Target determinant (TD1): {Store, Branch}

Start:  $\{Store, Branch\}^+ = \{Store, Branch\}$

First iteration through F:

- In fd1, the determinant {Store, Branch} is a subset of  $\{Store, Branch\}^+$ —so,  $\{Store, Branch\}^+ = \{Store, Branch\}^+ \cup \{Location\} = \{Store, Branch, Location\}$ .
- In fd2 and fd3, the respective determinants, Customer and Vendor, are not subsets of  $\{Store, Branch\}^+$ —so, no change in  $\{Store, Branch\}^+$ .

- In fd4, the determinant  $\{\text{Store, Branch}\}$  is a subset of  $\{\text{Store, Branch}\}^+$ —so,  $\{\text{Store, Branch}\}^+ = \{\text{Store, Branch}\}^+ \cup \text{Sq\_ft} = \{\text{Store, Branch, Location, Sq\_ft}\}$ .
- In fd5, the determinant Product is not a subset of  $\{\text{Store, Branch}\}^+$ —so, no change in  $\{\text{Store, Branch}\}^+$ .
- In fd6, the determinant  $\{\text{Store, Branch}\}$  is a subset of  $\{\text{Store, Branch}\}^+$ —so,  $\{\text{Store, Branch}\}^+ = \{\text{Store, Branch}\}^+ \cup \text{Manager} = \{\text{Store, Branch, Location, Sq\_ft, Manager}\}$ .
- In fd7, the determinant  $\{\text{Manager}\}$  is a subset of  $\{\text{Store, Branch}\}^+$ —so,  $\{\text{Store, Branch}\}^+ = \{\text{Store, Branch}\}^+ \cup \{\text{Store, Branch}\} = \{\text{Store, Branch, Location, Sq\_ft, Manager}\}$ .
- In fd8, the determinant  $\{\text{Store}\}$  is a subset of  $\{\text{Store, Branch}\}^+$ —so,  $\{\text{Store, Branch}\}^+ = \{\text{Store, Branch}\}^+ \cup \{\text{Type}\} = \{\text{Store, Branch, Location, Sq\_ft, Manager, Type}\}$ .
- First iteration through F completed.

Second iteration through F:

- No further change in the Closure  $[\{\text{Store, Branch}\} \mid F]$  due to fd1 through fd8; so step 2 terminates.

End:

$$\{\text{Store, Branch}\}^+ = \{\text{Store, Branch, Location, Sq\_ft, Manager, Type}\}$$

**Step 3.** If  $\text{TD1}^+$  is precisely the set of all attributes of URS, then TD1 is a candidate key of URS. If so, skip to step 7.

$\{\text{Store, Branch}\}^+$  is not precisely the set of all attributes of URS1; therefore,  $\{\text{Store, Branch}\}$  is not a candidate key of URS1.

**Step 4.** If  $\text{TD1}^+$  is not precisely the set of all attributes of URS, select a FD whose determinant is not a subset of  $\text{TD1}^+$  as the next target FD. The determinant of this FD will be TD2.

Compute the attribute closure  $[\text{TD2} \mid F_c]$ .

Target FD – fd2:  $\text{Customer} \rightarrow \text{Address}$ ;

Target determinant (TD2):  $\text{Customer}$

Start:  $\text{Customer}^+ = \text{Customer}$

First iteration through F:

- In all FDs in F but fd2, the respective determinants are not subsets of  $\text{Customer}^+$ —so, no change in  $\text{Customer}^+$ ; due to fd1, fd3, fd4, fd5, fd6, fd7, and fd8.
- In fd2, the determinant  $\text{Customer}$  is a subset of  $\text{Customer}^+$ —so,  $\{\text{Customer}\}^+ = \text{Customer}^+ \cup \text{Address} = \{\text{Customer, Address}\}$ .

Second iteration through F:

- No further change in the Closure  $[\text{Customer} \mid F]$ ; so, step 4 terminates.

End:

$$\text{Customer}^+ = \{\text{Customer, Address}\}$$

**Step 5.** If  $TD2^+$  is precisely the set of all attributes of URS, then  $TD2$  is a candidate key of URS. Otherwise, if  $\{TD1^+ \cup TD2^+\}$  is precisely the set of all attributes of URS, then  $\{TD1 \cup TD2\}$  is a candidate key of URS.

If either one is true, skip to step 7. Since either one is not true, continue step 5.

**Customer**<sup>+</sup> is not precisely the set of all attributes of URS1; therefore, **Customer** is not a candidate key of URS1.

$\{\text{Store, Branch}\}^+ \cup \text{Customer}^+ = \{\text{Store, Branch, Location, Sq_ft, Manager, Type, Customer, Address}\}$ .

$\{\text{Store, Branch}\}^+ \cup \text{Customer}^+$  is not precisely the set of all attributes of URS1; therefore,  $\{\text{Store, Branch, Customer}\}$  is not a candidate key of URS1 either.

**Step 6.** Since at this point no candidate key has emerged, repeat steps 4 and 5 using the next target FD. Repeat steps 4 and 5 until an attribute set K is derived such that  $K^+$  is precisely the set of all attributes of URS.

Then, and only then, K is a candidate key of URS.

Target FD – fd3: **Vendor** → **Product**;

Target determinant (TD3): **Vendor**

Start:  $\text{Vendor}^+ = \text{Vendor}$

First iteration through  $F_c$ :

- In all FDs in F except fd3 and fd5, the respective determinants are not subsets of  $\text{Vendor}^+$ —so, no change in  $\text{Vendor}^+$  due to fd1, fd2, fd4, fd6, fd7, and fd8.
- In fd3, the determinant **Vendor** is a subset of  $\text{Vendor}^+$ —so,  $\text{Vendor}^+ = \text{Vendor}^+ \cup \text{Product} = \{\text{Vendor, Product}\}$ .
- In fd5, the determinant **Product** is a subset of  $\text{Vendor}^+$ —so,  $\text{Vendor}^+ = \text{Vendor}^+ \cup \text{Price} = \{\text{Vendor, Product, Price}\}$ .
- First iteration through  $F_c$  completed.

Second iteration through  $F_c$ :

- No further change in the Closure  $[\text{Vendor} \mid F]$ ; so, step 6 terminates.

End:

$\text{Vendor}^+ = \{\text{Vendor, Product, Price}\}$

$\text{Vendor}^+$  is not precisely the set of all attributes of URS1; therefore, **Vendor** is not a candidate key of URS1.

$\{\text{Store, Branch}\}^+ \cup \text{Customer}^+ \cup \text{Vendor}^+ = \{\text{Store, Branch, Location, Sq_ft, Manager, Type, Customer, Address, Vendor, Product, Price}\}$

$\{\text{Store, Branch}\}^+ \cup \text{Customer}^+ \cup \text{Vendor}^+$  is indeed precisely the set of all attributes of URS1; therefore,  $\{\text{Store, Branch, Customer, Vendor}\}$  is a candidate key of URS1.

Observe that  $\{\text{Store, Branch, Customer, Vendor}\}$  is a superkey of URS1 and that none of its proper subsets is a superkey of URS1, confirming that the irreducible superkey  $\{\text{Store, Branch, Customer, Vendor}\}$  is a candidate key of URS1.

### 7.3.1.2 Deducing the Other Candidate Key(s) of URS1 by Synthesis

**Step 7.** If F contains an FD,  $fd_x$ , where a candidate key of URS is a dependent, then the determinant of  $fd_x$  is also a candidate key of URS.

There are no FDs in F, where  $\{\text{Store, Branch, Customer, Vendor}\}$  is a dependent. Therefore, proceed to step 8.

**Step 8.** When a candidate key of URS is a composite attribute, evaluate if the key attribute is a dependent in an FD,  $fd_y$ , in  $F$  for each key attribute (atomic or composite). If so, then the determinant of  $fd_y$ , by the pseudotransitivity rule, can replace the key attribute under consideration, thus yielding additional candidate key(s) of URS.

Since in  $\text{Manager} \rightarrow \{\text{Store}, \text{Branch}\}$  (see  $fd_7$ ), the dependent,  $\{\text{Store}, \text{Branch}\}$ , is a subset of the candidate key  $\{\text{Store}, \text{Branch}, \text{Customer}, \text{Vendor}\}$ , using the pseudotransitivity rule,  $\{\text{Manager}, \text{Customer}, \text{Vendor}\}$  is extracted as another candidate key of URS1.

Since there is no other key attribute, atomic or composite, of the candidate key,  $\{\text{Store}, \text{Branch}, \text{Customer}, \text{Vendor}\}$ , that is a dependent in any other FD in  $F$ , continue to step 9.

**Step 9.** Repetition of steps 7 and 8 for every candidate key of URS will systematically reveal all the other candidate key(s), if any, of URS.

The only other candidate key of URS1 so far is  $\{\text{Manager}, \text{Customer}, \text{Vendor}\}$ . In this case, application of steps 7 and 8 do not yield any more candidate keys.

In summary, URS1, where the set of FDs denoted by  $F$  prevail, has two candidate keys. They are:

$\{\text{Store}, \text{Branch}, \text{Customer}, \text{Vendor}\}$  and  $\{\text{Manager}, \text{Customer}, \text{Vendor}\}$

### 7.3.2 Deriving Candidate Keys by Decomposition

Since, by definition, a relation schema must have a superkey, every relation schema has at least one default superkey—the set of all its attributes. In the decomposition method for deriving a candidate key, for a given URS and the set of FDs that prevail over it, we start by setting the collection of all attributes of URS as its superkey,  $K$ . We then arbitrarily remove one attribute at a time from  $K$  and check if the collection of the remaining attributes,  $K'$ , continues to satisfy the uniqueness condition of a superkey in URS. The checking is done using the FDs in  $F$  that prevail over URS. If the test fails, the attribute removed is restored to  $K'$ . The process continues until  $K'$  is reduced to a superkey that is not further reducible, thus becoming a candidate key of URS.

When URS has multiple candidate keys, the candidate key returned through this process of decomposition of a superkey depends on the order in which attributes are removed from URS. The method is algorithmically stated in Table 7.3. The following example demonstrates the application of this algorithm to generate the first candidate key of URS.

Derivation of First Candidate Key of URS	
Step 1	Set superkey, $K$ , of URS = $\{A_1, A_2, A_3, \dots, A_n\}$
Step 2	Remove an attribute $A_i$ , ( $i = 1, 2, 3, \dots, n$ ) from URS such that $\{K - A_i\}$ is still a superkey, $K'$ , of URS.
Note: In order for $K'$ to be a superkey of URS, the FD: $(K' \rightarrow A_i)$ should persist in $F^+$ .	
Step 3	Repeat step 2 above recursively until $K'$ is further irreducible.
The irreducible $K'$ is a candidate key of URS under the set of FDs, $F$ .	

**TABLE 7.3** An algorithm for the derivation of candidate key(s) by decomposition of the superkey given a universal relation schema:  $URS = \{A_1, A_2, A_3, \dots, A_n\}$  and the set of FDs over URS:  $F = \{fd_1, fd_2, fd_3, \dots, fd_m\}$

**Derivation of First Candidate Key of URS**

- Step 4 If F contains an FD,  $fd_x$ , where a candidate key of URS is a dependent, then the determinant of  $fd_x$  is also a candidate key of URS.
- Step 5 When a candidate key of URS is a composite attribute, for each key attribute (atomic or composite):
  - Evaluate if the key attribute is a dependent in an FD,  $fd_y$ , in F.
  - If so, then the determinant of  $fd_y$ , by the rule of pseudotransitivity, can replace the key attribute under consideration, thus yielding additional candidate key(s) of URS.
- Step 6 Repetition of steps 4 and 5 above for every candidate key of URS will systematically reveal all the other candidate key(s), if any, of URS.

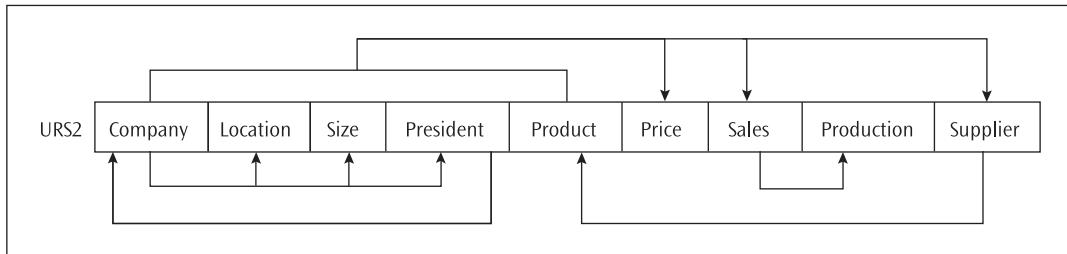
**TABLE 7.3** An algorithm for the derivation of candidate key(s) by decomposition of the superkey given a universal relation schema:  $URS = \{A_1, A_2, A_3, \dots, A_n\}$  and the set of FDs over URS:  $F = \{fd_1, fd_2, fd_3, \dots, fd_m\}$  (continued)

**7.3.2.1 Deriving the First Candidate Key by Decomposition**

Suppose there is a set of nine functional dependencies, F, as follows:

fd1: <b>Company</b> → <b>Location</b> ;	fd2: <b>Company</b> → <b>Size</b> ;	fd3: <b>Company</b> → <b>President</b> ;
fd4: <b>{Product, Company}</b> → <b>Price</b> ;	fd5: <b>Sales</b> → <b>Production</b> ;	fd6: <b>{Company, Product}</b> → <b>Sales</b> ;
fd7: <b>{Product, Company}</b> → <b>Supplier</b> ;	fd8: <b>Supplier</b> → <b>Product</b> ;	fd9: <b>President</b> → <b>Company</b>

Here is a universal relation schema that includes all these FDs:  
**URS2 (Company, Location, Size, President, Product, Price, Sales, Production, Supplier)**  
 The set of FDs in F may also be expressed via a dependency diagram, as shown in Figure 7.6.



**FIGURE 7.6** Dependency diagram for the relation schema URS2

Here are the steps to determine the candidate key(s) of URS2 by decomposing the superkey:

**Step 1.** Set superkey, K, of URS =  $\{A_1, A_2, A_3, \dots, A_n\}$

**K = {Company, Location, Size, President, Product, Price, Sales, Production, Supplier}**

**Step 2.** Remove an attribute  $A_i$ , ( $i = 1, 2, 3, \dots, n$ ) from URS such that  $\{K - A_i\}$  is still a superkey,  $K'$ , of URS.

Note: In order for  $K'$  to be a superkey of URS, the FD:  $(K' A_i)$  must persist in  $F^+$ .

Let us arbitrarily remove the attribute from the left end of  $K$ —that is, remove **Company** from  $K$ .

Then,

$$K' = (K - \text{Company}) = \{\text{Location, Size, President, Product, Price, Sales, Production, Supplier}\}$$

In order to know if  $K'$  is a superkey of URS2, we need to check if  $(K' \rightarrow \text{Company})$  is in  $F^+$ .

Since  $(\text{President} \rightarrow \text{Company})$  is in  $F$  (see fd9) and  $(\text{President} \subset K')$ , using the augmentation rule of Armstrong's axioms, we infer that  $(K' \rightarrow \text{Company})$  is in  $F^+$ . Therefore,  $K'$  is a superkey of URS2.

**Step 3.** Repeat step 2 recursively until  $K'$  is further irreducible.

Continue removing attributes (arbitrarily) from the left of  $K'$ . Accordingly, next remove the attribute **Location**.

Then,

$$K' = (K' - \text{Location}) = \{\text{Size, President, Product, Price, Sales, Production, Supplier}\}$$

The pared-down  $K'$  remains a superkey of URS2 because, since fd1:  $(\text{Company} \rightarrow \text{Location})$  and  $(K' \rightarrow \text{Company})$  are in  $F^+$ , then so is  $(K' \rightarrow \text{Location})$ .

Likewise, using fd2, we can conclude that after the removal of **Size** from  $K'$ ,

$$K' = \{\text{President, Product, Price, Sales, Production, Supplier}\}$$

continues to remain a superkey of URS2.

Next, when **President** is removed from the current  $K'$ , the pared-down  $K' = \{\text{Product, Price, Sales, Production, Supplier}\}$  fails to persist as a superkey of URS2 because  $(K' \rightarrow \text{Company, Location, Size, President})$  is no longer present in  $F^+$ .

So, **President** is added back to  $K'$ , restoring  $K'$  as a superkey of URS2:

$$K' = \{\text{President, Product, Price, Sales, Production, Supplier}\}$$

Even when **Product** is removed from  $K'$ , the reduced  $K' = \{\text{President, Price, Sales, Production, Supplier}\}$  continues to remain as a superkey of URS2 because since  $\text{Supplier} \rightarrow \text{Product}$  (fd8), we can infer that  $K' \rightarrow \text{Product}$ . In addition,  $K'$  will also functionally determine all other dependent attributes in FDs where **Product** is the determinant. It just happens that in this example there are no non-trivial FDs in  $F^+$  where **Product** is the determinant.

What happens when **Price** is removed from the current  $K'$ ? Does the trimmed-down  $K' = \{\text{President, Sales, Production, Supplier}\}$  persist as a superkey of URS2?

Since **Price** is not a part of a determinant in any FD in  $F$ , it is sufficient if  $K' \rightarrow \text{Price}$  is in  $F^+$  in order for  $K'$  to remain a superkey of URS2. From fd8, fd9, and fd4, we can conclude that  $\{\text{President, Supplier}\} \rightarrow \text{Price}$ . Since  $\{\text{President, Supplier}\} \subset K'$ , we further infer that  $K' \rightarrow \text{Price}$  is in  $F^+$ . Therefore,  $K'$  remains a superkey of URS2.

Along similar lines of recursive refinement by dropping attributes from  $K'$  from left to right, it can be shown that **Sales** and **Production** can be removed from  $K'$  while the successively reduced  $K'$  persists as the superkey of URS2. However, removal of **Supplier** from  $K'$  at the end results in  $K'$  not persisting as a superkey of URS2. As a consequence, **Supplier** is added back to  $K'$ , restoring  $K'$  as a superkey of URS2. At this point, it can be seen that:

$$K' = \{\text{President, Supplier}\}$$

is a superkey of URS2 and does not remain a superkey of URS2 if further reduced. Consequently,  $\{\text{President, Supplier}\}$  becomes a candidate key (CK1) of URS2.

### 7.3.2.2 Deducing the Other Candidate Key(s) of URS2

**Step 4.** If F contains an FD,  $fdx$ , where a candidate of URS is a dependent, then the determinant of  $fdx$  is also a candidate key of URS.

There are no FDs in F, where **{President, Supplier}** is a dependent. Therefore, proceed to step 5.

**Step 5.** When a candidate key of URS is a composite attribute, evaluate if the key attribute is a dependent in an FD,  $fdy$ , in F for each key attribute (atomic or composite). If so, then the determinant of  $fdy$ , by the pseudotransitivity rule, can replace the key attribute under consideration, thus yielding additional candidate key(s) of URS.

Given that **{President, Supplier}** is a candidate key of URS2, **President** is a key attribute and **Supplier** is a key attribute.

**President** is a dependent in  $fd3: \text{Company} \rightarrow \text{President}$ . Therefore, **{Company, Supplier}** is a candidate key (CK2) of URS2.

**Supplier** is a dependent in  $fd7: \{\text{Product}, \text{Company}\} \rightarrow \text{Supplier}$ . Therefore, **{President, Product, Company}** and **{Company, Product, Company}** result from the application of the pseudotransitivity rule on CK1 and CK2, respectively. However, since **President**  $\rightarrow$  **Company** and **Company**  $\rightarrow$  **Company**, **Company** is redundant in both cases. Therefore, **{President, Product}** and **{Company, Product}** become candidate keys (CK3 and CK4) of URS2.

To sum up, in this example we have discovered the following four candidate keys for URS2 under the set of functional dependencies, F:

- {Company, Product};**
- {Company, Supplier};**
- {President, Product};**
- {President, Supplier}**

Observe that any one of these could have been the first candidate key of URS2 derived through the process of superkey decomposition, depending on the order of removal of attributes in the derivation process.

### 7.3.3 Deriving a Candidate Key—Another Example

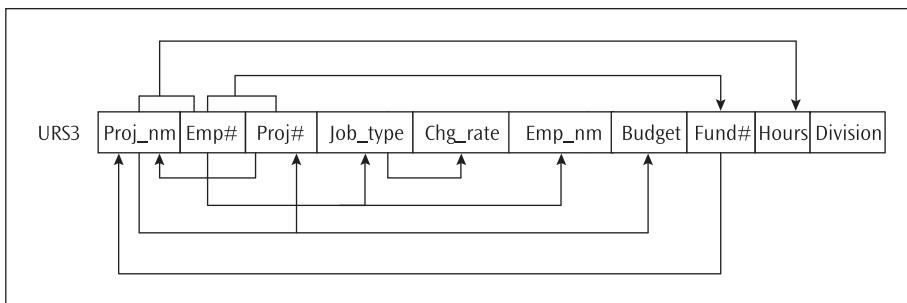
A third example demonstrating the derivation of candidate key(s) from a set of FDs is presented in this section. Given the universal relation schema:

**URS3 (Proj\_nm, Emp#, Proj#, Job\_type, Chg\_rate, Emp\_nm, Budget, Fund#, Hours, Division)**  
and the following set of functional dependencies, F, prevailing over URS3:

fd1: <b>Proj#</b> $\rightarrow$ <b>Proj_nm</b> ;	fd2: <b>Job_type</b> $\rightarrow$ <b>Chg_rate</b> ;	fd3: <b>Emp#</b> $\rightarrow$ <b>Emp_nm</b> ;
fd4: <b>Proj_nm</b> $\rightarrow$ <b>Budget</b> ;	fd5: <b>Fund#</b> $\rightarrow$ <b>Proj_nm</b> ;	fd6: <b>{Proj_nm, Emp#}</b> $\rightarrow$ <b>Hours</b> ;
fd7: <b>Proj_nm</b> $\rightarrow$ <b>Proj#</b> ;	fd8: <b>Emp#</b> $\rightarrow$ <b>Job_type</b> ;	fd9: <b>{Proj#, Emp#}</b> $\rightarrow$ <b>Fund#</b>

what is/are the candidate key(s) of URS3?

The set of FDs in F may also be expressed via a dependency diagram, as shown in Figure 7.7.



**FIGURE 7.7** Dependency diagram for the relation schema, URS3

For this example, the solution using both the decomposition technique and the synthesis technique are demonstrated.

### 7.3.3.1 Deriving a Candidate Key by Decomposition

**Step 1.** Set superkey, K, of URS = {A<sub>1</sub>, A<sub>2</sub>, A<sub>3</sub>, …., A<sub>n</sub>}

$$K = \{\text{Proj\_nm, Emp\#, Proj\#, Job\_type, Chg\_rate, Emp\_nm, Budget, Fund\#, Hours, Division}\}$$

**Step 2.** Remove an attribute A<sub>i</sub>, (i = 1, 2, 3, …., n) from URS such that {K – A<sub>i</sub>} is still a superkey, K', of URS. Note: In order for K' to be a superkey of URS, the FD: (K' → A<sub>i</sub>) must persist in F<sup>+</sup>.

Let us arbitrarily remove the attribute from the right end of K—that is, remove **Division** from K.

Then:

$$K' = (K - \text{Division}) = \{\text{Proj\_nm, Emp\#, Proj\#, Job\_type, Chg\_rate, Emp\_nm, Budget, Fund\#, Hours}\}$$

In order to know if K' is a superkey of URS3, we need to check if the FD (K' → Division) is in F<sup>+</sup>.

A perusal of F indicates that none of the subsets of K' functionally determines **Division**. Therefore, (K' → Division) cannot be present in F<sup>+</sup> either. Accordingly, K' is not a superkey of URS3. In fact, **Division** cannot be removed from K in order to construct a superkey of URS3. More precisely, any superkey of URS3 must include **Division**.

Removal of **Hours** from K to create K' results in:

$$K' = (K - \text{Hours}) = \{\text{Proj\_nm, Emp\#, Proj\#, Job\_type, Chg\_rate, Emp\_nm, Budget, Fund\#, Division}\}$$

and K' is now a superkey of URS3 since the FD (K' → Hours) exists in F<sup>+</sup>.

Working along similar lines, using fd9, fd4, fd3, fd2, fd8, and fd7, respectively, we can see that removal of the attributes **Fund#**, **Budget**, **Emp\_nm**, **Chg\_rate**, **Job\_type** and **Proj#** pares down K' to **{Proj\_nm, Emp#, Division}** without affecting its role as a superkey of URS3. Furthermore, as a superkey, K' is further irreducible because none of its proper subsets, **{Proj\_nm}**, **{Emp#}**, **{Division}**, **{Proj\_nm, Emp#}**, **{Proj\_nm, Division}**, or **{Emp#, Division}**, qualifies as a superkey of URS3. Thus, we infer that **{Proj\_nm, Emp#, Division}** is a candidate key (CK1) of URS3.

### 7.3.3.2 Deducing the Other Candidate Key(s) of URS3

**Step 3.** If F contains an FD,  $fd_x$ , where a candidate of URS is a dependent, then the determinant of  $fd_x$  is also a candidate key of URS.

There are no FDs in F, where  $\{\text{Proj\_nm}, \text{Emp\#}, \text{Division}\}$  is a dependent. Therefore, proceed to step 4.

**Step 4.** When a candidate key of URS is a composite attribute, evaluate if the key attribute is a dependent in an FD,  $fd_y$ , in F for each key attribute (atomic or composite). If so, then the determinant of  $fd_y$ , by the pseudotransitivity rule, can replace the key attribute under consideration, thus yielding additional candidate key(s) of URS.

Given that  $\{\text{Proj\_nm}, \text{Emp\#}, \text{Division}\}$  is a candidate key of URS3, its proper subsets,  $\{\text{Proj\_nm}\}$ ,  $\{\text{Emp\#}\}$ ,  $\{\text{Division}\}$ ,  $\{\text{Proj\_nm}, \text{Emp\#}\}$ ,  $\{\text{Proj\_nm}, \text{Division}\}$ ,  $\{\text{Emp\#}, \text{Division}\}$ , are all key attributes. All except  $\{\text{Proj\_nm}\}$  do not participate in any FD in F as a dependent, and  $\{\text{Proj\_nm}\}$  is a dependent in  $fd_1$  as well as  $fd_5$ . Therefore, by applying the pseudotransitivity rule, we deduce from  $fd_1$  and  $fd_5$  that  $\{\text{Proj\#}, \text{Emp\#}, \text{Division}\}$  and  $\{\text{Fund\#}, \text{Emp\#}, \text{Division}\}$  are also candidate keys of URS3.

### 7.3.3.3 Deriving Candidate Key(s) by Synthesis

The following steps demonstrate the application of the synthesis approach to derive the candidate key(s) of URS3 under the set of constraints  $F_c$ .

**Step 1.** Derive the minimal cover  $F_c$  for the set of functional dependencies, F, that prevails over the URS.

There are no redundant attributes or redundant FDs in F. The reader can verify this using the algorithm given at the end of Section 7.3.2.2. Thus, in this example,  $F_c = F$ .

**Step 2.** Select the FD with maximum number of attributes constituting the determinant as the starting point. Let us call this FD the target FD, and let us call the determinant of this first FD the target determinant (TD1). Note: If more than one such target FD exists, select one of them for now.

Compute the attribute closure  $[TD1 \mid F_c]$ .

Target FD –  $fd_6: \{\text{Proj\_nm}, \text{Emp\#}\} \rightarrow \text{Hours}$ ;

Target determinant (TD1):  $\{\text{Proj\_nm}, \text{Emp\#}\}$

Start:  $\{\text{Proj\_nm}, \text{Emp\#}\}^+ = \{\text{Proj\_nm}, \text{Emp\#}, \text{Hours}\}$

First iteration through  $F_c$ :

- In  $fd_1$  and  $fd_2$ , the respective determinants,  $\text{Proj\#}$  and  $\text{Job\_type}$ , are not subsets of  $\{\text{Proj\_nm}, \text{Emp\#}\}^+$ —so, no change in  $\{\text{Proj\_nm}, \text{Emp\#}\}^+$ .
- In  $fd_3$ , the determinant  $\{\text{Emp\#}\}$  is a subset of  $\{\text{Proj\_nm}, \text{Emp\#}\}^+$ —so,  $\{\text{Proj\_nm}, \text{Emp\#}\}^+ = \{\text{Proj\_nm}, \text{Emp\#}\}^+ \cup \text{Emp\_nm} = \{\text{Proj\_nm}, \text{Emp\#}, \text{Hours}, \text{Emp\_nm}\}$ .
- In  $fd_4$ , the determinant  $\{\text{Proj\_nm}\}$  is a subset of  $\{\text{Proj\_nm}, \text{Emp\#}\}^+$ —so,  $\{\text{Proj\_nm}, \text{Emp\#}\}^+ = \{\text{Proj\_nm}, \text{Emp\#}\}^+ \cup \text{Budget} = \{\text{Proj\_nm}, \text{Emp\#}, \text{Hours}, \text{Emp\_nm}, \text{Budget}\}$ .
- In  $fd_5$ , the determinant  $\text{Fund\#}$  is not a subset of  $\{\text{Proj\_nm}, \text{Emp\#}\}^+$ —so, no change in  $\{\text{Proj\_nm}, \text{Emp\#}\}^+$ .
- $fd_6$  has already been considered at the start of the first iteration through F—so, no change in  $\{\text{Proj\_nm}, \text{Emp\#}\}^+$  at this time.

- In fd7, the determinant  $\{\text{Proj\_nm}\}$  is a subset of  $\{\text{Proj\_nm}, \text{Emp\#}\}^+$ —so,  $\{\text{Proj\_nm}, \text{Emp\#}\}^+ = \{\text{Proj\_nm}, \text{Emp\#}\}^+ \cup \text{Proj\#} = \{\text{Proj\_nm}, \text{Emp\#}, \text{Hours}, \text{Emp\_nm}, \text{Budget}, \text{Proj\#}\}$ .
- In fd8, the determinant  $\{\text{Emp\#}\}$  is a subset of  $\{\text{Proj\_nm}, \text{Emp\#}\}^+$ —so,  $\{\text{Proj\_nm}, \text{Emp\#}\}^+ = \{\text{Proj\_nm}, \text{Emp\#}\}^+ \cup \text{Job\_type} = \{\text{Proj\_nm}, \text{Emp\#}, \text{Hours}, \text{Emp\_nm}, \text{Budget}, \text{Proj\#}, \text{Job\_type}\}$ .
- In fd9, the determinant  $\{\text{Proj\#, Emp\#}\}$  is a subset of  $\{\text{Proj\_nm}, \text{Emp\#}\}^+$ —so,  $\{\text{Proj\_nm}, \text{Emp\#}\}^+ = \{\text{Proj\_nm}, \text{Emp\#}\}^+ \cup \text{Fund\#} = \{\text{Proj\_nm}, \text{Emp\#}, \text{Hours}, \text{Emp\_nm}, \text{Budget}, \text{Proj\#, Job\_type, Fund\#}\}$ .
- First iteration through  $F_c$  completed.

Second iteration through  $F_c$ :

- fd1 does not lead to any changes to  $\{\text{Proj\_nm}, \text{Emp\#}\}^+$  since both the determinant and dependent in fd1 ( $\text{Proj\#}$  and  $\text{Proj\_nm}$ , respectively) are already present in  $\{\text{Proj\_nm}, \text{Emp\#}\}^+$ .
- In fd2, the determinant  $\{\text{Job\_type}\}$  is a subset of  $\{\text{Proj\_nm}, \text{Emp\#}\}^+$ —so,  $\{\text{Proj\_nm}, \text{Emp\#}\}^+ = \{\text{Proj\_nm}, \text{Emp\#}\}^+ \cup \text{Chg\_rate} = \{\text{Proj\_nm}, \text{Emp\#}, \text{Hours}, \text{Emp\_nm}, \text{Budget}, \text{Proj\#, Job\_type, Fund\#, Chg\_rate}\}$ .
- No further change in the Closure  $[\{\text{Proj\_nm}, \text{Emp\#}\} \mid F]$  due to fd3 through fd9; so step 2 terminates.

$\{\text{Proj\_nm}, \text{Emp\#}\}^+ = \{\text{Proj\_nm}, \text{Emp\#, Hours, Emp\_nm, Budget, Proj\#, Job\_type, Fund\#, Chg\_rate}\}$

**Step 3.** If  $\text{TD1}^+$  is precisely the set of all attributes of URS, then  $\text{TD1}$  is a candidate key of URS. If so, skip to step 7.

$\{\text{Proj\_nm}, \text{Emp\#}\}^+$  is not precisely the set of all attributes of URS3; therefore,  $\{\text{Proj\_nm}, \text{Emp\#}\}$  is not a candidate key of URS3.

**Step 4.** If  $\text{TD1}^+$  is not precisely the set of all attributes of URS, select an FD whose determinant is not a subset of  $\text{TD1}^+$  as the next target FD; the determinant of this FD will be  $\text{TD2}$ .

Compute the attribute closure  $[\text{TD2} \mid F_c]$ .

There are no FDs in  $F_c$  whose determinant is not a subset of  $\{\text{Proj\_nm}, \text{Emp\#}\}^+$ .

Observe that the attribute **Division** does not participate in any FD in  $F$  while it is present in URS3. This indicates the independent state of this attribute in URS3. One way to formalize the presence of the attribute **Division** in URS3 is to portray it through an implicit trivial FD in  $F^+$ —viz., **Division** → **Division**. Therefore, let  $\text{fd10}$  be **Division** → **Division**.

Target FD – fd10: **Division** → **Division**;

Target determinant (TD2): **Division**

Start:  $\text{Division}^+ = \text{Division}$ ;

First iteration through  $F$ :

- In all FDs in  $F$  but  $\text{fd10}$ , the respective determinants are not subsets of  $\text{Division}^+$ —so, no change in  $\text{Division}^+$  due to fd1, fd2, fd3, fd4, fd5, fd6, fd7, fd8, and fd9.
- Since  $\text{fd10}$  is the target FD, it is already accounted for in  $\text{Division}^+$ .
- No further change in the Closure  $[\text{Division} \mid F]$ ; so, step 4 terminates.

End: **Division<sup>+</sup>** = **Division**

**Step 5.** If TD2<sup>+</sup> is precisely the set of all attributes of URS, then TD2 is a candidate key of URS. Otherwise, if {TD1<sup>+</sup> ∪ TD2<sup>+</sup>} is precisely the set of all attributes of URS, then {TD1 ∪ TD2} is a candidate key of URS.

If either one is true, skip to step 7.

**Division<sup>+</sup>** is not precisely the set of all attributes of URS3; therefore, **Division** is not a candidate key of URS3.

{Proj\_nm, Customer#}<sup>+</sup> ∪ Division<sup>+</sup> = {Proj\_nm, Emp#, Hours, Emp\_nm, Budget, Proj#, Job\_type, Fund#, Chg\_rate, Division}

{Proj\_nm, Customer#}<sup>+</sup> ∪ Division<sup>+</sup> is precisely the set of all attributes of URS3; therefore, {Proj\_nm, Customer#, Division} is a candidate key of URS3 under F.

Observe that {Proj\_nm, Customer#, Division} is a superkey of URS3 and that none of its proper subsets is a superkey of URS3, confirming that the irreducible superkey {Proj\_nm, Customer#, Division} is a candidate key of URS3.<sup>10</sup> Had we chosen fd9: {Proj#, Emp#} → Fund# as the target FD and therefore {Proj#, Emp#} as the target determinant (TD1) in the first step, we would have ended up with {Proj#, Customer#, Division} as the candidate key of URS3 under F.

#### 7.3.3.4 Deducing the Other Candidate Key(s) of URS3 by Synthesis

The steps involved in discovering other candidate keys of URS3 are the same as those employed in the decomposition method discussed in Section 7.3.3.2 and so are not reiterated here.

In summary, URS3, where the set of FDs denoted by F are preserved, has three candidate keys:

{Proj#, Emp#, Division}  
{Proj\_nm, Emp#, Division}  
{Fund#, Emp#, Division}

#### 7.3.4 Prime and Non-Prime Attributes

Chapter 6 introduced the concepts of key and non-key attributes (see Section 6.3.1). The ideas are reaffirmed here. An attribute, atomic or composite, in a relation schema, R, is called a *key attribute* if it is a proper subset of any candidate key of R. Likewise, attributes that are not subsets of a candidate key of R become non-key attributes. In other words, a *non-key attribute* is not a member of any candidate key of R.

In the example in Section 7.2.3.2, **Company**, **President**, **Product**, and **Supplier**, individually, are all key attributes of URS2 since each one is a proper subset of some candidate key of URS2. On the other hand, **Location**, **Size**, **Price**, **Sales**, and **Production** in URS2 are, individually or in groups, non-key attributes. The composite attribute {Company, Product} is not a key attribute because it is not in the *proper* subset of the candidate key {Company, Product} of URS2. The same is true for every one of the other candidate keys of URS2: {Company, Supplier}; {President, Product}; and {President, Supplier}.

Is {Company, Location} or {Product, Sales} a key attribute of URS2? The answer is “No,” even though one of the attributes in each of these two composite attributes is a key

---

<sup>10</sup>For a review of the properties of superkey and candidate key, see Section 6.3.1 in Chapter 6.

attribute. Any attribute, atomic or composite, in a relation schema R that fails the test for a key attribute (being a proper subset of a candidate key) is a non-key attribute, with the exception of the candidate key(s) of R—which are, in fact, neither key attributes nor non-key attributes in R. This is because a candidate key is a subset of itself and thus fails the test for non-key attribute. It is, however, not a proper subset of itself and thus fails the test for a key attribute.

Based on this discussion, we have an alternative definition for a candidate key from this point forward:

*A candidate key of a relation schema, R, fully functionally determines all attributes of R.*

While the choice of primary key from among the candidate keys is essentially arbitrary, some rules of thumb are often helpful:<sup>11</sup>

- A candidate key with the least number of attributes may be a good choice.
- A candidate key whose attributes are numeric and/or of small sizes may be easy to work with from a developer's perspective.
- A candidate key that is a determinant in a functional dependency in F rather than  $F^+$  may be a good choice because it is probably semantically obvious from the user's perspective.
- Surrogate keys (especially DBMS-developed sequence numbers) should be used only as a last resort because they don't offer semantic reference points to the user community.

Primary keys were discussed in Chapter 6 (see Section 6.3.1). As a refresher, a primary key is an irreducible unique identifier like any other candidate key of a relation schema, but a primary key is in addition bound by the entity integrity constraint—that is, none of the attributes constituting a primary key is allowed to have “null” values. Suppose **{Company, Product}** is the primary key of URS2 in our example. Then, neither **Company** nor **Product** can have null values in any tuple (in any state) of the relation, URS2. Similar to a key attribute, any attribute, atomic or composite, in a relation schema, R, that is a proper subset of the primary key of R is called a **prime attribute**. An attribute of R that is not a member of the primary key is a **non-prime attribute except when it is a candidate key of R**. Any candidate key of R not chosen as the primary key is referred to as an **alternate key** of R and, like the primary key, is neither a prime nor a non-prime attribute of R. Here are some examples that provide additional clarification.

Consider URS2, the example from Section 7.3.2.1:

**URS2 (Company, Location, Size, President, Product, Price, Sales, Production, Supplier)** where, under the FDs specified in F with reference to URS2, the candidate keys are: **{Company, Product}; {Company, Supplier}; {President, Product}; and {President, Supplier}** and the chosen primary key is **{Company, Product}**.

Table 7.4 shows the key and non-key attributes and the prime and non-prime attributes for URS2.

---

<sup>11</sup>No matter which candidate key has been chosen to be the primary key of a relation schema, the normalization process may spontaneously force some changes. We will have an opportunity to observe this phenomenon in Chapter 9.

Role of the Attribute		
Attribute	Key/Non-Key Attribute	Prime/Non-Prime Attribute
Company	Key attribute	Prime attribute
Location	Non-key attribute	Non-prime attribute
Size	Non-key attribute	Non-prime attribute
President	Key attribute	Non-prime attribute
Product	Key Attribute	Prime attribute
{Company, Product}	<i>Candidate key</i>	<i>Primary key</i>
{President, Product}	<i>Candidate key</i>	<i>Alternate key</i>
Price	Non-key attribute	Non-prime attribute
Sales	Non-key attribute	Non-prime attribute
Production	Non-key attribute	Non-prime attribute
Supplier	Key attribute	Non-prime attribute
{Company, Supplier}	<i>Candidate key</i>	<i>Alternate key</i>
{President, Supplier}	<i>Candidate key</i>	<i>Alternate key</i>

Note: Any composite attribute that includes one or more non-key attribute(s) is a non-key attribute.

**TABLE 7.4** Attribute roles in URS2

As another example, consider URS3 from Section 7.3.3:

**URS3 (Proj\_nm, Emp#, Proj#, Job\_type, Chg\_rate, Emp\_nm, Budget, Fund#, Hours, Division)**, where, under the FDs specified in F with reference to URS3, the candidate keys are **{Proj#, Emp#, Division}**; **{Proj\_nm, Emp#, Division}**; and **{Fund#, Emp#, Division}** and the chosen primary key is **{Proj#, Emp#, Division}**.

Table 7.5 shows the key and non-key attributes and the prime and non-prime attributes for URS3.

Role of the Attribute		
Attribute	Key/Non-Key Attribute	Prime/Non-Prime Attribute
Proj_nm	Key attribute	Non-prime attribute
Emp#	Key attribute	Prime attribute
{Proj_nm, Emp#}	Key attribute	Non-prime attribute
Proj#	Key attribute	Prime attribute
{Proj#, Emp#}	Key attribute	Prime attribute
Job_type	Non-key attribute	Non-prime attribute
Chg_rate	Non-key attribute	Non-prime attribute
Emp_nm	Non-key attribute	Non-prime attribute
Budget	Non-key attribute	Non-prime attribute
Fund#	Key attribute	Non-prime attribute
{Fund#, Emp#}	Key attribute	Non-prime attribute
Hours	Non-key attribute	Non-prime attribute
Division	Key attribute	Prime attribute
{Proj_nm, Division}	Key attribute	Non-prime attribute
{Emp#, Division}	Key attribute	Prime attribute
{Proj#, Division}	Key attribute	Prime attribute
{Fund#, Division}	Key attribute	Non-prime attribute
{Proj_nm, Emp#, Division}	<i>Candidate key</i>	<i>Alternate key</i>
{Proj#, Emp#, Division}	<i>Candidate key</i>	<i>Primary key</i>
{Fund#, Emp#, Division}	<i>Candidate key</i>	<i>Alternate key</i>

Note: Any composite attribute that includes one or more non-key attribute(s) is a non-key attribute.

389

**TABLE 7.5** Attribute roles in URS3

## Chapter Summary

Normalization is a technique that facilitates systematic validation of the participation of attributes in a relation schema from the perspective of data redundancy. One of the main concepts associated with normalization is functional dependency. An FD in a relation schema, R, is a constraint of the form  $A \rightarrow B$ , where an attribute, B (atomic or composite), is dependent on attribute A (atomic or composite) if each value, a, of A is associated with exactly one value, b, of B.

Examination of functional dependencies in a relation schema is important because certain functional dependencies (i.e., those that are undesirable) can lead to insertion, deletion, and update anomalies via data redundancies in the associated relation instances, collectively known as modification anomalies. Since functional dependencies, desirable or undesirable, arise from the business rules embedded in the user requirements specification, they cannot be conveniently discarded if undesirable. Therefore, the data redundancies and modification anomalies are removed by decomposing the relation schema such that the undesirable functional dependencies are rendered desirable. The example in Section 7.1 illustrates this.

The functional dependencies in a relation schema that are semantically obvious from the business rules are often explicitly specified and are collectively referred to as F. All possible FDs that can be inferred from the set F plus the set F itself constitute the closure of F. The closure of F is denoted as  $F^+$ . Armstrong's axioms are a set of seven inference rules pertaining to functional dependencies that are used to derive  $F^+$ . Table 7.1 in Section 7.2.2 summarizes the inference rules for functional dependencies.

It is possible to progressively synthesize a candidate key from a set of functional dependencies through the systematic application of the principle of closure of an attribute set. A second method for identifying the candidate key(s) of a relation schema uses a top-down approach of decomposition. In this method, given a set of functional dependencies that prevail over a Universal Relation Schema (URS), the superkey, K, consisting of all attributes of URS, is progressively decomposed by arbitrarily removing one attribute at a time from K until  $K'$ , a superkey that is not further reducible (i.e., no proper subset of K has the uniqueness property), results, yielding a candidate key. The other candidate keys of URS are derived using F and the initial candidate key by the application of the pseudotransitivity rule of Armstrong's axioms. Three examples are used to illustrate the use of Armstrong's axioms to derive candidate keys of a relation schema using the method of synthesis and the method of decomposition. Finally, definitions of prime/non-prime and key/non-key attributes are presented, along with a handful of rules of thumb to choose a primary key from among the candidate keys.

## Exercises

1. What is the purpose of the normalization technique in the data modeling process?
2. Explain why data redundancy exists for the attributes **Discount** and **Location** in the STOCK table in Figure 7.1c.
3. Explain functional dependency between two attributes.
4. Why can functional dependency not be inferred from a particular relation state?

5. Identify the set of functional dependencies in the relation instance CAR shown next. Does this constitute the minimal cover for the set of functional dependencies present in CAR? If it is not a minimal cover, derive a minimal cover.

**CAR**

Model	#Cylinders	Origin	Tax	Fee
Camry	4	Japan	15	30
Mustang	6	USA	0	45
Fiat	4	Italy	18	30
Accord	4	Japan	15	30
Century	8	USA	0	60
Mustang	4	Canada	0	30
Monte Carlo	6	Canada	0	45
Civic	4	Japan	15	30
Mustang	4	Mexico	15	30
Mustang	6	Mexico	15	45
Civic	4	Korea	15	30

6. What is the difference between  $F$ ,  $F^+$ , and  $F_c$ ?
7. What is the purpose of Armstrong's axioms?
8. Suppose  $F \{fd_1, fd_2\}$  consists of the following FDs:
- fd1: **Ssn** → {**Ename**, **Bdate**, **Address**, **Dnumber**}
- fd2: **Dnumber** → {**Dname**, **Dmgrssn**}
- Which of Armstrong's axioms allows the following additional FDs to be inferred?
- Ssn** → {**Dname**, **Dmgrssn**}
  - Ssn** → **Ssn**
  - Dnumber** → **Dname**
  - Ssn** → **Dname**
9. Why is it useful to know all the candidate keys of a relation schema?
10. Describe the two approaches used in this book to derive candidate keys.
11. What is the difference between (a) a prime attribute and a non-prime attribute and (b) a key and non-key attribute?
12. Given  $R(X, A, Z, B)$  and  $A \rightarrow \{B, Z\}$ , what is (are) the candidate key(s) of  $R$ ?

13. Consider the universal relation schema INVENTORY (Store#, Item, Vendor, Date, Cost, Units, Manager, Price, Sale, Size, Color, Location) and the constraint set F {fd1, fd2, fd3, fd4, fd5, fd6, fd7}, where:

fd1: {Item, Vendor} → Cost	fd2: {Store#, Date} → {Manager, Sale}
fd3: {Store#, Item, Date} → Units	fd4: Manager → Store#
fd5: Cost → Price	fd6: Item → {Size, Color}
fd7: Vendor → Location	

392

- a. Do the FDs shown constitute a minimal cover of F? If not, derive a minimal cover?
  - b. Derive the candidate key(s) of URS using the synthesis approach.
  - c. Derive the candidate key(s) of URS by using the decomposition approach.
14. Given the set of functional dependencies F {fd1, fd2, fd3, fd4, fd5, fd6, fd7, fd8, fd9, fd10} where:

fd1: Tenant# → {Name, Job, Phone#, Address}	fd2: Job → Salary
fd3: Name → Gender	fd4: Phone# → Address
fd5: {Name, Phone#} → {Tenant, Deposit}	fd6: County → Tax_rate
fd7: Area → {Rent, County}	fd8: Survey# → Lot
fd9: {Lot, County} → {Survey#, Area}	fd10: {Survey#, Area} → County

- a. Construct the universal relation schema that includes (i.e., preserves) the set of FDs in F.
  - b. Do the FDs shown constitute a minimal cover of F? If not, derive a minimal cover.
  - c. Derive the candidate key(s) of F.
  - d. Select the primary key and justify your choice.
  - e. Considering your primary key and candidate key(s), distinguish between (1) key versus non-key attributes and (2) prime versus non-prime attributes.
15. Given the set of functional dependencies F {fd1, fd2, fd3, fd4, fd5, fd6, fd7, fd8, fd9, f10, f11}, where:

fd1: Client → Office	fd2: Stock → {Exchange, Dividend}
fd3: Broker → Profile	fd4: Company → Stock
fd5: Client → {Risk_profile, Analyst}	fd6: Analyst → Broker
fd7: {Stock, Broker} → {Investment, Volume}	fd8: Stock → Company
fd9: Investment → {Commission, Return}	fd10: {Stock, Broker} → Client
fd11: Account → Assets	

- Construct the Universal Relation Schema that includes (i.e., preserves) the set of FDs in F.
  - Do the FDs shown constitute a minimal cover of F? If not, derive a minimal cover.
  - Derive the candidate key(s) of F.
  - Select the primary key and justify your choice.
  - Considering your primary key and candidate key(s), distinguish between (1) key versus non-key attributes and (2) prime versus non-prime attributes.
16. Given the Universal Relation Schema **URS (A, B, C, D, F, G)** and the set of FDs prevailing over URS **F {fd1, fd2, fd3, fd4, fd5}**, where:

fd1: A → G;	fd2: {A, B} → {C, D};	fd3: {B, C} → {F, G};
fd4: G → B;	fd3: {B, D} → A;	fd5: C → G

- Derive a canonical cover of F.
  - Derive all the candidate keys of URS.
17. Given the Universal Relation Schema **URS (A, B, C, G, W, X, Y, Z)** and the set of FDs prevailing over URS, **F {fd1, fd2, fd3, fd4, fd5}**, where:

fd1: B → G;	fd2: {X, Z} → {Z, Y, B};	fd3: {Y, A} → {C, G};
fd4: C → W;	fd5: {X, Z} → G;	

- Derive a canonical cover of F.
- Derive the candidate key(s) of URS.
- Is the fd  $\{X, A, Z\} \rightarrow W$  in  $F^+$ ?
- Is the decomposition D {R1, R2}, where R1 (X, Z, Y, A, B) and R2 (A, C, Y, G, W) lossless?
- Is the decomposition D {R3, R4}, where R3 (X, Z, Y, A, C) and R4 (C, X, G, W, Z) lossless?



# CHAPTER

# 8

# NORMAL FORMS BASED ON FUNCTIONAL DEPENDENCIES

In Chapter 7, we saw how certain functional dependencies (FDs) can create data redundancy problems in a relation schema. This chapter shows how the process of normalization can be used to resolve these problems. Recall that *normalization* is a technique that facilitates systematic validation of the participation of attributes in a relation schema from a perspective of data redundancy.

This chapter flows as follows. Section 8.1 introduces normalization as a technique to facilitate systematic validation of the goodness of design of a relation schema. The first, second, and third normal forms (1NF, 2NF, and 3NF) are explained with appropriate examples in Subsections 8.1.1, 8.1.2, and 8.1.3, respectively. Boyce-Codd Normal Form (BCNF) is presented as a stronger version of the 3NF in Section 8.1.4. The lossless-join property and dependency preservation are then presented, in Section 8.1.5, as two critical side effects that require attention in the normalization process. The motivating exemplar originally introduced in Section 7.1 of the previous chapter is revisited in Section 8.2 to provide an explanation for and illustration of the logic behind the normalization process. Section 8.3 gives a comprehensive example of normalizing a universal relation schema subject to a defined set of FDs. Section 8.4 briefly discusses denormalization, and Section 8.5 presents the use of reverse engineering in data modeling.

## 8.1 NORMALIZATION

---

Data redundancy and the consequent modification (insertion, deletion, and update) anomalies can be traced to “undesirable” functional dependencies in a relation schema. What is an undesirable functional dependency? Any FD in a relation schema, R, where the determinant is a candidate key of R is a **desirable FD** because it will not cause data redundancy and the consequent modification anomalies. Where the determinant of an FD in R is *not* a candidate key of R, the FD will cause data redundancy and the consequent modification anomalies and so is an **undesirable FD**.

So, what can we do with the undesirable FDs? The source of all FDs, desirable and undesirable, is the set of user-specified business rules and so must be incorporated in the database system. Therefore, FDs cannot be selectively ignored or discarded because they are undesirable. The only solution is to somehow render the undesirable FDs desirable, and the process of doing this is called **normalization**.

**Normal forms (NFs)** provide a stepwise progression toward the goal of a fully normalized relation schema that is guaranteed to be free of data redundancies that cause modification anomalies from a functional dependency perspective.<sup>1</sup> A relation schema is said to be in a particular normal form if it satisfies certain prescribed criteria; otherwise, the relation schema is said to violate that normal form. First normal form (1NF) reflects one of the properties of a relation schema—that is, by definition, a relation schema is in 1NF. The normal forms associated with functional dependencies are second normal form (2NF), third normal form (3NF), and Boyce-Codd Normal Form (BCNF).<sup>2</sup>

The violations of each of these normal forms signal the presence of a specific type of “undesirable” FD. When a relation schema violates a certain normal form, it can be interpreted as equivalent to an inadvertent mixing of entity types belonging to two different entity classes in a single entity type. Therefore, by appropriately decomposing the relation schema, the undesirable FD causing the violation of a specific normal form can be rendered desirable in the resulting set of relation schemas—that is, the relational schema.

It is important to note that the normalization process is anchored to the candidate key of a relation schema, R. The assessment of normal form can be based on the primary key or any candidate key of R. This is not an issue when R has only one candidate key. Even when R has multiple candidate keys, normalization based on any and every candidate key (including the primary key) will yield the same set of normalized relation schemas. Therefore, we will use the primary key as the basis for evaluating and normalizing a relation schema. This does not by any means contradict the assertion that an FD in R is undesirable only when the determinant of that FD is not a *candidate key* of R.

The following sections delineate each of the normal forms using meaningful examples. Later in the chapter, we will address the situation of generating a fully normalized relational schema from a given set of FDs.

### 8.1.1 First Normal Form (1NF)

**First normal form (1NF)** imposes conditions so that a base relation that is physically stored as a file does not contain records with a variable number of fields. This is accomplished by prohibiting multi-valued attributes, composite attributes, and combinations thereof in a relation schema. As a consequence, the value of an attribute in a tuple of a relation can be neither a set of values nor another tuple. Such a constraint in effect prevents relations from containing other relations.<sup>3</sup> In essence, 1NF, by definition, requires that the domain of an attribute include only atomic values and that the value of an attribute in a relation’s tuple must be a single value from the domain of that attribute.

---

<sup>1</sup>A fully normalized relation schema from the perspective of functional dependencies need not be completely free of data redundancies and the consequent modification anomalies if multi-valued dependencies are present in the relation schema. This will be addressed in Chapter 9.

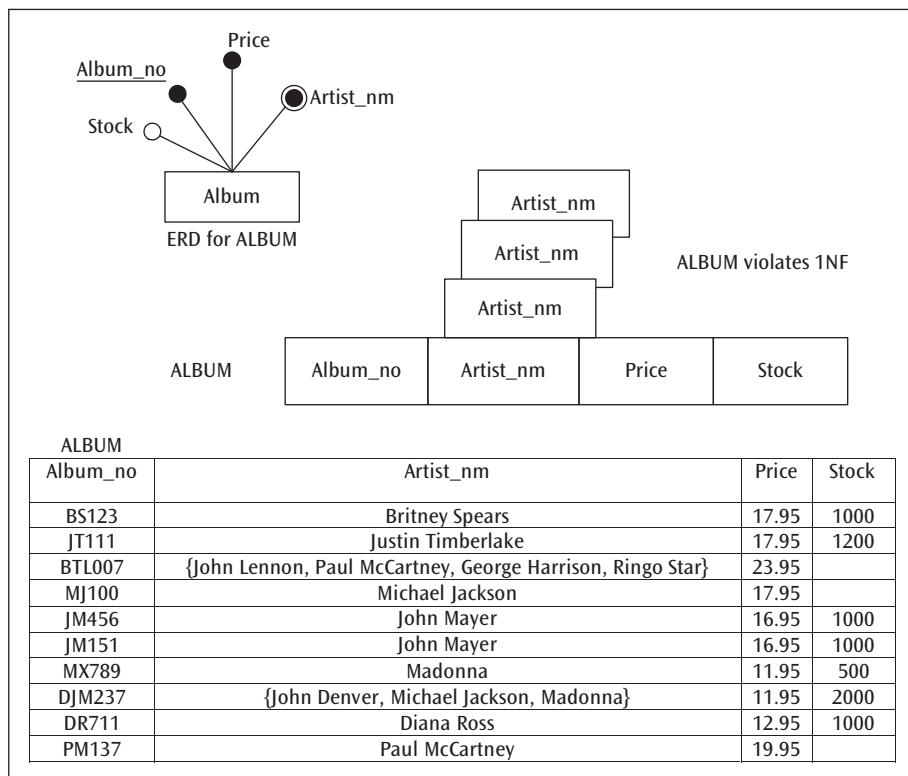
<sup>2</sup>E. F. Codd first proposed the 1NF, 2NF, and 3NF in 1972. Later, it was discovered that under certain conditions (i.e., FDs) a relation schema in 3NF continues to have data redundancies, causing modification anomalies. A revised, stronger definition of the 3NF was then proposed by Boyce and Codd in 1974, which came to be known as Boyce-Codd Normal Form (BCNF).

<sup>3</sup>This constraint is relaxed in object-relational database systems, which allow non-1NF relations.

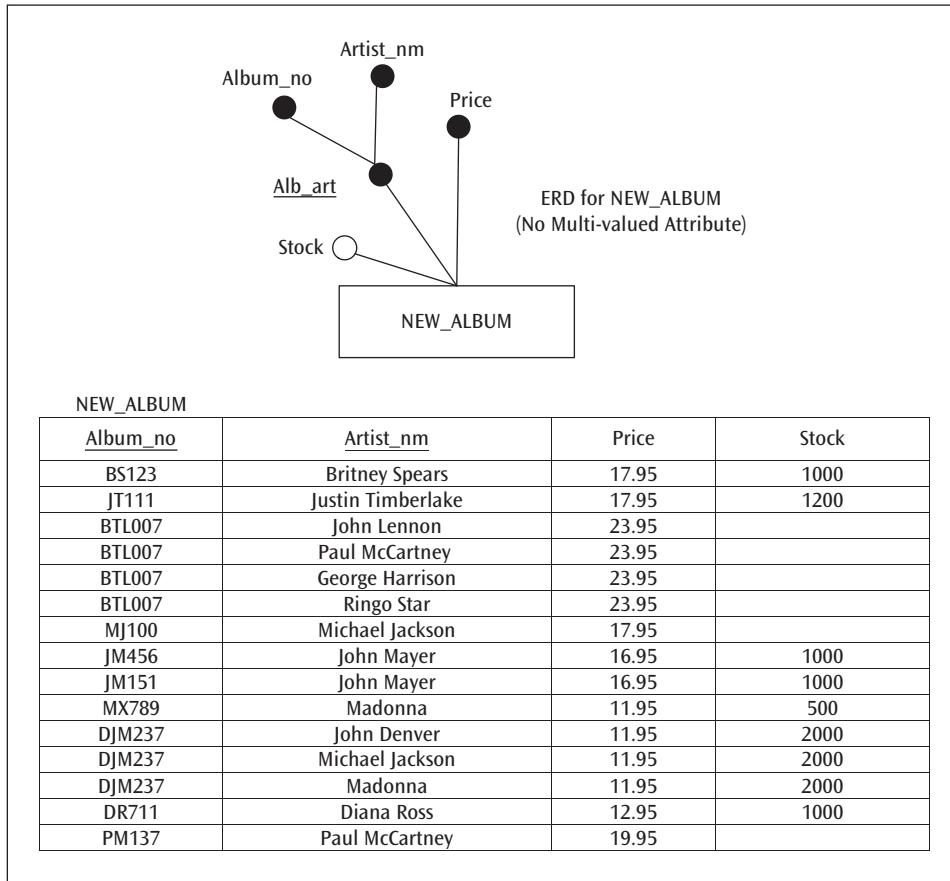
## DEFINITION

**1NF definition:** A schema  $R$  is in 1NF only when the attributes comprising the schema are atomic and single-valued. Unless a schema is in 1NF, it is not a “relation schema.” That is, a relation schema is, by definition, in 1NF.

Consider the schema ALBUM and the corresponding instance of ALBUM shown in Figure 8.1a. Here, for a given **Album\_no**, there is a single specific value of **Price** (i.e.,  $\text{Album\_no} \rightarrow \text{Price}$ ) and a single specific value of **Stock** (i.e.,  $\text{Album\_no} \rightarrow \text{Stock}$ ). On the other hand, either there are multiple values for **Artist\_nm** associated with an **Album\_no** or the domain of **Artist\_nm** does not have atomic values. In either case, 1NF is violated. In fact, by definition, ALBUM is not even a relation. The solution to render ALBUM in 1NF is to simply expand the relation so that there is a tuple for each (atomic) **Artist\_nm** for a given **Album\_no**. This is shown in NEW\_ALBUM (Figure 8.1b), which is in 1NF, with **{Album\_no, Artist\_nm}** as its primary key.



**FIGURE 8.1a** An example of 1NF violation

**FIGURE 8.1b** Resolution of 1NF violation of Figure 8.1a

### 8.1.2 Second Normal Form (2NF)

The second normal form (2NF) is based on a concept known as full functional dependency.

A functional dependency of the form  $Z \rightarrow A$  is a full functional dependency if, and only if, no proper subset of  $Z$  functionally determines  $A$ . In other words, if  $Z \rightarrow A$  and  $X \rightarrow A$ , and  $X$  is a proper subset of  $Z$ , then  $Z$  does not *fully* functionally determine  $A$ —that is,  $Z \rightarrow A$  is not a full functional dependency; it is a **partial dependency**.

#### DEFINITION

**2NF definition:** A relation schema  $R$  is in 2NF if every non-prime attribute in  $R$  is fully functionally dependent on the primary key of  $R$ —that is, a non-prime attribute is not functionally dependent on a proper subset of the primary key of  $R$ .

Partial dependency of a non-prime attribute on the primary key of a relation schema, R, is one form of an undesirable FD, because this amounts to the presence of an FD in R, where the determinant is not a candidate key of R.

Consider the 1NF relation instance (table) NEW\_ALBUM in Figure 8.1. The relation schema for NEW\_ALBUM and a copy of the relation instance are shown in Figure 8.2a. A review of the data in NEW\_ALBUM reveals the presence of the following minimal (canonical) cover of FDs in it:

$$F_c: fd1: \text{Album\_no} \rightarrow \text{Price}; fd2: \text{Album\_no} \rightarrow \text{Stock}$$

Note that:

$$\text{Album\_no} \not\rightarrow \text{Artist\_nm}$$

R: NEW\_ALBUM (Album\_no, Artist\_nm, Price, Stock)

NEW\_ALBUM

Album_no	Artist_nm	Price	Stock
BS123	Britney Spears	17.95	1000
JT111	Justin Timberlake	17.95	1200
BTL007	John Lennon	23.95	
BTL007	Paul McCartney	23.95	
BTL007	George Harrison	23.95	
BTL007	Ringo Star	23.95	
MJ100	Michael Jackson	17.95	
JM456	John Mayer	16.95	1000
JM151	John Mayer	16.95	1000
MX789	Madonna	11.95	500
DJM237	John Denver	11.95	2000
DJM237	Michael Jackson	11.95	2000
DJM237	Madonna	11.95	2000
DR711	Diana Ross	12.95	1000
PM137	Paul McCartney	19.95	

$$F: fd1: \text{Album\_no} \rightarrow \text{Price}; fd2: \text{Album\_no} \rightarrow \text{Stock}$$

$$F^+: fd12: \text{Album\_no} \rightarrow (\text{Price}, \text{Stock}); fd12x: (\text{Album\_no}, \text{Artist\_nm}) \rightarrow (\text{Price}, \text{Stock});$$

Candidate key of NEW\_ALBUM: (Album\_no, Artist\_nm); Primary key: (Album\_no, Artist\_nm)

fd1 and fd2 (fd12) violate 2NF in NEW\_ALBUM

**FIGURE 8.2a** An example of 2NF violation

What is the primary key of NEW\_ALBUM? Using Armstrong's axioms, we can infer that fd12:  $\text{Album\_no} \rightarrow \{\text{Price}, \text{Stock}\}$  and fd12x:  $\{\text{Album\_no}, \text{Artist\_nm}\} \rightarrow \{\text{Price}, \text{Stock}\}$  exist in  $F^+$ . Therefore,  $\{\text{Album\_no}, \text{Artist\_nm}\}$  is a candidate key of NEW\_ALBUM; being the only candidate key, it becomes the primary key of NEW\_ALBUM. Are there any "undesirable" FDs in NEW\_ALBUM? The answer is "Yes." Given that  $\{\text{Album\_no}, \text{Artist\_nm}\}$  is the primary key of NEW\_ALBUM, fd1 and fd2 (or fd12) reflects a partial dependency of a non-prime attribute on the primary key of NEW\_ALBUM. Therefore, 2NF is violated in NEW\_ALBUM.

Now let us examine the relation instance of NEW\_ALBUM to see if there are any data redundancies in it that lead to modification anomalies. It is obvious that both **Price** and **Stock** are repeated for every **Artist\_nm** for a given **Album\_no** (e.g., see **Album\_no** BTL007). Are there any modification anomalies in NEW\_ALBUM?

If we want to change the value of **Price** or **Stock** of **Album\_no** BTL007, four tuples in the relation require update—a clear case of *update anomaly*. The anomaly is due to the fact that it is possible to erroneously post different values for **Price** and **Stock** for the same value of **Album\_no** in the four different tuples.

Also, if we want to add a new tuple, (**Album\_no**: XY111, **Price**: 17.95, and **Stock**: 100), to NEW\_ALBUM, it is not possible to do so without knowing a value for **Artist\_nm** because the primary key of NEW\_ALBUM is {**Album\_no**, **Artist\_nm**} and a prime attribute, **Artist\_nm**, cannot have a null value. This is an *insertion anomaly* simply because of the inability to add a genuine tuple to the database.

In order to delete **Album\_no** BTL007, four tuples in the relation NEW\_ALBUM must be deleted. This is an example of a *deletion anomaly*. If all four tuples are not deleted, the information conveyed by the data in the relation is distorted; hence the anomaly.

The next question is, How do we know that the undesirable FDs identified earlier (i.e., fd1 and fd2) indeed cause the data redundancies and the associated modification anomalies? The way this can be verified is to eliminate the undesirable FDs from NEW\_ALBUM (that is, rendering them desirable) and see if the data redundancies and the associated anomalies persist. The resolution of 2NF violation is a two-step process that decomposes the target relation schema with undesirable FDs into multiple relation schemas that are free from undesirable FDs:

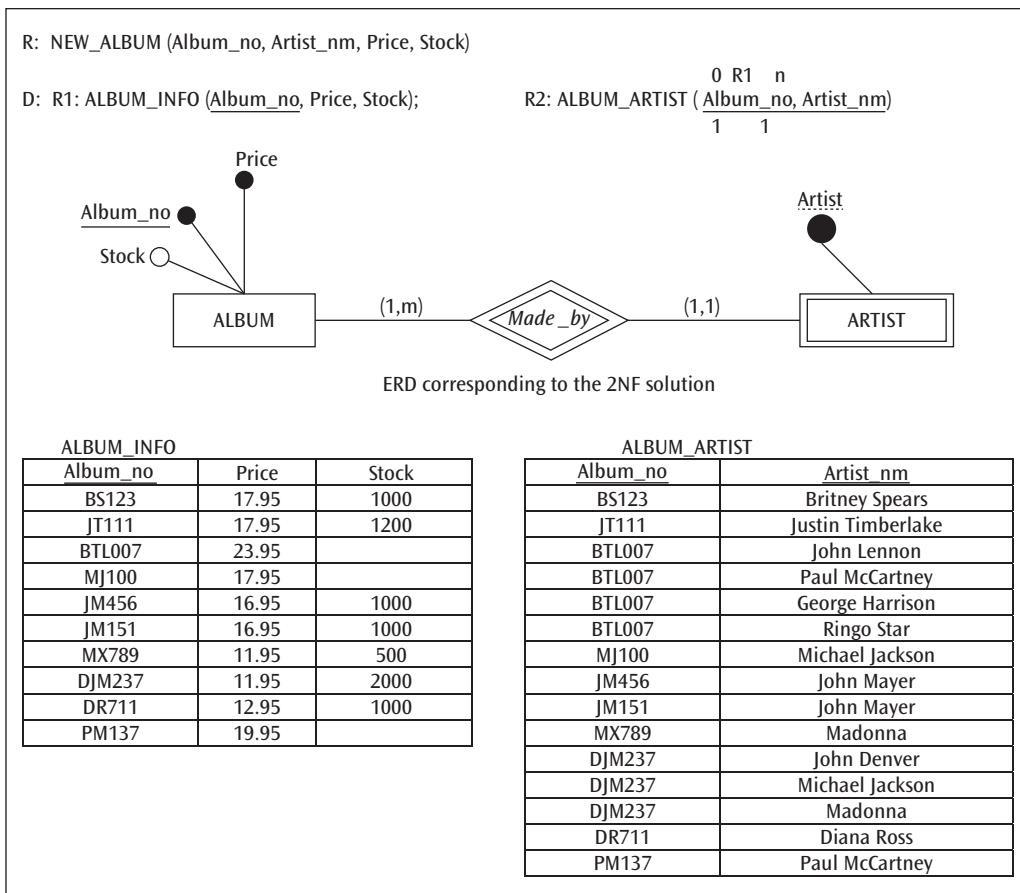
1. Pull out the undesirable FD(s) from the target relation schema as a separate relation schema.
2. Retain the determinant of the pulled-out relation schema as an attribute (foreign key) or attributes of the leftover target relation schema to facilitate reconstruction of the original target relation schema.

Applying this two-step process to NEW\_ALBUM, we have the decomposition, D:

D: ALBUM\_INFO (**Album\_no**, **Price**, **Stock**); and ALBUM\_ARTIST (**Album\_no**, **Artist\_nm**)

*Note:* ALBUM\_INFO and ALBUM\_ARTIST are arbitrary (meaningful) names assigned to the decomposed relation schemas.

All non-prime attributes in ALBUM\_INFO are fully functionally dependent on the primary key of ALBUM\_INFO, and in ALBUM\_ARTIST there are no non-prime attributes. So, both are in 2NF. Reviewing the corresponding decomposed relations (see Figure 8.2b), it is clear that there are no data redundancies causing modification anomalies in either relation. It is now possible to insert the tuple (**Album\_no**: XY111, **Price**: 17.95, and **Stock**: 100) in the database (i.e., in ALBUM\_INFO) without having a value for **Artist\_nm**. If the **Price** or **Stock** of **Album\_no** BTL007 changes, the corresponding update requires change of just one tuple in spite of the fact that the album involves multiple artists. So, we can infer that the resolution of undesirable FDs causing the 2NF violation eliminated the data redundancies and the associated modification anomalies.

**FIGURE 8.2b** Resolution of 2NF violation of Figure 8.2a

How about the presence of **Album\_no** as an attribute in both ALBUM\_INFO and ALBUM\_ARTIST? Is it not data redundancy? It is, but this is known as **controlled redundancy** (see Section 1.4.3 in Chapter 1) as long as the referential integrity constraint between the two relation schemas is enforced by specifying the inclusion dependency:

$$\text{ALBUM_ARTIST}.\{\text{Album}_\text{no}\} \subseteq \text{ALBUM\_INFO}.\{\text{Album}_\text{no}\}$$

which can also be used to reconstruct the original target schema, NEW\_ALBUM. Observe that the violation of 2NF is a concern only when the primary key of a relation schema is a composite attribute. If the primary key is an atomic attribute, a partial dependency is impossible.

### 8.1.3 Third Normal Form (3NF)

The **third normal form (3NF)** is based on the concept of **transitive dependency**. Consider a relation schema R (X, A, B), where X, A, and B are pair-wise disjoint atomic or composite attributes, X is the primary key of R, and A and B are non-prime attributes. If  $A \rightarrow B$  (or  $B \rightarrow A$ ) in R, then B (or A) is also said to be transitively dependent on X, the primary key of R. This is another form by which an undesirable FD appears in a relation schema.

Clearly, this form of FD is by definition undesirable, because the determinant in the FD is, again, not a candidate key of R. Fundamentally, the source of the problem is not the principle of transitivity itself, because had A (or B) been another candidate key (alternate key) of R, the transitive nature of  $X \rightarrow B$  (or  $X \rightarrow A$ ) does not yield an undesirable FD. In essence, the problem boils down to a non-prime attribute functionally determining another non-prime attribute. Nonetheless, this is not a violation of 2NF, because  $A \rightarrow B$  (or  $B \rightarrow A$ ) in R is not a partial dependency. In fact, R is in 2NF.

## DEFINITION

*3NF definition: A relation schema R is in 3NF if no non-prime attribute is functionally dependent on another non-prime attribute in R.*

402

While violations of 2NF and 3NF are independent effects, since these two normal forms are labeled as such (i.e., 2NF and 3NF), it is customary to specify that in order for a relation schema to be in 3NF, it should also be in 2NF.

As an example, consider the relation schema:

**FLIGHT (Flight#, Origin, Destination, Mileage)**

and the set of FDs, F, where:

fd1: **Flight# → Origin**; fd2: **Flight# → Destination**; fd3: **{Origin, Destination} → Mileage**;

In order to assess the normal form of FLIGHT, we first need to identify the candidate keys of FLIGHT. To that end, using Armstrong's axioms, we can infer that  $F^+$  includes the following FDs:

fd12: **Flight# → {Origin, Destination}**;—Union rule

fd3x: **Flight# → Mileage**;—Transitivity rule

fd123: **Flight# → {Origin, Destination, Mileage}**;—Union rule

Thus, we see that **Flight#** is a candidate key of FLIGHT. Since there are no other candidate keys of FLIGHT, **Flight#** becomes the primary key of FLIGHT. Since the primary key of FLIGHT is an atomic attribute, a 2NF violation is impossible in FLIGHT. So, FLIGHT is in 2NF. Is it also in 3NF? No, because fd3 (i.e., **{Origin, Destination} → Mileage**) causes a transitive dependency in FLIGHT, because a composite non-prime attribute, **{Origin, Destination}** functionally determines another non-prime attribute, **Mileage**. Thus, 3NF is violated in FLIGHT.

Let us now explore if there are data redundancies in FLIGHT. Figure 8.3 displays an instance of a relation for FLIGHT. Note that the relation instance precisely reflects the FDs specified. Data redundancy is exemplified in FLIGHT by the repetition of distance 1058 from Chicago to Dallas in more than one tuple.

Are there any modification anomalies in FLIGHT?

If the tuple identified by **Flight# DL507** is deleted, the information that Seattle to Denver is 1537 miles is inadvertently lost from the database, an example of a *deletion anomaly*.

Addition of a tuple to FLIGHT to indicate that the mileage for Cincinnati to Houston is 1100 (**Origin: 'Cincinnati'**, **Destination: 'Houston'**, **Mileage: 1100**) is not possible without a **Flight#** identifying this route. Since **Flight#** is the primary key of FLIGHT, it cannot have null values. This is an *insertion anomaly*.

Once again, if the normalization of FLIGHT to 3NF by the removal of the undesirable FD:

**{Origin, Destination} → Mileage**

eliminates the modification anomalies, we can infer that the data redundancy and the consequent modification anomalies are due to the presence of the undesirable FD in FLIGHT.

R: FLIGHT (Flight#, Origin, Destination, Mileage)

#### FLIGHT

Flight#	Origin	Destination	Mileage
DL507	Seattle	Denver	1537
DL123	Chicago	Dallas	1058
DL723	Boston	St. Louis	1214
DL577	Denver	Los Angeles	1100
DL5219	Minneapolis	St. Louis	580
DL357	Chicago	Dallas	1058
DL555	Denver	Houston	1100
DL5237	Cleveland	St. Louis	580
DL5271	Chicago	Cleveland	300

F: fd1: Flight# → Origin;      fd2: Flight# → Destination;  
           fd3: (Origin, Destination) → Mileage

FLIGHT is in 1NF (No composite or multi-valued attributes in FLIGHT)

F<sup>+</sup>: F;  
       fd12: Flight# → (Origin, Destination);      fd3x: Flight# → Mileage;  
       fd123: Flight# → (Origin, Destination, Mileage)

Candidate key of FLIGHT: Flight#      Primary key of FLIGHT: Flight#

fd1 & fd2 are desirable FDs – why?

Determinant in both cases is a candidate key (primary key) of FLIGHT

fd3 does not violate 2NF (not a partial dependency) – FLIGHT, therefore, is in 2NF  
           but, fd3 violates 3NF – why?

Non-prime determines another non-prime in FLIGHT

**Solution:**

D: R1: FLIGHT (Flight#, Origin, Destination);      R2: DISTANCE (Origin, Destination, Mileage)  
                         1                                    1

#### FLIGHT

Flight#	Origin	Destination
DL507	Seattle	Denver
DL123	Chicago	Dallas
DL723	Boston	St. Louis
DL577	Denver	Los Angeles
DL5219	Minneapolis	St. Louis
DL357	Chicago	Dallas
DL555	Denver	Houston
DL5237	Cleveland	St. Louis
DL5271	Chicago	Cleveland

#### DISTANCE

Origin	Destination	Mileage
Seattle	Denver	1537
Chicago	Dallas	1058
Boston	St. Louis	1214
Denver	Los Angeles	1100
Minneapolis	St. Louis	580
Denver	Houston	1100
Cleveland	St. Louis	580
Chicago	Cleveland	300

**FIGURE 8.3** An example of 3NF violation and resolution

The resolution of 3NF violation is accomplished by applying the same two-step process used earlier to resolve the 2NF violation (see Section 8.1.2). To review, the two-step process is:

1. Pull out the undesirable FD(s) from the target relation schema as a separate relation schema.
2. Retain the determinant of the pulled-out relation schema as an attribute (foreign key) or attributes of the leftover target relation schema to facilitate reconstruction of the original target relation schema.

Accordingly, we have the decomposition, D<sup>4</sup>:

D: DISTANCE (**Origin, Destination, Mileage**); FLIGHT (**Flight#, Origin, Destination**)

Note that DISTANCE is an arbitrary (meaningful) name assigned to the decomposed relation schema. The leftover target relation schema retains the same name as FLIGHT.

There are no 3NF violations in the decomposed set of relation schemas, DISTANCE and FLIGHT. So, the solution has yielded a relational schema that is in 3NF. A review of the corresponding decomposed relations (see Figure 8.3) reveals that there are no data redundancies causing modification anomalies in either relation of the 3NF design. It is now possible to insert the tuple (**Origin**: ‘Cincinnati’, **Destination**: ‘Houston’, **Mileage**: 1100) in the database (i.e., in DISTANCE) without having a value for **Flight#**. Deletion of the tuple identified by **Flight#** 507 in FLIGHT no longer gets rid of the information that Seattle to Denver is 1537 miles from the database (i.e., from DISTANCE). So, we can infer that the resolution of undesirable FDs causing the 3NF violation eliminated the data redundancies and the associated anomalies. The controlled redundancy between DISTANCE and FLIGHT in the 3NF design via the referencing attributes **{Origin, Destination}** establishes referential integrity constraint between the two relation schemas as reflected in the inclusion dependency:

**FLIGHT.{Origin, Destination} ⊆ DISTANCE.{Origin, Destination}**

which can also be used to reconstruct the original target relation schema.

#### 8.1.4 Boyce-Codd Normal Form (BCNF)

After Codd (1972) proposed the first three normal forms, it was discovered that the 3NF did not satisfactorily handle a more general case of undesirable functional dependencies. In other words, data redundancies and the consequent modification anomalies due to functional dependencies can persist even after a relation schema is normalized to 3NF. In particular, modification anomalies persist if the following pertain:

- A relation schema has at least two candidate keys,
- Both candidate keys are composite attributes, and
- There is an attribute overlap between the two candidate keys.

In order to rectify this inadequacy, Boyce and Codd (1974) proposed an improved version of 3NF definition. Since the definition, while simpler, is strictly stronger than the original definition of 3NF, it is given a different name, **Boyce-Codd Normal Form (BCNF)**.

---

<sup>4</sup>There are two other ways to decompose FLIGHT. The merits/demerits of those solutions are discussed later in this chapter (see Section 8.1.5).

Consider a relation schema R (X, A, B, C), where X, A, B, and C are pair-wise disjoint atomic or composite attributes. Suppose the set of FDs (a minimal cover over R), F, given here, prevail over R:

F: fd1: {X, A} → B; fd2: {X, A} → C; and fd3: B → A

Using Armstrong's axioms, we first infer from fd1 and fd2 that {X, A} is a candidate key of R. Then, based on fd3, we infer that {X, B} is another candidate key of R (using the pseudotransitivity rule of Armstrong's axioms). Choosing {X, A} as the primary key of R, R is in 2NF because there are no partial dependencies in R.

R is also in 3NF because there is no transitive dependency of a non-prime attribute on the primary key.

*Note:* B → A (fd3) does not violate 3NF because A is a *prime* attribute. In fact, in fd3 a *non-prime* attribute determines a *prime* attribute.

Therefore, R is, indeed, in 3NF when evaluated on the basis of {X, A} as the primary key of R. Observe that B → A, by definition, is an undesirable FD in R simply because B is not a candidate key of R.

## DEFINITION

*BCNF definition:* A relation schema R is in BCNF if for every non-trivial functional dependency in R, the determinant is a superkey of R.

The immediate questions, then, ought to be: "Is there any data redundancy in R? If so, does it cause any modification anomalies?" To explore this condition, let us review an example.

**STU\_SUB (Stu#, Subject, Teacher, Ap\_score) subject to**

F: fd1: {Stu#, Subject} → Teacher; fd2: {Stu#, Subject} → Ap\_score;

fd3: Teacher → Subject

It is obvious that {Stu#, Subject} is a candidate key of STU\_SUB. Choosing this as the primary key of STU\_SUB, fd1, fd2, and fd3 does not violate either 2NF or 3NF. In fact, it can be shown that no FD in F<sup>+</sup> violates 2NF or 3NF.

A relation instance for STU\_SUB appears in Figure 8.4, where one can observe that {Subject, Teacher} pairs are redundantly recorded. Does this cause any anomalies? Since Teacher → Subject, if we want to add a new Teacher for a Subject (e.g., Teacher: 'Salter,' Subject: 'English'), it is not possible to do so unless a corresponding Stu# is also provided. Likewise, if Campbell is no longer advising and is being replaced by, say, Smith, multiple tuples require modification. These are cases of *insertion anomaly* and *update anomaly*, respectively, in STU\_SUB. In short, STU\_SUB is in 3NF and yet modification anomalies are present in it. An examination of F (or F<sup>+</sup>) reveals that fd3 (Teacher → Subject) in STU\_SUB is an undesirable FD because the determinant in fd3, Teacher, is not a candidate key of STU\_SUB. Since fd3 is not a trivial dependency (i.e., the dependent in fd3 is not a subset of the determinant of fd3), the fact that Teacher is not a superkey of STU\_SUB causes a BCNF violation in STU\_SUB per the definition of BCNF.

R: STU\_SUB (Stu#, Subject, Teacher, Ap\_score)

<u>Stu#</u>	<u>Subject</u>	Teacher	Ap_score
IH123	Chemistry	Raturi	4
IH123	English	Stephan	4
IH235	History	Walker	5
IH357	English	Campbell	4
IH571	Chemistry	Raturi	3
IH235	English	Campbell	4

F            fd1: (Stu#, Subject) → Teacher;                      fd2: (Stu#, Subject) → Ap\_score;  
              fd3: Teacher → Subject

STU\_SUB is in 1NF (No composite or multi-valued attributes in STU\_SUB)

Candidate keys of STU\_SUB are: (Stu#, Subject); (Stu#, Teacher)

Primary key of STU\_SUB: (Stu#, Subject) [Chosen for this example]

fd1 & fd2 are desirable FDs – why?

Determinant in both cases is a candidate key (primary key) of STU\_SUB

fd3 does not violate 2NF (not a partial dependency)  
STU\_SUB, therefore, is in 2NF for the chosen primary key

fd3 does not violate 3NF (non-prime attribute not a determinant of another non-prime attribute)  
STU\_SUB, therefore, is in 3NF

But, fd3 violates BCNF – why?

Determinant is not a candidate key of STU\_SUB  
*(Non-prime determines a prime in STU\_SUB)*

### Solution:

D:      R1:      TEACH\_SUB (Teacher, Subject);      R2:      STU\_AP (Stu#, Teacher, Ap\_score)

TEACH_SUB	
Teacher	Subject
Raturi	Chemistry
Stephan	English
Walker	History
Campbell	English

<u>Stu_AP</u>	<u>Teacher</u>	<u>Ap_score</u>
IH123	Raturi	4
IH123	Stephan	4
IH235	Walker	5
IH357	Campbell	4
IH571	Raturi	3
IH235	Campbell	4

**FIGURE 8.4** An example of BCNF violation and resolution

Again, if the removal of the undesirable FD, fd3 (**Teacher → Subject**), from STU\_SUB eliminates the modification anomalies, we can infer that the data redundancy and the consequent anomalies are due to the presence of the undesirable FD in STU\_SUB. Let us see if the removal of fd3, the undesirable FD, eliminates the BCNF violation from STU\_SUB also. The resolution of BCNF violation is accomplished by applying the same two-step process used earlier to resolve the 2NF and 3NF violations (see Sections 8.1.2 and 8.1.3):

1. Pull out the undesirable FD(s) from the target relation schema as a separate relation schema.
2. Retain the determinant of the pulled-out relation schema as an attribute (foreign key) or attributes of the leftover target relation schema to facilitate reconstruction of the original target relation schema.

Accordingly, we have the decomposition, D:

D: TEACH\_SUB (**Teacher, Subject**); STU\_AP (**Stu#, Teacher, Ap\_score**)

Note: TEACH\_SUB and STU\_AP are arbitrary (meaningful) names assigned to the decomposed relation schemas.<sup>5</sup>

There are no BCNF violations in the decomposed set of relation schemas TEACH\_SUB and STU\_AP, because in both relation schemas the determinant of the only FD present in each is a superkey of the respective relation schemas. So, the solution has yielded a relational schema that is in BCNF. Reviewing the corresponding decomposed relations (see Figure 8.4), it is seen that there are no data redundancies causing modification anomalies in either relation of the BCNF design. It is now possible to insert the tuple (**Teacher**: ‘Salter,’ **Subject**: ‘English’) in the database (i.e., in TEACH\_SUB) without having a value for **Stu#**. Replacement of Campbell as a teacher requires change of attribute value in just one tuple (i.e., in TEACH\_SUB). So, we can infer that the resolution of the undesirable FD causing the BCNF violation eliminated the data redundancies and the associated modification anomalies. The **controlled redundancy** between TEACH\_SUB and STU\_AP in the BCNF design via the referencing attributes (**Teacher**) establishes referential integrity constraint between the two relation schemas, as reflected in the inclusion dependency:

**STU\_AP.{Teacher} ⊆ TEACH\_SUB.{TEACHER}**

which can also be used to reconstruct the original target relation schema.

### 8.1.5 Side Effects of Normalization

So far, we have approached the normalization process strictly from the perspective of eliminating data redundancies that lead to modification anomalies. Normalization entails decomposition of the target relation schema, R, into multiple relation schemas, D: {R<sub>1</sub>, R<sub>2</sub>, . . . . . R<sub>n</sub>}, such that the join of {R<sub>1</sub>, R<sub>2</sub>, . . . . . R<sub>n</sub>} is strictly equal to R. First, we must make sure that each attribute in the target relation schema, R, is present in some relation schema, R<sub>i</sub>, in the decomposition, D—that is, all attributes of R should collectively appear in D (no attribute can be lost in the decomposition). This basic condition of decomposition is called **attribute preservation**. In addition, two other critical aspects of

---

<sup>5</sup>Note that (Stu#, Teacher) → Ap\_score is in F<sup>+</sup> and is also a candidate key of STU\_SUB. By choosing (Stu#, Teacher) as the candidate key, the BCNF violation can be viewed as a 2NF violation and resolved accordingly to produce the same answer. There is another BCNF decomposition of STU\_SUB. The merits/demerits of this solution are discussed later in this chapter (see Section 8.1.5.3).

design require attention during the decomposition that results from normalization. They are *dependency preservation* and *lossless-join<sup>6</sup> decomposition*. These two independent properties are an expected requirement of an “ideal” design, and both are always tied to a set of functional dependencies, F that holds over the relation schema, R being normalized.

### 8.1.5.1 Dependency Preservation

A relation schema, R, under scrutiny for normalization preserves all FDs, F specified over R in a single relation schema. It is logical to expect that any decomposition of R continue to preserve the minimal cover ( $F_c$ ) of all FDs (F is always one of the covers, but not necessarily a minimal cover because F may include FDs that are redundant) that hold in R. Since the *specified* functional dependencies, F, arise from business rules, they will have to be accounted for in the database implementation. In other words, each FD in F represents a constraint on the database. Therefore, a robust logical database design should fully account for the set of FDs, F specified on a relation schema, R, across the decomposition, D of R. Preservation of  $F_c$  is equivalent to fully accounting for F. By definition, preservation of  $F_c$  implies that each FD in  $F_c$  should either directly appear in *single* decomposed relation schemas in D or be inferable from the FDs that appear in *single* decomposed relation schemas. Only then is the decomposition considered **dependency-preserving**.

If there is a need to join two or more decomposed relation schemas to ascertain an FD in  $F_c$ , then the decomposition is *not* dependency-preserving. At the same time, it is not necessary that the exact dependencies present in F appear in one of the relation schemas in the decomposition D. *It is sufficient if the union of the FDs that hold on individual relation schemas of D is equivalent to  $F_c$* —that is, a cover for F that can generate  $F^+$ . When a modification (insertion, deletion, or update) is made to the database, the DBMS should be able to check that the semantics of the relations in the database are not corrupted by the modification—that is, one or more FDs specified in F do not hold in the database any longer because of the modification. Efficient validation requires that FDs be verified from single relation schemas—that is, without joining two or more relations for verification.

Let us review the examples of normalization from the previous sections to see if the normalized decompositions are dependency-preserving. In the 2NF example (see Section 8.1.2):

F: fd1: **Album\_no → Price**; fd2: **Album\_no → Stock**  
is specified over the relation schema:

R: NEW\_ALBUM (**Album\_no, Artist\_nm, Price, Stock**)

The 2NF decomposition of NEW\_ALBUM is:

D: R1: ALBUM\_INFO (**Album\_no, Price, Stock**); R2: ALBUM\_ARTIST (**Album\_no, Artist\_nm**)

First of all, the decomposition is attribute-preserving since the union of all attributes in D is exactly the same as the attributes in R. The union of the FDs that hold on individual relation schemas of D are:

**Album\_no → Price**; and **Album\_no → Stock**

Since this set of FDs is exactly equivalent to F specified on R, the 2NF decomposition, D, is dependency-preserving. Note that it is impossible to decompose NEW\_ALBUM any other way to achieve a 2NF design.

---

<sup>6</sup>The lossless-join property is also referred to as non-additive join property or sometimes non-loss property.

Also note that  $F^+$  contains other non-trivial FDs:

$F^+ : \text{fd12: } \{\text{Album\_no}\} \rightarrow \{\text{Price, Stock}\}; \text{ fd12x: } \{\text{Album\_no, Artist_nm}\} \rightarrow \{\text{Price, Stock}\}$

fd12 and fd12x can be generated from the FDs preserved in D—specifically, in this case, from ALBUM\_INFO using Armstrong's axioms.

Next, consider the 3NF example in Section 8.1.3. Here:

$F : \text{fd1: } \{\text{Flight\#}\} \rightarrow \{\text{Origin}\}; \text{ fd2: } \{\text{Flight\#}\} \rightarrow \{\text{Destination}\}; \text{ fd3: } \{\text{Origin, Destination}\} \rightarrow \{\text{Mileage}\}$

is specified over the relation schema:

$R : \text{FLIGHT } (\{\text{Flight\#}, \text{Origin}, \text{Destination}, \text{Mileage}\})$

The 3NF decomposition of FLIGHT is:

$D : R1: \text{FLIGHT } (\{\text{Flight\#}, \text{Origin}, \text{Destination}\}); R2: \text{DISTANCE } (\{\text{Origin, Destination, Mileage}\})$

Once again, the decomposition is attribute-preserving since the union of all attributes in D is exactly the same as the attributes in R. The union of the FDs that hold on individual relation schemas of D is:

$\text{Flight\#} \rightarrow \{\text{Origin}\}; \text{ and Flight\#} \rightarrow \{\text{Destination}\}; \text{ (in R1) and } \{\text{Origin, Destination}\} \rightarrow \{\text{Mileage}\}$   
(in R2)

Since this set of FDs is exactly equivalent to F specified on R, the 3NF decomposition, D, is dependency-preserving.

$F^+$  contains other non-trivial FDs listed here:

$F^+ : \text{fd12: } \{\text{Flight\#}\} \rightarrow \{\text{Origin, Destination}\}; \text{ fd3x: } \{\text{Flight\#}\} \rightarrow \{\text{Mileage}\};$   
 $\text{fd123: } \{\text{Flight\#}\} \rightarrow \{\text{Origin, Destination, Mileage}\}$

All the FDs in  $F^+$  can be generated from the FDs preserved in D using Armstrong's axioms.

Unlike the 2NF example, here, two other decompositions of FLIGHT are possible:

$D : R1a: \text{FLIGHT\_A } (\{\text{Flight\#}, \text{Origin}, \text{Destination}\}); R2a: \text{DISTANCE\_A } (\{\text{Flight\#}, \text{Mileage}\})$   
and:

$D : R1b: \text{FLIGHT\_B } (\{\text{Flight\#}, \text{Mileage}\}); R2b: \text{DISTANCE\_B } (\{\text{Origin, Destination, Mileage}\})$

Interestingly, both decompositions yield a design that is in 3NF. Both decompositions are attribute-preserving since the union of all attributes in D in each case is exactly the same as the attributes in R.

Let us explore if these two designs are dependency-preserving. The union of the FDs that hold on individual relation schemas of D: {R1a, R2a} is:

$\text{Flight\#} \rightarrow \{\text{Origin}\}; \text{ and Flight\#} \rightarrow \{\text{Destination}\}; \text{ (in R1a) and Flight\#} \rightarrow \{\text{Mileage}\}$  (in R2a)

Observe that this set of FDs is not a cover for F because it is impossible to deduce  $\{\text{Origin, Destination}\} \rightarrow \{\text{Mileage}\}$  of F from this set of FDs. Therefore, D: {R1a, R2a} is not a dependency-preserving design.

Likewise, for the second design, D: {R1b, R2b}, the union of the FDs that hold on individual relation schemas of D is:

$\text{Flight\#} \rightarrow \{\text{Mileage}\}; \{\text{Origin, Destination}\} \rightarrow \{\text{Mileage}\}$

Observe that from this set of FDs it is impossible to derive fd1:  $\text{Flight\#} \rightarrow \{\text{Origin}\}$ ; and fd2:  $\text{Flight\#} \rightarrow \{\text{Destination}\}$  of F. Therefore, D: {R1b, R2b} is not a dependency-preserving design either.

Let us now compare an example of a dependency-preserving decomposition (D: {R1, R2}) with another decomposition where dependency is not preserved (D: {R1a, R2a}) to get a better grip on what dependency preservation actually means. The table at the top of Figure 8.5 displays the relation instance for FLIGHT that is not in 3NF. Nonetheless, the FD  $\{\text{Origin, Destination}\} \rightarrow \{\text{Mileage}\}$  is preserved in FLIGHT. The dependency-preserving decomposition D: {FLIGHT, DISTANCE} and a decomposition, D{FLIGHT\_A, DISTANCE\_A}, that does not preserve the FD,  $\{\text{Origin, Destination}\} \rightarrow \{\text{Mileage}\}$ , are also included in Figure 8.5.

R: FLIGHT (Flight#, Origin, Destination, Mileage)

**FLIGHT**

Flight#	Origin	Destination	Mileage
DL507	Seattle	Denver	1537
DL123	Chicago	Dallas	1058
DL723	Boston	St. Louis	1214
DL577	Denver	Los Angeles	1100
DL5219	Minneapolis	St. Louis	580
DL357	Chicago	Dallas	1058
DL555	Denver	Houston	1100
DL5237	Cleveland	St. Louis	580
DL5271	Chicago	Cleveland	300

0<--- R2 ---> n

D: R1: FLIGHT (Flight#, Origin, Destination); R2: DISTANCE (Origin, Destination, Mileage)

1            1

**FLIGHT**

Flight#	Origin	Destination
DL507	Seattle	Denver
DL123	Chicago	Dallas
DL723	Boston	St. Louis
DL577	Denver	Los Angeles
DL5219	Minneapolis	St. Louis
DL357	Chicago	Dallas
DL555	Denver	Houston
DL5237	Cleveland	St. Louis
DL5271	Chicago	Cleveland

**DISTANCE**

Origin	Destination	Mileage
Seattle	Denver	1537
Chicago	Dallas	1058
Boston	St. Louis	1214
Denver	Los Angeles	1100
Minneapolis	St. Louis	580
Denver	Houston	1100
Cleveland	St. Louis	580
Chicago	Cleveland	300

Legal

Illegal

0 R2 n

D: R1a: FLIGHT\_A (Flight#, Origin, Destination); R2a: DISTANCE\_A (Flight, Mileage)

1            1

**FLIGHT\_A**

Flight#	Origin	Destination
DL507	Seattle	Denver
DL123	Chicago	Dallas
DL723	Boston	St. Louis
DL577	Denver	Los Angeles
DL5219	Minneapolis	St. Louis
DL357	Chicago	Dallas
DL555	Denver	Houston
DL5237	Cleveland	St. Louis
DL5271	Chicago	Cleveland

**DISTANCE\_A**

Flight	Mileage
DL507	1537
DL123	1058
DL723	1214
DL577	1100
DL5219	580
DL357	1058
DL555	1100
DL5237	580
DL5271	300

Legal

Legal

**Note:** Shading indicates that addition of the tuple is “legal.” Strike-out indicates that addition of the tuple is “illegal.”

**FIGURE 8.5** A demonstration of the dependency preservation property

The clarification sought here is about what it means to not preserve the FD **{Origin, Destination} → Mileage**. Suppose we want to add a new flight (**Flight#**: DL111, **Origin**: Seattle, **Destination**: Denver, **Mileage**: 1300). In the 3NF decomposition that is dependency-preserving (the set of tables in the middle of Figure 8.5), it is legal to add the tuple (**Flight#**: DL111, **Origin**: Seattle, **Destination**: Denver) to FLIGHT, while it is not possible to add the tuple (**Origin**: Seattle, **Destination**: Denver, **Mileage**: 1300) to DISTANCE because **{Origin, Destination}** is the primary key of DISTANCE. Since a tuple with values (**Origin**: Seattle, **Destination**: Denver, **Mileage**: 1537) already exists in DISTANCE, another tuple that contains a duplicate value for the primary key cannot be added. Thus, **{Origin, Destination} → Mileage** continues to mean that for a given value of **{Origin, Destination}**, say ('Seattle', 'Denver'), there is a single, specific value of **Mileage**, say 1537—that is, the FD is preserved. Also, when the two relations (R1 and R2) are joined, the FD **{Origin, Destination} → Mileage** will continue to be preserved as in the original FLIGHT table from which the decomposition occurred.

On the other hand, in the 3NF decomposition that is *not* dependency-preserving (the set of tables at the bottom of Figure 8.5), it certainly is legal to add the tuple (**Flight#**: DL111, **Origin**: Seattle, **Destination**: Denver) to FLIGHT, and it is equally legal to add the tuple (**Flight#**: DL111, **Mileage**: 1300) to DISTANCE. Clearly, it is not possible to verify the FD **{Origin, Destination} → Mileage** from any *single* relation. If there is a need to combine (join) multiple relations (in this case, two) to check for an FD, then, by definition, that dependency is not preserved in the relational schema.

What does this entail? This can be seen by joining the two relations, R1a and R2a. When the two relations are joined, observe that the FD **{Origin, Destination} → Mileage** is not preserved because there is a tuple (**Flight#**: DL111, **Origin**: Seattle, **Destination**: Denver, **Mileage**: 1537) and another tuple (**Flight#**: DL111, **Origin**: Seattle, **Destination**: Denver, **Mileage**: 1300) in the joined relation; this means that for ('Seattle', 'Denver') we do not have a single, specific value of **Mileage**; there are two values, 1537 and 1300. This demonstrates the seriousness of failing to preserve the FD, **{Origin, Destination} → Mileage**. Essentially, the database has been contaminated with incorrect data. In short, failure to preserve the specified functional dependencies renders the resulting database vulnerable to contamination in the context of the business rules conveyed by the specified functional dependencies.

A similar analysis can be conducted about the other 3NF decomposition, D: {R1b, R2b}. It is important to note that the simple algorithm reflected by the two-step process prescribed will always yield a dependency-preserving decomposition for a relation schema that violates 2NF or 3NF (e.g., D: {R1, R2}).

### 8.1.5.2 Lossless-Join (Non-Additive Join) Property

The basic principle behind a **lossless-join decomposition** is that the decomposition of a relation schema, R, should be strictly reversible—that is, losslessly reversible in that the reversal should yield the original target relation intact with no loss of tuple or no additional spurious tuples. The condition stated holds on all legal states of a relation schema—"legal" state in the sense that the relation state conforms to the set of FDs, F, specified on the relation schema, R. It is important to note that the lossless-join property:

1. Is always stated with respect to a set of FDs, and
2. Is predicated on the premise that the join attributes in the decomposition are non-null values.

Even though the decomposition of a relation schema during normalization may ultimately yield a set of multiple relation schemas, in the stepwise process of normalization, each step typically involves a decomposition that produces two relation schemas—that is, a **binary decomposition**. Therefore, we will address the lossless-join property in binary decompositions.

Formally, a decomposition  $D: \{R1, R2\}$  of a relation schema,  $R$ , is lossless (non-additive) with respect to a set of FDs,  $F$  specified on  $R$ , if for every relation state  $r$  of  $R$  that satisfies  $F$ , the natural join of  $r(R1)$  and  $r(R2)$  strictly yields  $r(R)$ , from which the projections  $r(R1)$  and  $r(R2)$  emerged.

Note that the term “loss” in lossless-join implies loss of information, not loss of tuples. In fact, *loss join* occurs when the natural join of  $r(R1)$  and  $r(R2)$  yields  $r'(R)$ , which includes additional spurious tuples beyond  $r(R)$  from which  $r(R1)$  and  $r(R2)$  are projected. The additional spurious tuples amount to loss of information because their presence corrupts the semantics of the source relation schema,  $R$ —that is, the FDs specified on  $R$  no longer hold good in  $r'(R)$ . Then, the decomposition is a loss-join decomposition, not a lossless-join decomposition.

At this point, let us revisit the FLIGHT example. The three 3NF solutions for this example are reproduced here:

$F: fd1: \text{Flight\#} \rightarrow \text{Origin}; fd2: \text{Flight\#} \rightarrow \text{Destination}; fd3: \{\text{Origin}, \text{Destination}\} \rightarrow \text{Mileage}$  is specified over the relation schema:

$R: \text{FLIGHT } (\text{Flight\#}, \text{Origin}, \text{Destination}, \text{Mileage})$

The three 3NF solutions of FLIGHT are the decompositions:

D: R1: FLIGHT ( <u>Flight#</u> , Origin, Destination);	R2: DISTANCE ( <u>Origin</u> , <u>Destination</u> , Mileage)
D: R1a: FLIGHT_A ( <u>Flight#</u> , Origin, Destination);	R2a: DISTANCE_A ( <u>Flight#</u> , Mileage)
D: R1b: FLIGHT_B ( <u>Flight#</u> , Mileage);	R2b: DISTANCE_B ( <u>Origin</u> , <u>Destination</u> , Mileage)

Figure 8.6a shows an instance of FLIGHT and the decompositions corresponding to the three 3NF solutions, each with its associated reversals. In case 1,  $D: \{\text{FLIGHT}, \text{DISTANCE}\}$  is strictly reversible because  $\{\text{FLIGHT} * \text{DISTANCE}\}$  yields exactly the source relation instance,  $\{\text{FLIGHT}\}$ . Thus, the decomposition is, by definition, a lossless-join decomposition. In the second case also, the natural join  $\{\text{FLIGHT}_A * \text{DISTANCE}_A\}$  of the projections  $\{\text{FLIGHT}_A\}$  and  $\{\text{DISTANCE}_A\}$  produces the source relation instance  $\{\text{FLIGHT}\}$  intact. So, this solution is also a lossless-join decomposition. In case 3, however, the natural join  $\{\text{FLIGHT}_B * \text{DISTANCE}_B\}$  of the projections  $\{\text{FLIGHT}_B\}$  and  $\{\text{DISTANCE}_B\}$  produces a relation state of  $\{\text{FLIGHT}\}$  that contains four additional tuples beyond what is present in the source relation instance  $\{\text{FLIGHT}\}$ . A close examination of this relation state reveals that the presence of these new tuples changes the semantics incorporated in the original design of the relation schema, FLIGHT. For instance, the source relation instance  $\{\text{FLIGHT}\}$  precisely indicates that Flight# 577 flies from Denver to Los Angeles. However, from the case 3 solution we infer that Flight# 577 flies from Denver to Los Angeles as well as from Denver to Houston, which is incorrect according to the FDs specified over the relation schema, FLIGHT. Therefore, even though we have a 3NF solution, the result is not a lossless-join decomposition.

## Normal Forms Based on Functional Dependencies

413

R: FLIGHT (Flight#, Origin, Destination, Mileage)

FLIGHT

Flight#	Origin	Destination	Mileage
DL507	Seattle	Denver	1537
DL123	Chicago	Dallas	1058
DL723	Boston	St. Louis	1214
DL577	Denver	Los Angeles	1100
DL5219	Minneapolis	St. Louis	580
DL357	Chicago	Dallas	1058
DL555	Denver	Houston	1100
DL5237	Cleveland	St. Louis	580
DL5271	Chicago	Cleveland	300

**Case 1:**

$0 < \dots R2 \dots > n$

D: {FLIGHT, DISTANCE} where R1: FLIGHT (Flight#, Origin, Destination); R2: DISTANCE (Origin, Destination, Mileage)

1      1

FLIGHT

Flight#	Origin	Destination
DL507	Seattle	Denver
DL123	Chicago	Dallas
DL723	Boston	St. Louis
DL577	Denver	Los Angeles
DL5219	Minneapolis	St. Louis
DL357	Chicago	Dallas
DL555	Denver	Houston
DL5237	Cleveland	St. Louis
DL5271	Chicago	Cleveland

DISTANCE

Origin	Destination	Mileage
Seattle	Denver	1537
Chicago	Dallas	1058
Boston	St. Louis	1214
Denver	Los Angeles	1100
Minneapolis	St. Louis	580
Denver	Houston	1100
Cleveland	St. Louis	580
Chicago	Cleveland	300

{FLIGHT \* DISTANCE}

Flight#	Origin	Destination	Mileage
DL507	Seattle	Denver	1537
DL123	Chicago	Dallas	1058
DL723	Boston	St. Louis	1214
DL577	Denver	Los Angeles	1100
DL5219	Minneapolis	St. Louis	580
DL357	Chicago	Dallas	1058
DL555	Denver	Houston	1100
DL5237	Cleveland	St. Louis	580
DL5271	Chicago	Cleveland	300

**Case 2:**

$0 \ R2 \ n$

D: {FLIGHT\_A, DISTANCE\_A} where R1a: FLIGHT\_A (Flight#, Origin, Destination); R2a: DISTANCE\_A (Flight, Mileage)

1      1

FLIGHT\_A

Flight#	Origin	Destination
DL507	Seattle	Denver
DL123	Chicago	Dallas
DL723	Boston	St. Louis
DL577	Denver	Los Angeles
DL5219	Minneapolis	St. Louis
DL357	Chicago	Dallas
DL555	Denver	Houston
DL5237	Cleveland	St. Louis
DL5271	Chicago	Cleveland

DISTANCE\_A

Flight	Mileage
DL507	1537
DL123	1058
DL723	1214
DL577	1100
DL5219	580
DL357	1058
DL555	1100
DL5237	580
DL5271	300

{FLIGHT\_A \* DISTANCE\_A}

Flight#	Origin	Destination	Mileage
DL507	Seattle	Denver	1537
DL123	Chicago	Dallas	1058
DL723	Boston	St. Louis	1214
DL577	Denver	Los Angeles	1100
DL5219	Minneapolis	St. Louis	580
DL357	Chicago	Dallas	1058
DL555	Denver	Houston	1100
DL5237	Cleveland	St. Louis	580
DL5271	Chicago	Cleveland	300

**Case 3:**

$0 \ R2 \ n$

D: {FLIGHT\_B, DISTANCE\_B} where R1b: FLIGHT\_B (Origin, Destination, Mileage); R2b: DISTANCE\_B (Flight, Mileage)

1      1

FLIGHT\_B

Origin	Destination	Mileage
Seattle	Denver	1537
Chicago	Dallas	1058
Boston	St. Louis	1214
Denver	Los Angeles	1100
Minneapolis	St. Louis	580
Denver	Houston	1100
Cleveland	St. Louis	580
Chicago	Cleveland	300

DISTANCE\_B

Flight	Mileage
DL507	1537
DL123	1058
DL723	1214
DL577	1100
DL5219	580
DL357	1058
DL555	1100
DL5237	580
DL5271	300

{FLIGHT\_B \* DISTANCE\_B}

Flight#	Origin	Destination	Mileage
DL507	Seattle	Denver	1537
DL123	Chicago	Dallas	1058
DL723	Boston	St. Louis	1214
DL577	Denver	Los Angeles	1100
DL5219	Minneapolis	St. Louis	580
DL5219	Cleveland	St. Louis	580
DL357	Chicago	Dallas	1058
DL555	Denver	Los Angeles	1100
DL555	Denver	Houston	1100
DL5237	Minneapolis	St. Louis	580
DL5237	Cleveland	St. Louis	580
DL5271	Chicago	Cleveland	300

Note: Spurious tuples causing a loss-join decomposition are marked by shading.

**FIGURE 8.6a** A demonstration of lossless and loss-join decompositions

A review of this example in Section 8.1.5.1 combined with the above analysis indicates that while all three solutions yield 3NF decompositions, the following distinctions should be noted by the reader:

- {FLIGHT, DISTANCE} delivers a lossless-join decomposition that is also dependency-preserving with respect to the FDs, F, specified on the relation schema FLIGHT.
- {FLIGHT\_A, DISTANCE\_A} delivers a lossless-join decomposition. However, the decomposition is *not* dependency-preserving with respect to the FDs, F, specified on the relation schema FLIGHT.
- {FLIGHT\_B, DISTANCE\_B} delivers a *loss-join* decomposition that also fails to preserve the FDs, F, specified on the relation schema FLIGHT.

414

Clearly, the decomposition {FLIGHT, DISTANCE} is the only acceptable design. Once again, note that the simple algorithm incorporated in the two-step decomposition process always yields 2NF and 3NF solutions that are dependency-preserving and also possess the lossless-join property.

A test for verifying the lossless-join property of a binary decomposition can be specified as follows. A decomposition D: {R1, R2} of a relation schema, R, is a lossless-join decomposition with respect to a set of FDs, F that holds on R, if and only if  $F^+$  contains:

- either the FD  $(R1 \cap R2) \rightarrow R1$
- or the FD  $(R1 \cap R2) \rightarrow R2$

In other words, the attribute(s) common to R1 and R2 must contain a candidate key of either R1 or R2.<sup>7</sup> In our example, the join attribute in case 1 solution is {Origin, Destination}, which is the primary key of DISTANCE. Likewise, in the decomposition in case 2, the join attribute, Flight#, is the primary key of both FLIGHT\_A and DISTANCE\_A. Therefore, both these solutions offer lossless-join decompositions, as confirmed by the data in Figure 8.6a. The third solution presented, however, fails the test for lossless-join decomposition prescribed above, because the join attribute in this case, Mileage, does not contain a candidate key in either FLIGHT\_B or DISTANCE\_B. Once again, the spurious tuples in the natural join {FLIGHT\_B \* DISTANCE\_B} (see Figure 8.6a) confirm this.

### 8.1.5.3 BCNF Design, Dependency-Preservation, and Lossless-Join Decomposition

Decomposition of a relation schema that is in 3NF but is in violation of BCNF poses an interesting trade-off. Let us study this using the STU\_SUB example from Section 8.1.4. Here, F:

fd1: {Stu#, Subject} → Teacher;      fd2: {Stu#, Subject} → Ap\_score;      fd3: Teacher → Subject

is specified over the relation schema:

R:STU\_SUB (Stu#, Subject, Teacher, Ap\_score)

STU\_SUB is in 3NF but violates BCNF since Teacher, the determinant in fd3, is not a superkey of STU\_SUB.

---

<sup>7</sup>A more general specification of the test is: The FD  $(R1 \cap R2) \rightarrow (R1 - R2)$  or the FD  $(R1 \cap R2) \rightarrow (R2 - R1)$ .

From an alternative viewpoint, from F, it is possible to infer that:

fd4: {**Stu#, Teacher**} → **Subject**;

fd5: {**Stu#, Teacher**} → **Ap\_score**;

exist in  $F^+$ . Thus, {**Stu#, Teacher**} is a candidate key of **STU\_SUB** and, if treated as the primary key of **STU\_SUB**, renders fd3 a violation of 2NF.

Either way, the decomposition of **STU\_SUB** derived in Section 8.1.4 as the solution is:

D: R1: TEACH\_SUB (**Teacher**, **Subject**);

R2: STU\_AP (**Stu#**, **Teacher**, **Ap\_score**)

First of all, the decomposition is attribute-preserving since the union of all attributes in D is exactly the same as the attributes in R. The solution is a lossless-join decomposition because  $(TEACH\_SUB \cap STU\_AP)$  is the attribute **Teacher**, which is the candidate key of **TEACH\_SUB**. This can be verified by doing a natural join of the relations **TEACH\_SUB** and **STU\_AP** that appear in Figure 8.4. This is shown in Figure 8.6b.

Incidentally, it is important to note that while  $G\{fd1, fd2, fd3\}$  is a cover (in fact, a minimal cover) for  $F^+$ ,  $G'\{fd1, fd3, fd4\}$  is not a cover for  $F^+$ .

Is the solution D {R1, R2} dependency-preserving? The union of the FDs that hold on individual relation schemas of D is:

**Teacher** → **Subject** (in R1) and {**Stu#, Teacher**} → **Ap\_score** (in R2)

Clearly, this set of FDs is not a cover for F. In other words, while fd3 (**Teacher** → **Subject**) is preserved, fd1 {**Stu#, Subject**} → **Teacher** and fd2 {**Stu#, Subject**} → **Ap\_score** are not preserved in the solution. Thus the solution is not dependency-preserving.

Is there another decomposition that preserves all the FDs specified in F? Clearly, the two-step decomposition procedure that yields a 2NF decomposition and a 3NF decomposition possessing both the dependency-preserving and lossless-join properties has failed to produce a BCNF solution that preserves both the properties. A second solution is shown here:

D: R1a: TEACH\_SUB1 (**Teacher**, **Subject**);

R2a: STU\_AP1 (**Stu#**, **Subject**, **Ap\_score**)

The decomposition is in BCNF. The union of the FDs that hold on individual relation schemas of D {R1a, R2a} is:

**Teacher** → **Subject** (in R1a) and {**Stu#, Subject**} → **Ap\_score** (in R2a)

Once again, this set of FDs is not a cover for F. In other words, while this solution preserves fd2 and fd3, fd1 is not preserved in this solution. In fact, *there is no BCNF solution that can preserve fd1*. Moreover, this second solution also fails to produce a lossless-join decomposition, as can be seen by applying the prescribed test for lossless-join decomposition. In this case,  $(TEACH\_SUB1 \cap STU\_AP1)$  is the attribute **Subject**, which is neither the candidate key of **TEACH\_SUB1** nor the candidate key of **STU\_AP1**. Accordingly, the decomposition yields a *loss-join*. This can be verified by constructing the natural join of the relations **TEACH\_SUB1** and **STU\_AP1**, as shown in Figure 8.6b. The spurious tuples that result from the natural join  $(TEACH\_SUB1 * STU\_AP1)$  are a clear proof for the absence of lossless-join property in the solution.

R: STU\_SUB (Stu#, Subject, Teacher, Ap\_score)  
 STU\_SUB

Stu#	Subject	Teacher	Ap_score
IH123	Chemistry	Raturi	4
IH123	English	Stephan	4
IH235	History	Walker	5
IH357	English	Campbell	4
IH571	Chemistry	Raturi	3
IH235	English	Campbell	4

F      fd1: (Stu#, Subject) → Teacher;  
 fd2: (Stu#, Subject) → Ap\_score;  
 fd3: Teacher → Subject

Option 1

D:      R1: TEACH\_SUB (Teacher, Subject);      R2: STU\_AP (Stu#, Teacher, Ap\_score)  
 0      R1 n  
 1      1

STU\_AP

Stu#	Teacher	Ap_score
IH123	Raturi	4
IH123	Stephan	4
IH235	Walker	5
IH357	Campbell	4
IH571	Raturi	3
IH235	Campbell	4

TEACH\_SUB

Teacher	Subject
Raturi	Chemistry
Stephan	English
Walker	History
Campbell	English

(STU\_AP \* TEACH\_SUB)

Stu#	Subject	Teacher	Ap_score
IH123	Chemistry	Raturi	4
IH123	English	Stephan	4
IH235	History	Walker	5
IH357	English	Campbell	4
IH571	Chemistry	Raturi	3
IH235	English	Campbell	4

Option 2

D:      R1a: TEACH\_SUB1 (Teacher, Subject);      R2a: STU\_AP1 (Stu#, Subject, Ap\_score)  
 0      R1a n  
 1      1

STU\_AP1

Stu#	Subject	Ap_score
IH123	Chemistry	4
IH123	English	4
IH235	History	5
IH357	English	4
IH571	Chemistry	3
IH235	English	4

TEACH\_SUB1

Teacher	Subject
Raturi	Chemistry
Stephan	English
Walker	History
Campbell	English

(STU\_AP1 \* TEACH\_SUB1)

Stu#	Subject	Teacher	Ap_score
IH123	Chemistry	Raturi	4
IH123	English	Stephan	4
IH235	History	Walker	5
IH357	English	Stephan	4
IH357	English	Campbell	4
IH571	Chemistry	Raturi	3
IH235	English	Stephan	4
IH235	English	Campbell	4

**Note:** Spurious tuples causing a loss-join decomposition are marked by shading.

**FIGURE 8.6b** Lossless-join and dependency preservation in a BCNF resolution

In sum, since both solutions do not provide full dependency preservation, the designer's choice ought to be the solution that generates at least lossless-join projections, if BCNF design is desired. If one has to aim for one of these two properties, a lossless-join condition is an absolute must (Elmasri and Navathe, 2010, p. 357).

Assuming that we always seek a lossless-join condition, if we are forced to choose between BCNF without preserving dependencies and 3NF with preserved dependencies, it is generally preferable to opt for the latter. After all, if one can't test for dependency preservation efficiently, one either pays a high penalty in system performance or risks the integrity of the data in the database. Neither is an attractive alternative. Thus, the limited amount of redundancy allowed under 3NF is regarded as the lesser of the two evils.

Therefore, the design goals can be expressed in two basic options:

### **Option 1**

BCNF

Lossless-join

Dependency preservation

This is the ideal option and is achieved only when a relational schema is in 3NF and there are no BCNF violations in the relational schema, because then the relational schema is also in BCNF.

If the above design cannot be achieved, we may have to settle for:

### **Option 2**

3NF

Lossless-join

Dependency preservation

That said, with the advent of materialized views,<sup>8</sup> it is possible to always achieve Option 1 rather cost-effectively. In the absence of BCNF (Option 2), the application developer assumes the responsibility to keep redundant data consistent programmatically when modifications to the database occur. If we opt for a BCNF design (Option 1), the cost of application programming incurred in Option 2 is eliminated. However, we need to supplement the BCNF design with a materialized view for each unpreserved FD in the minimal cover of F. The advantage of this approach is that the DBMS takes care of the maintenance of the materialized views as modifications occur in the source relations, thus assuring preservation of the associated dependencies. While the DBMS overhead for the maintenance of materialized views requires consideration, BCNF violations are usually few and far between. Costs and inefficiencies associated with application programming in this situation are often far more burdensome than the DBMS overhead.

---

<sup>8</sup>A *view* defines a “virtual” relation schema constructed from one or more base relation schemas; unlike base relations, a view does not store data; the value of a view at any given time is a ‘derived’ relation and results from the evaluation of a specified relational expression at that time. A view is just a logical window to view selected data (attributes and tuples) from one or a set of relation schemas. *Materialized views* (also known as *snapshots*) are also derived like views except that they are stored in the database and refreshed on every modification (i.e., maintained current by the DBMS as modifications occur) in the source relations from where the materialized views are generated.

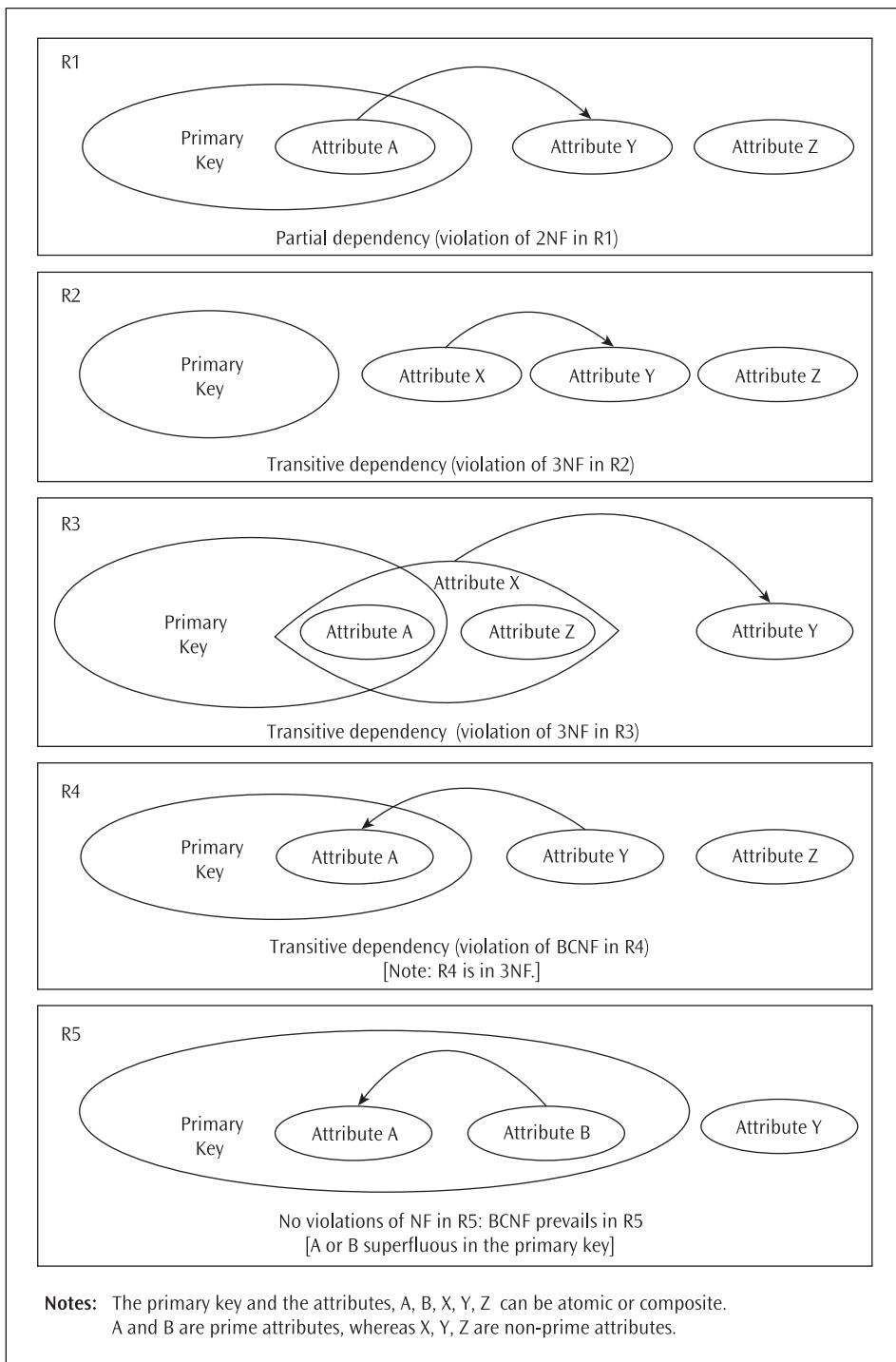
In conclusion, where a dependency-preserving BCNF design is not possible, it is generally preferable to opt for BCNF (Option 1) and supplement it with materialized views to preserve dependencies (Silberschatz, Korth, and Sudarshan, 2010).

### 8.1.6 Summary Notes on Normal Forms

There are several points about normal forms and the normalization process that are noteworthy. Normalization systematically eliminates data redundancies that cause modification anomalies by decomposing the target relation schema that contains undesirable FDs. The decomposition is done in such a way that the FDs that were “undesirable” in the target relation schema are not discarded, but are rendered “desirable” in the resulting relational schema (set of relation schemas). Figure 8.7 summarizes normal form violations due to functional dependencies in a nutshell.

Observe that in the normalization process, there is no specific merit in resolving partial dependencies (2NF violations) before transitive dependencies (3NF violations) are resolved. It simply happens that historically 3NF has been defined with an assumption that a relation schema is tested for 2NF first and then for 3NF. Next, the definition of BCNF is the simplest and yet the strongest and most general among the definitions of normal forms associated with functional dependencies. Interestingly, it can be seen that a 2NF violation is also a BCNF violation. Similarly, a 3NF violation is also a BCNF violation. That is, BCNF, by definition, subsumes 2NF and 3NF. Nonetheless, in practice, it is useful to view the normal form violations in terms of partial dependency (2NF violation) and transitive dependency (3NF violation) and BCNF violation so that the database designer may systematically approach the problem and its solution in incremental steps. Attempting to solve all normal form violations due to functional dependencies as BCNF violations can be overwhelming because the approach to the solution can get tentative. In this context, hereafter we will use the term “immediate” violation to refer to violations of 2NF, 3NF, and BCNF so that we do not arbitrarily refer to any undesirable FD as a BCNF violation and be right.

For a database design to be robust, it is not enough for the designer to focus exclusively on the elimination of data redundancies that lead to modification anomalies. The designer must also pay attention to the dependency preservation property and lossless-join property of the design. Given  $R | F$ , it is possible to derive a dependency-preserving non-loss decomposition,  $D$  of  $R$ , that does not violate 3NF.

**FIGURE 8.7** Normal forms based on functional dependencies in a nutshell

## 8.2 THE MOTIVATING EXEMPLAR REVISITED

At this point, normalization concepts have been presented by analyzing 1NF, 2NF, 3NF, and BCNF in isolation. In practice, however, normal form violations rarely occur in isolation. Therefore, we now explore a comprehensive approach to normalization by studying how, given a set of FDs derived from user-specified business rules, one develops a fully normalized relational schema from the *universal relation schema* that depicts the stated functional dependencies.

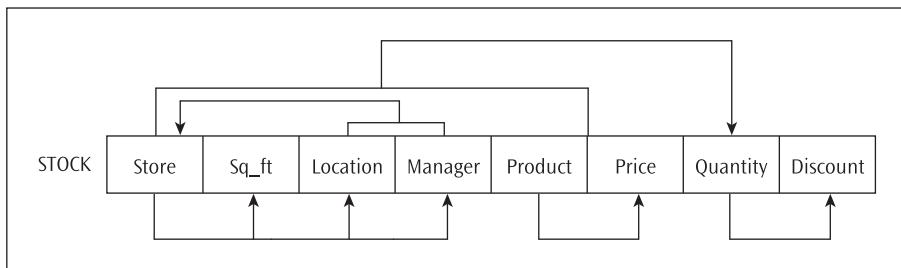
To begin with, let us revisit the motivating exemplar in Chapter 7 (see Section 7.1) and understand the process that transpired resulting in the decomposition of STOCK (Figure 7.1c) into STORE, PRODUCT, INVENTORY, and DISC\_STRUCTURE (Figure 7.3).

Based on the data in the relation instance of STOCK in Figure 7.1c, we construct the following minimal cover of FDs, F, for the relation schema:

**R: STOCK (Store, Location, Sq\_ft, Manager, Product, Price, Quantity, Discount)**

fd1: Store → Location;	fd2: Store → Sq_ft;	fd3: Store → Manager;
fd4: Product → Price;	fd5: {Store, Product} → Quantity	fd6: Quantity → Discount
fd7: {Manager, Location} → Store		

The dependency diagram in Figure 8.8a provides a pictorial version of F.



**FIGURE 8.8a** Dependency diagram for the relation schema STOCK

STOCK is in 1NF because there are no composite or multi-valued attributes in it.

Using Armstrong's axioms, we can derive **{Store, Product}** as one of the candidate keys and choose it as the primary key of STOCK. The other candidate key is **{Manager, Location, Product}**.

On the basis of **{Store, Product}** as the primary key of STOCK, fd1, fd2, fd3, and fd4 violate 2NF in STOCK; fd6 violates 3NF in STOCK; and fd7 violates BCNF in STOCK.

Neither 2NF nor 3NF nor BCNF is violated in STOCK by fd5.

Applying the two-step normalization process to the partial dependencies  $fd1$ ,  $fd2$ ,  $fd3$ , and  $fd4$  causing the 2NF violation, we have the decomposition D:

D: {R1, R2, R3, R4, R5}

where:

R1: STORE_LOC ( <u>Store</u> , Location);	R2: STORE_SIZE ( <u>Store</u> , Sq_ft);
R3: STORE_MGR ( <u>Store</u> , Manager);	R4: PRODUCT ( <u>Product</u> , Price);
R5: INVENTORY ( <u>Store</u> , Product, Quantity, Discount)	

and the inclusion dependencies:

$INVENTORY.\{Store\} \subseteq STORE\_LOC.\{Store\}$

$INVENTORY.\{Product\} \subseteq PRODUCT.\{Product\}$

The decomposed relational schema, D, consisting of the five (arbitrarily named) relation schemas—STORE\_LOC, STORE\_SIZE, STORE\_MGR, PRODUCT, and INVENTORY—does not have any 2NF violations anymore; that is, D is in 2NF. In addition, the decomposition is lossless and preserves all the FDs in F.

In fact, STORE\_LOC, STORE\_SIZE, STORE\_MGR, and PRODUCT are also in 3NF and BCNF.

However, INVENTORY is in violation of 3NF because  $fd6: Quantity \rightarrow Discount$  causes a transitive dependency in INVENTORY. Application of the two-step normalization process, once again, leads to the following decomposition of INVENTORY:

R5a: DISC\_STRUCTURE (Quantity, Discount); R5b: INVENTORY (Store, Product, Quantity)

and the inclusion dependency:

$INVENTORY.\{Quantity\} \subseteq DISC\_STRUCTURE.\{Quantity\}$

Both the decomposed relation schema, DISC\_STRUCTURE, and the leftover relation schema, INVENTORY (R5b), are in 3NF as well as in BCNF.

Thus, a BCNF solution yields the following result. The relation schema STOCK where 2NF, 3NF, and BCNF violations were present is replaced by the relational schema that contains the set of relation schemas STORE\_LOC, STORE\_SIZE, STORE\_MGR, PRODUCT, INVENTORY, and DISC\_STRUCTURE, as shown next:

D: {R1, R2, R3, R4, R5a, R5b}

where:

R1: STORE_LOC ( <u>Store</u> , Location);	R2: STORE_SIZE ( <u>Store</u> , Sq_ft);
R3: STORE_MGR ( <u>Store</u> , Manager);	R4: PRODUCT ( <u>Product</u> , Price);
R5a: DISC_STRUCTURE ( <u>Quantity</u> , <u>Discount</u> );	R5b: INVENTORY ( <u>Store</u> , <u>Product</u> , <u>Quantity</u> )

and the inclusion dependencies:

$INVENTORY.\{Store\} \subseteq STORE\_LOC.\{Store\}$

$INVENTORY.\{Product\} \subseteq PRODUCT.\{Product\}$

$INVENTORY.\{Quantity\} \subseteq DISC\_STRUCTURE.\{Quantity\}$

In addition, the decomposition is lossless and preserves all the FDs in F except  $fd7$ .

Since the primary keys of R1, R2, and R3 are the same, the three relations can be consolidated into one without compromising the normal form attained—that is, BCNF.

Observe that, incidentally, this consolidation also preserves fd7, and {Manager, Location} is a candidate key of R123. The resulting solution is of the form:

D: {R123, R4, R5a, R5b}  
where:

R123: STORE ( <u>Store</u> , <u>Location</u> , <u>Sq. ft</u> , <u>Manager</u> );	R4: PRODUCT ( <u>Product</u> , <u>Price</u> );
R5a: DISC_STRUCTURE ( <u>Quantity</u> , <u>Discount</u> );	R5b: INVENTORY ( <u>Store</u> , <u>Product</u> , <u>Quantity</u> )

and the inclusion dependencies:

$$\begin{aligned}\text{INVENTORY.}\{\text{Store}\} &\subseteq \text{STORE.}\{\text{Store}\} \\ \text{INVENTORY.}\{\text{Product}\} &\subseteq \text{PRODUCT.}\{\text{Product}\} \\ \text{INVENTORY.}\{\text{Quantity}\} &\subseteq \text{DISC_STRUCTURE.}\{\text{Quantity}\}\end{aligned}$$

The solution yields a BCNF design, retains the lossless-join property, and preserves all the FDs in F. In addition, the design is parsimonious (tighter) because the number of relation schemas in the design is only four, as opposed to the six relation schemas in the previous solution. Clearly, this is a superior design. This is how the set of tables in Figure 7.3, portraying an instance of this relational schema decomposed from the relation schema shown in Figure 7.1, is produced.

In Section 7.1, while previewing the transformation of the STOCK relation in Figure 7.1 to the set of relations in Figure 7.3, we asked three questions. Now, we can answer these questions with clarity.

- *How do we systematically identify data redundancies?*  
Since we now know that the root cause of data redundancies and the resulting modification anomalies in a relation schema R is undesirable FDs, we first identify the FDs that prevail over R. The only desirable FDs in R are the ones where the determinant is a candidate key of R. The rest, the undesirable FDs, manifest in R as partial dependencies (violation of 2NF) and transitive dependencies (violation of 3NF or BCNF).
- *How do we know how to decompose the base relation schema under investigation?*  
The objective is to render all undesirable FDs in R desirable. This is done by decomposing R into a set of relation schemas D: {R1, R2, R3, . . . , Rn} such that the FDs captured in each relation schema of D become desirable FDs. The decomposition is done by systematically resolving the 2NF, 3NF, and BCNF violations, starting from R and then in the successive versions of D.
- *How do we know that the decomposition is correct and complete (without looking at sample data)?*  
A decomposition is “correct” when:
  - The decomposition is attribute-preserving. That is, all attributes of R collectively appear in D (no attribute is lost in the decomposition).
  - The join of D: {R1, R2, . . . , Rn} is strictly equal to R.
  - The decomposition does not have any modification anomalies due to FDs. The relational schema D: {R1, R2, . . . , Rn}, when in BCNF, assures absence of modification anomalies due to FDs and thus minimal data redundancies.

A decomposition is “complete” when it is a dependency-preserving lossless-join decomposition. Preservation of FDs is a verification process and is accomplished by inspecting the decomposition to see if the union of the FDs that hold on individual relation schemas of D is a cover for F (i.e., can generate  $F^+$ ). This is demonstrated in Section 8.1.5.1. One can also test for the lossless-join property. The method of testing is presented in Section 8.1.5.2.

As an alternative to the systematic, rather comprehensive, approach used above, let us try a simplistic method of arriving at a design, paying no heed to a systematic normalization process. Suppose we simply do the following:

- Map each FD in the canonical cover to a relation schema.
- Consolidate relation schemas with the same primary key into a single relation schema.
- Eliminate redundant relation schemas. (Note that a relation schema R in a relational schema is redundant if it is a subset of another relational schema S in the relational schema).

Now, let’s see if we arrive at a parsimonious design that is free of any normal form violation. In the first step, we have:

- R1: STORE\_LOC (Store, Location);
- R2: STORE\_SIZE (Store, Sq\_ft);
- R3: STORE\_MGR (Store, Manager);
- R4: PRODUCT (Product, Price);
- R5: DISC\_STRUCTURE (Quantity, Discount);
- R6: INVENTORY (Store, Product, Quantity);
- R7: MGR\_LOC (Location, Manager, Stock)

Consolidation leads to:

- R123: STORE (Store, Location, Sq\_ft, Manager)
- R4: PRODUCT (Product, Price);
- R5: DISC\_STRUCTURE (Quantity, Discount);
- R6: INVENTORY (Store, Product, Quantity)
- R7: MGR\_LOC (Location, Manager, Stock)

Observe that R7 is a proper subset of R123 and so is redundant. Thus, we have a final design D {R123, R4, R5, R6}. Interestingly, this solution is identical to the normalized solution arrived at earlier. So, why bother with the laborious normalization process?

Based on the above examples, it appears, at first glance, that a simple mapping of every FD in F to a relation schema, and consolidation of relation schemas with the same primary key to a single relation schema, produce the solution sought without any concern about even pursuing the normalization process. BCNF violations often present trade-off conditions in the solution: Dependency preservation may have to be sacrificed to achieve a lossless-join design, or a certain level of data redundancy may have to be tolerated in order to achieve a lossless-join design that is also dependency-preserving. Either way, the simplistic approach we used will not catch such nuances in the result; a systematic approach to the problem when a set of FDs over a relation schema violates various normal forms is, therefore, very much needed. The examples that follow will clearly demonstrate this issue.

## 8.3 A COMPREHENSIVE APPROACH TO NORMALIZATION

In practice, the database designer encounters the normalization task when assessing the quality of a relation schema in the context of the set of FDs that prevails over it. Each relation schema in a relational schema (the set of relation schemas constituting the logical data model) is evaluated individually. Often, most of the relation schemas are already in BCNF by virtue of the accurate allocation of attributes to the entity types in the conceptual modeling stage. However, a few relation schemas do suffer from erroneous attribute allocation, often due to inadvertent representation of a set of related entity types as a single entity type in the ERD (see Section 7.1 for an illustration). In this book, we assume that a conceptual design (ER modeling) precedes the logical design, but the issue is even more relevant when a set of FDs is directly derived from the business rules embedded in the user requirements specification. In our case, we start with a single universal relation schema (URS) constructed as a collection of attributes present in the set of FDs, and use the process of normalization to develop the complete logical schema—in our case, the relational schema.

Given a set of FDs prevailing over a URS either constructed from the set of FDs or mapped from an entity type of an ERD, data redundancies and modification anomalies due to undesirable FDs in the URS usually occur in the form of immediate violations of 2NF, 3NF, and/or BCNF. Although a single relation schema may be, in general, the best design from a data retrieval perspective, the normal form violations indicate that a single relation schema is seldom an effective design since modification anomalies (as indicated by the normal form violations) often impose restrictions on effective use and maintenance of the resulting database. Therefore, the ideal goal of database design is to develop a fully normalized relational schema, preferably in BCNF, that is parsimonious—that is, a tight design containing the least number of relation schemas.

Normalization algorithms do not always yield a parsimonious design, especially if the set of FDs, F, provided is not a minimal cover of F. Where multiple minimal covers are possible, these algorithms usually yield a solution based on some minimal cover, which need not be the most desirable solution—for example, not necessarily a parsimonious solution. When the design goal requires evaluation of trade-offs among a BCNF solution, lossless-join decomposition, and dependency preservation, human intervention is almost always necessary. Furthermore, it is rather critical that the database designer has a thorough understanding of the design.

For these reasons, we resort to a heuristic approach based on a few design guidelines for developing a normalized relational schema given a set of FDs. We have already been initiated into this process somewhat informally when we revisited the motivating exemplar from Section 7.1 of Chapter 7 in Section 8.2. In this section, we examine a comprehensive approach where trade-off among a BCNF solution, a lossless-join decomposition, and dependency preservation in the normalization process is dealt with. In the interest of continuity and as a means to better learning, we use the same three examples that were used for the derivation of candidate keys in Chapter 7.

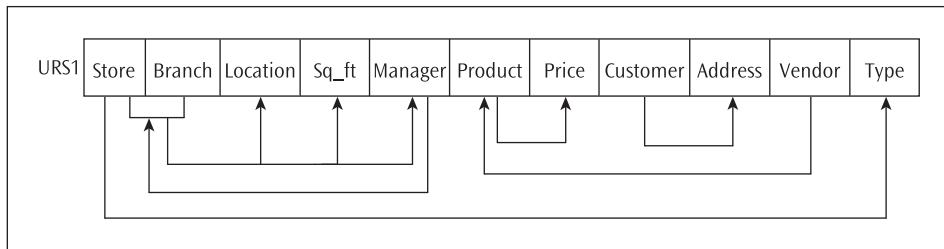
### 8.3.1 Case 1

Let us investigate the example presented in Section 7.3.1. The URS and the set of FDs that hold in it are reproduced here:

**URS1 (Store, Branch, Location, Sq\_ft, Manager, Product, Price, Customer, Address, Vendor, Type)**

fd1: {Store, Branch} → Location;	fd5: Product → Price;
fd2: Customer → Address;	fd6: {Store, Branch} → Manager;
fd3: Vendor → Product;	fd7: Manager → {Store, Branch};
fd4: {Store, Branch} → Sq_ft;	fd8: Store → Type;

The set of FDs in F may also be expressed via a dependency diagram, as shown in Figure 8.8b.



**FIGURE 8.8b** Dependency diagram for the relation schema URS1

425

### 8.3.1.1 A Simplistic Approach Ignoring Normalization

Is URS1 normalized? Before answering this question, let us first ignore normalization and write a relational schema by simply mapping every FD to a relation schema and consolidating the relation schemas with the same primary key into single relation schemas. The result is shown here:

D: {R1, R2, R3, R4, R5, R6}

where:

R1: STORE ( <u>Store</u> , <u>Type</u> );	R2: BRANCH ( <u>Store</u> , <u>Branch</u> , <u>Location</u> , <u>Sq_ft</u> , <u>Manager</u> );
R3: CUSTOMER ( <u>Customer</u> , <u>Address</u> );	R4: VENDOR ( <u>Vendor</u> , <u>Product</u> );
R5: PRODUCT ( <u>Product</u> , <u>Price</u> );	R6: MANAGER ( <u>Manager</u> , <u>Store</u> , <u>Branch</u> )

Is this design free of data redundancies/modification anomalies? The answer is “Yes,” because every relation schema in D is in BCNF. Is this design dependency-preserving? Again, the answer is “Yes.” Does the design exhibit lossless-join property? The answer is a resounding “Yes.” Is the design parsimonious? A careful inspection of the relational schema reveals that there is a tighter design that does not compromise on the other three properties. R2 and R6 can be consolidated without any adverse effect on the established condition of a dependency-preserving BCNF solution with an enduring lossless-join

property. This is possible because R2 and R6 have at least one common candidate key. Consequently, a superior design is of the form:

D: {R1, R26, R3, R4, R5}

where:

R1: STORE ( <u>Store</u> , <u>Type</u> );	R3: CUSTOMER ( <u>Customer</u> , <u>Address</u> );
R4: VENDOR ( <u>Vendor</u> , <u>Product</u> );	R5: PRODUCT ( <u>Product</u> , <u>Price</u> );
R26: BRANCH ( <u>Store</u> , <u>Branch</u> , <u>Location</u> , <u>Sq_ft</u> , <u>Manager</u> );	

If the above design is correct and complete, there is no utility in normalization. So, an important question arises: *Is the above design correct and complete?* In Section 8.2, three conditions were specified to determine if a design is correct. Accordingly, the above design is attribute-preserving and each relation schema in the relational schema D {R1, R26, R3, R4, R5} is in BCNF. Does the join of D {R1, R26, R3, R4, R5} strictly yield R? The answer is “No” because while R1 and R26 as well as R4 and R5 can be joined, the results of these joins and R3 are not joinable. So, the design is incorrect. One way to ratify or refute this is to see if we can arrive at the same solution through the normalization process. In order to do that, we return to the original question: Is URS1 normalized?

### 8.3.1.2 The Normalization Approach

The fact that URS1 is a relation schema clarifies that it is in 1NF. Are there any data redundancies in URS1? We know that any violation of 2NF, 3NF, or BCNF implies that data redundancies/modification anomalies attributable to undesirable FDs are present in URS1. Violations of 2NF, 3NF, and/or BCNF are always with reference to a candidate key—invariably, the primary key. Therefore:

**Step 1: Identify the candidate keys of URS1, given the set of FDs, F**—In Chapter 7, we derived the candidate keys of URS1 (see Section 7.3.1.) as:

{**Store**, **Branch**, **Customer**, **Vendor**} and {**Manager**, **Customer**, **Vendor**}

**Step 2: Choose a primary key for URS1**—Since the choice of primary key from among the candidate keys is essentially arbitrary (see Section 6.3.1), using the rules of thumb prescribed in Chapter 7 (see Section 7.3.4), let us choose {**Manager**, **Customer**, **Vendor**} as the primary key of URS1. Then, the other candidate key, {**Store**, **Branch**, **Customer**, **Vendor**}, becomes the alternate key of URS1.

**Step 3: Record the immediate normal form violated in URS1 with respect to the primary key by each of the FDs in F**—The normal form violations are shown in Table 8.1.

URS1 (Store, Branch, Location, Sq_ft, Manager, Product, Price, Customer, Address, Vendor, Type)						
FD	Cover	Role	Determinant	Dependent(s)	NF violated in URS1	
					PK = CK2	AK = CK1
fd1	F		(Store, Branch)	Location	3NF	2NF
fd2	F		Customer	Address	2NF	2NF
fd3	F		Vendor	Product	2NF	2NF
fd4	F		(Store, Branch)	Sq_ft	3NF	2NF
fd5	F		Product	Price	3NF	3NF
fd6	F		(Store, Branch)	Manager	BCNF	2NF
fd7	F		Manager	(Store, Branch)	2NF	BCNF
fd8	F		Store	Type	3NF	2NF
fdx	F <sup>+</sup>		(Store, Branch)	Location, Sq_ft, Manager	3NF	2NF
fdy	F <sup>+</sup>	CK1	(Store, Branch, Customer, Vendor)	Location, Sq_ft, Manager, Address, Product, Price, Type	None	None
fdz	F <sup>+</sup>	CK2	(Manager, Customer, Vendor)	Location, Sq_ft, Store, Branch, Address, Product, Price, Type	None	None

Note: While we are only interested in the normal form violations predicated upon the primary key, the table above also shows the normal form violations with respect to the alternate key just to demonstrate that the same FD can violate different normal forms depending on the candidate key on which the evaluation is based.

**TABLE 8.1** Normal form violations in URS1

**Step 4: Resolve 2NF and 3NF violations in URS1; the sequence of resolution is immaterial—** We use the two-step process prescribed in Sections 8.1.2 and 8.1.3 to resolve the normal form violations. It is important to note that as Step 4 is executed recursively, URS1 ceases to remain a single relation schema. Instead, in each successive execution of this step, the URS1 used is a revised set of decomposed relation schemas, as shown here:

#### Execution 1:

Input: URS1	FD: fd2: <b>Customer → Address</b>	Violation: 2NF in URS1
<i>Resolution:</i> Decomposition URS1 {R1, R0}		
where:		
R1: CUSTOMER ( <u>Customer</u> , Address); <span style="float: right;">in BCNF</span>		
R0: LORS1 ( <u>Manager</u> , <u>Customer</u> , <u>Vendor</u> , <u>Store</u> , <u>Branch</u> , <u>Location</u> , <u>Sq_ft</u> , <u>Product</u> , <u>Price</u> , <u>Type</u> )		
# LORS1.{Customer} ⊆ CUSTOMER.{Customer} <span style="float: right;">lossless-join</span>		
Note: LORS1 is an acronym for <i>Leftover Relation Schema 1</i> . The input to the next execution of this step is URS1 {R1, R0}.		

**Execution 2:**

Input: URS1 {R1, R0}

FD: fd3: **Vendor → Product**

Violation: 2NF in LORS1

*Resolution:* Decomposition URS1 {R1, R2, R0}  
where:

R1: CUSTOMER ( <u>Customer, Address</u> );	in BCNF
R2: VENDOR ( <u>Vendor, Product</u> )	in BCNF
R0: LORS2 ( <u>Manager, Customer, Vendor, Store, Branch, Location, Sq_ft, Price, Type</u> )	
# LORS2.{Customer} ⊆ CUSTOMER.{Customer}	lossless-join
# LORS2.{Vendor} ⊆ VENDOR.{Vendor}	lossless-join

The input to the next execution of this step is URS1 {R1, R2, R0}.

**Execution 3:**

Input: URS1 {R1, R2, R0}

FD: fd5: **Product → Price**

Violation: 3NF in ?

Observe that the attributes in fd5 have been fragmented in previous decompositions. In order to evaluate the effect of fd5 properly, fd5 will have to be restored. This is accomplished by moving the dependent in fd5 (**Price**) to the relation schema R2, where the determinant of fd5 (**Product**) now resides. Here is the revised URS1 {R1, R2, R0}:

R1: CUSTOMER ( <u>Customer, Address</u> );	in BCNF
R2: VENDOR ( <u>Vendor, Product, Price</u> )	Violates 3NF in R2
R0: LORS2 ( <u>Manager, Customer, Vendor, Store, Branch, Location, Sq_ft, Type</u> )	
# LORS2.{Customer} ⊆ CUSTOMER.{Customer}	Lossless-join
# LORS2.{Vendor} ⊆ VENDOR.{Vendor}	Lossless-join

Accordingly, the violation of 3NF by fd5 in URS1 has moved from LORS2 to R2 and is now resolved as follows:

*Resolution:* Decomposition URS1 {R1, R2, R3, R0}  
where:

R1: CUSTOMER ( <u>Customer, Address</u> );	in BCNF
R2: VENDOR ( <u>Vendor, Product</u> );	in BCNF
R3: PRODUCT ( <u>Product, Price</u> );	in BCNF
R0: LORS3 ( <u>Manager, Customer, Vendor, Store, Branch, Location, Sq_ft, Type</u> )	
# LORS3.{Customer} ⊆ CUSTOMER.{Customer}	Lossless-join
# LORS3.{Vendor} ⊆ VENDOR.{Vendor}	Lossless-join
# VENDOR.{Product} ⊆ PRODUCT.{Product}	Lossless-join

The input to the next execution of this step is URS1 {R1, R2, R3, R0}.

**Execution 4:**

Input: URS1 {R1, R2, R3, R0}	FD: fd7: <b>Manager</b> → { <b>Store</b> , <b>Branch</b> }	Violation: 2NF in LORS3
------------------------------	--	-------------------------

*Resolution:* Decomposition URS1 {R1, R2, R3, R4, R0}

where:

R1: CUSTOMER ( <u>Customer</u> , Address);	in BCNF
R2: VENDOR ( <u>Vendor</u> , Product);	in BCNF
R3: PRODUCT ( <u>Product</u> , Price);	in BCNF
R4: MANAGER ( <u>Manager</u> , Store, Branch)	in BCNF
R0: LORS4 ( <u>Manager</u> , <u>Customer</u> , <u>Vendor</u> , <u>Location</u> , Sq_ft, Type)	
# LORS4.{Customer} ⊆ CUSTOMER.{Customer}	Lossless-join
# LORS4.{Vendor} ⊆ VENDOR.{Vendor}	Lossless-join
# VENDOR.{Product} ⊆ PRODUCT.{Product}	Lossless-join
# LORS4.{Manager} ⊆ MANAGER.{Manager}	

429

Note that the resolution of normal form violation due to fd7 also resolves the BCNF violation in LORS3 due to fd6: {**Store**, **Branch**} → **Manager**.

The input to the next execution of this step is URS1 {R1, R2, R3, R4, R0}.

**Execution 5:**

Input: URS1 {R1, R2, R3, R4, R0}	FD: fd8: <b>Store</b> → <b>Type</b>	Violation: 3NF in ?
----------------------------------	-------------------------------------	---------------------

Once again, the attributes in fd8 have been fragmented in previous decompositions. In order to evaluate the effect of fd8 properly, fd8 will have to be restored. This is accomplished by moving the dependent in fd8 (**Type**) to the relation schema R4, where the determinant of fd8 (**Store**) now resides. The affected relation schemas in URS1 {R1, R2, R3, R4, R0} are R4 and R0, as shown here:

R4: MANAGER ( <u>Manager</u> , <u>Store</u> , <u>Branch</u> , <u>Type</u> )	Violates 3NF in R4
R0: LORS4 ( <u>Manager</u> , <u>Customer</u> , <u>Vendor</u> , <u>Location</u> , Sq_ft)	

{**Store**, **Branch**} is a candidate key of R4 based on fd6. Thus, it is possible to designate {**Store**, **Branch**} as the primary key of R4, in which case **Manager** becomes an alternate key of R4. Then, the same fd6 is seen to violate 2NF in R4. Either way, the resolution of the normal form violation due to fd6 yields the same decomposition given next.

*Resolution:* Decomposition URS1 {R1, R2, R3, R4, R5, R0}

where:

R1: CUSTOMER ( <u>Customer</u> , Address);	in BCNF
R2: VENDOR ( <u>Vendor</u> , Product);	in BCNF

R3:	PRODUCT ( <u>Product</u> , Price);	in BCNF
R4:	MANAGER ( <u>Manager</u> , Store, Branch)	in BCNF
R5:	STORE ( <u>Store</u> , Type)	in BCNF
R0:	LORS5 ( <u>Manager</u> , Customer, Vendor, Location, Sq_ft)	
#	LORS5.{Customer} $\subseteq$ CUSTOMER.{Customer}	Lossless-join
#	LORS5.{Vendor} $\subseteq$ VENDOR.{Vendor}	Lossless-join
#	VENDOR.{Product} $\subseteq$ PRODUCT.{Product}	Lossless-join
#	LORS5.{Manager} $\subseteq$ MANAGER.{Manager}	Lossless-join
#	MANAGER.{Store} $\subseteq$ STORE.{Store}	Lossless-join

Since URS1 {R1, R2, R3, R4, R5, R0} is still not fully normalized—for instance, LORS5 is not in BCNF—we continue with the normalization process. Accordingly, the input to the next execution of this step is URS1 {R1, R2, R3, R4, R5, R0}.

#### Execution 6:

Input: URS1 {R1, R2, R3, R4, R5, R0}	FD: fdx = {fd1 U fd4 U fd6}	Violation: 3NF in ?
	fdx: {Store, Branch} $\rightarrow$ {Location, Sq_ft, Manager};	

The attributes in fdx have been fragmented in previous decompositions. In order to evaluate the effect of fdx properly, fdx will have to be restored. This is accomplished by moving the dependent in fdx **{Location, Sq\_ft, Manager}** to the relation schema R4, where the determinant of fdx **(Store, Branch)** now resides. R4 and R0 are the affected relation schemas in URS1 {R1, R2, R3, R4, R5, R0}:

R4:	MANAGER ( <u>Manager</u> , Store, Branch, Location, Sq_ft)
R0:	LORS5 ( <u>Manager</u> , Customer, Vendor)

Since **{Store, Branch}** is a candidate key of R4, fdx does not violate any normal form in R4. Hence, R4 is in BCNF. As can be seen, LORS5 is also in BCNF.

**Step 5: Resolve all BCNF violations in URS1; the sequence of resolution is immaterial—There are no other BCNF violations in URS1 {R1, R2, R3, R4, R5, R0}.**

Thus, the final design URS1 {R1, R2, R3, R4, R5, R0} that is free from modification anomalies is as follows:

R1:	CUSTOMER ( <u>Customer</u> , Address);	In BCNF
R2:	VENDOR ( <u>Vendor</u> , Product);	In BCNF
R3:	PRODUCT ( <u>Product</u> , Price);	In BCNF
R4:	MANAGER ( <u>Manager</u> , Store, Branch, Location, Sq_ft)	In BCNF
R5:	STORE ( <u>Store</u> , Type)	In BCNF

R0: LORS5 ( <u>Manager, Customer, Vendor</u> )		
#	LORS5.{Customer} $\subseteq$ CUSTOMER.{Customer}	Lossless-join
#	LORS5.{Vendor} $\subseteq$ VENDOR.{Vendor}	Lossless-join
#	VENDOR.{Product} $\subseteq$ PRODUCT.{Product}	Lossless-join
#	LORS5.{Manager} $\subseteq$ MANAGER.{Manager}	Lossless-join
#	MANAGER.{Store} $\subseteq$ STORE.{Store}	Lossless-join

*Is the above design correct and complete?* The above design is attribute-preserving, and each relation schema in the relational schema URS1 {R1, R26, R3, R4, R5, R0} is in BCNF. Does the join of D {R1, R26, R3, R4, R5} strictly yield R? Yes, it does. Therefore, the solution is correct. Since the design also yields a lossless-join decomposition that is also dependency-preserving, the solution is also complete.

It is crucial to observe that LORS5 is part of the final relational schema. Without LORS5 as a part of the final decomposed URS1, the solution is incomplete and therefore incorrect. The initial approach of constructing the relational schema by simply mapping the FDs in F to relation schemas explored at the start of Section 8.3.1 fails to generate LORS5 as a relation schema in the design and therefore is flawed.<sup>9</sup> The solution generated above via normalization demonstrates that the normalization process is indispensable for analyzing a universal relation schema and the set of FDs holding over it, thereby generating a correct and complete relational schema.

### 8.3.2 Case 2

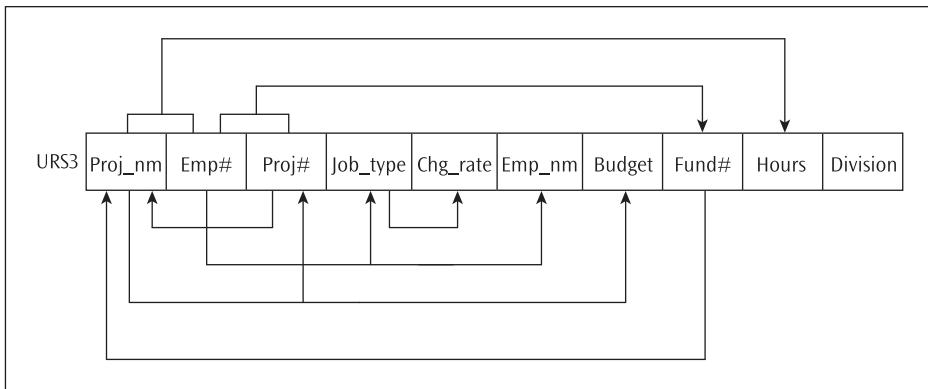
Let us next review the example presented in Section 7.3.3, this time as a second exercise in normalization. The URS and the set of FDs that prevail over it are reproduced here:

**URS3 (Proj\_nm, Emp#, Proj#, Job\_type, Chg\_rate, Emp\_nm, Budget, Fund#, Hours, Division)**

fd1: Proj# $\rightarrow$ Proj_nm;	fd2: Job_type $\rightarrow$ Chg_rate;	fd3: Emp# $\rightarrow$ Emp_nm;
fd4: Proj_nm $\rightarrow$ Budget;	fd5: Fund# $\rightarrow$ Proj_nm;	fd6: {Proj_nm, Emp#} $\rightarrow$ Hours;
fd7: Proj_nm $\rightarrow$ Proj#;	fd8: Emp# $\rightarrow$ Job_type;	fd9: {Proj#, Emp#} $\rightarrow$ Fund#

The set of FDs in F may also be expressed via a dependency diagram, as shown in Figure 8.8c.

<sup>9</sup>This is further clarified when the relational schema is reverse engineered to an ERD (see Section 8.4.1). A simpler decomposition algorithm is available if a 3NF solution is acceptable; see Elmasri and Navathe (2010, p. 342) for details.



**FIGURE 8.8c** Dependency diagram for the relation schema URS3

Convinced from the previous example that arbitrary development of relation schemas from a set of FDs yields an incomplete and incorrect solution, let us reject that approach and proceed with the evaluation of URS3 from the normalization perspective.<sup>10</sup> URS3 is in 1NF by virtue of its definition as a relation schema—that is, there are no multi-valued attributes or composite attributes in URS3, and all FDs in F are preserved in URS3. Lossless-join property is not an issue since URS3 is a single relation schema. If URS3 is in BCNF, it is guaranteed that there will be no data redundancies/modification anomalies in URS3 due to functional dependencies. In order to check this, we need to know at least one candidate key of URS3, since normal form violations can be checked only with respect to candidate keys (or a primary key, it being one of the candidate keys of the relation schema). So, we start with Step 1 of the normalization heuristic.

**Step 1: Identify the candidate keys of URS3 given the set of FDs, F**—In Chapter 7, we derived the candidate keys of URS3 (see Section 7.3.3.) as:

{Proj#, Emp#, Division}, {Proj\_nm, Emp#, Division}, and {Fund#, Emp#, Division}

**Step 2: Choose a primary key for URS3**—Since the choice of primary key from among the candidate keys is essentially arbitrary, using the rules of thumb prescribed in Chapter 7, let us choose {Proj#, Emp#, Division} as the primary key of URS3. Then, the other two candidate keys, {Proj\_nm, Emp#, Division} and {Fund#, Emp#, Division}, become the alternate keys of URS3.

**Step 3: Record the immediate normal form violated in URS3 with respect to the primary key by each of the FDs in F**—Based on the *primary key* chosen, viz., {Proj#, Emp#, Division}, normal forms are violated in URS3 by:

fd1	fd2	fd3	fd4	fd5	fd6	fd7	fd8	Fd9
2NF	3NF	2NF	3NF	3NF	3NF	BCNF	2NF	2NF

<sup>10</sup>The unconvinced reader may, as an exercise, construct the relational schema by mapping each FD to a relation schema, do some consolidation based on Armstrong's axioms, and compare the results with the normalized solution.

**Step 4: Resolve 2NF and 3NF violations in URS3; the sequence of resolution is immaterial**—We now know that this step is executed recursively as necessary and that in successive executions of this step, the URS3 is progressively revised to a set of decomposed relation schemas. Since we have already observed the successive, iterative execution of this step for each of the 2NF and 3NF violations in the first example, we handle them collectively in one execution of this step in this example. Solving for the 2NF and 3NF violations in URS3, we have:

Decomposition: URS3 {R1, R2, R3, R4, R5, R0}  
where:

R1: PROJECT ( <u>Proj#</u> , Proj_nm, Budget);	in BCNF
R2: FUND ( <u>Fund#</u> , Proj_nm)	in BCNF
R3: EMPLOYEE ( <u>Emp#</u> , Emp_nm, Job_type)	in BCNF
R4: JOB ( <u>Job_type</u> , Chg_rate)	in BCNF
R5: ASSIGNMENT ( <u>Proj#</u> , Emp#, Fund#, Hours)	in 3NF (violates BCNF)*
R0: LORS1 ( <u>Proj#</u> , Emp#, Division)	in BCNF
# LORS1.{Proj#, Emp#} ⊆ ASSIGNMENT.{Proj#, Emp#}	Lossless-join
# ASSIGNMENT.{Emp#} ⊆ EMPLOYEE.{Emp#}	Lossless-join
# ASSIGNMENT.{Proj#} ⊆ PROJECT.{Proj#}	Lossless-join
# ASSIGNMENT.{Fund#} ⊆ FUND.{Fund#}	Lossless-join
# EMPLOYEE.{Job_type} ⊆ JOB.{Job_type}	Lossless-join
# FUND.{Proj_nm} ⊆ PROJECT.{Proj_nm}	Lossless-join**

\*Violation of BCNF due to the FD in  $F^+$  Fund# → Proj#  
\*\*Note that Proj\_nm is an alternate key of R1.

While at this stage of the solution, the revised relational schema URS3 is not in BCNF, it possesses the lossless-join property, and all FDs in F are preserved. Note that, incidentally, the decomposition of URS3 for the resolution of 2NF and 3NF violations eliminated the BCNF violation due to fd7 by the migration of Proj\_nm to R1. However, the decomposition has seeded a new BCNF violation in R5.

**Step 5: Resolve all BCNF violations in URS3; the sequence of resolution is immaterial**—The only BCNF violation in the URS3 {R1, R2, R3, R4, R5, R0} is the one in:

R5: ASSIGNMENT (Proj#, Emp#, Fund#, Hours)

The resolution of the BCNF violation using the prescribed two-step process results in:

R5a: FUND_A ( <u>Fund#</u> , Proj#)	in BCNF
R5b: ASSIGNMENT ( <u>Fund#</u> , Emp#, Hours)	in BCNF

R5a can be consolidated with R2, since the determinant of the FDs preserved in both is the same—viz., **Fund#**. Thus, we have:

R2a:	FUND ( <u>Fund#</u> , Proj#, Proj_nm)	violates 3NF
R5b:	ASSIGNMENT ( <u>Fund#</u> , Emp#, Hours)	in BCNF

Since **Proj#** and **Proj\_nm** functionally determine each other, as per fd1 and fd7, one of the two attributes moves to R1 in the process of decomposing R2a, to eliminate the 3NF violation. Thus, we have:

R2:	FUND ( <u>Fund#</u> , Proj_nm)	in BCNF
R5b:	ASSIGNMENT ( <u>Fund#</u> , Emp#, Hours)	in BCNF

So, a solution in BCNF can be written as:

Decomposition: URS3 {R1, R2, R3, R4, R5b, R0}

where:

R1:	PROJECT ( <u>Proj#</u> , Proj_nm, Budget);	in BCNF
R2:	FUND ( <u>Fund#</u> , Proj_nm)	in BCNF
R3:	EMPLOYEE ( <u>Emp#</u> , Emp_nm, Job_type)	in BCNF
R4:	JOB ( <u>Job_type</u> , Chg_rate)	in BCNF
R5b:	ASSIGNMENT ( <u>Fund#</u> , Emp#, Hours)	in BCNF
R0:	LORS1 ( <u>Proj#</u> , Emp#, Division)	in BCNF
#	LORS1.{Proj#} $\subseteq$ PROJECT.{Proj#}	Lossless-join*
#	LORS1.{Emp#} $\subseteq$ EMPLOYEE.{Emp#}	Lossless-join*
#	ASSIGNMENT.{Emp#} $\subseteq$ EMPLOYEE.{Emp#}	Lossless-join
#	ASSIGNMENT.{Fund#} $\subseteq$ FUND.{Fund#}	Lossless-join
#	EMPLOYEE.{Job_type} $\subseteq$ JOB.{Job_type}	Lossless-join
#	FUND.{Proj_nm} $\subseteq$ PROJECT.{Proj_nm}	Lossless-join

\*Indicates revision to relationships via changes in inclusion dependencies in order to indicate a lossless-join navigation path.

The relational schema, while eradicating modification anomalies due to functional dependencies, is not dependency-preserving because it is impossible to deduce

fd6:  $\{Proj\_nm, Emp\# \} \rightarrow Hours$  and fd9:  $\{Proj\#, Emp\# \} \rightarrow Fund\#$

from the union of the FDs that hold in the *individual* relation schemas of URS3. In other words, from the union of the *individual* relation schemas of URS3 in the solution above,  $F^+$  cannot be generated. The earlier lossless-join relationship between R5 and R0 is no

longer present; instead, the decomposition has created a loss-join relationship between R5b and R0. A more robust design is to propagate the revision of R5b to R0 and thus retain the originally established lossless-join relationship between R0 and R5.

**Step 6: Propagate the revisions to the primary keys in the BCNF resolutions to the primary keys of related relation schemas**—In this example, the propagation involves just one relation schema (R0):

R5b:	ASSIGNMENT ( <u>Fund#</u> , <u>Emp#</u> , Hours)	in BCNF
R0:	LORS2 ( <u>Fund#</u> , <u>Emp#</u> , Division)	in BCNF

This design is superior because the lossless-join property is fully restored. Thus, the final form of the BCNF design that retains the lossless-join property is:

Decomposition: URS3 {R1, R2, R3, R4, R5b, R0}  
where:

R1:	PROJECT ( <u>Proj#</u> , Proj_nm, Budget);	in BCNF
R2:	FUND ( <u>Fund#</u> , Proj_nm)	in BCNF
R3:	EMPLOYEE ( <u>Emp#</u> , Emp_nm, Job_type)	in BCNF
R4:	JOB ( <u>Job_type</u> , Chg_rate)	in BCNF
R5b:	ASSIGNMENT ( <u>Fund#</u> , <u>Emp#</u> , Hours)	in BCNF
R0:	LORS2 ( <u>Fund#</u> , <u>Emp#</u> , Division)	in BCNF
#	LORS2.{Fund#, Emp#} $\subseteq$ ASSIGNMENT.{Fund#, Emp#}	Lossless-join
#	ASSIGNMENT.{Emp#} $\subseteq$ EMPLOYEE.{Emp#}	Lossless-join
#	ASSIGNMENT.{Fund#} $\subseteq$ FUND.{Fund#}	Lossless-join
#	EMPLOYEE.{Job_type} $\subseteq$ JOB.{Job_type}	Lossless-join
#	FUND.{Proj_nm} $\subseteq$ PROJECT.{Proj_nm}	Lossless-join

The final solution is attribute-preserving and the join of URS3 {R1, R2, R3, R4, R5b, R0} does strictly yield R. All the relation schemas in URS3 {R1, R2, R3, R4, R5b, R0} are in BCNF. Therefore, the solution is correct. As for completeness, the solution is a lossless-join design, but is not dependency-preserving. The only way to compensate for the dependencies that are not preserved in this solution is to supplement the relational schema with the necessary materialized views for covering the lost FDs. In this case, the two lost FDs happen to have the same determinant and so can be captured in a single materialized view of the form:

#### MV1: SCHEDULE (Proj#, Emp#, Fund#, Hours)

Finally, while the choice of primary key at the beginning of the normalization process (Step 2) may have some impact on the execution of the normalization steps prescribed, the final solution will be the same or a close equivalent no matter which candidate key is chosen as the primary key. Verification of this is left as an exercise for the reader.

### 8.3.3 A Fast-Track Algorithm for a Non-Loss, Dependency-Preserving Solution

Now that we know how to generate a normalized decomposition of a URS containing several normal form violations and how to manage the side effects of dependency preservation and loss/non-loss join conditions, we are ready to consider a fast-track algorithm to achieve a non-loss 3NF solution that is guaranteed to also be dependency-preserving. But then, it is important to note that this algorithm will yield only a 3NF solution. The resulting relational schema can contain immediate BCNF violations. Thus, the only merit of this algorithm is the quick route to a 3NF design; the modeler/designer may then use this solution as the input to further normalization to achieve a BCNF design. The algorithm is presented here:

436

**Input:** URS | F

- Derive the canonical cover  $G_c$  of F.
- Derive the candidate keys of  $URS \upharpoonright G_c$ .
- For each FD in  $G_c$ , create a relation schema in the decomposition D.
- If none of the relation schemas in D contains a candidate key of URS, create one more relation schema in D that contains attributes that form a candidate key of URS.
- Eliminate redundant relation schemas from the resulting relational schema (database schema).
- Consolidate D to a parsimonious set of relation schemas by combining relation schemas in D that share the same primary key.

#### 8.3.3.1 Case 2 Revisited

Let us use the previous example (URS3) to demonstrate the utility of this algorithm. Here are the URS and the set of FDs that prevail over it:

**URS3 (Proj\_nm, Emp#, Proj#, Job\_type, Chg\_rate, Emp\_nm, Budget, Fund#, Hours, Division)**

and:

**F {fd1, fd2, fd3, fd4, fd5, fd6, fd7, fd8, fd9}**

where:

fd1: Proj# → Proj_nm;	fd2: Job_type → Chg_rate;	fd3: Emp# → Emp_nm;
fd4: Proj_nm → Budget;	fd5: Fund# → Proj_nm;	fd6: {Proj_nm, Emp#} → Hours;
fd7: Proj_nm → Proj#;	fd8: Emp# → Job_type;	fd9: {Proj#, Emp#} → Fund#

The first step in the fast-track algorithm asks for developing the canonical cover ( $G_c$ ) of F. Here, it just happens that  $G_c = F$ . Note that since fd1 and fd7 indicate mutual referencing between the attributes **Proj#** and **Proj\_nm**, the dependent in fd5 can also be **Proj#**, essentially providing an alternate canonical cover for F. This property will be invoked later, in the development of the normalized solution.

Next, we need to derive the candidate keys of URS3. Since this was already done once (see Section 7.3.3), we have the candidate keys of URS, which are the following:

**{Proj#, Emp#, Division}, {Proj\_nm, Emp#, Division}, and {Fund#, Emp#, Division}**

The next two steps in the fast-track algorithm dictate that we specify a relation schema for each FD in  $G_c$  and that we specify another relation schema containing the attributes of a candidate key of the URS if this set of attributes is not present in one of the specified relation schemas. Accordingly, we have R1 through R9 to meet the first rule and R0 to fulfill the second rule:

- R1: PROJECT (Proj#, Proj\_nm);
- R2: JOB (Job\_type, Chg\_rate);
- R3: EMPLOYEE (Emp#, Emp\_nm);
- R4: BUDGET (Proj\_nm, Budget);
- R5: FUND (Fund#, Proj\_nm);
- R6: HOURS (Proj\_nm, Emp#, Hours);
- R7: PROJ (Proj\_nm, Proj#);
- R8: EMP\_JOB (Emp#, Job\_type);
- R9: PRJ\_EMP (Proj#, Emp#, Fund#);
- R0: PRJ\_EMP\_DIV (Proj#, Emp#, Division)

Since R1 is a subset of R7, R1 is redundant and can be eliminated. Next, the consolidations based on common primary key will lead to a relational schema as shown here:

- R2: JOB (Job\_type, Chg\_rate);
- R38: EMPLOYEE (Emp#, Emp\_nm, Job\_type);
- R47: PROJECT (Proj\_nm, Proj#, Budget);
- R5: FUND (Fund#, Proj\_nm);
- R6: HOURS (Proj\_nm, Emp#, Hours);
- R9: PRJ\_EMP (Proj#, Emp#, Fund#);
- R0: PRJ\_EMP\_DIV (Proj#, Emp#, Division)

Observe that, in this solution, all relation schemas except R9 are in BCNF. R9 violates BCNF since  $\text{Fund\#} \rightarrow \text{Proj\#}$  is in  $F^+$ . The design is dependency-preserving. On first glance, it may appear that there are several loss-join conditions in the design (e.g., between R6 and R9, between R6 and R0, and between R5 and R6). While this algorithm does not explicitly handle this condition, the fact that attributes **Proj#** and **Proj\_nm** are mutually substitutable based on the FDs fd1 and fd7, slightly revising  $F_c$  as the input will indeed resolve this issue. For now, based on this property of substitutability due to the mutual

referencing between the attributes **Proj#** and **Proj\_nm**, it is seen that R6 and R9 can be consolidated as follows:

**R69: PRJ\_EMP (Proj#, Emp#, Fund#, Hours)**

Thus, the final design generated by the fast-track algorithm leads to a non-loss 3NF solution that is dependency-preserving, as shown here:

R2:	JOB ( <u>Job_type</u> , Chg_rate);
R38:	EMPLOYEE ( <u>Emp#, Emp_nm, Job_type</u> );
R47:	PROJECT ( <u>Proj_nm, Proj#, Budget</u> );
R5:	FUND ( <u>Fund#, Proj_nm</u> );
R69:	<b>PRJ_EMP (<u>Proj#, Emp#, Fund#, Hours</u>)</b>
R0:	<b>PRJ_EMP_DIV (<u>Proj#, Emp#, Division</u>)</b>
#	PRJ_EMP_DIV.{Proj#, Emp#} ⊆ PRJ_EMP.{Proj#, Emp#}
#	PRJ_EMP.{Emp#} ⊆ EMPLOYEE.{Emp#}
#	PRJ_EMP.{Proj#} ⊆ PROJECT.{Proj#}
#	PRJ_EMP.{Fund#} ⊆ FUND.{Fund#}
#	EMPLOYEE.{Job_type} ⊆ JOB.{Job_type}
#	FUND.{Proj_nm} ⊆ PROJECT.{Proj_nm}

Observe that the solution here is identical to the one arrived at in Step 4 of Case 2 in Section 8.3.2. Having arrived at this stage of solution rather quickly, the only task that remains is to resolve the BCNF violation in R69 caused by the FD in  $F^+ \text{Fund\#} \rightarrow \text{Proj\#}$ . While this resolution is exactly the same as in Step 5 of Case 2 in Section 8.3.2, a slight variation in the approach is shown here:

R69b:	<b>FUND_PRJ (<u>Fund#, Proj#</u>)</b>
R69a:	<b>PRJ_EMP (<u>Emp#, Fund#, Hours</u>)</b>

Given the substitutable nature of the attributes **Proj#** and **Proj\_nm**, R69b can be seen as a proper subset of R47 and hence redundant.

It is crucial to note that this decomposition to resolve the BCNF violation in R69 changes the primary key of R69a. Three consequences result from this process:

- The final relational schema in BCNF guarantees total eradication of modification anomalies due to functional dependencies.
- The solution is not dependency-preserving. Two FDs preserved in the 3NF solution are no longer preserved in this BCNF solution. They are:  
fd6: {Proj\_nm, Emp#} → Hours; fd9: {Proj#, Emp#} → Fund#
- The lossless-join that prevailed in the 3NF solution is now violated in the BCNF solution. That is, a non-loss join between R0 and R69 present in the 3NF solution is now violated in R0 and R69a.

Any time a decomposition of a relation schema occurs to resolve an immediate BCNF violation, the primary key of one of the binary projections (a relation schema) will change. This can disrupt a prevailing non-loss condition in a relational schema and requires a remedial action. As stipulated in Step 6 of Case 2 in Section 8.3.2, the revision to the primary key of the relation schema in the BCNF decomposition should be propagated to all the related relation schemas in the relational schema. Accordingly, the current R0 gets revised as follows:

**R0: PRJ\_EMP\_DIV (Fund#, Emp#, Division)**

Thus, the final non-loss BCNF solution is identical to that in Case 2, presented in Section 8.3.2:

R2:	JOB ( <u>Job_type</u> , Chg_rate);	
R38:	EMPLOYEE (Emp#, Emp_nm, Job_type);	
R47:	PROJECT (Proj_nm, Proj#, Budget);	
R5:	FUND ( <u>Fund#</u> , Proj_nm);	
R69:	FUND_EMP ( <u>Fund#</u> , Emp#, Hours);	
R0:	<b>FUND_EMP_DIV (<u>Fund#</u>, Emp#, Division)</b>	
#	FUND_EMP_DIV.{Fund#} ⊆ FUND_EMP.{Fund#}	Lossless-join
#	FUND_EMP.{Emp#} ⊆ EMPLOYEE.{Emp#}	Lossless-join
#	FUND_EMP.{Fund#} ⊆ FUND.{Fund#}	Lossless-join
#	EMPLOYEE.{Job_type} ⊆ JOB.{Job_type}	Lossless-join
#	FUND.{Proj_nm} ⊆ PROJECT.{Proj_nm}	Lossless-join

Observe that the price paid to achieve a non-loss BCNF solution is the inability to preserve the following two FDs:

fd6: {Proj\_nm, Emp#} → Hours; and fd9: {Proj#, Emp#} → Fund#

A compensatory mechanism for this shortcoming using “materialized views” is presented in Section 8.3.2.

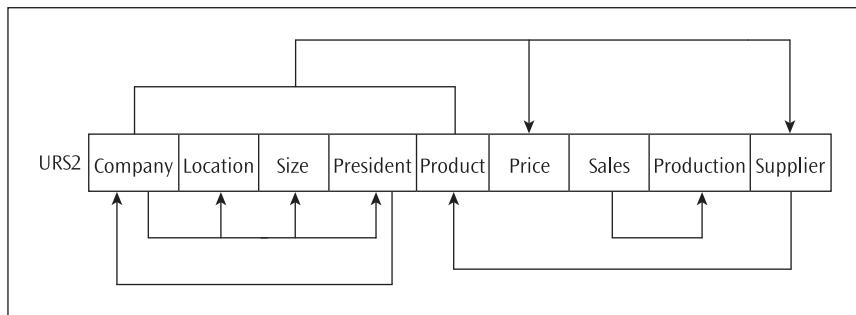
### 8.3.3.2 Case 3

The next example, first presented in Section 7.3.2.1, offers a different variation for a normalization exercise. The URS and the set of FDs that prevail over URS are reproduced here:

**URS2 (Company, Location, Size, President, Product, Price, Sales, Production, Supplier)**

fd1: Company → Location;	fd2: Company → Size;	fd3: Company → President;
fd4: {Product, Company} → Price;	fd5: Sales → Production;	fd6: {Company, Product} → Sales;
fd7: {Product, Company} → Supplier;	fd8: Supplier → Product;	fd9: President → Company

The set of FDs in F may also be expressed via a dependency diagram, as shown in Figure 8.8d.



440

**FIGURE 8.8d** Dependency diagram for the relation schema URS2

Let us step through the fast-track algorithm to quickly arrive at the non-loss, dependency-preserving stage of the normalization process.

**Step 1:** Derive the canonical cover  $G_c$  of F.

Here,  $G_c = F$ .

In addition, since fd3 and fd9 indicate mutual referencing between the attributes **Company** and **President**, it is obvious that another canonical cover can be derived by substituting the attribute **President** for **Company** in the relevant FDs in F ( $G_c$ ).

**Step 2:** Derive the candidate keys of  $URS \setminus G_c$ .

In Chapter 7, we already derived the candidate keys of URS2 (see Section 7.3.2) as follows:

{**Company, Product**}; {**Company, Supplier**}; {**President, Product**}; and {**President, Supplier**}.

**Step 3:** For each FD in  $G_c$ , create a relation schema in the decomposition D.

Since there are nine FDs in  $G_c$ , we have D: {R1, R2, R3, R4, R5, R6, R7, R8, R9} where:

R1: CO_LOC ( <u>Company</u> , Location);	R4: CO_PRD_PRC ( <u>Company</u> , Product, Price);
R2: CO_SZ ( <u>Company</u> , Size);	R6: CO_PRD_SAL ( <u>Company</u> , Product, Sales);
R3: CO_PR ( <u>Company</u> , President);	R7: PRODUCT ( <u>Company</u> , Product, Supplier);
R8: SUPPLIER ( <u>Supplier</u> , Product);	R5: SALES ( <u>Sales</u> , Production);
R9: PR_CO ( <u>President</u> , Company);	

**Step 4:** If none of the relation schemas in D contains a candidate key of URS, create one more relation schema in D that contains attributes that form a candidate key of URS.

R4, R6, and R7 in D shown in Step 3 each contain a candidate key of URS2; hence, there is no need to create an additional relation schema.

**Step 5:** Eliminate redundant relation schemas from the resulting relational schema (database schema).

In the presence of R3, it can be seen that R9 is redundant and should be eliminated. In addition, R8 is a proper subset of R7 and hence, by definition, redundant; it should therefore be eliminated.

**Step 6:** Consolidate D to a parsimonious set of relation schemas by combining relation schemas in D that share the same primary key.

The consolidation leads to D: {R123, R467, R5} where:

R123: COMPANY ( <u>Company</u> , Location, Size, President);	R5:SALES ( <u>Sales</u> , Production);
R467: CO_PRD ( <u>Company</u> , Product, Price, Sales, Supplier);	
# CO_PRD.{Company} ⊆ COMPANY.{Company}	Lossless-join
# CO_PRD.{Sales} ⊆ SALES.{Sales}	Lossless-join

The final design generated by the fast-track algorithm shown above is a non-loss 3NF solution that is also dependency-preserving. The only additional step required at this point is to examine each relation schema in the design for possible violation of BCNF. A critical point to note here is that one must evaluate every FD preserved in each relation schema since it is possible that some of the FDs are present in  $F^+$ . In this example, fd8: **Supplier → Product** violates BCNF in R467; therefore, modification anomalies will persist in this solution.

Decomposition of R467 to resolve the BCNF violation using the basic two-step algorithm prescribed in Sections 8.1.2, 8.1.3, and 8.1.4 yields the following:

**R8: SUPPLIER (Supplier, Product); R46: CO\_SUP (Company, Price, Sales, Supplier)**

Observe that the resolution of immediate BCNF violation has resulted in a change in the primary key of R46. However, in this example, there is no ripple effect (disruption) on the non-loss status in the rest of the relation schemas in the solution, obviating any need to propagate changes. On the other hand, the BCNF solution is not dependency-preserving anymore. The FDs in  $G_c$  that are not preserved in the BCNF solution are as follows:

fd4: {Product, Company} → Price;      fd6: {Company, Product} → Sales;  
fd7: {Product, Company} → Supplier.

The final non-loss BCNF solution is presented here:

D: {R123, R46, R5, R8}

where:

R123: COMPANY ( <u>Company</u> , Location, Size, President);	R5:SALES ( <u>Sales</u> , Production);
R46: CO_SUP ( <u>Company</u> , Price, Sales, Supplier);	R8: SUPPLIER ( <u>Supplier</u> , Product)
# CO_SUP.{Company} ⊆ COMPANY.{Company}	Lossless-join
# CO_SUP.{Supplier} ⊆ SUPPLIER.{Supplier}	Lossless-join
# CO_SUP.{Sales} ⊆ SALES.{Sales}	Lossless-join

Since **Company** → **President** (fd3) and **President** → **Company** (fd9), either **Company** or **President** can be the primary key of R123 and the other becomes the alternate key of R123. Also, using the rule of pseudotransitivity, since **President** → **Company**, **President** can replace **Company** in R46 to generate an equivalent relation schema, as follows:

R0: CO_SUP ( <u>President</u> , Sales, Price, <u>Supplier</u> )	in BCNF
# CO_SUP.{President} ⊆ COMPANY. {President}	Lossless-join

## 8.4 DENORMALIZATION

442

The jovial remark, “Normalize until it hurts, and denormalize until it works,” while certainly funny, also indicates that the concept of denormalization is ill-understood. The general case against normalization is that the process results in lots of logically separate relations (tables), leading to lots of physically separate stored files and the consequent data retrieval inefficiencies.

**Denormalization** entails combining relations so that they are easier to query. The combined relations may lose their normalized status in that they may reintroduce data redundancies eliminated by the normalization process. The general misunderstanding, however, is that denormalization always improves data retrieval performance.

Formally, denormalization may be defined as replacing a set of (often normalized) relation schemas D {R1, R2, . . . . . Rn} by their join R, such that projecting R over the set of attributes of R1, R2, . . . . . Rn, respectively, is guaranteed to yield the original set D. The objective is to reduce the number of joins that may be required during the run time of queries (data retrieval) by including some of these joins structurally as a part of the database design.

As an example, consider the following relation schema:

R: CUSTOMER (Id, Name, Street, City, State, Zip\_code).

From the semantics of this general scenario, it is obvious that an FD of the form:

Zip\_code → {City, State}

holds on CUSTOMER, resulting in a violation of 3NF. In a strict normalization paradigm, one would decompose R to eliminate the 3NF violation, leading to the solution D {R1, R2}, where:

R1: ZIP (Zip\_code, City, State); R2: CUSTOMER (Id, Name, Street, Zip\_code)

While the normalization process eliminates data redundancies and the associated modification anomalies in the design, most queries on CUSTOMER may require joining the two relations R1 and R2, while a denormalized R eliminates the repeated join operation and the attendant retrieval inefficiencies. The semantics of the scenario indicates that a **Zip\_code** is rarely deleted, added, or updated. Thus, the expected modification anomalies are not a serious practical problem. In this case, one may opt for the denormalized design instead of the normalized solution. However, execution of any query that exclusively seeks **Zip\_code** data needs to unnecessarily access a relatively larger relation (R) and be accordingly less efficient.

As a second example, consider a scenario where employees of a large corporation have access to a handful of mutual fund companies for their retirement investments. Suppose the following relation schema:

R: EMPLOYEE (Emp#, Name, Age, Salary, M\_fund#, Fund\_nm, Fund\_mgr)

represents this scenario. The semantics of the scenario would suggest the presence of an FD  $M\_fund\# \rightarrow \{Fund\_nm, Fund\_mgr\}$ , causing a violation of 3NF in R. In a strict normalization paradigm, one would decompose R to eliminate the 3NF violation, leading to the solution D {R1, R2}, where:

R1: FUND (M\_fund#, Fund\_nm, Fund\_mgr);

R2: EMPLOYEE (Emp#, Name, Age, Salary, M\_fund#)

Queries on EMPLOYEE requiring retirement financial data leads to joining R1 and R2, while the unnormализed (or denormalized) R is a better option in such situations. However, suppose there are thousands of employees and just a handful of these retirement mutual funds. Any query requiring only mutual fund data will execute rather inefficiently in the denormalized design. In this case, it may be worthwhile to evaluate a normalized design supplemented by a materialized view for the joined relation.

In short, it must be noted that denormalization, contrary to conventional beliefs, need not improve data retrieval performance in all circumstances. Further, denormalization is considered, in general, only during physical database design, where there are sometimes other options available to improve data retrieval efficiency.

Denormalization can be a meaningful strategy in the logical data modeling tier also. This application of denormalization is usually not recognized by academics or practitioners. When a relation schema is in immediate violation of BCNF (that is, the relation schema is in 3NF, but violates BCNF), solving the design for BCNF always fails to preserve some of the functional dependencies in F. Under these circumstances, one is apt to denormalize the design to 3NF in order to preserve all functional dependencies in F. Solution Option 2 discussed in Section 8.1.5.3 is an ideal example for this.

## **8.5 ROLE OF REVERSE ENGINEERING IN DATA MODELING**

---

The term **reverse engineering** originates in hardware development and refers to the idea of working backwards *in a systematic manner* with a view to discovering how a product works. The focus of software engineering over the decades has been in the area of “forward engineering”—that is, developing *clearly understood* new systems, while one of the major concerns in the practitioners’ world has been upgrading *poorly understood* old systems. The critical role of reverse engineering in the software development arena has been acknowledged since the early 1990s (*Communications of the ACM*, May 1994). The goal of reverse engineering in a software development environment is to understand how existing software systems work. Reverse engineering encompasses a wide array of tasks. Central to these tasks is identifying data and process components of existing software systems and the relationships within and across these components.

Motives suggested for database reverse engineering traditionally include migration from past database paradigms of hierarchical and network data architectures to the contemporary relational or object-oriented paradigms. But the more mundane task of migrating between different implementations within the relational paradigm is equally relevant from a reverse-engineering perspective. In addition, reverse engineering sheds light on poorly documented software systems. The starting point of reverse engineering in these cases is an existing operational software system, and the task at hand is to analyze data patterns to distill emerging data structure and behaviors.

This section of the book focuses on reverse-engineering data models to generate a high-level conceptual description of the data architecture. Reverse-engineering studies have revealed that the original schemas are often fundamentally flawed or violate “good” design practices (Premelani and Blaha, 1994). This section presents a unique use of reverse engineering in the data modeling task in order to preempt occurrence of such flaws and failures through a better understanding of the design. To that end, the scope of this section is restricted to reverse-engineering normalized relational schemas to their conceptual counterparts—namely, ERDs.

When a relation schema directly mapped from an entity type of an ERD requires normalization, it simply means that a set of related independent entity types have been erroneously represented as a single entity type in the ERD. Thus, reverse engineering a normalized relational schema to an ERD reveals how the ERD should have been to begin with. Such “discovery” enriches the designers’ understanding of the application domain being modeled. Section 7.1 provided a case in point. If the source relation schema is not an entity type from an ERD, but, instead, a URS constructed from a set of FDs directly arising from the business rules embedded in the user requirements specification, it is all the more valuable to depict the normalized solution as an ERD, so that it can serve as a presentation/communication device among the designers and the users.

Unfortunately, reverse engineering is far from being a fully automated technique. Some automation has been tried (e.g., Chiang, et al., 1994) using heuristic approaches. In the following subsections, we use a heuristic approach based on a few guidelines for reverse engineering a relational schema to an ERD. The reader is encouraged to review Section 7.1, where a reverse-engineered ERD was shown (in Figure 7.4), before continuing with the rest of this section. The reverse-engineering heuristic is stated in Figure 8.9.

- » Step 1 Translate the normalized relational schema to an information-preserving logical schema based on available information.
  - Denote alternate keys as unique [Q]; Enclose composite attributes in braces [ ]
  - Indicate the parent label on top of the foreign key along with the parent side structural constraints of the relationship type—default value for min and max when information not available are 0 and n respectively (see Section 6.7.2 of Chapter 6 for grammar)
  - Indicate child side structural constraints of the relationship type right below the foreign key along with the deletion rule—default value for min when information not available is 0 and default deletion rule is R except when the cardinality ratio is 1:1; then, leave unfilled if rule not specified. (see Section 6.7.2 of Chapter 6 for grammar)
- » Step 2 Transform the information-preserving logical schema to a Design-Specific ER
  - Map each logical scheme to an entity type:
    - ◆ If the primary key of a logical scheme Lx is a proper subset of the primary key of another logical scheme Ly, map Ly as a weak entity type in the ERD with Lx as its identifying parent
    - ◆ If the primary key of a logical scheme Lx is a concatenation of the primary keys of multiple logical schemes Ly, Lz, etc., map Lx as a gerund entity type in the ERD with Ly, Lz, etc. as its identifying parents
    - ◆ All other logical schemes are mapped as base entity types
  - Represent the foreign key attribute(s) in a logical scheme by a relationship type in the ERD\*:
    - ◆ Establish the relationship type
    - ◆ Connect the relationship type to the parent (referenced) and child (referencing) entity types – if the child is a weak entity type then the relationship type is mapped as an identifying relationship type
    - ◆ Map the (min, max) to the appropriate edge of the relationship type
  - Map attributes of individual logical schemes to corresponding entity types in the ERD:
    - ◆ Map atomic and composite attributes
    - ◆ Underline unique identifiers (The primary key and alternate keys of a logical scheme are unique identifiers of the corresponding entity type)
    - ◆ Partial key of a weak entity is denoted by a dotted underline
- » Step 3 Abstract the Design-Specific ER diagram up to a Presentation Layer ERD.
  - Transform gerund entity types to n-way relationship types. Attributes of the gerund entity type remain as attributes of the relationship type
  - Any relationship with a gerund entity type is transformed to a relationship type with the cluster entity type that the gerund represents
  - A weak entity type not participating in any relationship other than the identifying relationship is transformed to a multi-valued (atomic/composite) attribute of the parent entity type

\*The foreign key attribute is removed from the referencing (child) entity type unless the attribute plays some other role in the entity type in which case the attribute is retained in the entity type.

**FIGURE 8.9** Heuristic to reverse engineer a relational schema to an ERD

### 8.5.1 Reverse Engineering the Normalized Solution of Case 1

To begin with, URS1 in 1NF is of the following form:

**URS1 (Store, Branch, Location, Sq\_ft, Manager, Product, Price, Customer, Address, Vendor, Type)**

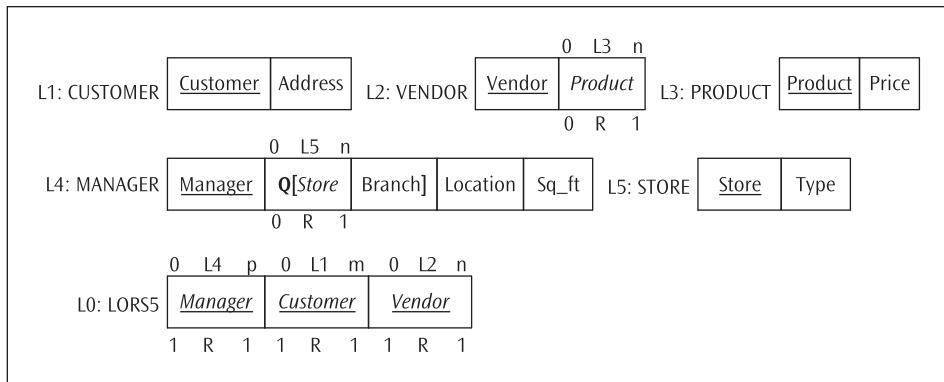
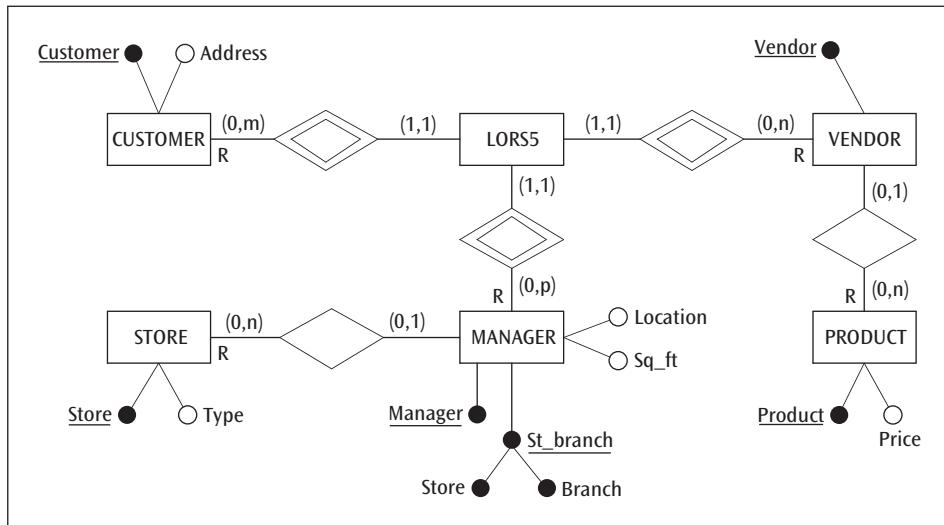
When fully normalized to BCNF, the resulting relational schema looks like this:

URS1 {R1, R2, R3, R4, R5, R0}

where:

R1:	CUSTOMER ( <u><b>Customer</b></u> , Address);	in BCNF
R2:	VENDOR ( <u><b>Vendor</b></u> , Product);	in BCNF
R3:	PRODUCT ( <u><b>Product</b></u> , Price);	in BCNF
R4:	MANAGER ( <u><b>Manager</b></u> , Store, Branch, Location, Sq_ft)	in BCNF
R5:	STORE ( <u><b>Store</b></u> , Type)	in BCNF
R0:	LORS5 ( <u><b>Manager</b></u> , <u><b>Customer</b></u> , <u><b>Vendor</b></u> )	
#	LORS5.{Customer} $\subseteq$ CUSTOMER.{Customer}	Lossless-join
#	LORS5.{Vendor} $\subseteq$ VENDOR.{Vendor}	Lossless-join
#	VENDOR.{Product} $\subseteq$ PRODUCT.{Product}	Lossless-join
#	LORS5.{Manager} $\subseteq$ MANAGER.{Manager}	Lossless-join
#	MANAGER.{Store} $\subseteq$ STORE.{Store}	Lossless-join

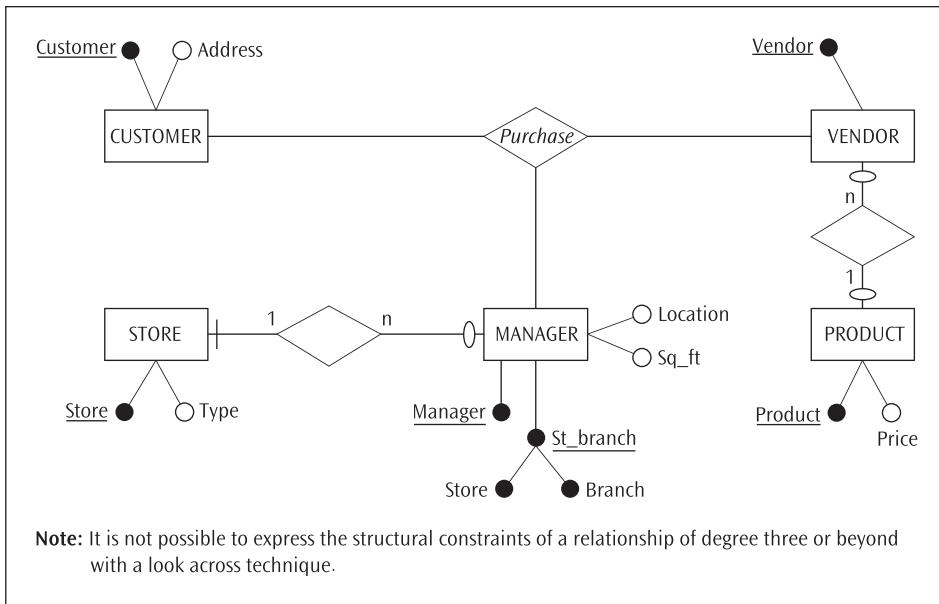
During the discussion of normalization, we moved the information-preservation issue to the background. As we get ready to reverse engineer the normalized URS1 {R1, R2, R3, R4, R5, R0}, the first step prescribed in the reverse-engineering heuristic is to translate the normalized relational schema to an information-preserving logical schema *based on available information and assumed default properties, where information is not immediately available*. The resulting information-preserving logical schema is shown in Figure 8.10a. Next, following Step 2, we construct one entity type for each scheme in the logical schema using base and weak entity types. Since every foreign key in the logical schema represents a relationship, we then systematically replace every foreign key with a relationship type between the referencing entity type (the logical schema carrying the foreign key) and the referenced entity type (the parent entity type in this relationship). At this point, we have a skeletal ERD with no attributes allocated to the entity types. The attributes that are left in the logical schema after the replacement of foreign key attributes are mapped to the corresponding entity types, appropriately indicating non-key attributes with optional property, underlining the unique identifier (primary key and unique attributes in the logical schema), and marking the partial keys by a dotted underline. Figure 8.10b portrays this design-specific reverse engineering.

**FIGURE 8.10a** Information-preserving logical schema for URS1**FIGURE 8.10b** Design-Specific ERD reverse engineered from the logical schema in Figure 8.10a

Finally, the Design-Specific ERD is abstracted up one more notch based on the following procedure:

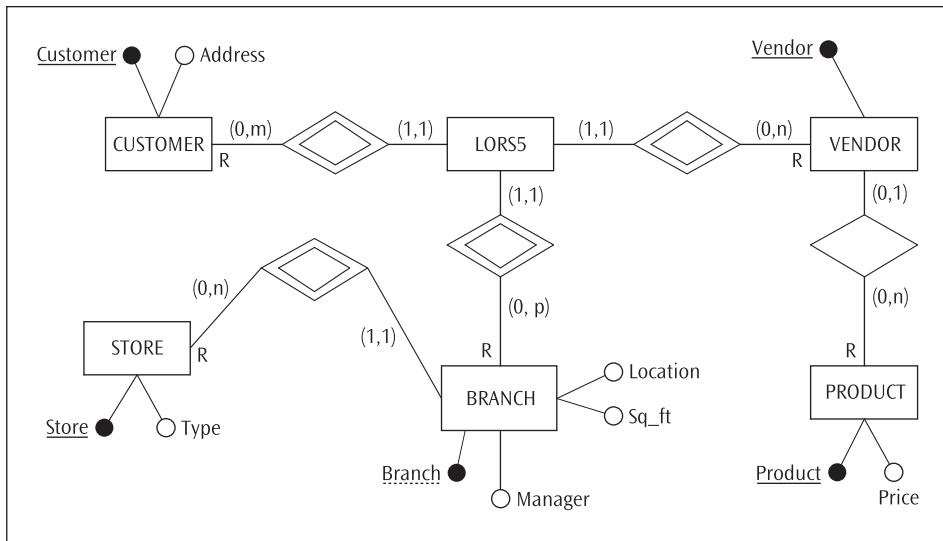
- The gerund entity types are transformed into m:n relationships between the participating entity types.
- Weak entity types with no relationship other than the identifying relationship, and no attributes other than the partial keys are transformed into multi-valued attributes of the identifying parent.
- The (min, max) grammar, for the structural constraints of relationship types is replaced by cardinality ratio and participation constraint expressed as independent constructs.

The Presentation Layer ERD for this case is shown in Figure 8.10c.

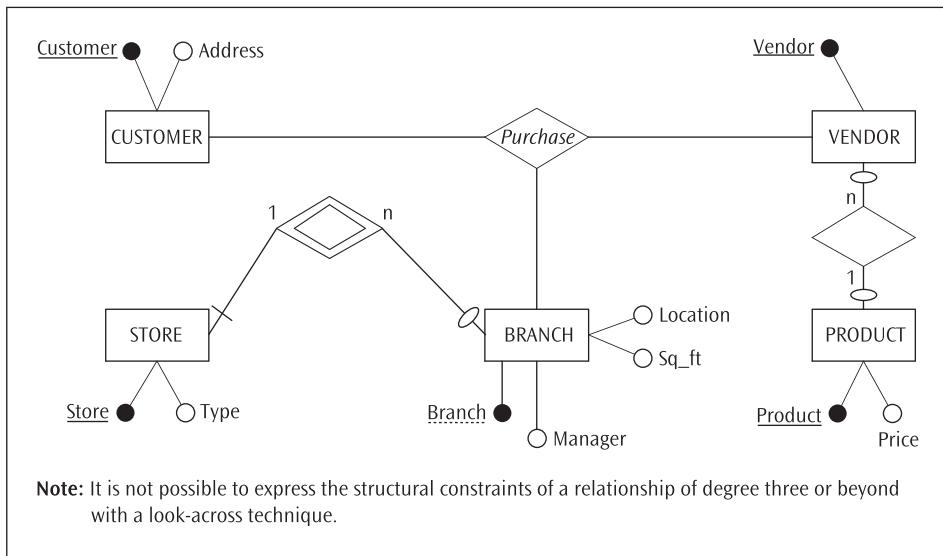


**FIGURE 8.10c** Presentation Layer ERD reverse engineered from the Design-Specific ERD in Figure 8.10b

Reverse engineering a relational schema can generate more than one solution—that is, more than one ERD that can produce the same relational schema. From a database-implementation perspective, the effect is insignificant since the same relational schema is generated. However, at the conceptual level, the designer/user may relate better to one ERD than the other. While presenting an equivalent ERD for the readers' review, we do not make any comparative assessment of the efficacy of two different reverse engineered ERDs in this book. Figure 8.11a is an alternative design equivalent to the reverse engineered Design-Specific ERD shown in Figure 8.10b. By selecting **{Store, Branch}** as the primary key instead of **Manager** in R4, R4 can be depicted as a weak entity child of R5 in the ERD with no impact on the relational schema. The corresponding Presentation Layer ERD appears in Figure 8.11b. The fact that the attribute **Manager** is a unique identifier of **BRANCH** does not appear in this ERD. This is because a weak entity type, by definition, does not have an independent unique identifier. Therefore, this information should be carried in the list of semantic integrity constraints that accompanies the ERD.



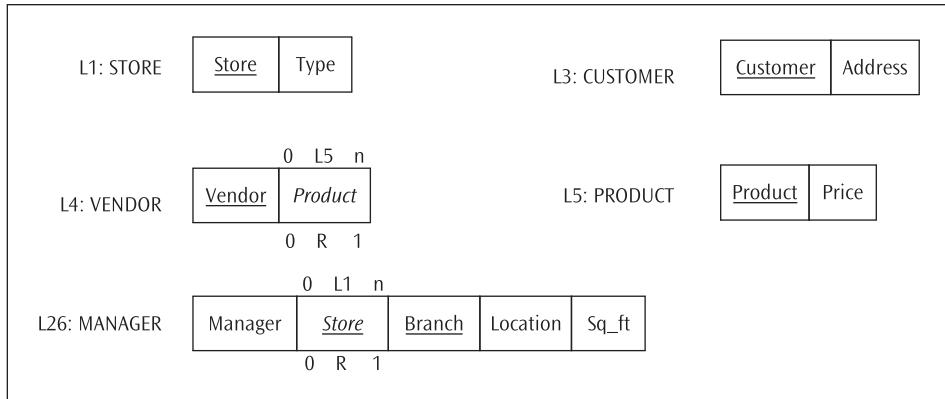
**FIGURE 8.11a** An alternative design equivalent to the Design-Specific ERD in Figure 8.10b



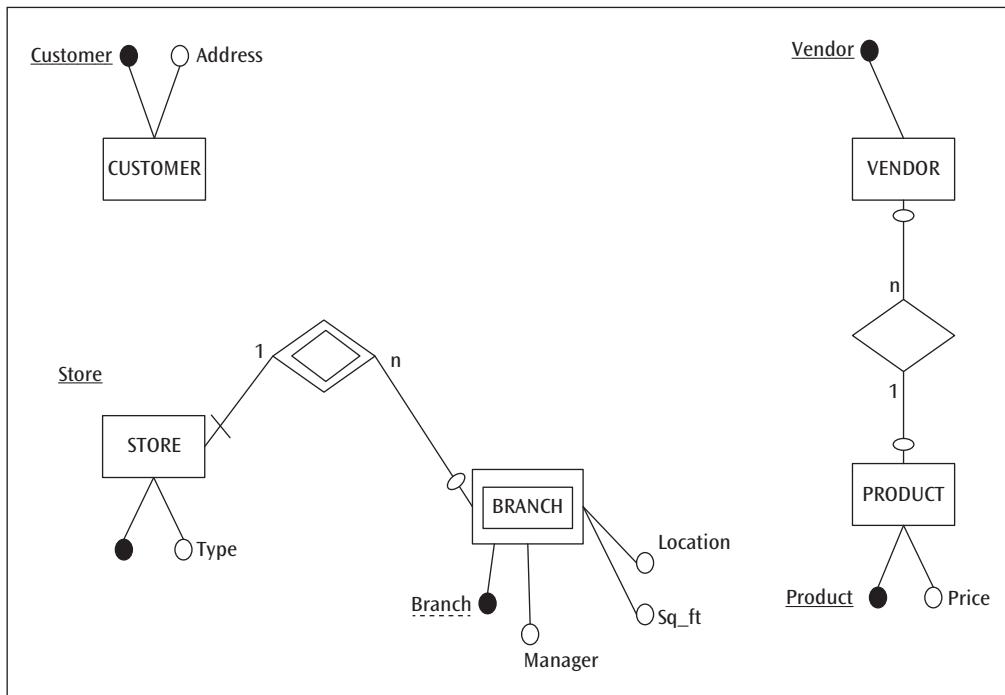
**FIGURE 8.11b** Presentation Layer ERD reverse engineered from the Design-Specific ERD in Figure 8.11a

The logical schema in Figure 8.12a is the translation of the relational schema constructed by directly mapping the FDs in F to relation schemas instead of following the normalization process. At the end of Section 8.3.1, it is pointed out that the solution from this arbitrary approach fails to capture a relation schema—viz., LORS5 (**Manager**, **Customer**, **Vendor**)—that the normalized solution yields. The consequence of this error

becomes obvious in the ERD that is reverse engineered from the logical schema depicted in Figure 8.12a, in that we see three islands of ERDs that are unconnected (see Figure 8.12b). Since the original source of the reengineering task is a single relation schema—viz., URS1—the ERD in Figure 8.12b cannot be correct.



**FIGURE 8.12a** Logical schema for URS1 {L1, L3, L4, L5, L26} constructed by directly mapping FDs to relation schemas



**FIGURE 8.12b** Presentation Layer ERD for the logical schema in Figure 8.12a

### 8.5.2 Reverse Engineering the Normalized Solution of URS2 (Case 3)

URS2 in 1NF is of the following form:

**URS2 (Company, Location, Size, President, Product, Price, Sales, Production, Supplier)**

When fully normalized to BCNF, the resulting relational schema looks like this:

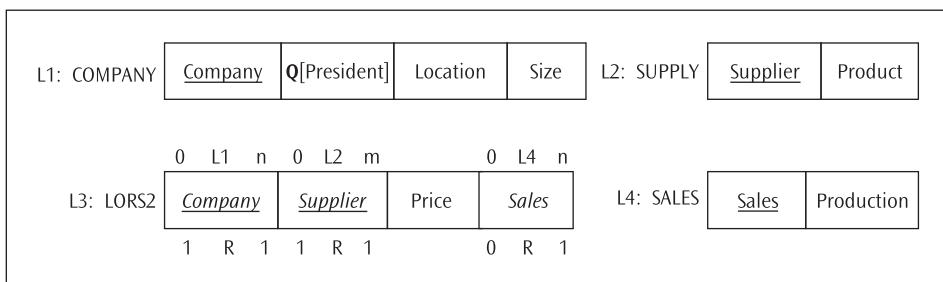
URS2 {R1, R2, R3, R0}

where:

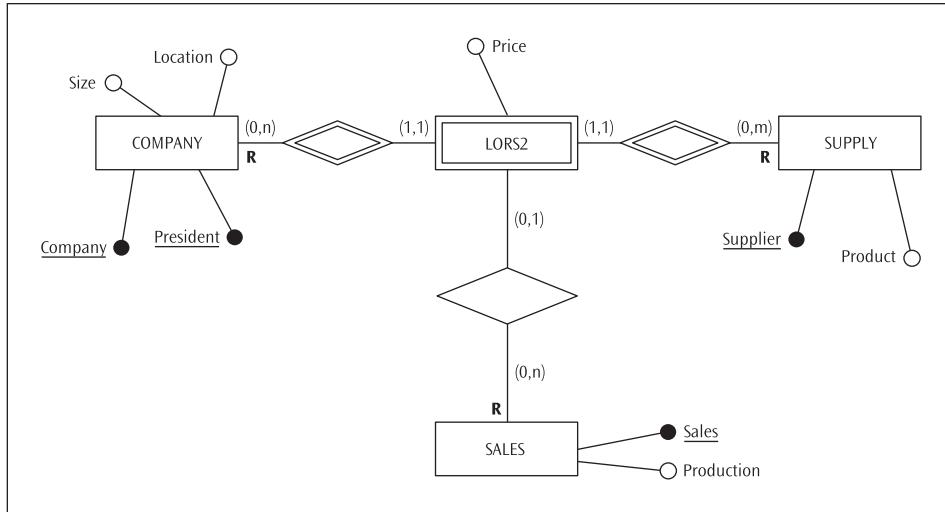
R1:	COMPANY ( <u>Company</u> , <u>Location</u> , <u>Size</u> , <u>President</u> );	in BCNF
R2:	SALES ( <u>Sales</u> , <u>Production</u> )	in BCNF
R3:	SUPPLY ( <u>Supplier</u> , <u>Product</u> )	in BCNF
R0:	LORS2 ( <u>Company</u> , <u>Sales</u> , <u>Price</u> , <u>Supplier</u> )	in BCNF
#	LORS2.{ <u>Company</u> } $\subseteq$ COMPANY.{ <u>Company</u> }	Lossless-join
#	LORS2.{ <u>Supplier</u> } $\subseteq$ SUPPLY.{ <u>Supplier</u> }	Lossless-join

451

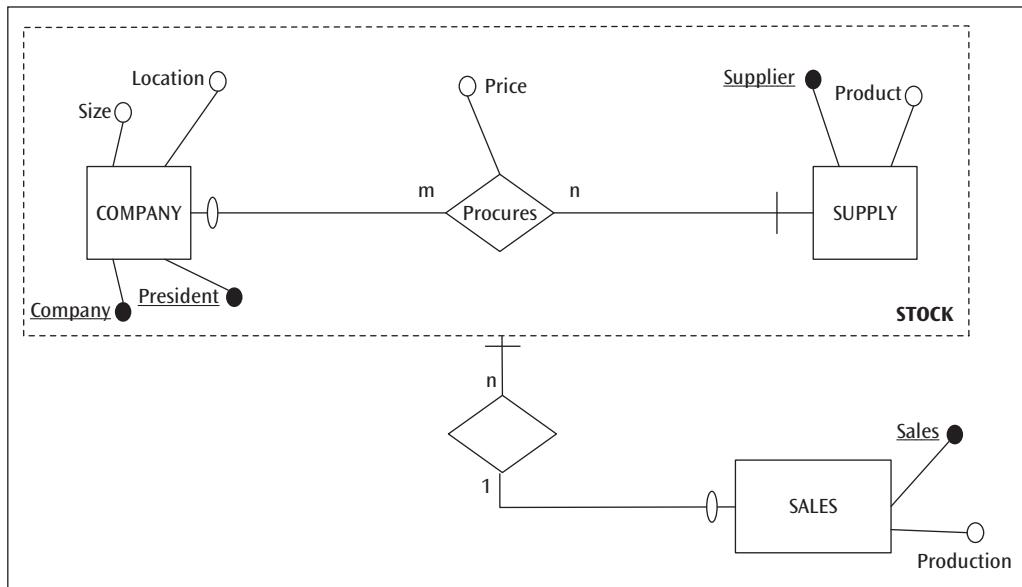
The information-preserving logical schema based on available information and assumed default properties is shown in Figure 8.13a. Next, following Step 2 in Figure 8.9, the Design-Specific ERD is created. The last step is to abstract up the Design-Specific ERD to the Presentation Layer ERD. This is done by following the procedure suggested in Step 3 of the reverse-engineering heuristic. The two layers of ERD reverse-engineered from the logical schema shown in Figure 8.13a appear in Figures 8.13b and 8.13c, respectively.



**FIGURE 8.13a** Logical schema for URS2



**FIGURE 8.13b** Design-Specific ERD reverse engineered from the logical schema in Figure 8.13a



**FIGURE 8.13c** Presentation Layer ERD reverse engineered from the Design-Specific ERD in Figure 8.13b

### 8.5.3 Reverse Engineering the Normalized Solution of URS3 (Case 2)

URS3 in 1NF is of the following form:

**URS3 (Proj\_nm, Emp#, Proj#, Job\_type, Chg\_rate, Emp\_nm, Budget, Fund#, Hours, Division)**

When fully normalized to BCNF, the resulting relational schema looks like this:

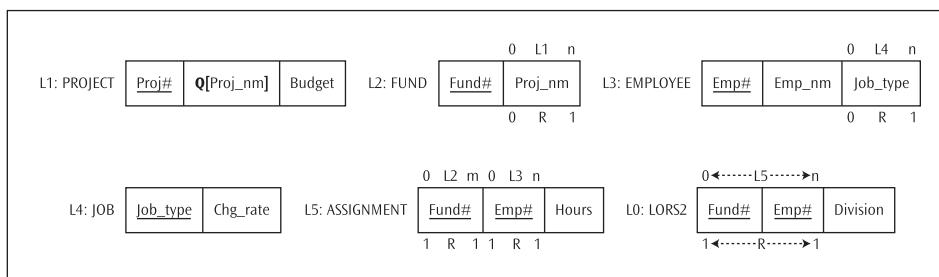
URS3 {R1, R2, R3, R4, R5b, R0}

where:

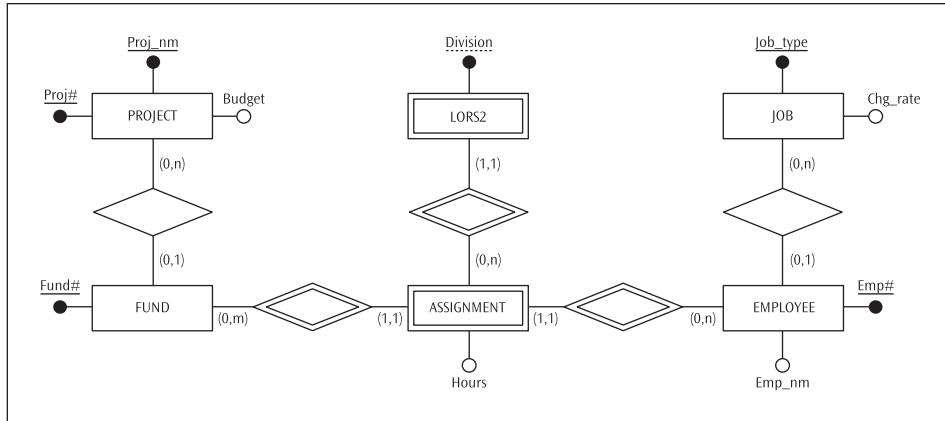
R1:	PROJECT ( <u>Proj#</u> , <u>Proj_nm</u> , <u>Budget</u> );	in BCNF
R2:	FUND ( <u>Fund#</u> , <u>Proj_nm</u> )	in BCNF
R3:	EMPLOYEE ( <u>Emp#</u> , <u>Emp_nm</u> , <u>Job_type</u> )	in BCNF
R4:	JOB ( <u>Job_type</u> , <u>Chg_rate</u> )	in BCNF
R5b:	ASSIGNMENT ( <u>Fund#</u> , <u>Emp#</u> , <u>Hours</u> )	in BCNF
R0:	LORS2 ( <u>Fund#</u> , <u>Emp#</u> , <u>Division</u> )	in BCNF
#	LORS2.{Fund#, Emp#} $\subseteq$ ASSIGNMENT.{Fund#, Emp#}	Lossless-join
#	ASSIGNMENT.{Emp#} $\subseteq$ EMPLOYEE.{Emp#}	Lossless-join
#	ASSIGNMENT.{Fund#} $\subseteq$ FUND.{Fund#}	Lossless-join
#	EMPLOYEE.{Job_type} $\subseteq$ JOB.{Job_type}	Lossless-join
#	FUND.{Proj_nm} $\subseteq$ PROJECT.{Proj_nm}	Lossless-join

453

The information-preserving logical schema based on available information and assumed default properties is shown in Figure 8.14a. Next, following Step 2, the Design-Specific ERD is created (see Figure 8.14b).

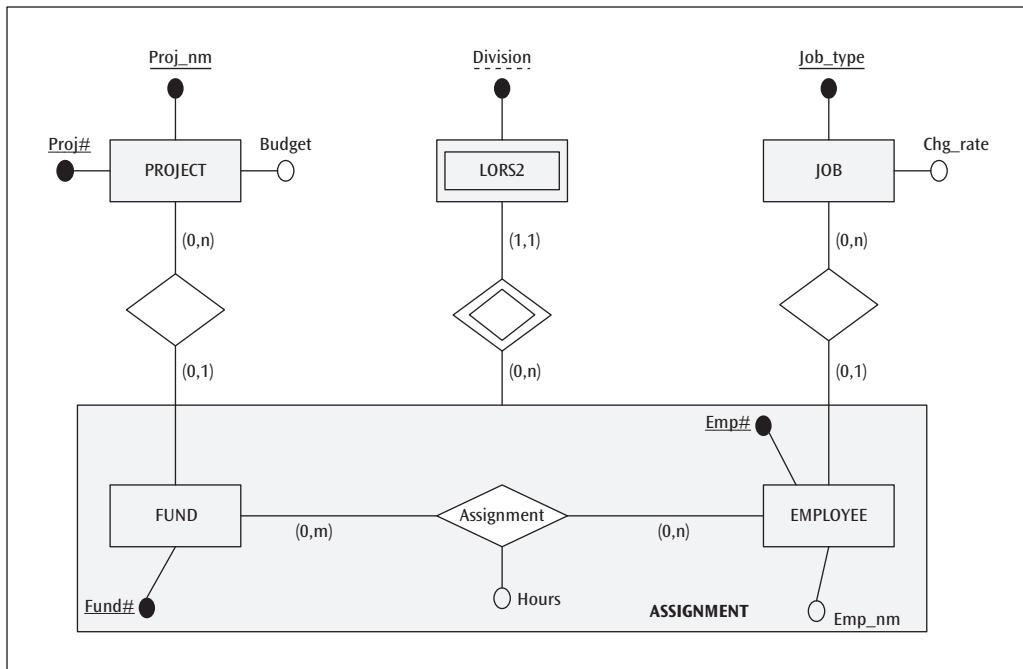


**FIGURE 8.14a** Logical schema for URS3

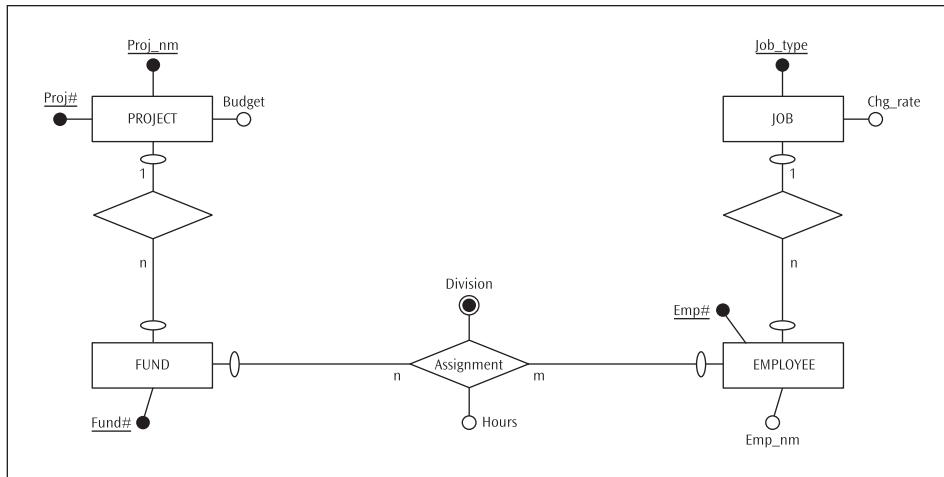


**FIGURE 8.14b** Design-Specific ERD reverse engineered from the logical schema in Figure 8.14a

The last step is to reverse engineer the Design-Specific ERD to the Presentation layer ERD. This is done by following the procedure suggested in Step 3 of the reverse-engineering heuristic in Figure 8.9. The reverse-engineered Presentation Layer ERD is shown in Figure 8.14c. An additional level of abstraction of Figure 8.14c appears in Figure 8.14d. In fact, only Figure 8.14d represents the Presentation Layer ERD for the given relational schema.

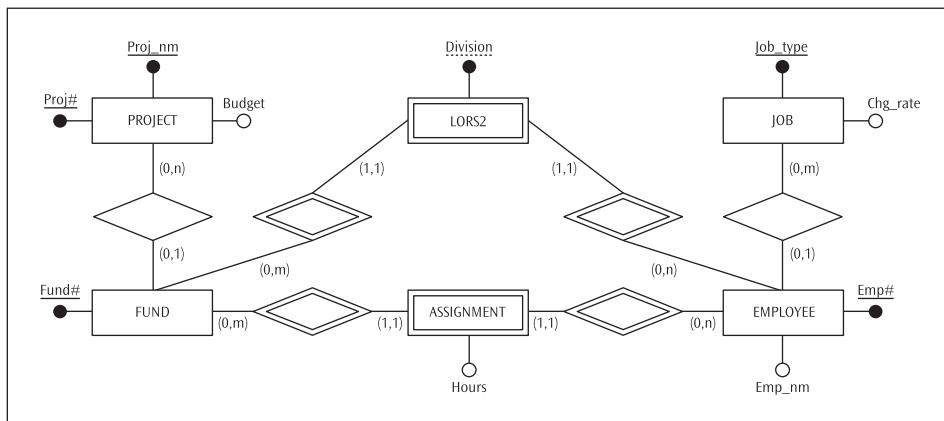


**FIGURE 8.14c** Design-Specific ERD reverse engineered from Figure 8.14b to a higher level of abstraction

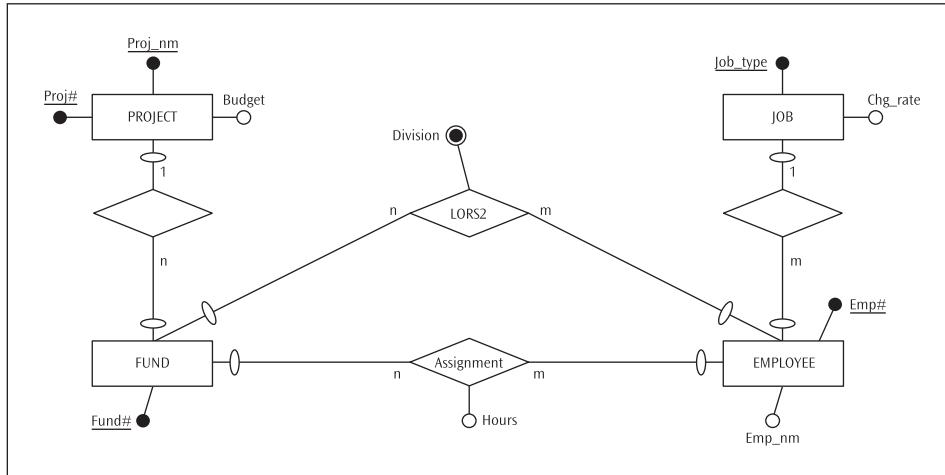


**FIGURE 8.14d** Presentation Layer ERD reverse engineered from the Design-Specific ERD in Figure 8.14c

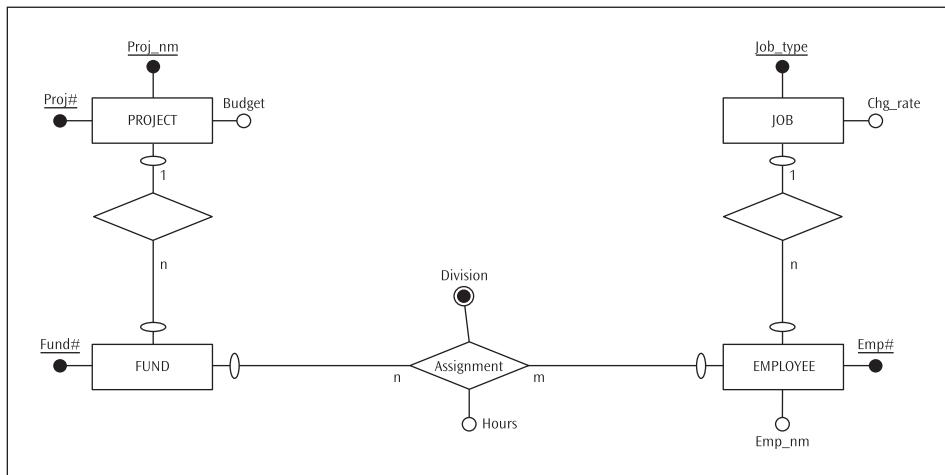
Once again, a different ERD equivalent to the one in Figures 8.14b-d—equivalent in the sense it generates the same relational schema—is shown in Figure 8.15a-c. The variation in the design arises from the fact that based on the final relational schema in BCNF, R0 can be modeled in the ERD as either the weak entity child of R5b or a weak entity type with two identifying parents, viz., R2 and R3.



**FIGURE 8.15a** An alternative design equivalent to the Design-Specific ERD in Figure 8.15b



**FIGURE 8.15b** Presentation Layer ERD reverse engineered from the Design-Specific ERD in Figure 8.15a



**FIGURE 8.15c** Presentation Layer ERD reverse engineered from the Design-Specific ERD in Figure 8.15a: an alternative design

## Chapter Summary

The root cause of data redundancy and the consequent modification anomalies in a database is the presence of undesirable functional dependencies in a relation schema. Any FD whose determinant is not a candidate key of the relation schema in which the FD holds is an undesirable FD. FDs emerge from business rules of the application domain and so cannot be ignored or discarded when they are undesirable. The set of FDs that prevail over a relation schema is referred to as F. This chapter explores the solution for this problem.

Normalization is the mechanism capable of systematically weeding out undesirable FDs from a relation schema. The undesirable FDs manifest themselves either as partial dependencies or as transitive dependencies in a relation schema. Normalization prescribes a method to render the undesirable FDs desirable. This is done by decomposing a relation schema with undesirable FDs to a set of relation schemas so that, in each relation schema of the decomposed set, the determinant of the preserved FD is the candidate key of that relation schema.

Normal forms (NFs) provide a stepwise progression towards attaining the goal of a fully normalized relational schema that is guaranteed to be free of data redundancies that cause modification anomalies from a functional dependency perspective. First normal form (1NF) defines a relation schema—that is, a schema that is not in 1NF is not a relation schema. Elimination of partial dependencies establishes 2NF. Two variations of transitive dependencies are resolved by 3NF and Boyce-Codd Normal Form (BCNF). A relation schema in BCNF is guaranteed to be free of modification anomalies due to functional dependencies.

A relational schema (that is, a set of relation schemas) in BCNF does not necessarily result in a good database design. Relational schemas created through the decomposition process should also exhibit the lossless-join and dependency preservation properties. The lossless-join property is critical to ensure that spurious tuples are not generated by a join operation between two relations in the relational schema. The dependency preservation property, which ensures that each functional dependency is represented in some individual relation schema resulting after the decomposition, is sometimes sacrificed.

After introducing first, second, third, and BCNSs plus the lossless-join and dependency preservation properties in the context of a series of individual examples, the motivating exemplar introduced originally in Chapter 7 was reexamined to illustrate how a set of functional dependencies derived from user-specified business rules can be used to develop a fully normalized relational schema. This discussion answers the three questions presented in Section 7.1: (a) How are data redundancies identified? (b) How is the base relation schema decomposed? and (c) How can the decomposition be evaluated for correctness and completeness?

This chapter presented a comprehensive approach to the normalization process that incorporates trade-offs among a BCNF design, a lossless-join decomposition, and dependency preservation. The approach involves (a) the derivation of the candidate keys along with the primary key of the initial universal relational schema (URS), (b) the identification and resolution of all second and third normal form violations in the URS, (c) the resolution of all Boyce-Codd normal form violations in the URS, and (d) an evaluation of whether the relational schema is dependency-preserving and yields a lossless-join decomposition.

At that point, once the various issues and trade-offs associated with the normalization process had been explained, a fast-track algorithm to quickly achieve a non-loss and dependency-preserving solution was introduced. A reader who completely understands the ramifications of the various issues associated with normalization can use this algorithm to quickly achieve a non-loss

3NF design that is also dependency-preserving, and can then use the traditional process of normalization to resolve the BCNF violation, if any, in the relational schema.

When a relational schema is in BCNF, it may not be possible to achieve both lossless-join and dependency-preservation properties in the design. One alternative in this situation is to accept a relational schema in third normal form and handle the potential data-redundancy problems via application programs. Often, a superior alternative is to establish a BCNF design that possesses the lossless-join property and then use materialized views to capture the few functional dependencies that are not preserved.

Since the early 1990s, reverse engineering has drawn the attention of database researchers. Traditionally, reverse engineering seeks to examine operational software systems with a view to analyzing data patterns to extract data structures and behaviors. This chapter presented a unique use of reverse engineering in the data modeling task in order get a better grip on design errors. To that end, the scope of Section 8.5 is restricted to reverse-engineering normalized relational schemas to their conceptual counterparts, viz., ERDs. Reverse engineering a normalized relational schema to an ERD reveals how the ERD should have been to begin with. Such discovery enriches the designers' understanding of the application domain being modeled. The chapter concluded with a description and a few demonstrations of a heuristic to reverse engineer a relational schema to a Presentation Layer ERD.

## Exercises

---

1. What is the source of functional dependencies?
2. What is the difference between a desirable and an undesirable functional dependency? Describe the nature of the problems caused by undesirable functional dependencies. What prevents us from simply ignoring undesirable functional dependencies?
3. What is the role of normalization in the database design process?
4. Figure 8.1 illustrates how a first normal form violation of ALBUM can be resolved. Suppose that a single album could never have more than four artists. Describe another approach for defining ALBUM that does not violate first normal form.
5. Suppose  $\{A, B\} \rightarrow C$  in a relation schema R. Under what condition would this not reflect a full functional dependency?
6. Consider the relation instance of the STU-CLASS relation schema:  
**STU-CLASS (Snum, Sname, Major, Cname, Time, Room)**

Snum	Sname	Major	Cname	Time	Room
0110	KHUMAWALA	ACCOUNTING	BA482	MW3	C-150
0110	KHUMAWALA	ACCOUNTING	BD445	TR2	C-213
0110	KHUMAWALA	ACCOUNTING	BA491	TR3	C-141
1000	STEDRY	ANTHROPOLOGY	AP150	MWF9	D-412
1000	STEDRY	ANTHROPOLOGY	BD445	TR2	C-213
2000	KHUMAWALA	STATISTICS	BA491	TR3	C-141

Snum	Sname	Major	Cname	Time	Room
2000	KHUMAWALA	STATISTICS	BD445	TR2	C-213
3000	GAMBLE	ACCOUNTING	BA482	MW3	C-150
3000	GAMBLE	ACCOUNTING	BP490	MW4	C-150

- Identify at least one update anomaly, one insertion anomaly, and one deletion anomaly.
7. What are the undesirable functional dependencies in the relation instance of the STUDENT-CLASS relation schema shown in Exercise 6?
  8. Consider the following relation instance of the CAR relation schema:

459

**CAR**

Model	#cylinders	Origin	Tax	Fee
Camry	4	Japan	15	30
Mustang	6	USA	0	45
Fiat	4	Italy	18	30
Accord	4	Japan	15	30
Century	8	USA	0	45
Mustang	4	Canada	0	30
Monte Carlo	6	Canada	0	45
Civic	4	Japan	15	30
Mustang	4	Mexico	15	30
Mustang	6	Mexico	15	45
Civic	4	Korea	15	30

- a. What is the minimal cover of functional dependencies that exists in CAR?
  - b. What is (are) the candidate key(s) of CAR?
  - c. If there is more than one candidate key, select a primary key from among the candidate keys.
  - d. Based on the primary key, what is the immediate Normal Form violated in CAR?
  - e. Develop a dependency-preserving solution that eliminates this Normal Form violation.  
Is your solution a lossless-join decomposition?
9. Consider the following relation instance of the SPORT relation schema:

**SPORT**

Stu#	Sport	Coach
125	Football	Register
140	Basketball	Lambert

- | Stu# | Sport      | Coach    |
|------|------------|----------|
| 220  | Baseball   | Register |
| 246  | Basketball | Lambert  |
- a. What is the minimal cover of functional dependencies that exists in SPORT?
- b. What is (are) the candidate key(s) of SPORT?
- c. If there is more than one candidate key, select a primary key from among the candidate keys.
- d. Based on the primary key, what is the normal form violated in SPORT?
- e. Develop a dependency-preserving solution that eliminates these Normal Form violations. Is your solution a lossless-join decomposition?
10. Given a relation schema R (A, B, C) that is in 3NF:
- State the conditions under which modification anomalies can exist.
  - Express the conditions stated above in terms of functional dependencies.
  - For the conditions stated in Step a, modification anomalies need not exist. State the functional dependencies for this case.
11. What is the difference between 3NF and Boyce-Codd Normal Form (BCNF)?
12. Consider the relation instance EXAM (Student, Subject, Rank). The meaning of an EXAM tuple is that a specified student examined in a specified subject achieves a specified rank in the class. Further, no two students obtain the same rank in the same subject.

## EXAM

Student	Subject	Rank
McGrady	Math	3
Howard	Math	4
McGrady	English	2
Jackson	English	1
Yao	Math	1
Yao	Chemistry	1
Sura	Math	2
Ward	English	3
Taylor	Chemistry	2
Taylor	Math	5
Ewing	Chemistry	3

- What is the minimal cover of functional dependencies that exists in EXAM?
- What is (are) the candidate key(s) of EXAM?
- If there is more than one candidate key, select a primary key from among the candidate keys.
- Based on your primary key, what normal form violations exist in EXAM?

13. Consider the relation schema PATIENT\_VISIT (Patient, Hospital, Doctor) and the relation instance given here:

Patient	Hospital	Doctor
Smith	Methodist	D. Cooley
Lee	St. Luke's	Z. Zhang
Marks	Methodist	D. Cooley
Marks	St. Luke's	W. Lowe
Lou	Hermann	R. Duke

461

In addition, suppose the following semantic rules exist:

- Each patient may be a patient in several hospitals.
  - For each hospital, a patient may have only one doctor.
  - Each hospital has several doctors.
  - Each doctor uses only one hospital.
  - Each doctor treats several patients in one hospital.
- a. What is the minimal cover of functional dependencies that exists in PATIENT\_VISIT?
  - b. What is (are) the candidate key(s) of PATIENT\_VISIT?
  - c. If there is more than one candidate key, select a primary key from among the candidate keys.
  - d. Based on the primary key chosen, what normal form violations exist in PATIENT\_VISIT?
  - e. Develop a solution that eliminates the Normal Form violations.
  - f. Is your solution a lossless-join decomposition?
  - g. Is your solution dependency-preserving? If not, how can dependency preservation be achieved? Is this revised solution in BCNF?
  - h. Provide a solution that meets all three of the following conditions: (1) is in BCNF, (2) is dependency-preserving, and (3) is a lossless-join decomposition.
14. Why are attribute preservation, dependency preservation, and lossless-join decomposition requirements of a fully normalized database design?
15. Consider the relation schema ACTIVITY (Stu#, Sport, Cost) and associated relation instance shown here:

F: fd1: Stu# → Sport; fd2: Stu# → Cost; fd3: Sport → Cost

Stu#	Sport	Cost
100	SKIING	200
150	TENNIS	50
175	KARATE	50
200	TENNIS	50

- a. What is the minimal cover of ACTIVITY?
- b. What, if any, immediate Normal Form violation exists in ACTIVITY?
- c. Discuss each of the following three decompositions in the context of attribute preservation, dependency preservation, and lossless-join decomposition:
- D: R1a: STU-SPORT (Stu#, Sport) R2a: SPORT-COST (Sport, Cost)
- D: R1b: STU-SPORT (Stu#, Sport) R2b: STU-COST (Stu#, Cost)
- D: R1c: STU-COST (Stu#, Cost) R2c: SPORT-COST (Sport, Cost)
16. What is the difference between a loss-join decomposition and a lossless-join decomposition?
17. Consider relation SUPPLY (S#, Sname, P#, Qty) with supplier names unique such that:
- F: fd1: S# → Sname; fd2: Sname → S#; {S#, P#} → Qty
- S# represents a supplier number, Sname represents a supplier name, P# represents a part number, and Qty represents the quantity of a specific part supplied by a specific supplier.

462

**SUPPLY**

<b>S#</b>	<b>Sname</b>	<b>P#</b>	<b>Qty</b>
S1	SMITH	P1	300
S1	SMITH	P2	300
S1	SMITH	P3	400
S1	SMITH	P4	200
S1	SMITH	P5	100
S1	SMITH	P6	100
S2	CLARK	P1	300
S2	CLARK	P2	400
S3	MORRIS	P2	200
S4	MCNARY	P2	200
S4	MCNARY	P4	300
S4	MCNARY	P5	400

- a. Is there a 3NF violation in SUPPLY? If yes, explain. If no, is there a BCNF violation in SUPPLY? Explain.
- b. Decompose SUPPLY if necessary so that the resulting relational schema is in BCNF. Is your design attribute-preserving, dependency-preserving, and a lossless-join decomposition? Explain.

18. Consider the relation schema CLASS with attributes **Student**, **Subject**, and **Teacher**. The meaning of this relation is that the specified student is taught the specified subject by the specified teacher. Assume that semantic rules, depicted by the following functional dependencies, exist:

$\{Student, Subject\} \rightarrow Teacher$

$Teacher \rightarrow Subject$

$Subject \not\rightarrow Teacher$

$\{Student, Teacher\} \rightarrow Subject$

a. What do these rules mean in words? Are all these rules necessary? If not, explain which are not needed.

b. Is the following sample data consistent with these rules? Why or why not?

Student	Subject	Teacher
Smith	Math	White
Smith	Physics	Green
Jones	Math	White
Jones	Physics	Brown

- c. What causes CLASS to contain a BCNF violation? What anomalies does it exhibit?
- d. Decompose CLASS if necessary so that the resulting relational schema is in BCNF. Is your design attribute-preserving, dependency-preserving, and a lossless-join decomposition? Explain.
19. This exercise is a variation of Exercise 18. Consider the following functional dependencies prevailing over the relation schema CLASS (Student, Subject, Teacher):

$\{Student, Subject\} \rightarrow Teacher$

$Teacher \not\rightarrow Subject$

$Subject \not\rightarrow Teacher$

$\{Student, Teacher\} \not\rightarrow Subject$

a. What do these rules mean in words? Are all these rules necessary? If not, explain which are not needed.

b. Is the following sample data consistent with these rules? Why or why not?

Student	Subject	Teacher
Smith	Math	White
Smith	Physics	White
Jones	Math	Green
Jones	Physics	White

- c. Does this version of CLASS satisfy the requirements of BCNF? Is it free of modification anomalies triggered by undesirable functional dependencies? Is it free of modification anomalies altogether?

20. Given the relation schema FLIGHT (Gate#, Flight#, Date, Airport, Aircraft, Pilot) and the constraint set F {fd1, fd2, fd3}, where:

fd1: {Airport, Flight#, Date} → Gate; fd2: {Flight#, Date} → Aircraft

fd3: {Flight, Date} → Pilot

- List the candidate key(s) of FLIGHT.
- For each candidate key, indicate the immediate normal form violated in FLIGHT by each of the functional dependencies given above.
- If FLIGHT is not in BCNF, design a relational schema that
  - is in BCNF, and
  - yields a lossless-join decomposition
- Are all functional dependencies in F preserved? If not, which are not preserved?

21. Consider the universal relation schema INVENTORY (Store#, Item, Vendor, Date, Cost, Units, Manager, Price, Sale, Size, Color, Location) and the constraint set F {fd1, fd2, fd3, fd4, fd5, fd6, fd7} introduced originally in Chapter 7, Exercise 13, where:

fd1: {Item, Vendor} → Cost

fd2: {Store#, Date} → {Manager, Sale}

fd3: {Store#, Item, Date} → Units

fd4: Manager → Store#

fd5: Cost → Price

fd6: Item → {Size, Color}

fd7: Vendor → Location

- Confirm that F is a minimal cover for the set of functional dependencies given above.
  - List the candidate key(s) of INVENTORY.
  - For each candidate key, indicate the immediate Normal Form violated in INVENTORY by each of the functional dependencies given above.
  - If INVENTORY is not in BCNF, design a relational schema that
    - is in BCNF so that all modification anomalies due to functional dependencies are eradicated, and
    - yields *all* lossless-join decompositions
  - List the functional dependencies in F that are not preserved in this design.
  - Show the final design. The design should be parsimonious (i.e., minimal set in BCNF). *Also, clearly indicate entity integrity and referential integrity constraints.*
  - Revise the above design so that all dependencies are preserved in a lossless-join decomposition with the least sacrifice in the achieved level of normal form.
22. Given the set of functional dependencies F {fd1, fd2, fd3, fd4, fd5, fd6, fd7, fd8, fd9, fd10} introduced originally in Chapter 7, Exercise 14:

fd1: Tenant# → {Name, Job, Phone#, Address}

fd2: Job → Salary

fd3: Name → Gender

fd4: Phone# → Address

fd5: {Name, Phone#} → {Tenant, Deposit}

fd6: County → Tax\_rate

fd7: Area → {Rent, County}

fd8: Survey# Lot

fd9: {Lot, County} → {Survey#, Area}

fd10: {Survey#, Area} → County

- a. Using the universal relation schema (URS) and its associated primary key developed in Chapter 7, Exercise 14, indicate the immediate Normal Form violated in each of the functional dependencies given above and explain how the particular Normal Form is violated in each case.
  - b. If the URS is not in BCNF, design a relational schema that:
    - is in BCNF so that all modification anomalies due to functional dependencies are eradicated, and
    - yields all lossless-join decompositions
  - c. List the functional dependencies in F that are not preserved in this design.
  - d. Show the final design. The design should be parsimonious (i.e., minimal set in BCNF). *Also, clearly indicate entity integrity and referential integrity constraints.*
  - e. Revise the above design so that *all* dependencies are preserved in a lossless-join decomposition with the least sacrifice in the achieved level of normal form.
  - f. Reverse engineer the design to the conceptual level and show it as a Presentation Layer ERD.
23. Given the set of functional dependencies F {fd1, fd2, fd3, fd4, fd5, fd6, fd7, fd8, fd9, f10, f11} introduced originally in Chapter 7, Exercise 15:

fd1: Client → Office	fd2: Stock → {Exchange, Dividend}
fd3: Broker → Profile	fd4: Company → Stock
fd5: Client → {Risk_profile, Analyst}	fd6: Analyst → Broker
fd7: {Stock, Broker} → {Investment, Volume}	fd8: Stock → Company
fd9: Investment → {Commission, Return}	fd10: {Stock, Broker} → Client
fd11: Account → Assets	

- a. Using the universal relation schema (URS) and its associated primary key developed in Chapter 7, Exercise 15, indicate the *immediate* Normal Form violated in each of the functional dependencies given above and explain how the particular normal form is violated in each case.
- b. Decompose the URS to arrive at a design/schema that is in BCNF. In each decomposition, identify the primary key and the functional dependencies accounted for. Demonstrate that each decomposition is a lossless-join decomposition.
- c. Show the final design. The design should be parsimonious (minimal set in BCNF). *Clearly indicate entity integrity and referential integrity constraints.*
- d. Indicate the functional dependencies in F that are not preserved in the BCNF design and specify materialized view(s) for the same.
- e. Reverse engineer the design to the conceptual level and show it is a Presentation Layer ERD.
- f. Revise the design above so that *all* dependencies are preserved in a lossless-join decomposition, with the least sacrifice in the achieved level of normal form.

24. Given the URS R (A, B, C, D, E, I, J, K, O, U, X, Y, Z, P, Q) and

F {fd1, fd2, fd3, fd4, fd5, fd6, fd7, fd8, fd9, fd10, fd11, fd12, fd13} prevailing over R, where:

fd1: $U \rightarrow K;$	fd2: $\{A, C\} \rightarrow E;$	fd3: $O \rightarrow J;$	fd4: $B \rightarrow D;$	fd5: $\{J, K\} \rightarrow O;$
fd6: $E \rightarrow C;$	fd7: $\{J, K\} \rightarrow U;$	fd8: $A \rightarrow B$	fd9: $O \rightarrow X;$	fd10: $U \rightarrow Y;$
fd11: $B \rightarrow Z;$	fd12: $A \rightarrow P;$	fd13: $E \rightarrow Q$		

- a. Derive the canonical cover ( $F_c$ ) of F.
- b. If URS is not in BCNF, derive a relational schema that:

- is in BCNF and attribute-preserving
- yields a fully lossless join decomposition
- preserves, if not all, at least *all possible* functional dependencies

The final design (relational schema) should be parsimonious. *Clearly indicate entity integrity and referential integrity constraints.*

- c. List the functional dependencies in F that are not preserved in your BCNF design.
  - d. Revise (denormalize) the above design so that *all* dependencies are preserved in a lossless join decomposition with the least sacrifice in the achieved level of normal form.
25. Given the URS (A, B, C, D, F, G) and the set of FDs prevailing over URS, F {fd1, fd2, fd3, fd4, fd5}, where:

fd1: $A \rightarrow G;$	fd2: $\{A, B\} \rightarrow \{C, D\};$	fd3: $\{B, C\} \rightarrow \{F, G\};$
fd4: $G \rightarrow B;$	fd5: $\{B, D\} \rightarrow A;$	fd6: $C \rightarrow G$

- a. Derive a canonical cover of F.
  - b. Derive all the candidate key(s) of URS.
  - c. Normalize URS to obtain a parsimonious, non-loss BCNF design.
  - d. List any FD(s) not preserved in this design.
26. Given the URS **ALLOY** (**Iron, Chromium, Platinum, Carbon**) and F {fd1, fd2, fd3}, where:

fd1: $\text{Iron} \rightarrow \text{Chromium, Platinum, Carbon};$
fd2: $\{\text{Chromium, Platinum}\} \rightarrow \text{Carbon, Iron};$
fd3: $\text{Carbon} \rightarrow \text{Chromium}$

There are two canonical covers ( $F_c$ ) for F.

- a. Derive both the canonical covers.
- b. Normalize ALLOY to generate dependency-preserving designs resulting from each of the two canonical covers.
- c. Normalize the two solutions developed in b) to non-loss BCNF designs.
- d. In the BCNF solution, list the FDs not preserved in each of the canonical covers.

# CHAPTER 9

# HIGHER NORMAL FORMS

The focus of Chapter 8 was data redundancies resulting from undesirable functional dependencies and the aspects of normalization that address this specific problem. This chapter completes the discussion of normalization by addressing normal forms that are beyond the purview of functional dependencies.

This chapter flows as follows. Section 9.1 introduces the concept of multi-valued dependency (MVD) in a relation schema. It begins with a motivation exemplar in Subsection 9.1.1 to help the reader appreciate the import of MVD intuitively; Subsections 9.1.2 and 9.1.3 then present the formal definition of MVD and the inference rules associated with MVD, respectively. In Section 9.2, fourth normal form (4NF) is introduced as the solution to eliminate data redundancies caused by MVDs. This is followed, in Section 9.3, by a comprehensive example describing the occurrence and resolution of 4NF violation in a relation schema. The generality of 4NF is discussed in Section 9.4 by showing how 4NF subsumes all the previously discussed normal forms. The topic of Section 9.5 is the concept of join-dependency and the associated normal form called Project/Join normal form (PJNF) or fifth normal form (5NF). A brief note on Domain/Key normal form (DKNF) in Section 9.6 concludes this chapter.

## 9.1 MULTI-VALUED DEPENDENCY

---

So far, we have examined modification anomalies (insertion, deletion, and update anomalies) due to data redundancies triggered by undesirable functional dependencies in a relation schema and have developed a solution to the problem via normalization. We concluded that when a relation schema is normalized to BCNF, the resulting BCNF design (relational schema) is guaranteed to be free of modification anomalies due to functional dependencies. In other words, the basis for 2NF, 3NF, and BCNF has been functional dependencies. However, there are modification anomalies that are not based on functional dependencies, as seen in the following sections.

### 9.1.1 A Motivating Exemplar for Multi-Valued Dependency

Consider an entity type **MUSICIAN** with attributes **Name**, **Age**, **Address**, **Phone#**, and **Band**. Also, assume that a musician can be identified by his/her name. In addition, suppose that a musician may be associated with several types of music (e.g., jazz, rock, classical) and may own several vehicles (e.g., car, jeep, truck, van). In other words, while **Name**, **Age**, **Address**, **Phone#**, and **Band** are single-valued attributes of **MUSICIAN**, **Music** and **Vehicle**

are multi-valued attributes of MUSICIAN. A schema representing MUSICIAN can be shown as: MUSICIAN (**Name**, **Age**, **Address**, **Phone#**, {**Music**}, {**Vehicle**}, **Band**)

where fd1: **Name** → **Age**, **Address**, **Phone#**, #, **Band**, and **Music**, and **Vehicle** (enclosed in {}) are multi-valued attributes.

MUSICIAN, by definition, is not a relation schema because it is not in 1NF due to the presence of multi-valued attributes **Music** and **Vehicle**.

MUSICIAN can be set in 1NF by reducing **Music** and **Vehicle** to single-valued attributes. This is accomplished by introducing the functional dependency (FD) fd2 in R0, where:

fd2: (**Name**, **Music**, **Vehicle**) → **Age**, **Address**, **Phone#**, **Band**

As a result, we have a 1NF schema:

R0: MUSICIAN (**Name**, **Music**, **Vehicle**, **Age**, **Address**, **Phone#**, **Band**)

Since MUSICIAN (R0) is in 1NF, it is a relation schema with (**Name**, **Music**, **Vehicle**) as its primary key.

In this relation schema, fd1 violates 2NF. The decomposition D: {R1, R}, where:

R1: MUSICIAN (**Name**, **Age**, **Address**, **Phone#**, **Band**)

R: MUSIC\_VEHICLE (**Name**, **Music**, **Vehicle**)

is not only in 2NF but also in 3NF and BCNF.

At this point, let us explore MUSIC\_VEHICLE further using an instance of the relation schema shown in Table 9.1. A tuple in this relation indicates that a musician (who has a name) is, say, learning different types of music and owns different kinds of vehicles.

Notably, the kinds of vehicles a musician owns are independent of the types of music he or she learns—that is, the attributes **Music** and **Vehicle** are independent of each other in MUSIC\_VEHICLE.

Name	Music	Vehicle
Kamath	Jazz	Jeep
Kamath	Jazz	Truck
Kamath	Jazz	Van
Kamath	Rock	Truck
Kamath	Rock	Jeep
Kamath	Rock	Van
McKinney	Classical	Van
McKinney	Rock	Van
McKinney	Classical	Car
McKinney	Rock	Car
Barron	Jazz	Jeep
Barron	Jazz	Car

TABLE 9.1 MUSIC\_VEHICLE

Although MUSIC\_VEHICLE is in BCNF, data redundancies exist in the relation. For instance, if Kamath starts learning country music, to keep the relation instance consistent, we need to add the following three tuples to the relation:

Kamath	Country	Jeep
Kamath	Country	Truck
Kamath	Country	Van

If we add only one or two of the above tuples, the semantics of the relation will be altered. For example, if we add only the first tuple from the set above to MUSIC\_VEHICLE, it will mean that Kamath learns country music only when owning a jeep. In other words, the fact that **Music** and **Vehicle** are independent attributes will be compromised in MUSIC\_VEHICLE. Therefore, an insertion anomaly is present. Likewise, if Kamath does not own a jeep anymore, the following two tuples will have to be deleted in order to keep the relation instance consistent with the implied semantics:

Kamath	Jazz	Jeep
Kamath	Rock	Jeep

This amounts to a deletion anomaly.

In short, MUSIC\_VEHICLE is in BCNF and yet modification anomalies persist in the relation schema. It is important to note that these modification anomalies are not due to undesirable FDs. There are no undesirable FDs in MUSIC\_VEHICLE. In fact, all FDs in MUSIC\_VEHICLE are trivial dependencies, such as  $\{\{\text{Name}, \text{Music}, \text{Vehicle}\} \rightarrow \text{Name}; \{\text{Name}, \text{Music}, \text{Vehicle}\} \rightarrow \{\text{Music}, \text{Vehicle}\}\}$ . A close examination of the relation MUSIC\_VEHICLE reveals that for a given value of **Name** (e.g., Kamath), the same set of **Music** that Kamath learns (jazz, rock) occurs for each value of **Vehicle** that Kamath owns (jeep, truck, van). Similarly, for the attribute value McKinney for **Name**, the set of values, classical and rock for **Music**, that McKinney learns occurs for each value of **Vehicle**, van and car, that McKinney owns. This pattern is due to what is called a multi-valued dependency (MVD) and is a direct consequence of 1NF because 1NF does not permit an attribute to have more than one value in a tuple.

### 9.1.2 Multi-Valued Dependency Defined

Given  $R(X, Y, Z)$ , where  $X$ ,  $Y$ , and  $Z$  are atomic or composite attributes and  $Z = R - (X \cup Y)$ ,  $X \rightrightarrows Y$  means that for a given value of  $X$  (say,  $x_1$ ) in any relation state  $r$  of  $R$ , the same set of  $Y$  values ( $y_1, y_2, \dots, y_n$ ) occurs for each value of  $Z$  ( $z_1, z_2, z_3, \dots, z_m$ ). In other words, a multi-valued dependency (MVD)  $X \rightrightarrows Y$  holds in  $R(X, Y, Z)$  iff (if and only if), whenever  $(x, y, z)$  and  $(x, y', z')$  are tuples of any relation state  $r$  of  $R$ , then so are  $(x, y', z)$  and  $(x, y, z')$ . Observe that the implication here is that  $Y$  and  $Z$  are multi-valued attributes “independent” of each other in  $R$ .

How can we detect an MVD in a relation schema? In a relation schema  $R(X, Y, Z)$  that is in BCNF, MVD  $X \Rightarrow\! Y$  exists iff the natural join of  $r(R1)$  and  $r(R2)$ , where  $R1(X, Y)$  and  $R2(X, Z)$  are projections of  $R$ , strictly yields  $r(R)$  for every relation state  $r$  of  $R$ . Let us apply this test on the relation state of MUSIC\_VEHICLE that appears in Section 9.1.1. Suppose we want to test if the MVD **Name  $\Rightarrow\! Music$**  exists in MUSIC\_VEHICLE. Then, the projections whose natural join we want to compute and compare with MUSIC\_VEHICLE are:

**N\_MUSIC (Name, Music) and N\_VEHICLE (Name, Vehicle)**

For the convenience of the reader the relation state of MUSIC\_VEHICLE from Section 9.1.1 along with the two projections, N\_MUSIC and N\_VEHICLE, are shown in Figure 9.1. Observe that the natural join ( $N\_MUSIC * N\_VEHICLE$ ) strictly yields MUSIC\_VEHICLE. Therefore, we conclude that the MVD **Name  $\Rightarrow\! Music$**  exists in MUSIC\_VEHICLE. According to the inference rule of complementation discussed in Section 9.1.3, **Name  $\Rightarrow\! Vehicle$**  also exists in MUSIC\_VEHICLE. In other words, MVDs occur in pairs and are often shown that way, as in **Name  $\Rightarrow\! Music | Vehicle$** .

In essence, if a binary decomposition that has the lossless-join property exists for a relation state of a relation schema  $R$ , then:

- There are non-trivial MVDs in the target relation schema  $R$ .
- The MVDs can be inferred from the lossless-join decompositions as the intersection of the two projections multi-determines the differences between the two projections.

To reinforce understanding, let us test if MVDs **Music  $\Rightarrow\! Vehicle | Name$**  exist in MUSIC\_VEHICLE. In this case, the projections of MUSIC\_VEHICLE whose natural join we want to compute and compare with MUSIC\_VEHICLE are:

**M\_NAME (Name, Music) and M\_VEHICLE (Music, Vehicle)**

These two projections of MUSIC\_VEHICLE and the natural join of the projections are shown at the bottom of Figure 9.1. Observe that the natural join ( $M\_NAME * M\_VEHICLE$ ) does not strictly yield MUSIC\_VEHICLE. The shaded tuples in the natural join at the bottom of Figure 9.1 are not present in MUSIC\_VEHICLE. Therefore, we conclude that the MVDs **Music  $\Rightarrow\! Vehicle | Name$**  do not exist in MUSIC\_VEHICLE.

### 9.1.3 Inference Rules for Multi-Valued Dependencies

Similar to the three root inference rules for FDs (the reflexivity rule, the augmentation rule, and the transitivity rule; see Section 7.2.2), there are four inference rules that characterize MVDs.

MUSIC_VEHICLE		
Name	Music	Vehicle
Kamath	Jazz	Jeep
Kamath	Jazz	Truck
Kamath	Jazz	Van
Kamath	Rock	Truck
Kamath	Rock	Jeep
Kamath	Rock	Van
McKinney	Classical	Van
McKinney	Rock	Van
McKinney	Classical	Car
McKinney	Rock	Car
Barron	Jazz	Jeep
Barron	Jazz	Car

Test: Name  $\not\rightarrow$  Music | Vehicle

N_MUSIC			N_VEHICLE			N_MUSIC * N_VEHICLE		
Name	Music		Name	Vehicle		Name	Music	Vehicle
Kamath	Jazz		Kamath	Jeep		Kamath	Jazz	Jeep
Kamath	Rock		Kamath	Truck		Kamath	Jazz	Truck
McKinney	Classical		Kamath	Van		Kamath	Jazz	Van
McKinney	Rock		McKinney	Van		McKinney	Rock	Truck
Barron	Jazz		McKinney	Car		Kamath	Rock	Jeep
			Barron	Jeep		Kamath	Rock	Van
			Barron	Car		McKinney	Classical	Van

Test Rated:  
Name  $\not\rightarrow$  Music | Vehicle  
holds in MUSIC\_VEHICLE

Test: Music  $\not\rightarrow$  Vehicle | Name

M_NAME			M_VEHICLE		M_NAME * M_VEHICLE		
Name	Music		Music	Vehicle	Name	Music	Vehicle
Kamath	Jazz		Jazz	Jeep	Kamath	Jazz	Jeep
Kamath	Rock		Jazz	Truck	Kamath	Jazz	Truck
McKinney	Classical		Jazz	Van	Kamath	Jazz	Van
McKinney	Rock		Jazz	Car	Kamath	Rock	Truck
Barron	Jazz		Rock	Jeep	Kamath	Rock	Jeep
			Rock	Truck	Kamath	Rock	Van
			Rock	Van	McKinney	Classical	Van
			Rock	Car	McKinney	Rock	Van
			Classical	Van	McKinney	Classical	Car
			Classical	Car	McKinney	Rock	Car

Test Failed:  
Music  $\not\rightarrow$  Vehicle | Name  
Does not hold in  
MUSIC\_VEHICLE

© 2015 Cengage Learning®

**FIGURE 9.1** Test to check for the presence of multi-valued dependencies

Given  $R(A, B, C, D)$ , where  $A, B, C$ , and  $D$  are arbitrary subsets, atomic or composite, of the set of attributes of a relation schema,  $R$ , the following inference rules apply to MVDs:

- Reflexivity rule: If  $B \subset A$ , then  $A \Rightarrow B$
- Complementation rule: If  $A \Rightarrow B$ , then  $A \Rightarrow [R - (A \cup B)]$
- Augmentation rule: If  $A \Rightarrow B$ , and  $C \subset D$ , then  $(A, D) \Rightarrow (B, C)$
- Transitivity rule: If  $A \Rightarrow B$ , and  $B \Rightarrow C$ , then  $A \Rightarrow (C - B)$

A few other inference rules can be derived from these four (e.g., union rule, decomposition rule, pseudotransitivity rule). In addition, there is an inference rule that essentially bridges an FD and an MVD<sup>1</sup>:

- **Replication rule:** If  $A \rightarrow B$ , then  $A \Rightarrow B$

472

The replication rule indicates that an MVD is a generalization of an FD in the sense that every FD is an MVD. Note that the converse is not true. More precisely, an FD is an MVD in which the set of dependent values matching a specific determinant value is always a *singleton set*.

From the above set of rules for FDs and MVDs, it is possible to infer the complete set of FDs in  $F^+$  and the MVDs that hold in any relation state  $r$  of  $R$  that satisfies  $V$  (the set of specified MVDs). The closure of  $V$  is referred to as  $V^+$ .

Another useful rule derived by Catriel, Fagin, and Howard (1977) is<sup>2</sup>:

If  $A \Rightarrow B$ , and  $(A, B) \rightarrow C$ , then  $A \rightarrow (C - B)$

The property of symmetry in an MVD emerges directly from the complementation rule. Accordingly, if an MVD  $X \Rightarrow Y$  holds in the relation schema  $R(X, Y, Z)$ , then so does the MVD  $X \Rightarrow Z$ . This is often represented as  $X \Rightarrow Y \sqcup Z$ . The MVD  $X \Rightarrow Y$  in the relation schema  $R(X, Y, Z)$  is a *trivial* MVD if either (i)  $Y \subset X$  or (ii)  $(X \cup Y) = R$  because in either case the MVD does not convey any additional constraint (meaning). Trivial MVDs are usually removed from  $V^+$  without any consequence. An MVD that satisfies neither (i) nor (ii) is a *non-trivial* MVD and requires attention.

## 9.2 FOURTH NORMAL FORM (4NF)

In Section 9.1.1, we saw that a relation schema in BCNF can still contain data redundancies and associated modification anomalies. However, the data redundancies in this case are not due to functional dependencies. The source of the problem here is what is known as multi-valued dependencies. The only desirable MVD in a relation schema,  $R$ , is an MVD whose determinant is a superkey of  $R$ . In other words, non-trivial MVDs that are not also FDs are always undesirable because the very presence of these MVDs induces data redundancies in a relation schema, resulting in modification anomalies. This problem is dealt with by **fourth normal form (4NF)**.

<sup>1</sup>There is a second inference rule called the **coalescence rule** linking FD to MVD, which is a little less intuitive: if  $A \Rightarrow B$ , and (i)  $B$  and  $D$  are disjoint, (ii)  $D \rightarrow C$ , and (iii)  $C \subset B$ , then  $A \rightarrow C$ .

<sup>2</sup>This rule helps us understand 2NF violation as a special case of 4NF violation.

## DEFINITION

**4NF defined:** A relation schema  $R$  is in 4NF if there are no non-trivial multi-valued dependencies in  $R$ , or the determinant of any non-trivial multi-valued dependency in  $R$  is a superkey of  $R$ .<sup>3</sup>

For instance, let us evaluate the relation schema from Section 9.1.1 for 4NF:

$R$ : MUSIC\_VEHICLE (**Name**, **Music**, **Vehicle**)

As seen in Section 9.1.2,  $V^+$  contains the MVDs  $\text{Name} \rightrightarrows \text{Music} \mid \text{Vehicle}$ . Therefore, MUSIC\_VEHICLE violates 4NF. The resolution of 4NF violation is accomplished by the decomposition strategy:

- Replace the target relation schema ( $R$ ) by the projections ( $R_1$  and  $R_2$ ) that contain the determinant and dependent present in each of the two MVDs.

Accordingly, we have the decomposition:

**Solution 1**

$D \{R_1, R_2\}$

where:

$R_1$ : N\_MUSIC (**Name**, **Music**);  $R_2$ : N\_VEHICLE (**Name**, **Vehicle**)

$D \{R_1, R_2\}$  is in 4NF because both  $R_1$  and  $R_2$  are free of any non-trivial MVD.

Other decompositions are also possible:

**Solution 2**

$D \{R_{1a}, R_{2a}\}$

where:

$R_{1a}$ : N1\_MUSIC (**Name**, **Music**);  $R_{2a}$ : N2\_VEHICLE (**Music**, **Vehicle**)

Does this decomposition resolve the 4NF violation in MUSIC\_VEHICLE? Yes, because both  $R_{1a}$  and  $R_{2a}$  are free of any non-trivial MVD. The difference between Solutions 1 and 2 is that Solution 1 is a lossless-join decomposition while Solution 2 is a loss-join decomposition, as shown in Figure 9.1.

How do we detect loss/lossless-join decomposition in a 4NF resolution? With reference to MVDs that hold on a relation schema  $R$ , a decomposition  $D: \{R_1, R_2\}$  is a lossless-join decomposition iff  $V^+$  contains:

- either the MVD  $(R_1 \cap R_2) \rightrightarrows (R_1 - R_2)$
- or the MVD  $(R_1 \cap R_2) \rightrightarrows (R_2 - R_1)$

Accordingly, an examination of the two solutions will reveal that Solution 1 is a lossless-join decomposition while Solution 2 is not. Note that Solution 2 does not follow the decomposition method prescribed earlier. That method prescribed always yields a lossless-join decomposition.

It seems reasonable to make a semantic conclusion that **Music** and **Vehicle** are *independent* multi-valued attributes of MUSIC\_VEHICLE. Let us review another

<sup>3</sup>This is an informal definition of 4NF. More formally, a relation schema  $R$  is in 4NF if for every non-trivial MVD  $X \rightrightarrows Z$  in  $V^+$  in  $R$ ,  $X$  is a superkey of  $R$ . Equivalently, it can be stated that  $R$  is in 4NF if it is in BCNF and the dependents in all non-trivial MVDs in  $R$  are singleton sets—that is, the non-trivial MVDs are also FDs whose determinants are candidate keys of  $R$ .

example where the relationship between multi-valued attributes is not semantically obvious.

Suppose, based on user-specified business rules, MVDs  $\text{Name} \Rightarrow \text{Skill} | \text{Music}$  are specified over the relation schema R: MUSIC\_SKILL (**Name**, **Skill**, **Music**). Then it is clear that MUSIC\_SKILL is in violation of 4NF. MUSIC\_SKILL can be decomposed following the decomposition strategy stated earlier and results in a 4NF relational schema of the form:

$$D \{R1, R2\}$$

where:

$$R1: N\_MUSIC (\underline{\text{Name}}, \underline{\text{Music}}); R2: N\_SKILL (\underline{\text{Name}}, \underline{\text{Skill}})$$

First, observe that the solution (i.e., the projections R1 and R2) yields a lossless-join binary decomposition. Also, the solution suggests that **Music** and **Skill** are *independent* multi-valued attributes of MUSIC\_SKILL and that this “independence” is the cause of the MVDs.

What if **Music** and **Skill** are multi-valued attributes that are *not* independent of each other? Will the MVDs  $\text{Name} \Rightarrow \text{Skill} | \text{Music}$  persist in this case in MUSIC\_SKILL? Once again, to get a better understanding, let us review a relation instance of MUSIC\_SKILL, which appears at the top of Figure 9.2.

The semantics of this relation can be interpreted as: Pixoto, Hathi, and LaMott each have certain skills and each know certain types of music. It also appears that Ms. Pixoto is a composer of jazz, classical, and rock but works as a critic of *only* classical. Likewise, even though Mr. Hathi is a composer as well as a critic, and is also an exponent of classical and rock, he composes only classical and critiques only rock. Finally, Mr. LaMott is a composer—that is all he does, and since his interest is only jazz, he composes only jazz. Clearly, **Skill** and **Music** appear to be multi-valued attributes with respect to **Name** in a non-1NF schema. However, with (**Name**, **Skill**, **Music**) as the primary key, this schema is a relation schema in BCNF (i.e., no immediate violations of 1NF, 2NF, 3NF, or BCNF are present in MUSIC\_SKILL). Also, semantically it appears that **Skill** and **Music** are not independent (i.e., **Name's** relationship with **Skill** is not independent of the **Name's** relationship with **Music**). How do we confirm or refute this? Do the MVDs  $\text{Name} \Rightarrow \text{Skill} | \text{Music}$  exist in MUSIC\_SKILL? Applying the method prescribed in Section 9.1.2, we produce the projections (**Name**, **Skill**) and (**Name**, **Music**), join the two back, and verify if the natural join of the two strictly produces the target relation instance from which the projections emerged.

A comparison of MUSIC\_SKILL with (**N\_MUSIC** \* **N\_SKILL**) reveals that this is not the case (see Figure 9.2). So, we reject the premise that MVDs  $\text{Name} \Rightarrow \text{Skill} | \text{Music}$  exist in MUSIC\_SKILL. The next question is: Are there any other MVDs in MUSIC\_SKILL?

The only other possible MVD pairs in MUSIC\_SKILL are  $\text{Music} \Rightarrow \text{Skill} | \text{Name}$  and  $\text{Skill} \Rightarrow \text{Name} | \text{Music}$ .

MUSIC_SKILL		
Name	Skill	Music
Pixoto	Composer	Jazz
Pixoto	Composer	Classical
Pixoto	Composer	Rock
Pixoto	Critic	Classical
Hathi	Composer	Classical
Hathi	Critic	Rock
LaMott	Composer	Jazz

Three Possible Binary Projections

N_MUSIC	
Name	Music
Pixoto	Jazz
Pixoto	Rock
Pixoto	Classical
Hathi	Classical
Hathi	Rock
LaMott	Jazz

N_SKILL	
Name	Skill
Pixoto	Composer
Pixoto	Critic
Hathi	Composer
Hathi	Critic
LaMott	Composer

S_MUSIC	
Skill	Music
Composer	Jazz
Composer	Classical
Composer	Rock
Critic	Classical
Critic	Rock

Test: Name  $\Rightarrow$  Skill | Music      Test: Music  $\Rightarrow$  Skill | Name      Test: Skill  $\Rightarrow$  Name | Music

N_MUSIC * N_SKILL		
Name	Skill	Music
Pixoto	Composer	Jazz
Pixoto	Composer	Rock
Pixoto	Composer	Classical
Pixoto	Critic	Jazz
Pixoto	Critic	Rock
Pixoto	Critic	Classical
Hathi	Composer	Rock
Hathi	Composer	Classical
Hathi	Critic	Rock
Hathi	Critic	Classical
LaMott	Composer	Jazz

N_MUSIC * S_MUSIC		
Name	Skill	Music
Pixoto	Composer	Jazz
Pixoto	Composer	Classical
Pixoto	Composer	Rock
Pixoto	Critic	Classical
Pixoto	Critic	Rock
Hathi	Composer	Classical
Hathi	Composer	Rock
Hathi	Critic	Classical
Hathi	Critic	Rock
LaMott	Composer	Jazz

N_SKILL * S_MUSIC		
Name	Skill	Music
Pixoto	Composer	Jazz
Pixoto	Composer	Classical
Pixoto	Composer	Rock
Pixoto	Critic	Classical
Pixoto	Critic	Rock
Hathi	Composer	Jazz
Hathi	Composer	Classical
Hathi	Composer	Rock
Hathi	Critic	Classical
Hathi	Critic	Rock
LaMott	Composer	Jazz
LaMott	Composer	Classical
LaMott	Composer	Rock

Test Failed:  
Name  $\Rightarrow$  Skill | Music  
Does not hold in MUSIC\_SKILL

Test Failed:  
Music  $\Rightarrow$  Skill | Name  
does not hold in MUSIC\_SKILL

Test Failed:  
Skill  $\Rightarrow$  Name | Music  
does not hold in MUSIC\_SKILL

Inference: MUSIC\_SKILL is in 4NF

**FIGURE 9.2** Testing MUSIC\_SKILL for violation of 4NF

In order to verify these, we have to test if  $(N\_MUSIC * S\_MUSIC)$  or  $(N\_SKILL * S\_MUSIC)$  strictly yield  $MUSIC\_SKILL$ . An inspection of these two natural joins in Figure 9.2 reveals that neither is true. Thus, we conclude that there are no MVDs in  $MUSIC\_SKILL$ , even when multi-valued attributes are present. Consequently,  $MUSIC\_SKILL$  is in 4NF and does not require any decomposition—in fact, decomposition of  $MUSIC\_SKILL$  is incorrect. Note that there is no need to make an “intuitive” judgment about the independence among the multi-valued attributes in a relation schema for inferring the presence or absence of MVDs; it can be scientifically tested and formally inferred based on the test prescribed earlier in this section.

### 9.3 RESOLUTION OF A 4NF VIOLATION—A COMPREHENSIVE EXAMPLE

476

Consider the schema:

**MUSICIAN** (**Name**, **Age**, **Ph#**, **Band**, **Rate**, {**Music**}, {**Skill**}, {**Dependent**})  
In this schema, **Music**, **Skill**, and **Dependent** are multi-valued attributes.

The stated set of FDs and MVD that prevail over **MUSICIAN** are:

F {fd1, fd2, fd3, fd4} and V{mvd1}

where:

fd1: <b>Name</b> → <b>Age</b> ;	fd2: <b>Name</b> → <b>Ph#</b> ;
fd3: <b>Name</b> → <b>Band</b> ;	fd4: <b>Band</b> → <b>Rate</b> ;
mvd1: <b>Name</b> ⇒ <b>Dependent</b>	

By the rule of complementation, we can infer that mvd2: **Name** ⇒ {**Music**, **Skill**}.<sup>4</sup> **MUSICIAN** is not in 1NF because of the presence of multi-valued attributes; for this reason, **MUSICIAN** is not even a relation schema. In order to transform **MUSICIAN** to a 1NF relation schema, we specify (**Name**, **Music**, **Skill**, **Dependent**) as the primary key of **MUSICIAN**. Thus we have a 1NF relation schema:

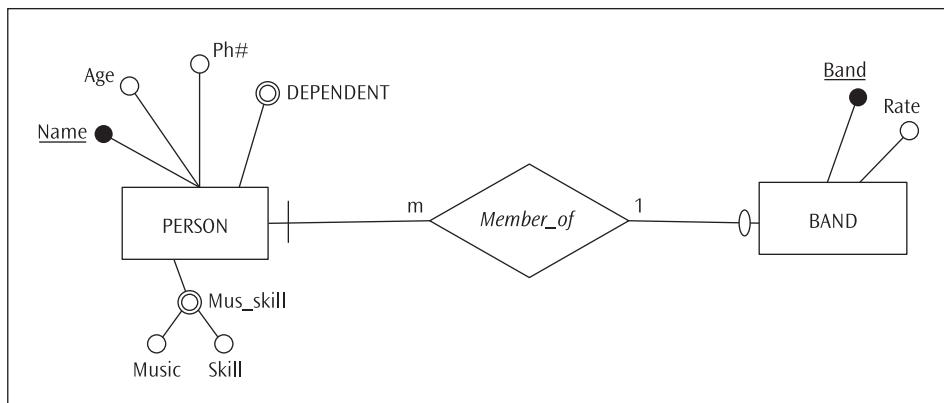
**MUSICIAN** (**Name**, **Age**, **Ph#**, **Band**, **Rate**, **Music**, **Skill**, **Dependent**)  
Eliminating normal form violations due to FDs yields the decomposition:  
R1: PERSON (**Name**, **Age**, **Ph#**, **Band**)  
R2: BAND (**Band**, **Rate**)  
R3: NMSD (**Name**, **Music**, **Skill**, **Dependent**)

R1 and R2 are in BCNF (in fact, in 4NF), and R3 violates 4NF because mvd1||mvd2 persists in R3. Solving R3 for 4NF violation yields:

R3a: FAMILY (**Name**, **Dependent**)  
R3b: NMS (**Name**, **Music**, **Skill**)

The Presentation Layer ERD reverse engineered from this solution enabling a deeper insight into the semantics of the scenario is shown in Figure 9.3.

<sup>4</sup>It is incorrect to conclude that **Name** ⇒ **Music**; and **Name** ⇒ **Skill**.

**FIGURE 9.3** Reverse engineered 4NF design scenario 1

Note: Participation constraints arbitrarily assumed

Suppose we add a few more attributes to the 1NF relation schema MUSICIAN, where the following FDs hold:

fd5: **Music → School;**

fd6: **Music → Year;**

fd7: **Skill → Department;**

fd8: **Dependent → Age;**

fd9: **Dependent → Gender;**

fd10: **Dependent → Relationship**

The revised 1NF relation schema will be of the form:

MUSICIAN (Name, Age, Ph#, Band, Rate, Music, Skill, Dependent, School, Year, Department, Age, Gender, Relationship)

Eliminating normal form violations due to the additional set of FDs, {fd5, fd6, fd7, fd8, fd9, fd10} in F, we have a 4NF design:

R1: PERSON (Name, Age, Ph#, Band);

R2: BAND (Band, Rate);

R3a: FAMILY (Name, Dependent);

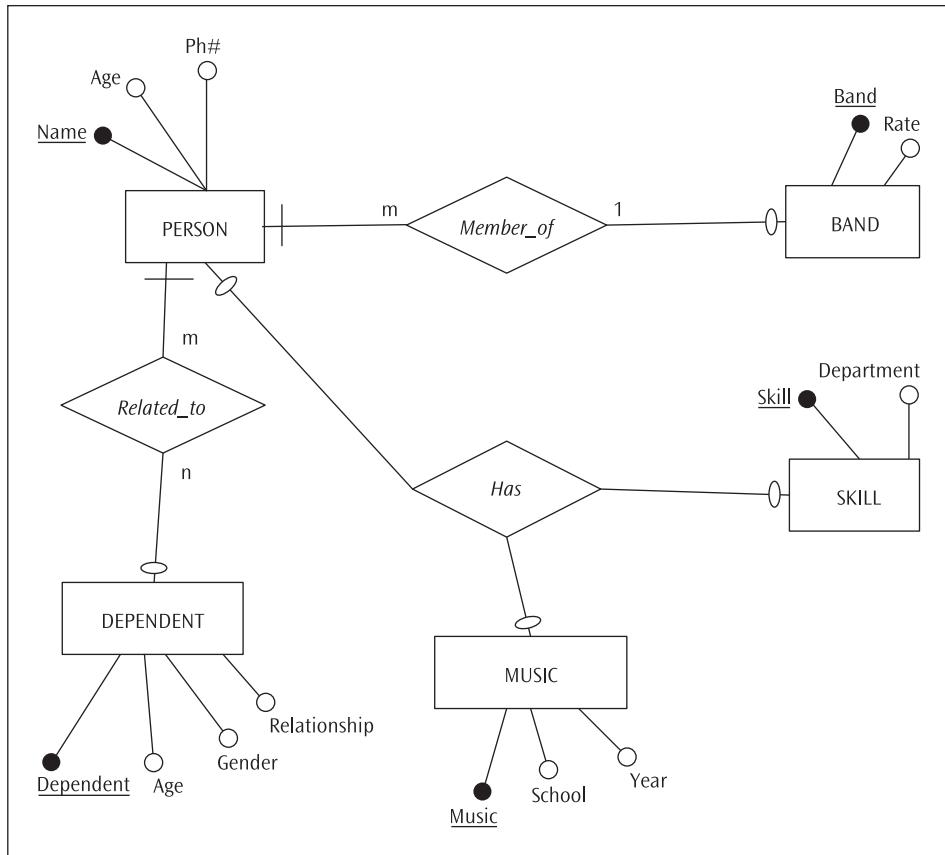
R3b: NMS (Name, Music, Skill);

R4: MUSIC (Music, School, Year);

R5: SKILL (Skill, Department);

R6: DEPENDENT (Dependent, Age, Gender, Relationship)

The Presentation Layer ERD reverse engineered from this revised solution appears in Figure 9.4.



**FIGURE 9.4** Reverse engineered 4NF design scenario 2  
Note: Participation constraints arbitrarily assumed

Comparing the two solutions along with the ERDs in Figures 9.3 and 9.4 provides interesting insights as to how identical relation schemas in the two solutions (R3a and R3b) depict somewhat different scenarios.

## 9.4 GENERALITY OF MULTI-VALUED DEPENDENCIES AND 4NF

Let us explore the following two assertions from Section 9.1.2 to uncover a broader theme underlying normalization that links the normal forms based on FDs (2NF, 3NF, and BCNF) with the normal form based on MVD (4NF):

- Assertion 1: If  $A \rightarrow B$ , then  $A \Rightarrow B$  (replication rule)
- Assertion 2: If a binary decomposition with the lossless-join property exists for a relation schema  $R$ , then non-trivial MVDs are present in  $R$ .

The replication rule simply states that an FD is a special case of an MVD since the FD satisfies the definition of an MVD. However, this occurs only under the special condition

that the dependent value is restricted to a *singleton set*—that is, for a given value of A, B has only one value.

To get a better grip on the concept, let us review the 2NF violation from this perspective using the example studied in Chapter 8 (see Sections 8.1.1 and 8.1.2). The non-1NF schema ALBUM (**Album\_no**, {**Artist\_nm**}, **Price**, **Stock**) with a multi-valued attribute, **Artist\_nm**, and an FD, **Album\_no** → **(Price, Stock)**, is first normalized to 1NF as:

NEW\_ALBUM (**Album\_no**, **Artist\_nm**, **Price**, **Stock**)

Since NEW\_ALBUM is in 1NF, it is a relation schema and the tenets of the relational theory can be applied to it. Does **Album\_no** ⇒ **Artist\_nm** in NEW\_ALBUM? This can be determined by comparing the natural join (ALBUM\_ARTIST \* ALBUM\_INFO) with NEW\_ALBUM, where:

D: ALBUM\_INFO (**Album\_no**, **Price**, **Stock**); and ALBUM\_ARTIST (**Album\_no**, **Artist\_nm**) represents a binary decomposition NEW\_ALBUM.

Figure 8.2 contains the relation instances pertaining to this example. By computing the natural join, (ALBUM\_ARTIST \* ALBUM\_INFO), it can be seen that the natural join indeed strictly yields the relation NEW\_ALBUM. Therefore, it is concluded that **Album\_no** ⇒ **Artist\_nm**. Then, by the rule of complementation, we have **Album\_no** ⇒ **(Price, Stock)**. In other words, for each value of **Album\_no** in NEW\_ALBUM, the same set of **(Price, Stock)** occurs for each value of **Artist\_nm**. However, **(Price, Stock)** is a singleton set. Because **(Price, Stock)** is a singleton set, the MVD, **Album\_no** ⇒ **(Price, Stock)**, is equivalent to the FD, **Album\_no** → **(Price, Stock)**, and that is why the primary key for ALBUM\_INFO is **Album\_no** instead of **(Album\_no, Price, Stock)**. Thus, it is seen that a 2NF violation is a special case of 4NF violation—that is, 4NF subsumes 2NF. The fact that the binary decomposition of NEW\_ALBUM yields a lossless-join decomposition signals the presence of MVD in NEW\_ALBUM, thus ratifying Assertion 2 above. The inference rule derived by Catriel, Fagin, and Howard (see Section 9.1.3) essentially conveys the same idea.

In a similar fashion, it can be shown that a 3NF violation and BCNF violation are also special cases of 4NF violation. The mere existence of a lossless-join decomposition of the target relation schema that violates 3NF or BCNF in itself signals the presence of an MVD in the relation schema, and that is how a 3NF and a BCNF violation can be seen as a 4NF violation. To gain deeper insight, the reader is encouraged to explore this further using the examples from Chapter 8 (see Sections 8.1.3 and 8.1.4). Identifying the MVDs that prevail in these cases is an interesting exercise to pursue. In short, a relation schema in 4NF also is in 3NF and BCNF.

Here is a summary of the salient properties of multi-valued dependencies and 4NF:

- A multi-valued attribute (MVA) can cause multi-valued dependencies (MVD) in a relation schema R.
- A MVA is necessary to cause MVD in a relation schema R.
- When MVAs cause MVDs, modification anomalies will be present in R if the determinant of any MVD present in R is not a super (candidate) key of R, even though there are no undesirable functional dependencies (FDs) in R.
- When MVAs don't cause MVDs in R, R is in 4NF.
- Independent MVAs in a relation schema cause MVDs and hence immediate violation of 4NF.
- MVAs in a relation schema dependent on each other do not cause MVDs and hence no immediate violation of 4NF.

## 9.5 JOIN-DEPENDENCIES AND FIFTH NORMAL FORM (5NF)

So far, we were able to resolve normal form violations due to undesirable FDs and MVDs through binary decompositions of the affected relation schemas. Join dependencies (JDs) pertain to conditions where *binary projections* are not sufficient to achieve a lossless-join decomposition and the associated issues.

### DEFINITION

**Join dependency (JD) defined:**  $JD(R_1, R_2, R_3, \dots, R_n)$  specified over a relation schema,  $R$ , states that every legal state  $r$  of  $R$  has a lossless-join decomposition into  $R_1, R_2, R_3, \dots, R_n$ . In other words, given  $(R_1, R_2, R_3, \dots, R_n)$  as projections of  $R$ , if the natural join of  $(R_1, R_2, R_3, \dots, R_n)$  produces every legal state  $r$  of  $R$ , then  $JD(R_1, R_2, R_3, \dots, R_n)$  exists in  $R$ . A  $JD(R_1, R_2, R_3, \dots, R_n)$  is trivial if one of the projections,  $R_i$ , is equal to  $R$  because this condition guarantees lossless-join property for any state  $r$  of  $R$  and essentially does not specify any constraint at all on  $R$ . Observe that given  $(R_1, R_2)$  as projections of  $R$ , the MVDs  $(R_1 \cap R_2) \Rightarrow (R_1 - R_2) | (R_2 - R_1)$  are equivalent to  $JD(R_1, R_2)$ . Thus, MVDs are essentially binary join dependencies possessing a number of algebraic properties similar to those of FDs. Since an FD can be seen as a special case of MVD (i.e., an MVD subsumes an FD), an FD is also subsumed by a JD. Therefore, it can be said that JD is the most general form of dependency constraint that deals with decomposition via projections and re-compositions via natural joins.

480

**Fifth normal form (5NF)** is about JDs. In simple terms, the presence of JDs in a relation schema  $R$  violates 5NF—that is, if a relation state  $r$  of  $R$  can be strictly reconstructed from the natural join of all of its projections,  $(R_1, R_2, R_3, \dots, R_n)$ , then a join-dependency is present in  $R$  (i.e., a constraint specifying JD has been imposed on  $R$ ) and  $R$  is not in 5NF. In order to establish 5NF,  $R$  should be replaced by its decomposition  $(R_1, R_2, R_3, \dots, R_n)$ . The set of relation schemas  $(R_1, R_2, R_3, \dots, R_n)$  in this case is in 5NF.

### DEFINITION

**5NF defined:** A relation schema  $R$  is in 5NF if there are no non-trivial join dependencies in  $R$ .<sup>5</sup> A relation schema that cannot be reconstructed by a natural join of all its projections does not have a JD imposed on it and so is already in 5NF and should not be decomposed to achieve 5NF.

Let us now review an illustration that clarifies the concept. SCHEDEULE\_X at the top of Figure 9.5 is the relation instance that remains after all the normal form violations due to functional dependencies have been resolved and decomposed from the original relation schema; it is represented as:

R: SCHEDEULE\_X (Prof\_name, Course#, Quarter)

<sup>5</sup>A technically more precise definition is:  $R$  is in 5NF if, for every non-trivial join-dependency,  $JD(R_1, R_2, \dots, R_n)$ , every  $R_i$  is a superkey of  $R$  (Elmasri and Navathe, 2010). A comprehensive (highly technical) discussion of 4NF and 5NF can be found in Johnson (1997).

**Business Rule:** If an instructor teaches a course, and that course is offered in certain quarters, then the instructor must teach that course in those quarters if s/he is teaching during those quarters.

R: SCHEDULE\_X (Prof\_name, Course#, Quarter); R1x: TAUGHT\_X (Prof\_name, Course#);  
R2x: TAUGHT\_DURING\_X (Prof\_name, Quarter); R3x: COURSE\_OFFERING\_X (Course#, Quarter)

SCHEDULE_X			
Prof_name	Course#		Quarter
Verstrate	IS812		Fall
Verstrate	IS812		Spring
Verstrate	IS832		Winter
Verstrate	IS330		Fall
Verstrate	IS330		Spring
Surendra	IS812		Fall
Surendra	IS812		Spring
Surendra	IS430		Fall
Surendra	IS430		Spring
Kim	IS821		Winter
Kim	IS430		Spring
Kim	IS430		Summer

1. Does the natural join of any two of the three tables below strictly yield the table above? The answer is "No." Therefore, SCHEDULE\_X is in 4NF.

TAUGHT\_X

Prof_name	Course#
Verstrate	IS812
Verstrate	IS832
Verstrate	IS330
Surendra	IS812
Surendra	IS430
Kim	IS821
Kim	IS430

TAUGHT\_DURING\_X

Prof_name	Quarter
Verstrate	Fall
Verstrate	Winter
Verstrate	Spring
Surendra	Fall
Surendra	Spring
Kim	Winter
Kim	Spring
Kim	Summer

COURSE\_OFFERING\_X

Course#	Quarter
IS330	Fall
IS330	Spring
IS430	Fall
IS430	Spring
IS430	Summer
IS812	Fall
IS812	Spring
IS821	Winter
IS832	Winter

2. Does the natural join of all the three tables above strictly yield SCHEDULE\_X above? The answer is "Yes," indicating presence of join-dependencies. Therefore, SCHEDULE\_X violates 5NF. In order to achieve 5NF in the design, SCHEDULE\_X must be replaced by the set of relation schemas {TAUGHT\_X, TAUGHT\_DURING\_X, COURSE\_OFFERING\_X}.

**FIGURE 9.5** An illustration of 5NF violation and resolution

There are no MVDs specified on SCHEDE\_X. In fact, we are able to verify this from the relation instance in Figure 9.5. The three possible decompositions that can suggest presence of non-trivial MVDs in SCHEDE\_X are:

- R1x:TAUGHT\_X (**Prof\_name**, **Course#**);
- R2x:TAUGHT\_DURING\_X (**Prof\_name**, **Quarter**)
- R3x:COURSE\_OFFERING\_X (**Course#**, **Quarter**)

If a natural join of any binary decomposition of R—namely, (R1x \* R2x) or (R1x \* R3x) or (R2x \* R3x)—is sufficient to strictly reconstruct R (non-loss composition), then the presence of MVD is evidenced. When verified with the relation instances in Figure 9.5, the reader will find that none of these natural joins strictly yields R. Then, there are no MVDs in R implying that R (i.e., SCHEDE\_X) is in 4NF. Is SCHEDE\_X in 5NF? In order to check this, we need to know if *JD* (R1x, R2x, R3x) exists. If the natural join of {R1x, R2x, R3x} strictly results in R, then we can conclude that *JD* (R1x, R2x, R3x) persists. Then, R (i.e., SCHEDE\_X) violates 5NF. When verified using the relation instances in Figure 9.5, the reader will find that a natural join of {R1x, R2x, R3x} indeed yields strictly R, meaning that *JD* (R1x, R2x, R3x) is present. Therefore, SCHEDE\_X violates 5NF. In order to restore the design to 5NF, SCHEDE\_X should be replaced by the set of relation schemas (TAUGHT\_X, TAUGHT\_DURING\_X, COURSE\_OFFERING\_X).

Let us now review the relation instance SCHEDE\_Y in Figure 9.6. The relation schema that represents this relation instance is:

- R: SCHEDE\_Y (**Prof\_name**, **Course#**, **Quarter**)

SCHEDE\_Y is in 4NF. The verification of this claim is left as an exercise to the reader. Is SCHEDE\_Y in 5NF? In order to check this, we need to know if *JD* (R1y, R2y, R3y) exists, where:

- R1y: TAUGHT\_Y (**Prof\_name**, **Course#**);
- R2y: TAUGHT\_DURING\_Y (**Prof\_name**, **Quarter**)
- R3y: COURSE\_OFFERING\_Y (**Course#**, **Quarter**)

Once again, if the natural join of {R1Y, R2Y, R3Y} strictly results in R, then we can conclude that *JD* (R1Y, R2Y, R3Y) persists in R. Then, R (i.e., SCHEDE\_Y) violates 5NF. When verified using the relation instances in Figure 9.6, we find that a natural join of {R1Y, R2Y, R3Y} does not strictly yield R, meaning that *JD* (R1Y, R2Y, R3Y) is not present in R. Therefore, SCHEDE\_Y does not violate 5NF—that is, SCHEDE\_Y is in 5NF. Replacing SCHEDE\_Y by the set of relation schemas (TAUGHT\_Y, TAUGHT\_DURING\_Y, COURSE\_OFFERING\_Y) will change the intended semantics of the design and amounts to an erroneous decomposition. Table 9.2 is provided as an aid to a better understanding of 5NF through a comparative review of SCHEDE\_X (Figure 9.5) and SCHEDE\_Y (Figure 9.6).

**Business Rule:** If an instructor teaches a course, and that course is offered in certain quarters, the instructor need not teach that course in those quarters just because s/he is teaching during those quarters.

- R: SCHEDULE\_Y (Prof\_name, Course#, Quarter); R1y: TAUGHT\_X (Prof\_name, Course#);  
 R2y: TAUGHT\_DURING\_X (Prof\_name, Quarter); R3y: COURSE\_OFFERING\_X (Course#, Quarter)

SCHEDULE\_Y

Prof_name	Course#	Quarter
Verstrate	IS812	Fall
Verstrate	IS812	Spring
Verstrate	IS832	Winter
Verstrate	IS330	Fall
Verstrate	IS330	Spring
Surendra	IS812	Fall
Surendra	IS812	Spring
Surendra	IS430	Fall
Surendra	IS430	Spring
Kim	IS821	Winter
Kim	IS430	Spring
Kim	IS430	Summer

3. Does the natural join of any two of the three tables below strictly yield the table above? The answer is "No." Therefore, SCHEDULE\_Y is in 4NF

TAUGHT\_Y

Prof_name	Course#
Verstrate	IS812
Verstrate	IS832
Verstrate	IS330
Surendra	IS812
Surendra	IS430
Kim	IS821
Kim	IS430

TAUGHT\_DURING\_Y

Prof_name	Quarter
Verstrate	Fall
Verstrate	Winter
Verstrate	Spring
Surendra	Fall
Surendra	Spring
Kim	Winter
Kim	Spring
Kim	Summer

COURSE\_OFFERING\_Y

Course#	Quarter
IS330	Fall
IS330	Spring
IS430	Fall
IS430	Spring
IS430	Summer
IS812	Fall
IS812	Spring
IS821	Winter
IS832	Winter

4. Does the natural join of all the three tables above strictly yield SCHEDULE\_Y above? The answer is "No," indicating absence of join-dependencies. Therefore, SCHEDULE\_Y is in 5NF. Replacing SCHEDULE\_Y by the set of relation schemas {TAUGHT\_Y, TAUGHT\_DURING\_Y, COURSE\_OFFERING\_Y} changes the intended semantics of the business rule stated above.

**FIGURE 9.6** An illustration of a relation schema in 5NF

Question	SCHEDULE_X	SCHEDULE_Y
Who taught IS812 in Winter	Nobody	Nobody
Who taught IS812 in Fall	Verstrate, Surendra	Surendra
Who taught IS812 in Spring	Verstrate, Surendra	Verstrate
Who taught IS430 in Fall	Surendra	Surendra
Who taught IS430 in Spring	Surendra, Kim	Surendra, Kim
Who taught IS430 in Summer	Kim	Kim
Notes	<p>Since Verstrate taught IS812 and he taught during Fall, Winter, and Spring, he taught IS812 if it was offered in any of these three quarters—i.e., Verstrate taught IS812 in Fall and Spring. <i>He didn't teach IS812 in Winter because IS812 was not offered in Winter, even though he taught in Winter.</i></p> <p>Since Surendra taught IS812 and he taught during Fall and Spring, he taught IS812 if it was offered in any of these two quarters—i.e., Surendra taught IS812 in both Fall and Spring.</p>	<p>Verstrate taught IS812 and Verstrate taught during Fall, Winter, and Spring. But, he taught IS812 <i>only in Spring</i>—i.e. even though IS812 was offered in Fall and Verstrate taught in Fall, he didn't teach IS812 in Fall. In short, Verstrate didn't teach IS812 whenever it was offered while he was teaching.</p> <p>Surendra taught IS812 and Surendra taught during Fall and Spring. But, he taught IS812 <i>only in Fall</i>—i.e. even though IS812 was offered in Spring and Surendra taught in Spring, he didn't teach IS812 in Spring. In short, Surendra didn't teach IS812 whenever it was offered while he was teaching.</p>
	No MVDs—in 4NF But <i>only</i> in 4NF; not in 5NF	No MVDs In 4NF and in 5NF
	<p>SCHEDULE_X can be reconstructed by joining the three of its projections TAUGHT_X, TAUGHT_DURING_X, and COURSE_OFFERING_X</p> <p>Therefore, join dependency is <b>present</b> in SCHEDULE_X.</p>	<p>SCHEDULE_Y <u>cannot</u> be reconstructed by joining the three of its projections TAUGHT_Y, TAUGHT_DURING_Y, and COURSE_OFFERING_Y</p> <p>Therefore, join dependency is <b>absent</b> in SCHEDULE_Y.</p>
	SCHEDULE_X cannot be reconstructed by joining any two of its three projections. That is why there are no MVDs in SCHEDULE_X and therefore it is in 4NF.	SCHEDULE_Y cannot be reconstructed by joining any two of its three projections. That is why there are no MVDs in SCHEDULE_Y and therefore it is in 4NF.

**TABLE 9.2** A comparative analysis of the presence and absence of 5NF violation

The relation instance MUSIC\_SKILL in Figure 9.2 is in 4NF. Is it also in 5NF? If not, what can be done to this relation instance to establish 5NF in MUSIC\_SKILL? On the other hand, if MUSIC\_SKILL is in 5NF, how can the relation instance be altered to depict a 4NF relation that violates 5NF? The reader should find this an interesting exercise.

A summary of the salient properties of join-dependencies (JD) and Project-Join Normal Form (PJNF, also termed 5NF) along with a clarification of how JD subsumes MVD and therefore FD and thus represents the general form of all dependencies predicated upon project and join relational algebra operations is presented here:

- A JD in a relation schema R pertains to conditions where the natural join of any proper subset of its projections results in the strict reconstruction of R.
- Therefore, a relation schema R that cannot be reconstructed by a natural join of any proper subset of its projections does not have a JD.
- A JD is trivial if the set of projections includes R.
- A JD ( $R_1, R_2, R_3, \dots, R_n$ ) specified over a relation schema R states that every legal state  $r$  of R has a lossless-join decomposition into  $R_1, R_2, R_3, \dots, R_n$ .
- Given  $(R_1, R_2, R_3, \dots, R_n)$  as a decomposition of R, if the natural join of  $(R_1, R_2, R_3, \dots, R_n)$  produces every legal state  $r$  of R, then JD  $(R_1, R_2, R_3, \dots, R_n)$  exists in R.
- A relation schema R is in 5NF if there are no non-trivial join-dependencies in R.
- Alternatively, R is in 5NF if for every non-trivial join-dependency, JD  $(R_1, R_2, \dots, R_n)$ , every  $R_i$  is a superkey of R.
- Presence of non-trivial join-dependency in a relation schema R violates 5NF in R except when the projections are superkeys of R.
- **5NF Solution:** If R has a non-trivial join-dependency, replace R with the appropriate proper subset of projections  $[R_1, R_2, R_3, \dots, R_n]$ .
- An MVD is a special case of JD where the decomposition is binary—that is, JD  $(R_1, R_2)$  in R is equivalent to MVD,  $(R_1 \cap R_2) \rightarrow\!\!\!> (R_1 - R_2) \sqcup (R_2 - R_1)$ .
- MVDs are essentially binary join-dependencies possessing a number of algebraic properties similar to those of FDs.
- Since an FD can be seen as a special case of MVD (i.e., an MVD subsumes an FD), an FD is also subsumed by a JD.
- JD is the most general form of dependency constraint that deals with decomposition through projections and re-compositions using natural joins.

485

While FDs portray binary relationships among a set of entity types, MVDs and JD pertain to ternary and n-ary relationship types, respectively. The salient characteristics of the relationship type intrinsic to the 4NF and 5NF relations are presented here:

- Erroneous decomposition of a relation schema R that is in 4NF renders a genuinely ternary relationship into two incorrect binary relationships.
- When a binary relationship between  $R_1 \& R_2$  and  $R_1 \& R_3$  is erroneously expressed as a ternary relationship among  $R_1, R_2$ , and  $R_3$ , 4NF is violated. A ternary relationship that can be reconstructed from any two of its binary projections should not be modeled as a ternary relationship.

- A ternary relationship that can be reconstructed by joining all three of its projections of degree two (but not just any two) is in 4NF but violates 5NF because join-dependency persists.
- A ternary relationship that cannot be reconstructed by joining all three of its projections of degree two is not only in 4NF, it is in 5NF, because join-dependency does not exist.
- A n-ary relationship that cannot be reconstructed from any combination of its projections is in 5NF (therefore, in 4NF also), because join-dependency does not exist.
  - A n-ary relationship that can be reconstructed from some combination of its projections violates 5NF, because join-dependency persists. To attain a 5NF design, replace the n-ary relationship with the same set of projections that produce the non-loss join—no more, no less.

Let us now revisit the two examples we have seen so far (Figures 9.5 and 9.6). They essentially indicate that either SCHEDELE\_Y, representing a ternary relationship type, or the decomposition {TAUGHT\_X, TAUGHT\_DURING\_X, COURSE\_OFFERING\_X} of SCHEDELE\_X, representing multiple binary relationships among the participating entity types, can exist if 5NF has to prevail. This gives an impression that simultaneous existence of both the ternary relationship type and multiple binary relationship types among the participating entity types is prohibited if 5NF condition has to be satisfied.

In Figure 9.7, the relation OFFERS is a direct replication of SCHEDELE\_Y; therefore, OFFERS is in 5NF, meaning that the three possible binary projections of OFFERS do not losslessly reconstruct OFFERS. Now, let us review the set of relations {CAN\_TEACH, AVAILABLE\_TO\_TEACH, COURSE\_OFFERING}. The first thing to observe here is that this is not a set of binary projections decomposed from OFFERS. A quick comparison of this set of binary relations with the set of binary projections decomposed from OFFERS, viz., {TAUGHT\_Y, TAUGHT\_DURING\_Y, COURSE\_OFFERING\_Y} (see Figure 9.6), will clarify this.

Clearly, the set of relations {CAN\_TEACH, AVAILABLE\_TO\_TEACH, COURSE\_OFFERING} in Figure 9.7 is semantically unrelated to the ternary relationship expressed by OFFERS. The naming of the three relations is an attempt to express the semantic difference. For instance, while Seligman does not offer any course in any quarter, the fact that she can teach the course IS430 is certainly of value independent of the facts about who offers what courses during which quarters and the like. Likewise, the fact that Barron is available to teach during Fall and Winter quarters adds independent value beyond the facts captured by the relation OFFERS. In short, simultaneous presence of relevant binary relationships in addition to a ternary relationship among the same participating entity types is not only possible but can contribute to the semantic richness of a data model and may even be expected in the users' requirement specifications.

<b>OFFERS</b>		
<b>Prof_name</b>	<b>Course#</b>	<b>Quarter</b>
Verstrate	IS812	Spring
Verstrate	IS832	Winter
Verstrate	IS330	Fall
Verstrate	IS330	Spring
Surendra	IS812	Fall
Surendra	IS430	Fall
Surendra	IS430	Spring
Kim	IS821	Winter
Kim	IS430	Spring
Kim	IS430	Summer

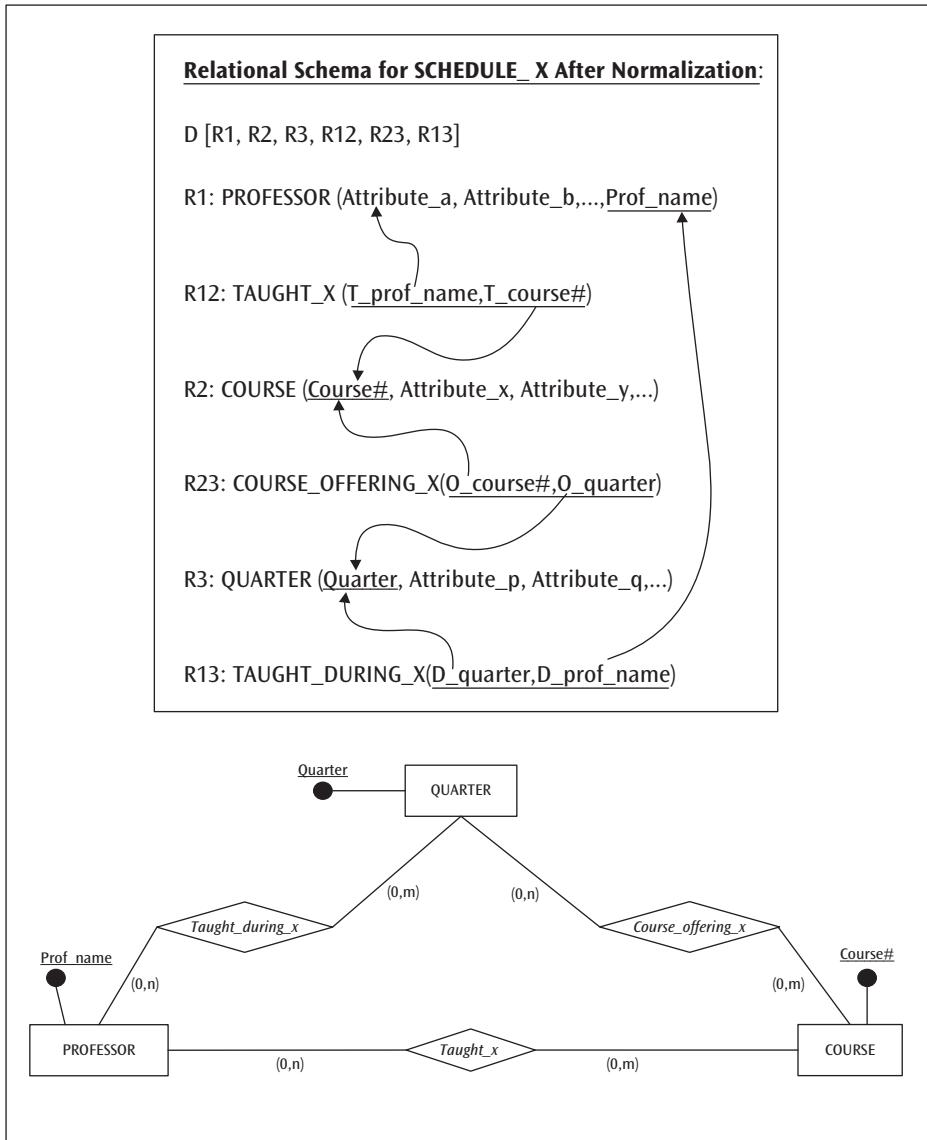
<b>CAN_TEACH</b>		<b>AVAILABLE_TO_TEACH</b>	
<b>Prof_name</b>	<b>Course#</b>	<b>Prof_name</b>	<b>Quarter</b>
Verstrate	IS812	Verstrate	Fall
Verstrate	IS832	Verstrate	Winter
Verstrate	IS330	Verstrate	Spring
Surendra	IS812	Surendra	Fall
Surendra	IS430	Surendra	Spring
<i>Kim</i>	<b>IS812</b>	<i>Barron</i>	<b>Fall</b>
Kim	IS821	<i>Barron</i>	<b>Winter</b>
Kim	IS430	Kim	Winter
<i>Seligman</i>	<b>IS430</b>	Kim	Spring
		Kim	Summer

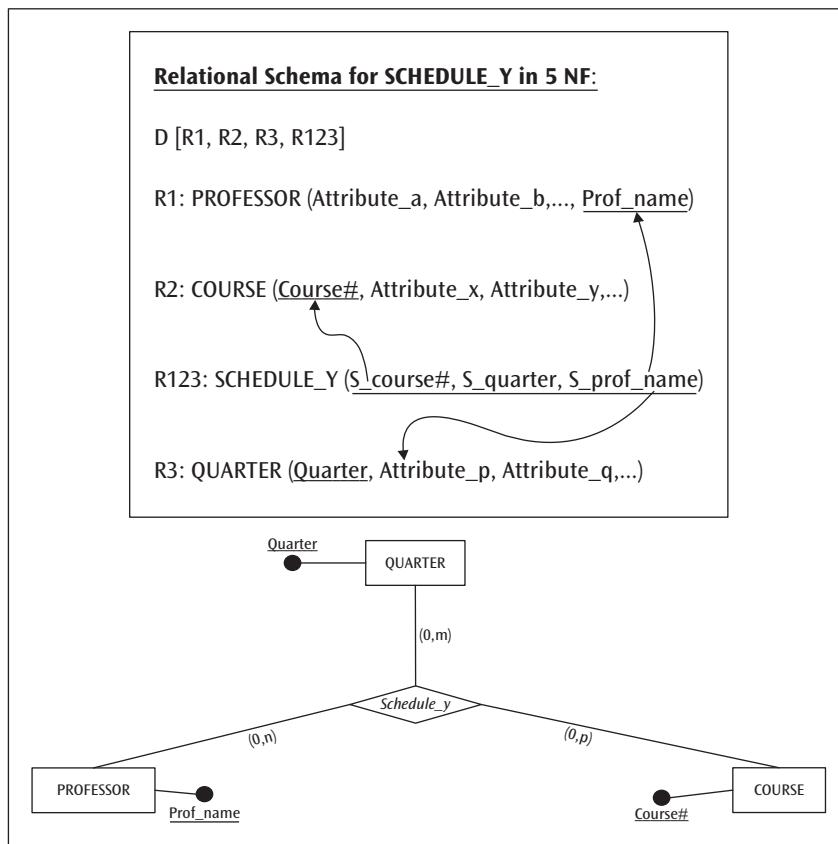
  

<b>COURSE_OFFERING</b>	
<b>Course#</b>	<b>Quarter</b>
IS330	Fall
IS330	Spring
<i>IS330</i>	<b>Summer</b>
<b>IS340</b>	<b>Winter</b>
IS430	Fall
IS430	Spring
IS430	Summer
IS812	Fall
IS812	Spring
IS821	Winter
IS832	Winter

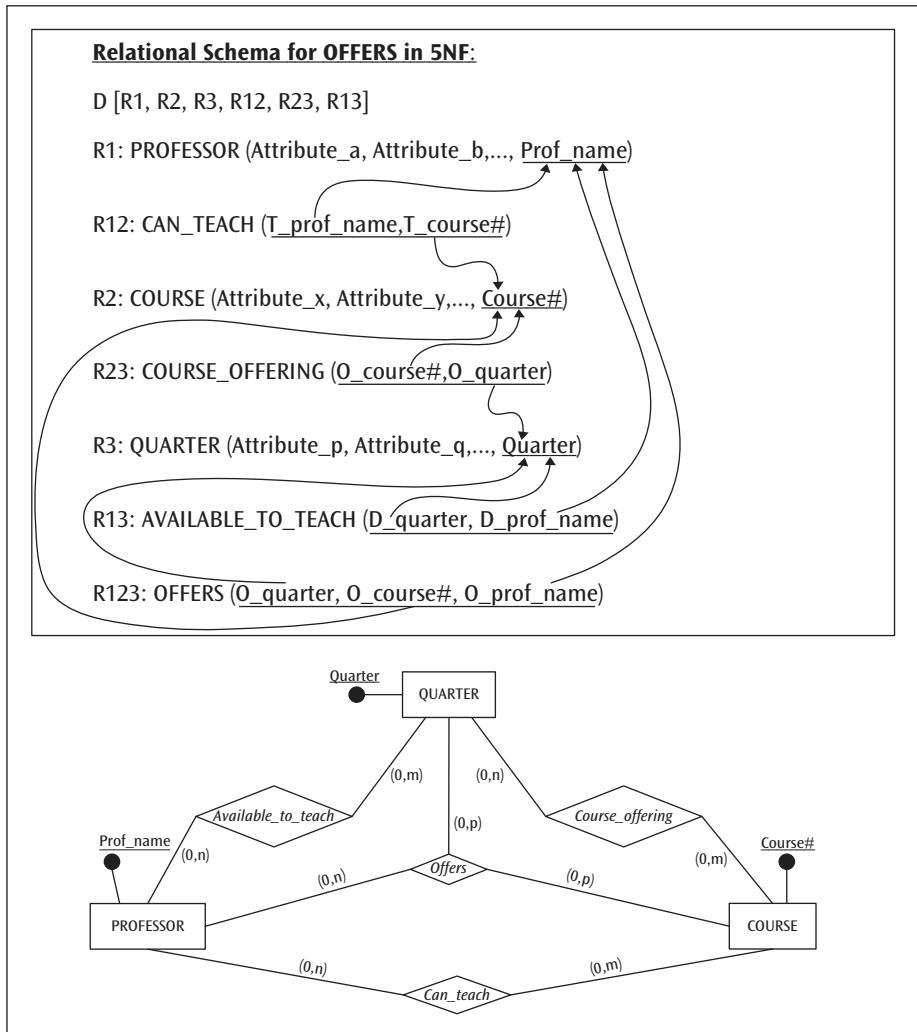
**FIGURE 9.7** A dataset exemplifying co-existing ternary and binary relationship types

Figures 9.8a, 9.8b, and 9.8c, which present relational schema and ERDs reverse engineered from them for the relations SCHEDELE\_X, SCHEDELE\_Y, and OFFERS, respectively, ought to complete the picture for the reader.

**FIGURE 9.8a** Relational Schema and ERD to model SCHEDULE\_X



**FIGURE 9.8b** Relational Schema and ERD to model SCHEDULE\_Y



**FIGURE 9.8c** Relational Schema and ERD to model co-existing ternary and binary relationships

## 9.6 A THOUGHT-PROVOKING EXEMPLAR

At this point, having covered ER modeling and normalization in significant depth, we are ready to examine an intriguing exemplar that at first may appear self-contradicting. We will evaluate the alternative solutions and discuss the consequences.

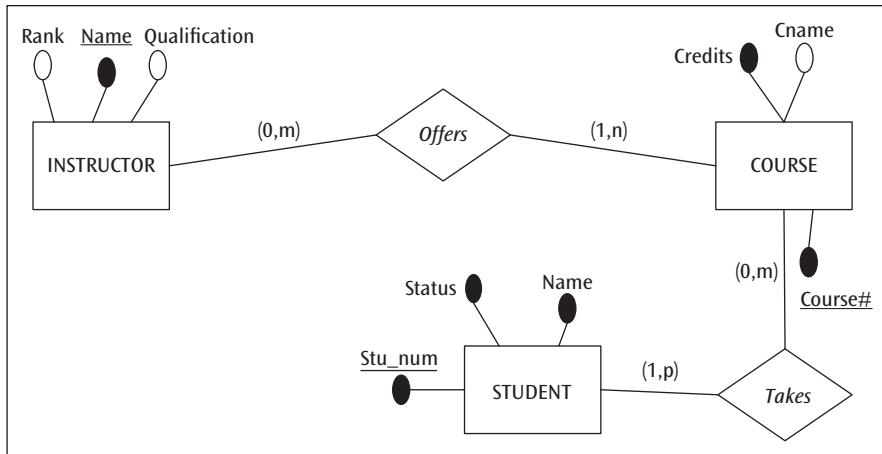
The ERD in Figure 9.9a is aimed at conveying the following business rules:

Every course is offered by some instructor—sometimes, by more than one instructor.

An instructor may offer several courses, or an instructor need not offer

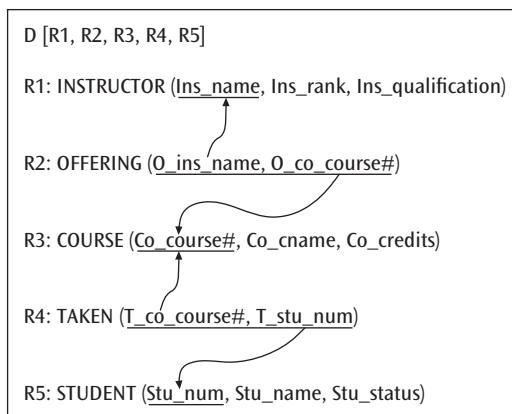
Every student takes several courses—that is, takes at least one course.

Every course may be taken by many students; also, a course may have no students.

**FIGURE 9.9a** ERD for an intriguing exemplar

These semantics are conveyed with the use of two m:n relationships—one between COURSE and INSTRUCTOR and the other between COURSE and STUDENT—along with the appropriate participation constraints.

Prior to mapping this ERD to the logical tier, the m:n relationship types will have to be decomposed to gerund entity types. We ought to be able to bypass this step and proceed to map the ERD to the logical tier. Thus, the relational schema will contain one relation schema for each of the three entity types, INSTRUCTOR, COURSE and STUDENT. The two relationship types, *Offers* and *Takes*, that capture m:n relationships will resolve to gerund entity types in the Design-Specific ER diagram and appear as relation schemas OFFERING and TAKEN bridging the relation schemas COURSE and INSTRUCTOR and the relation schemas COURSE and STUDENT, respectively. The relational schema mapped from the ERD in Figure 9.9a is displayed in Figure 9.9b as D [R1, R2, R3, R4, R5].

**FIGURE 9.9b** Relational schema for the ERD in Figure 9.9a

Here, two observations are worthwhile:

While the mapping yields a dependency-preserving BCNF solution, there is a loss join between the relations OFFERING and TAKEN.

However, there is a non-loss navigation path available in this solution, an informed compromise deserving consideration.

That said, let us next derive the FDs reflected in D and specify the canonical cover of these FDs in  $D \mid F$  as  $F_c$ . The  $F_c$  specified here should also prevail over R, the universal relation schema (URS) derived through the natural join of R1, R2, R3, R4, and R5. The URS and the  $F_c$  prevailing over it are shown in Figure 9.9c.

$F_c [fd1, fd2, fd3, fd4, fd5, fd6]$ <b><i>derived from</i></b> D [R1, R2, R3, R4, R5] where	
fd1: Ins_name → Ins_rank;	fd2: Ins_name → Ins_qualification;
fd3: Co_course# → Co_cname;	fd4: Co_course# → Co_credits;
fd5: Stu_num → Stu_name;	fd6: Stu_num → Stu_status.
Universal Relation Schema over which $F_c$ prevails:	
URS (Ins_name, Ins_rank, Ins_qualification, O_ins_name, O_co_course#, Semester, Co_course#, Co_cname, Co_credits, T_co_course#, T_stu_num, Grade, Stu_num, Stu_name, Stu_status)	

**FIGURE 9.9c** FDs derived from the relational schema in Figure 9.9b and the associated URS

Now that we have a URS and a set of FDs prevailing over it, we should be able to go through the normalization process, reverse engineer the resulting BCNF design to the conceptual tier (an ER model) and be able to verify if we arrive at the source ERD we started off with. The URS and  $F_c$  prevailing over it are reproduced here from Figure 9.9c:

URS (Ins\_name, Ins\_rank, Ins\_qualification, O\_ins\_name, O\_co\_course#, Semester, Co\_course#, Co\_cname, Co\_credits, T\_co\_course#, T\_stu\_num, Grade, Stu\_num, Stu\_name, Stu\_status)  
and

$F_c [fd1, fd2, fd3, fd4, fd5, fd6]$  prevailing over URS, where

fd1: Ins_name → Ins_rank;	fd2: Ins_name → Ins_qualification;
fd3: Co_course# → Co_cname;	fd4: Co_course# → Co_credits;
fd5: Stu_num → Stu_name;	fd6: Stu_num → Stu_status.

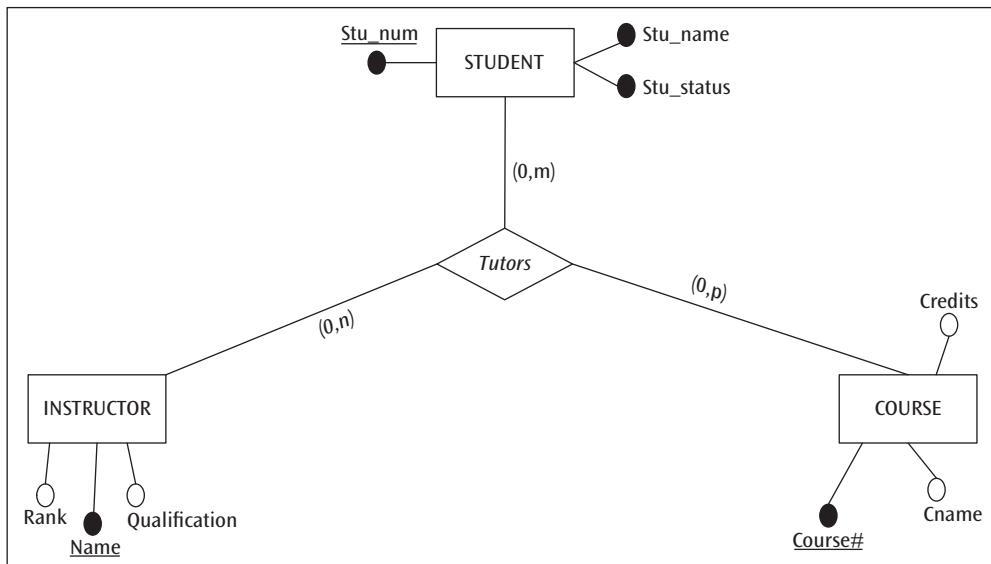
We note that the only candidate key (and therefore the primary key) of URS is (Ins\_name, Course#, Stu\_stu). Next, using the fast-track algorithm, we first arrive at the relation schemas portrayed in Figure 9.10a. Observe that the relation schema R0 results pursuant to the requirement of the algorithm that, in the absence of a candidate key of URS in R1, R2, or R3, a relation schema should be created containing a candidate key of URS.

R1: INSTRUCTOR (Ins\_name, Ins\_rank, Ins\_qualification);  
 R3: COURSE (Co\_course#, Co\_cname, Co\_credits);  
 R5: STUDENT (Stu\_num, Stu\_name, Stu\_status)  
 R0: ICS (L\_ins\_name, L\_co\_course#, L\_stu\_num)  
 $ICS.\{L\_ins\_name\} \subseteq INSTRUCTOR.\{Ins\_name\}$   
 $ICS.\{L\_co\_course\# \} \subseteq COURSE.\{Co\_course\# \}$   
 $ICS.\{L\_stu\_num\} \subseteq STUDENT.\{Stu\_num\}$

**FIGURE 9.10a** Normalized solution for the URS |  $F_c$  displayed in Figure 9.9c

This decomposition D [R1, R3, R5, R0] of URS is not only a non-loss dependency-preserving 3NF solution that the fast-track algorithm is expected to generate, it is also in BCNF. However, contrary to expectation, this solution differs from the relational schema in Figure 9.9b. Note that while the relational schema in Figure 9.9b is a lossy design, the solution derived here (Figure 9.10a) is a non-loss decomposition.

When reverse engineered from this solution, we obtain the ERD displayed in Figure 9.10b, a ternary-relationship type among the three entity types INSTRUCTOR, COURSE, and STUDENT that is drastically different from the two binary relationships proposed in the source ERD (see Figure 9.9a).



**FIGURE 9.10b** ERD reverse engineered from the relational schema displayed in Figure 9.10a

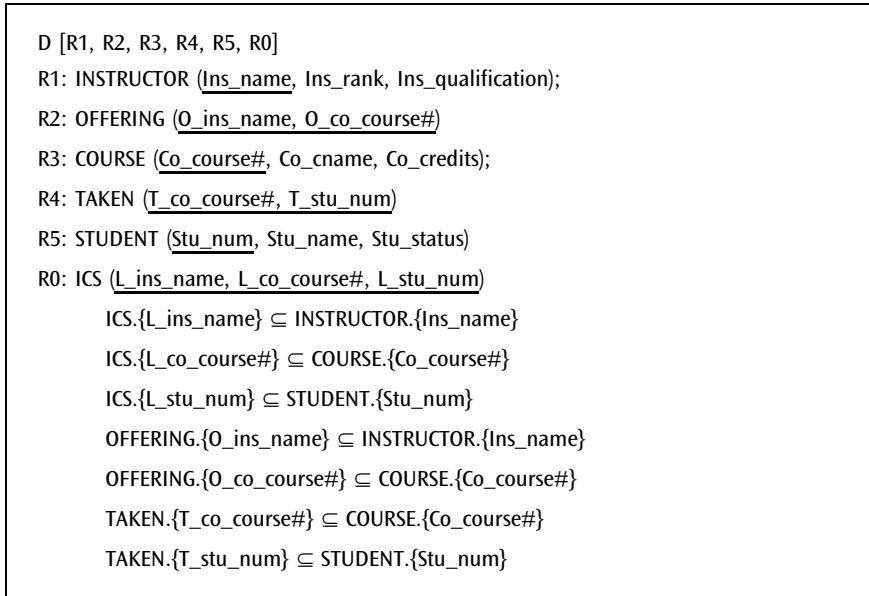
While multiple solutions are certainly acceptable, the semantics captured in these two solutions are drastically different. In fact, for the business rules specified at the beginning of this section, the second solution, depicting a ternary relationship type, is clearly

incorrect. Let us now examine the two relational schemas shown in Figures 9.9b and 9.10a. From a technical perspective, the two designs can be reconciled only when R0: ICS (L\_ins\_name, L\_co\_course#, L\_stu\_num) in Figure 9.10a can be decomposed to the two relation schemas, RX: OFFERING (L\_ins\_name, L\_co\_course#) and RY: TAKING (L\_co\_course#, L\_stu\_num). A close scrutiny of R0, RX, and RY reveals that this precise decomposition is possible when the MVD of the form  $L_{co\_course\#} \Rightarrow L_{ins\_name} | L_{stu\_num}$  is imposed on URS.

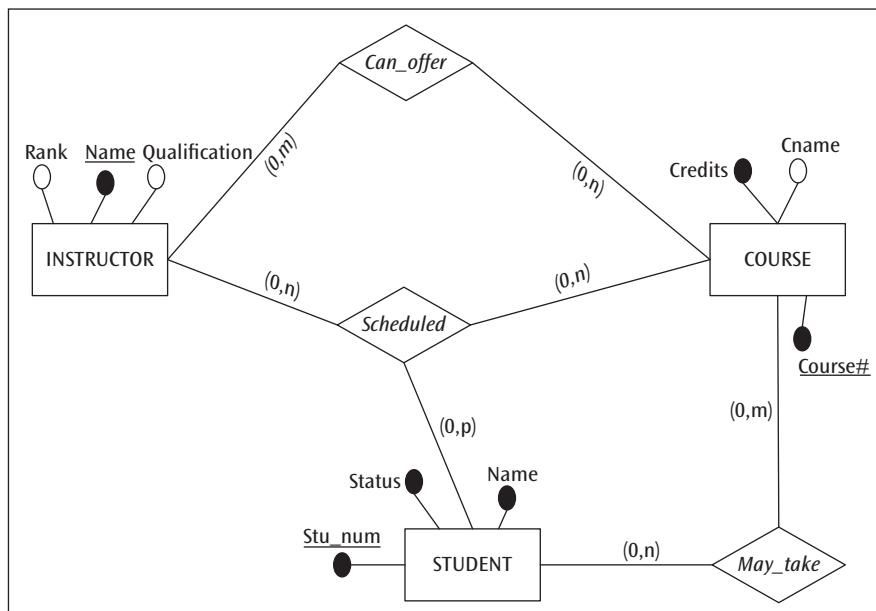
To this end, a review of the dependencies specified on the URS in Figure 9.9c based on the relational schema shown in Figure 9.9b simply focused on the FDs derived from R1, R3, and R5. While no non-trivial FDs are present in R2 and R4, R2 and R4 are mappings of the gerund entity types decomposed from the relationship types *Offers* and *Takes* in Figure 9.9a. While our original specification of the FDs prevailing in the relational schema in Figure 9.9b is technically correct, the overall specification of dependencies is indeed incomplete without capturing the effect of the two relationship types *Offers* and *Takes*. Now, we should be able to infer that while no non-trivial FDs are present in R3 and R5, they actually represent the MVD  $L_{co\_course\#} \Rightarrow L_{ins\_name} | L_{stu\_num}$  that we failed to specify in Figure 9.9c. In other words, this MVD pair represents the business rules that specify the two m:n relationships the entity type COURSE has in the ERD shown in Figure 9.9a. In short, had this MVD been specified in Figure 9.9c, the normalization process would not have yielded the solution presented in Figure 9.10a; instead, the solution would have been exactly the same as in Figure 9.9b, first mapped from the ERD in Figure 9.9a.

In Section 9.5, we discussed the rationale, possibility, and perhaps the need pursuant to requirement specification for a relational model that captures simultaneous existence of ternary and binary relationships among the participating entity types (e.g., SCHEDELE\_Y and OFFERS). Let us now explore a similar design in the current case. The relational schema and the ERD from which this relational schema is mapped are displayed in Figures 9.11a and 9.11b, respectively. While the ERD is technically legitimate and can/does make semantic sense, the relational schema from a pure technical perspective will be observed as having redundant relation schemas. A parsimonious design will evaluate the relation schemas OFFERING and TAKEN as redundant subsumable in the relation schema ICS. The dilemma here is rather straightforward:

- If MVD  $L_{co\_course\#} \Rightarrow L_{ins\_name} | L_{stu\_num}$  exists, the consequent 4NF violation in ICS will require decomposition of ICS to the two projections OFFERING and TAKEN; thus, ICS cannot exist.
- Alternatively, if ICS is in 4NF, decomposing ICS is invalid, and so the relation schemas OFFERING and TAKEN cannot exist.



**FIGURE 9.11a** Relational schema accommodating simultaneous presence of ternary and binary relationships among the participating entity types



**FIGURE 9.11b** ERD for the relational schema presented in Figure 9.11a

The underlying issue is the inability to specify the semantic unrelated, independent existence of one or more binary relationships simultaneously with the existence of a ternary relationship among three participating entity types in terms of dependencies (e.g., FDs, MVDs, and/or JDs). The only solution appears to be to tolerate apparent redundancies in a relational schema when semantics captured in the ERD obviate the absence of any redundancies.

An interesting variation to this design inadvertently resolves the conflict we discussed regarding Figures 9.11a and 9.11b. The only difference in the design presented in Figure 9.12a and 9.12b from the one we analyzed in Figures 9.11a and 9.11b () is that the relation schemas OFFERING and TAKEN each contain an attribute, meaning that **Semester** and **Grade** are attributes of the respective relationships. The presence of these attributes ensures that the relation schemas OFFERING and TAKEN are no longer subsets of the relation schema ICS and are no longer redundant; in fact, their presence in the relational schema is supported by the FDs  $\{O_{\text{co\_course}\#}, O_{\text{ins\_name}}\} \rightarrow \text{Semester}$  and  $\{T_{\text{co\_course}\#}, T_{\text{stu\_num}}\} \rightarrow \text{Grade}$ .

```

D [R1, R2, R3, R4, R5, R0]
R1: INSTRUCTOR (Ins_name, Ins_rank, Ins_qualification);
R2: OFFERING (O_ins_name, O_co_course#, Semester)
R3: COURSE (Co_course#, Co_cname, Co_credits);
R4: TAKEN (T_co_course#, T_stu_num, Grade)
R5: STUDENT (Stu_num, Stu_name, Stu_status)
R0: ICS (L_ins_name, L_co_course#, L_stu_num)
    ICS.{L_ins_name} ⊆ INSTRUCTOR.{Ins_name}
    ICS.{L_co_course#} ⊆ COURSE.{Co_course#}
    ICS.{L_stu_num} ⊆ STUDENT.{Stu_num}
    OFFERING.{O_ins_name} ⊆ INSTRUCTOR.{Ins_name}
    OFFERING.{O_co_course#} ⊆ COURSE.{Co_course#}
    TAKEN.{T_co_course#} ⊆ COURSE.{Co_course#}
    TAKEN.{T_stu_num} ⊆ STUDENT.{Stu_num}

```

**FIGURE 9.12a** A variation of the design presented in Figure 9.11a

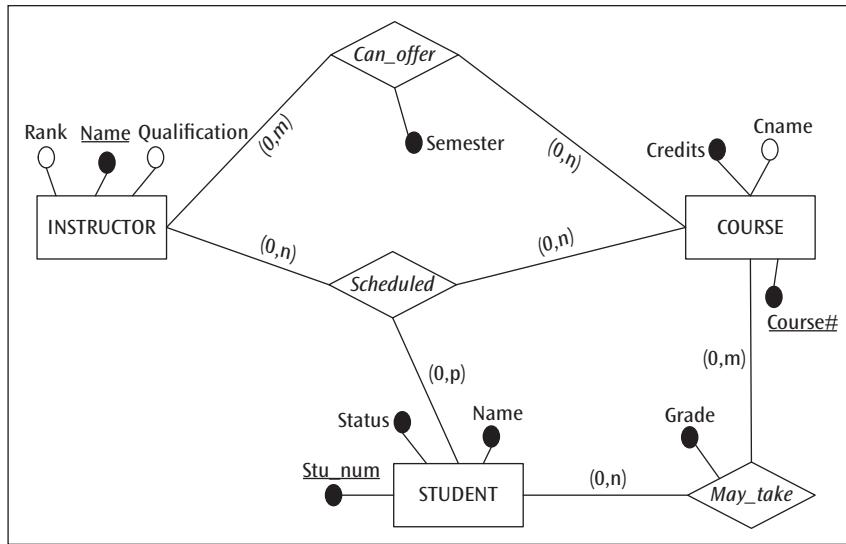


FIGURE 9.12b ERD for the relational schema presented in Figure 9.12a

## 9.7 A NOTE ON DOMAIN KEY NORMAL FORM (DK/NF)

Fagin (1981) proposed a normal form based exclusively on *domain constraints* and *key constraints*. It is named **domain key normal form (DK/NF)**. DKNF states that if every constraint on a relation schema, R, is a logical consequence of the domain constraints and key constraints that apply to R, then, the relation schema, R, is in DKNF. In other words, in order for a relation schema to be in DKNF, all constraints and dependencies that hold on valid relation states (e.g., FD, MVD, JD) are enforced through a set of domain constraints and key constraints. Fagin (1981) showed that a DKNF relation schema is necessarily in 5NF.

While the rules of DKNF are rather simple, DKNF is not always achievable. More importantly, verification of compliance (i.e., when DKNF is achieved) is difficult. There are no formal methods to systematically verify if a relation schema is in DKNF. Thus, while theoretically rigorous, DKNF has limited practical utility.

## Chapter Summary

---

While relation schemas in BCNF are free from modification anomalies due to undesirable functional dependencies, BCNF relation schemas that fail to achieve 4NF and 5NF are vulnerable to modification anomalies as a result of multi-valued dependencies (in the case of 4NF) and join-dependencies (in the case of 5NF).

A multi-valued dependency is defined as follows: In a relation schema  $R(X, Y, Z)$ , where  $X$ ,  $Y$ , and  $Z$  are atomic or composite attributes, a multi-valued dependency  $X \rightrightarrows Y$  exists if each value of  $X$  in any relation state  $r$  of  $R$  is associated with a set of  $Y$  values independent of the  $Z$  values with which  $X$  is associated. In a relation schema  $R(X, Y, Z)$  in BCNF, a multi-valued dependency  $X \rightrightarrows Y$  exists if and only if the natural join  $r(R1)$  and  $r(R2)$ , where  $R1(X, Y)$  and  $R2(X, Z)$  are projections of  $R$ , strictly yields  $r(R)$  for every relation state  $r$  of  $R$ . A relation schema  $R$  is in 4NF if it is in BCNF and has no non-trivial multi-valued dependencies.

A join dependency in a relation schema  $R$  pertains to conditions where the natural join of all of its projections results in the reconstruction of  $R$ . A relation schema that cannot be reconstructed by a natural join of all its projections does not have a join-dependency and is said to be in 5NF.

The normal forms 2NF through 5NF have considered constraints imposed by functional dependencies, multi-valued dependencies, or join-dependencies. Fagin (1981) suggests a generalized constraint that both infers the three kinds of dependencies and allows more generalized constraints. Constraints are explained here in terms of domain constraints of the relation schema's attributes and relation keys. Relation schemas that satisfy these constraints are said to be in domain key normal form (DK/NF). Unfortunately, other than checking to see if each constraint on a relation schema is a logical consequence of the definition of keys of a relation schema or domains of attributes, there are no formal methods to systematically verify if a relation schema is in DK/NF.

## Exercises

---

1. What is a multi-valued dependency?
2. Consider the instance of the relation SHIRT (Shirt#, Color, Size), where Shirt# is equivalent to a style number (e.g., style number 341 might be a shirt with a button-down collar, while style number 342 might be a shirt with an open collar, etc.). Observe that each Shirt# comes in a variety of colors and sizes.

Shirt#	Color	Size
341	White	Small
341	White	Medium
341	White	Large
341	Blue	Small
341	Blue	Medium
341	Blue	Large
341	Yellow	Medium
341	Yellow	Large

Shirt#	Color	Size
342	White	Medium
342	White	Large
342	Blue	Large

Does SHIRT possess the multi-valued dependency  $\text{Shirt\#} \Rightarrow \text{Color}\text{Size}$ ? Why or why not?

3. Which of the inference rules for multi-valued dependencies supports the statement that *multi-valued dependencies (MVDs) are a generalization of functional dependencies (i.e., a functional dependency is a special case of a multi-valued dependency)*?
4. Consider the following revised instance of the SHIRT relation in Exercise 2:

Shirt#	Color	Size
341	White	Large
342	White	Medium
343	White	Large
344	Blue	Large
345	Yellow	Large

499

Does this version of SHIRT possess the multi-valued dependency  $\text{Shirt\#} \Rightarrow \text{Color}\text{Size}$ ? Why or why not?

5. Consider the relation schema STUDENT (Sid, Shoe\_size, Marital\_status)
 

F: fd1: **Sid**  $\rightarrow$  **Shoe\_size**; fd2: **SID**  $\rightarrow$  **Marital\_status**

  - a. Does STUDENT possess a multi-valued dependency? Why or why not?
  - b. Does STUDENT possess a 4NF violation? Explain your answer.
6. How are multi-valued dependencies incorporated into the definition of 4NF?
7. Consider the relation instance STUDENT (Major, Stu\_id, Activity, Name, Phone):

Major	Stu_id	Activity	Name	Phone
Music	100	Swimming	Costello	444-5456
Accounting	100	Swimming	Costello	444-5456
Music	100	Tennis	Costello	444-5456
Accounting	100	Tennis	Costello	444-5456
Mathematics	150	Jogging	Brooks	444-5456
Mathematics	250	Sleeping	Abbott	665-5456
Music	200	Swimming	Costello	665-5456
Mathematics	250	Eating	Abbott	665-5456

Chapter 9

- 500
- a. Identify all multi-valued dependencies in STUDENT.
  - b. Describe insertion, deletion, and update anomalies that accompany each of the multi-valued dependencies identified above. Are any of these multi-valued dependencies also functional dependencies?
  - c. Decompose STUDENT into a relational schema (i.e., a set of relation schemas) that exhibit attribute preservation, dependency preservation, and the presence of a lossless-join decomposition.
8. Consider the relation instance COURSE\_OFFERED (Course, Teacher, Text):

Course	Teacher	Text
Math	White	Calculus I
Physics	Brown	B. Mechanics
Physics	Black	B. Mechanics
Physics	Black	Prin of Optics
Math	Black	B. Mechanics
Math	White	B. Mechanics
Eng Mech	White	Calculus I

- a. Does COURSE\_OFFERED possess a multi-valued dependency? If the answer is yes, show how this multi-valued dependency occurs. How do you resolve this condition?
  - b. Does COURSE\_OFFERED possess a join-dependency? If the answer is yes, show how this join-dependency occurs. How do you resolve this condition?
9. What is a join-dependency (JD), and in what way is it related to a multi-valued dependency?
10. What is the definition of fifth normal form (5NF)?
11. Is the relation instance SHIRT in Exercise 2 in 5NF? If the answer is yes, explain. If the answer is no, provide a solution.
12. Consider the universal relation schema STOCK (Symbol, Company, Exchange, Investor, Date, Price, Broker, Dividend) and the following dependencies set:
- fd: **Symbol → {Company, Exchange, Dividend}**
- mvd: **Symbol ⇒ {Investor, Broker}**
- a. Identify the primary key such that STOCK is in 1NF.
  - b. Explain the immediate normal forms violated by the functional dependencies and multi-valued dependencies.
  - c. Step through the normalization process to obtain a final design that is in 4NF. At each level of decomposition, indicate the primary key and dependencies resolved and explain the normal form achieved and violated in each one of the decomposed relation schemas. Explain how each of the decompositions is lossless. Finally, show all integrity constraints in the final relational schema.
  - d. Reverse engineer your final design to a Presentation Layer ERD.

13. Given the schema SCHEDULE (Prof, Office, Major, {Book}, Course, Quarter) along with F: fd1: **Prof** → {Office, Major} and V: mvd: **Prof** ⇒ **Book**, do the following:
- Identify the primary key of SCHEDULE such that SCHEDULE is a 1NF relation schema.
  - Normalize SCHEDULE to 4NF.
  - Indicate the entity integrity and referential integrity constraints.
  - Reverse engineer the design to a conceptual schema using the ER modeling grammar.
14. This exercise is based on four different inventory relation schemas, P, Q, R, and S. An instance of each relation schema is given here, each of which reflects a different design objective.

501

P	Q	R	S								
Part#	Color	Store#									
17	Red	A	17	Blue	A	17	Blue	A	17	Blue	A
17	Red	B	17	Green	A	17	Green	A	17	Green	A
17	Red	C	17	Black	A	17	Blue	B	17	Blue	B
18	White	B	17	Green	C	17	Green	B	17	Green	B
18	White	C	18	Green	A	17	Blue	C	17	Green	C
19	Red	B	18	Black	C	17	Green	C	18	Green	B
						18	Green	B	18	Green	C
						18	Green	C	18	Black	C
						18	Black	B			
						18	Black	C			

Answer each of the following questions:

- Identify the primary key of each relation schema.
- Indicate the functional dependencies and/or multi-valued dependencies present in each design.
- Evaluate each design for a 4NF or 5NF violation. Explain your conclusion in each case.
- Where there is a 4NF violation, provide a revised lossless-join design that is in 4NF.



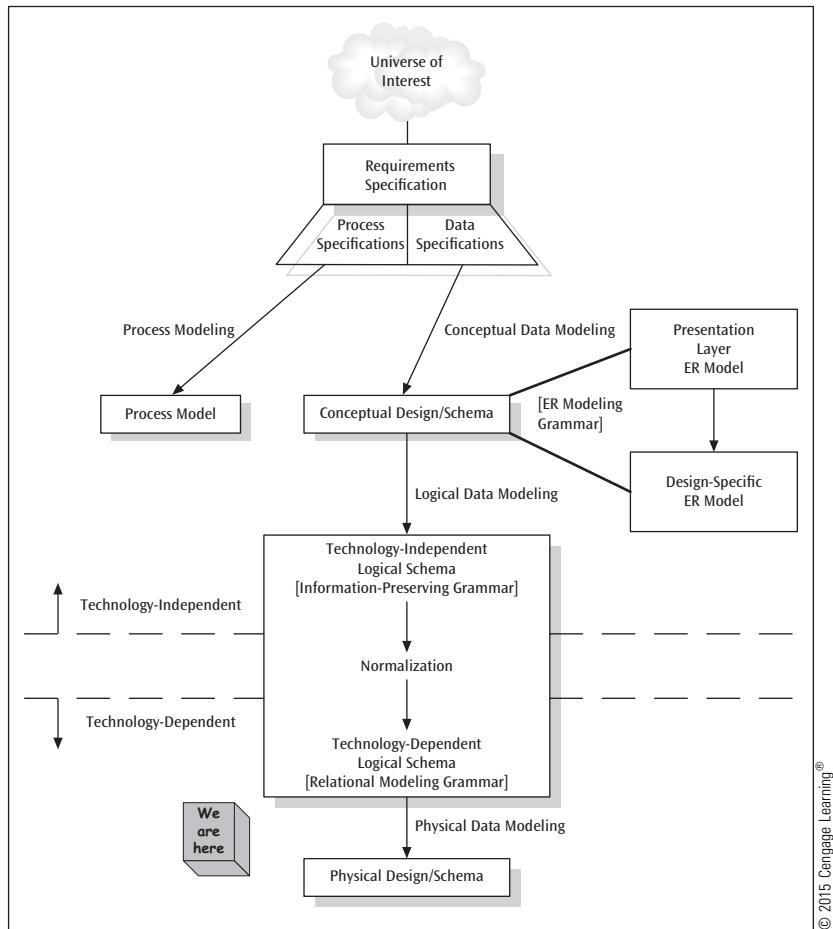
# PART IV

## DATABASE IMPLEMENTATION USING THE RELATIONAL DATA MODEL

### INTRODUCTION

---

The final phase of the data modeling life cycle is physical data modeling. At this point, we have an information-preserving logical schema normalized to the extent we want and ready for implementation. Because contemporary database systems are dominated by relational data models, as we transition from a technology-independent to a technology-dependent paradigm, our mapping of the logical schema to a physical data model will employ the technology-driven relational data modeling grammar. Accordingly, a relation becomes a *table*, the tuples are the *rows* of the table, and the attributes are the *columns* of the table. Figure IV.1 points out our location in the data modeling hierarchy.



**FIGURE IV.1** Roadmap for data modeling and database design

Structured Query Language (SQL) is the standard universally accepted by the relational database (RDB) community—the users and the RDBMS vendors—for the implementation of a relational database. The name SQL comes from its predecessor SEQUEL (Structured English Query Language) developed by IBM Research as an experimental product. SQL<sup>1</sup> is a comprehensive relational database language and comprises three sublanguages: (1) a data definition language (SQL/DDL) intended for the creation and alteration of tables and other associated structures such as views, domains, and schemas; (2) a data manipulation language (SQL/DML) aimed at data manipulation tasks; and (3) a data control language (SQL/DCL) geared to controlling database access. There are numerous idiosyncratic differences among commercial DBMS products. Nonetheless, the RDBMS vendors endeavor to meet a certain level of the SQL standards developed jointly by ANSI

<sup>1</sup>Officially pronounced *ess-que-ell* (Date, 2004, p. 4)—not *sequel*.

and ISO. Consequently, the users of ANSI/ISO standard SQL find migration and interoperability across RDBMS products relatively painless. At this writing, most commercial RDBMS products provide reasonable support to SQL-2003. Therefore, the discussions in Part IV are based on SQL-2003. Also, we provide only an overview of the salient features of SQL. The reader is directed to the vendor-specific reference manuals for the complete syntax of the language.

Chapter 10 begins the discussion with two sections that focus on database creation followed by Chapter 11 where relational algebra is introduced as a means to retrieve data from a relational database. A query expressed in relational algebra involves a series of operations which when executed in the order specified produces the desired results. Even though SQL uses some of the relational algebra operators explicitly, it is a high-level declarative language based on tuple relational calculus.<sup>2</sup> Chapter 12 is dedicated to an extensive coverage of SQL pertaining to data retrieval (querying). Sometimes, this portion of the DML is referred to as the data query language (DQL).

Chapter 13 covers additional features of the SQL language. The discussion begins with two sections that focus on built-in functions that facilitate working with strings, dates, and times. This is followed by sections that introduce the reader to SQL features for writing hierarchical queries, using extended group by clauses, working with analytical functions, and incorporating elements of spreadsheet modeling into the SQL SELECT statement.

---

<sup>2</sup>For more details on tuple relational calculus, see Elmasri and Navathe (2010).

# CHAPTER 10

## DATABASE CREATION

At the implementation stage of a database, the principal tasks include creating and modifying the database tables and other related structures, enforcing integrity constraints, and populating the database tables. Once created, data from these tables must be retrieved. Relational algebra, a mathematical expression of data retrieval methods prescribed by E. F. Codd, is introduced first as a means to specify the logic for data retrieval from a relational database. A query expressed in relational algebra involves a sequence of operations that, when executed in the order specified, produces the desired results. SQL is the most common way that relational algebra is implemented for data retrieval operations in a relational database.

In this chapter as well as in Chapters 11 through 13, we use the SQL-2003 language standard in the code for the SQL statements. It is not necessary that a commercial DBMS implementation include all the standard SQL constructs or follow the standard language syntax verbatim for all SQL constructs. It is highly likely that commercial DBMS products differ in the implementation of at least some of the SQL syntax. Also, some vendors offer additional non-standard SQL constructs. Therefore, the reader using the SQL scripts<sup>1</sup> presented in Chapters 10 through 13 may occasionally need to refer to the SQL reference material of the DBMS platform being used. To the extent that a DBMS product does not conform to a common syntactical standard, portability and migration across product platforms might be difficult.

Chapter 10 is divided into two sections. Section 10.1 discusses elements of SQL's data definition language used to create and alter the structure of base tables. Section 10.2 discusses three statements used to modify database tables: INSERT, DELETE, and MODIFY.

---

<sup>1</sup>A script is a command or series of commands usually stored in a file.

## 10.1 DATA DEFINITION USING SQL

As noted in the introduction of Part IV, a relation is called a “table” in SQL, and attributes and tuples are termed “columns” and “rows,” respectively. In reality, however, the formal object called a “relation” and the informal object called a “table” have several differences. For instance, a table can contain duplicate rows, while a relation is prohibited from having duplicate tuples. SQL tables are allowed to have null values, while relations are not. The columns of a table have a left-to-right ordering, but the attributes of a relation in a relational data model are unordered. Columns of a table may end up with duplicate column names (e.g., the result of a join of two tables with the same column name) or with no column name (a derived column); this, however, is not permissible in a relation. The rows of a table have a top-to-bottom ordering, but the tuples of a relation do not. A table is usually expected to have at least one column, whereas, technically, a relation with no attributes is permissible. Notwithstanding many such differences, in the context of a relational data model, a table can represent a concrete picture of an abstract object called a relation—that is, a table can “represent” a relation but is not precisely equivalent to a relation.

The data definition sublanguage of SQL, known as SQL/DDL, has three major constructs for creating and modifying databases: CREATE, ALTER, and DROP.

### 10.1.1 Base Table Specification in SQL/DDL

A relation schema (or, alternatively, a logical scheme) is specified in SQL/DDL by the CREATE TABLE statement. The result of a CREATE TABLE statement is referred to as a “base table” to indicate that the table is actually created, populated with rows of data, and stored as a physical file by the DBMS. In addition to defining the basic table name and column names, the CREATE TABLE statement has a rich syntax that can specify domain constraints on columns, an entity integrity constraint (specification of the primary key), uniqueness constraints (specification of alternate keys), foreign key constraints, deletion and update rules, and more. We present the CREATE TABLE statement in gradual increments using appropriate examples. Let us begin with a simple scenario of patients in a clinic placing orders for medications. The ER model (the ERD plus a list of semantic integrity constraints that cannot be incorporated in the ERD), the relational schema, and the corresponding information-preserving logical schema appear in Figures 10.1a through 10.1d.

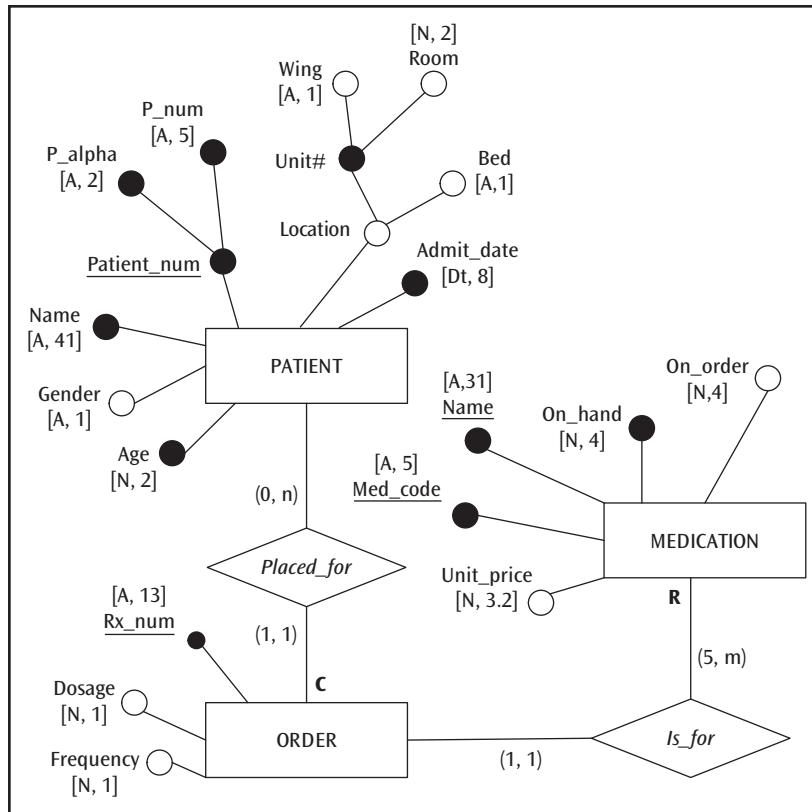
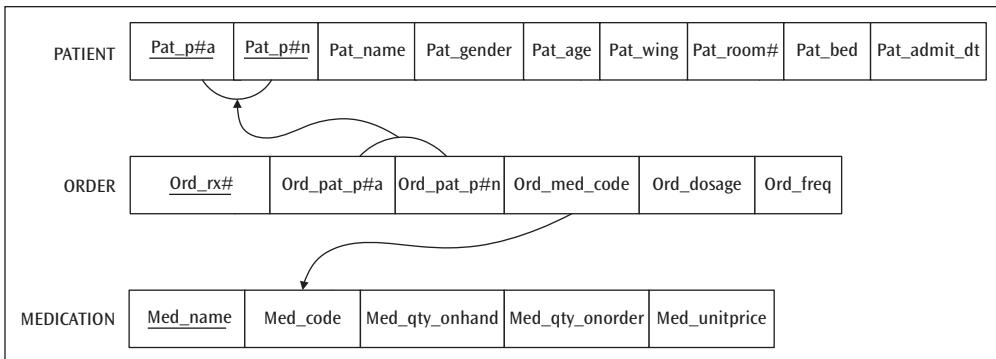


FIGURE 10.1a ERD: An excerpt from a hypothetical medical information system

> Constraint	Gender	IN ('M', 'F')
> Constraint	Age	IN (1 through 90)
> Constraint	Bed	IN ('A', 'B')
> Constraint	Unit_price	< 4.50
> Constraint	(Qty_onhand + Qty_onorder)	IN (1000 through 3000)
> Constraint	Dosage	DEFAULT 2
> Constraint	Dosage	IN (1 through 3)
> Constraint	Frequency	DEFAULT 1
> Constraint	Frequency	IN (1 through 3)

FIGURE 10.1b Semantic integrity constraints for the Design-Specific ERD

**FIGURE 10.1c** Relational schema for the ERD in Figure 10.1a

L1: PATIENT	<u>Pat_p#a</u> (A, 2)	Pat_p#n (A, 5)	Pat_name* (A, 41)	Pat_gender (A, 1)	Pat_age* (N, 2)	[Pat_wing(A, 1)]	Pat_room#)* (N, 3)	Pat_bed (A, 1)	Pat_admit_dt* (Dt, 8)
0 <----- L1 -----> n      5      L3      m									
L2: ORDER	Ord_rx# (A, 13)	Ord_pat_p#a (A, 2)	Ord_pat_p#n (A, 5)	Ord_med_code (A, 5)	Ord_dosage (N, 1)	Ord_freq (N, 1)			
1 <----- C -----> 1      0      R      1									
L3: MEDICATION	Med_name (A, 31)	Q[Med_code]* (A, 5)	Med_qty_onhand* (N, 4)	Med_qty_onorder (N, 4)	Med_unitprice (N, 3.2)				

**FIGURE 10.1d** An information-preserving logical schema for the ERD in Figure 10.1a

### 10.1.1.1 Syntax of the CREATE TABLE Statement

The implementation of the design shown in Figures 10.1a–d in a relational database entails creating the three base table structures first. At the minimum, the table specification includes the table name, the names of the columns in the table, and the data type for each column. The data type of a column is a part of the domain specification for the attribute and is sometimes referred to as the “type constraint” on the attribute. In other words, specifying the data type for a column is equivalent to imposing the type constraint on the attribute. The minimal form of a CREATE TABLE statement for the three base tables is shown in Box 1. SQL statements are *case-insensitive*, while string values referenced in the SQL statements are *case-sensitive*.

```

CREATE TABLE patient
(Pat_p#a      char (2),
Pat_p#n      char (5),
Pat_name     varchar (41),
Pat_gender   char (1),
Pat_age      smallint,
Pat_admit_dt date,
Pat_wing     char (1),
Pat_room#    integer,
Pat_bed      char (1)
);

CREATE TABLE medication
(Med_code      char (5),
Med_name      varchar (31),
Med_qty_onhand integer,
Med_qty_onorder integer,
Med_unitprice decimal(3,2)
);

CREATE TABLE order
(Ord_rx#      char (13),
Ord_pat_p#a  char (2),
Ord_pat_p#n  char (5),
Ord_med_code char (5),
Ord_dosage   smallint,
Ord_freq     smallint
);

```

## Box 1

Observe that while it is legal to use the same attribute name in different entity types in an ER diagram, relational database theory stipulates that attribute names must be unique over the entire relational schema. Accordingly, the attribute names in the relational/logical schema in this example follow the naming convention proposed in Chapter 6. The column names of a base table in SQL/DDL are considered to be ordered in the sequence in which they are specified in the CREATE TABLE syntax. However, when populated with data, the rows are not considered to be ordered within a base table. Common data types supported by SQL-2003 are grouped under *Number*, *String*, *Date/time*, and *Interval*, and they are listed in Table 10.1. DBMS vendors often build their own data types based on these four categories.

<b>Number Data Types</b>	
<b>numeric (p, s)</b> where p indicates precision and s indicates scale	Exact numeric type—literal representation of number values— <i>decimal portion exactly the size dictated by the scale</i> —storage length = (precision + 1) when scale is > 0—most DBMSs have an upper limit on p (e.g., 28)
<b>decimal (p, s)</b> where p indicates precision and s indicates scale	Exact numeric type—literal representation of number values— <i>decimal portion at least the size of the scale; but expandable to limit set by the specific DBMS</i> —storage length = (precision + 1) when scale is > 0 — most DBMSs have an upper limit on p (e.g., 28)
<b>integer or integer (p)</b> where p indicates precision	Exact numeric type—binary representation of large whole number values—often precision set by the DBMS vendor (e.g., 2 bytes)
<b>smallint or smallint (p)</b> where p indicates precision	Exact numeric type—binary representation of small whole number values—often precision set by the DBMS vendor (e.g., 1 byte)
<b>float (p)</b> where p indicates precision	Approximate numeric type—represents a given value in an exponential format—precision value represents the minimum size used, up to the maximum set by the DBMS
<b>real</b>	Approximate numeric type—represents a given value in an exponential format—has a default precision value set below that set for double precision data type by the DBMS
<b>double precision</b>	Approximate numeric type—represents a given value in an exponential format—has a default precision value set above that set for real data type by the DBMS
<b>String Data Types</b>	
<b>character (<math>\ell</math>) or char (<math>\ell</math>)</b> where $\ell$ indicates length	Fixed length character strings including blanks from the defined language set SQL_TEXT within a database—can be compared to other columns of the same type with different lengths or varchar type with different maximum lengths—most DBMS have an upper limit on $\ell$ (e.g., 255)
<b>character varying (<math>\ell</math>) or char (<math>\ell</math>) varying or varchar (<math>\ell</math>)</b> where $\ell$ indicates the maximum length	Variable length character strings except trailing blanks from the defined language set SQL_TEXT within a database—DBMS records actual length of column values—can be compared to other columns of the same type with different maximum lengths or char type with different lengths—most DBMS have an upper limit on $\ell$ (e.g., 2000)
<b>bit (<math>\ell</math>)</b> where $\ell$ indicates length	Fixed length binary digits (0,1)—can be compared to other columns of the same type with different lengths or bit varying type with different maximum lengths
<b>bit varying (<math>\ell</math>)</b> where $\ell$ indicates maximum length	Variable length binary digits (0,1)—can be compared to other columns of the same type with different maximum lengths or bit type with different lengths

**TABLE 10.1** Some common data types supported by SQL-2003

Date/Time & Interval Data Types	
<b>date</b>	10 characters long—format: yyyy-mm-dd—can be compared to only other date type columns—allowable dates conform to the Gregorian calendar
<b>time (p)</b>	Format: hh:mi:ss—sometimes precision ( <b>p</b> ) specified to indicate fractions of a second—the length of a TIME value is 8 characters, if there are no fractions of a second. Otherwise, the length is 8, plus the precision, plus one for the delimiter: hh:mi:ss.p—if no precision is specified, it is 0 by default—TIME can only be compared to other TIME data type columns
<b>timestamp (p)</b>	Format: yyyy:mm:dd hh:mi:ss.p—a timestamp length is nineteen characters, plus the precision, plus one for the precision delimiter—timestamp can only be compared to other timestamp data type columns
<b>interval (q)</b>	Represents measure of time—there are two types of intervals: year-month (yyyy:mm) which stores the year and month; and day-time (dd hh:mi:ss) which stores the days, hours, minutes, and seconds—the qualifier ( <b>q</b> ) known in some databases as the interval lead precision, dictates whether the interval is year-month or day-time—implementation of the qualifier value varies

**TABLE 10.1** Some common data types supported by SQL-2003 (continued)

Note that the “CREATE TABLE order” statement in Box 1 will generate an error; the word “order” (or “ORDER,” or any case of the word) cannot be used as a user-defined value for a table, column, or any construct in SQL because ORDER itself is an SQL construct and thus a reserved word. A list of SQL reserved words appears in Appendix A.

The CREATE TABLE statement is a single statement starting at “CREATE” and ending with a semicolon (;). The entire statement could be written on a single line, but it spans multiple lines to enhance clarity and readability. The general form of the syntax for the CREATE TABLE statement is:

```
CREATE TABLE table_name (comma-delimited list of table-elements);
```

where the following apply:

- *table\_name* is a user-supplied name for the base table.
- Each *table-element* in the list is either a column definition or a constraint definition.

There must be at least one column definition in order for a base table to exist. The basic syntax for a column definition is of the following form:

```
column_name representation [default-definition] [column-
constraint list]
```

where the following apply:

- *column\_name* is a user-supplied name for a column.
- *representation* specifies the relevant **data type**, or alternatively, the predefined *domain-name*.

The optional *default-definition* (optional elements are indicated by square brackets [ ]) specifies a default value for the column, which overrides any default value specified in the domain definition, if applicable. In the absence of an explicit default definition (directly or via the domain definition), the implicit assumption of a NULL value for the default prevails. The *default-definition* is of the following form:

```
DEFAULT ( literal | niladic-function | NULL )2
```

where the following apply:

- The | implies “or.”
- The optional column-constraint list specifies constraint-definition for the column.

The basic syntax for a *constraint-definition* follows this form:

```
[ CONSTRAINT constraint_name ] constraint-definition3
```

Constraints in a base table are declarative in nature and are imposed either on a single column in the table or on a set of columns in the table. The former is referred to as an attribute-level or column-level constraint, while the latter goes by the name tuple-level or row-level constraint. A constraint definition can be an independent table element (i.e., row-level constraint) or, if applicable to a specific column only, part of the column definition (i.e., column-level constraint). Constraint definitions include the following:

- The primary key definition—i.e., specification of an entity integrity constraint:

```
PRIMARY KEY (comma-delimited column list)
```

**Example:**

```
CONSTRAINT pk_pat PRIMARY KEY (Pat_p#a, Pat_p#n)
```

- An alternate key definition—i.e., specification of a uniqueness constraint:  
UNIQUE (comma-delimited column list)

**Example:**

```
CONSTRAINT unq_med UNIQUE (Med_code)
```

- A foreign key constraint—i.e., specification of a referential integrity constraint:

```
FOREIGN KEY (comma-delimited column list of referencing table)
    REFERENCES table_name (comma-delimited column list of referenced
    table)4
```

```
[ referential triggered action clause ]
```

---

<sup>2</sup>A niladic-function is a built-in function that takes no arguments (Date and Darwen, 1997, p. 55). The niladic-functions that are allowed here are: USER, CURRENT\_USER, SESSION\_USER, SYSTEM\_USER, CURRENT\_DATE, CURRENT\_TIME, and CURRENT\_TIMESTAMP.

<sup>3</sup>The CONSTRAINT constraint\_name phrase enclosed by [ ] is optional. However, in practice, it is *extremely* useful to use this phrase, especially for ease of later reference to the constraint by a name known to the creator of the table or an individual responsible for altering the table. We strongly suggest mandatory use of this phrase in constraint specification.

<sup>4</sup>If the referenced column list is the primary key of the referenced table, the specification of this column list is optional.

The *referential triggered action clause* facilitates specification of the action to be taken when the foreign key constraint is violated upon deletion of a referenced tuple or upon modification of the value of the referenced column list (primary key or alternate key, as the case may be). The action options are: CASCADE, SET NULL, SET DEFAULT, and RESTRICT. These actions are referred to as **reactive constraints** and must be qualified by the referential trigger ON DELETE or ON UPDATE. These action options are described and illustrated with examples in the context of a deletion referential trigger in Section 2.3.7 of Chapter 2 and are equally applicable to an update referential trigger.

**Example:**

```
CONSTRAINT fk_med FOREIGN KEY (Ord_med_code)
    REFERENCES medication (med_code)
    ON DELETE RESTRICT
    ON UPDATE CASCADE
```

514

- A check constraint definition in order to restrict column or domain values:

```
CHECK (conditional expression)
```

The *conditional expression* here can be of arbitrary complexity; it can even refer to other base tables in the database. Violation of a check constraint occurs when an attempt to create or modify a row in a base table causes the conditional expression stated in the constraint to evaluate as “false.”

**Example:**

```
CONSTRAINT chk_gender CHECK Gender IN ('M', 'F')
```

To enhance clarity, it is customary to include the column-level constraint definitions as clauses of the respective column definitions to which they belong, and to list only the row-level constraint definitions at the end of all the column definitions. This style of coding is demonstrated in the DDL code for the MEDICATION and ORDERS tables in Box 2. Notice that the name of the base table ORDER (SQL reserved word) has been changed to ORDERS (not a reserved word). An alternative practice is to code all constraint definitions at the end of all the column definitions, as shown in the DDL code for the PATIENT table in Box 2. Haphazard mixing of column definitions and constraint definitions in the DDL code is strictly discouraged in the interest of best programming practice.

At this point, we are ready to add other constraints in the CREATE TABLE statement. Let us incorporate the integrity constraints specified in the relational schema (Figure 10.1c) along with the semantic integrity constraints listed in Figure 10.1b in the SQL/DDL code that we already have. The resulting script is shown in Box 2.<sup>5</sup>

---

<sup>5</sup>In Chapters 11–13, table names are shown in all capital letters; that convention is preserved in the running text of this chapter and other chapters of Part IV. However, in this chapter, for clarity in distinguishing SQL keywords from table names, table names are sometimes shown in lowercase in SQL code.

```

CREATE TABLE patient
(Pat_p#a      char (2),
Pat_p#n      char (5),
Pat_name      varchar (41),
Pat_gender    char (1),
Pat_age       smallint,
Pat_admit_dt  date,
Pat_wing      char (1),
Pat_room#     integer,
Pat_bed       char (1),
CONSTRAINT pk_pat PRIMARY KEY (Pat_p#a, Pat_p#n),
CONSTRAINT chk_gender CHECK (Pat_gender IN ('M', 'F')),
CONSTRAINT chk_age CHECK (Pat_age IN (1 through 90)),
CONSTRAINT chk_bed CHECK (Pat_bed IN ('A', 'B'))
);

CREATE TABLE medication
(Med_code      char (5) CONSTRAINT unq_med UNIQUE,
Med_name      varchar (31) CONSTRAINT pk_med PRIMARY KEY,
Med_unitprice decimal (3,2) CONSTRAINT chk_unitprice CHECK (Med_unitprice < 4.50),
Med_qty_onhand integer,
Med_qty_onorder integer,
CONSTRAINT chk_qty CHECK ((Med_qty_onhand + Med_qty_onorder) BETWEEN 1000 AND 3000)
);

CREATE TABLE orders
(Ord_rx#      char (13) CONSTRAINT pk_ord PRIMARY KEY,
Ord_pat_p#a   char (2),
Ord_pat_p#n   char (5),
Ord_med_code  char (5) CONSTRAINT fk_med FOREIGN KEY REFERENCES medication (med_code),
Ord_dosage    smallint DEFAULT 2 CONSTRAINT chk_dosage CHECK (Ord_dosage BETWEEN 1 AND 3),
Ord_freq      smallint DEFAULT 1 CONSTRAINT chk_freq CHECK (Ord_freq IN (1,2,3)),
CONSTRAINT fk_pat FOREIGN KEY (Ord_pat_p#a, Ord_pat_p#n)
REFERENCES patient (Pat_p#a, Pat_p#n)
);

```

## Box 2

Observe that the DDL script produced on the basis of the relational schema (Figure 10.1c) does not fully capture all the information conveyed in the ERD. For instance, the optional property of some attributes, candidate keys of relation schemas, deletion rules, and participation constraints of relationships have not been mapped from the ERD to the relational schema and hence are not reflected in the DDL. An inspection of the information-preserving logical schema (Figure 10.1d) reveals that:

- **Pat\_name**, **Pat\_age**, **Pat\_admit\_dt**, **Med\_code**, and **Med\_qty\_onhand** are mandatory attributes—that is, cannot have null values in any tuple.
- **Med\_code** is the alternate key since **Med\_name** has been chosen as the primary key of the MEDICATION table.
- Participation of ORDER in the *Placed\_for* relationship is total.
- Participation of PATIENT in the *Placed\_for* relationship is partial.
- Participation of ORDER in the *Is\_for* relationship is total.
- Participation of MEDICATION in the *Is\_for* relationship is total.
- The deletion rule for the *Is\_for* relationship is restrict.
- The deletion rule for the *Placed\_for* relationship is cascade.
- **[Pat\_wing, Pat\_room]** is a composite attribute.
- **[Pat\_wing, Pat\_room, Pat\_bed]** is a composite attribute.

*Note:* The cardinality ratio of the form (1, n) in a relationship type is implicitly captured in the DDL specification via the foreign key constraint. Any (1, 1) cardinality ratio can be implemented using the UNIQUE constraint definition.

At this point, let us write the SQL/DML script a third time so as to capture all these constraint definitions. The revised DDL script appears in Box 3, with the added constraint definitions highlighted.

```

CREATE TABLE patient
(Pat_p#a      char (2),
Pat_p#n      char (5),
Pat_name     varchar (41) constraint nn_Patnm not null,
Pat_gender   char (1),
Pat_age      smallint constraint nn_Patage not null,
Pat_admit_dt date constraint nn_Patadmdt not null,
Pat_wing     char (1),
Pat_room#    integer,
Pat_bed      char (1),
CONSTRAINT pk_pat PRIMARY KEY (Pat_p#a, Pat_p#n),
CONSTRAINT chk_gender CHECK (Pat_gender IN ('M', 'F')),
CONSTRAINT chk_age CHECK (Pat_age IN (1 through 90)),
CONSTRAINT chk_bed CHECK (Pat_bed IN ('A', 'B'))
);

CREATE TABLE medication
(Med_code      char (5) CONSTRAINT nn_medcd not null CONSTRAINT unq_med UNIQUE,
Med_name     varchar (31) CONSTRAINT pk_med PRIMARY KEY,
Med_unitprice decimal (3,2) CONSTRAINT chk_unitprice CHECK (Med_unitprice < 4.50),
Med_qty_onhand integer CONSTRAINT nn_medqty not null,
Med_qty_onorder integer,
CONSTRAINT chk_qty CHECK ((Med_qty_onhand + Med_qty_onorder) BETWEEN 1000 AND 3000)
);

CREATE TABLE orders
(Ord_rx#      char (13) CONSTRAINT pk_ord PRIMARY KEY,
Ord_pat_p#a  char (2) CONSTRAINT nn_ord_pat_p#a not null,
Ord_pat_p#n  char (5) CONSTRAINT nn_ord_pat_p#n not null,
Ord_med_code char (5) CONSTRAINT fk_med REFERENCES medication (Med_code)
ON DELETE RESTRICT ON UPDATE RESTRICT,
Ord dosage   smallint DEFAULT 2 CONSTRAINT chk_dosage CHECK (Ord_dosage BETWEEN 1 AND 3),
Ord freq     smallint DEFAULT 1 CONSTRAINT chk_freq CHECK (Ord_freq IN (1, 2, 3)),
CONSTRAINT fk_pat FOREIGN KEY (Ord_pat_p#a, Ord_pat_p#n)
REFERENCES patient (Pat_p#a, Pat_p#n) ON DELETE CASCADE ON UPDATE CASCADE
);

```

Box 3

By default, a column is allowed to have null values in the rows. The “not null” constraint definitions take care of the mandatory attribute value specification in the corresponding columns in the associated base tables. Likewise, unless explicitly prohibited, a column can contain the same value in multiple rows. An alternate key (i.e., a candidate key not chosen as the primary key of a base table) is required to enforce the uniqueness constraint. This is accomplished by the UNIQUE constraint definition for the **Med\_code** column in the MEDICATION table.

Partial participation of a parent as well as a child in a relationship (i.e., min = 0) exists by default. Total participation of a parent in a relationship (i.e., min > 0) cannot be

enforced using any of the constraint definitions discussed so far. SQL-2003 offers another mechanism called “declarative assertion” to specify broader constraints at the database schema level. However, although part of the SQL-2003 standard, major database products do not support the declarative definition of an assertion. Total participation of a child in a relationship (i.e., min = 1) is enforced by specifying a “not null” constraint on the foreign key attribute(s) (e.g., not null constraint definition for **(Ord\_pat\_p#a, Ord\_pat\_p#n)** in the ORDERS table). The deletion rules (referential triggered action clause) are incorporated using the ON DELETE clause of the foreign key constraint definition.<sup>6</sup> Since, by definition, a relation schema has only atomic attributes, SQL/DDL does not provide for the specification of composite columns—all columns in a table are atomic.

#### 10.1.1.2 Syntax of the ALTER TABLE Statement

At this point, we have successfully created the table structures for the logical schema constituting the base tables PATIENT, MEDICATION, and ORDERS. Suppose we want to make some corrections or changes to one or more of these base table structures. The ALTER TABLE statement in SQL/DDL is used to accomplish this. The general form of the syntax for the ALTER TABLE statement is as follows:

```
ALTER TABLE table_name action;
```

where *table\_name* is the name of the base table being altered and *action* is one of the following:

- Adding a column or altering a column’s *default-definition* or removing the existing *default-definition* via the syntax:

```
ADD [ COLUMN ] column_definition
```

```
ALTER [ COLUMN ] column_name
{ SET default-definition | DROP DEFAULT }7
```

- Removing an existing column via the syntax:

```
DROP [ COLUMN ] column_name { RESTRICT | CASCADE }
```

- Adding to an existing set of constraints via the syntax:

```
ADD table_constraint_definition
```

- Removing a named constraint via the syntax:

```
DROP CONSTRAINT constraint_name { RESTRICT | CASCADE }
```

Suppose we want to add a column to the base table PATIENT to store the phone number of every patient. The SQL/DDL code to do this is as follows:

```
ALTER TABLE patient ADD Pat_phone# char (10);
```

---

<sup>6</sup>Similar to the ON DELETE clause, SQL/DDL offers an ON UPDATE clause for referential triggered action that is intended for specifying action to be taken when the referenced attribute(s) value (primary key or alternate key value) in a foreign key constraint is changed. Since in our example this is not specified, we defaulted to an assumption of same as ON DELETE clause specifications.

<sup>7</sup>Braces { } are used to specify that one of the items from the list of items separated by the vertical bar must be chosen.

Now, the rows of the base table PATIENT are capable of receiving values for the **Pat\_phone#** column. Since no default has been specified for the new column added to the table, the rows of PATIENT will, by default, have “null” value for the column, **Pat\_phone#**. Clearly, it is not possible to specify a “not null” constraint on the column until either a non-null default value is specified for the column or the column is populated with non-null values in all rows of the base table.

The column can be removed from the base table by either of these two statements:

```
ALTER TABLE patient  DROP Pat_phone# CASCADE;
```

or:

```
ALTER TABLE patient  DROP Pat_phone# RESTRICT;
```

Observe the definition of DROP behavior (i.e., CASCADE or RESTRICT) in the SQL/DML statement. The SQL-2003 standard requires the DROP behavior definition. The CASCADE option implies that all constraints and derived tables that reference the column also be dropped from the database schema. Likewise, the RESTRICT option prevents the dropping of the column, should any schema element that references the column exist. Also, SQL-2003 provides for the dropping of only one column per ALTER statement.

Suppose we want to specify a default value of \$3.00 for the unit price of all medications. This can be done as follows:

```
ALTER TABLE medication  ALTER Med_unitprice SET DEFAULT 3.00;
```

The default clause can be removed by using this statement:

```
ALTER TABLE medication  ALTER Med_unitprice DROP DEFAULT;
```

#### 10.1.1.3 A Best Practice Hint

Let us take a look at two methods of specifying the domain constraints on the column **Pat\_age** in the base table PATIENT. Here is the first method:

```
Pat_age smallint CONSTRAINT nn_Patage not null,  
CONSTRAINT chk_age CHECK (Pat_age IN (1 through 90))
```

Since the naming of the constraint definition is optional, it is possible to write the above DDL using a second method, as follows:

```
Pat_age smallint not null CHECK (Pat_age IN (1 through 90))
```

Clearly, the second method code appears simpler and more concise. However, if we decide to permit null values for **Pat\_age**, in method 2, the entire column definition has to be re-specified. On the other hand, using method 1, we simply drop the “not null” constraint, as shown here:

```
ALTER TABLE patient  DROP CONSTRAINT nn_patage CASCADE;
```

or:

```
ALTER TABLE patient  DROP CONSTRAINT nn_patage RESTRICT;
```

This is possible only because we named the constraint. While the DBMS names every constraint when we don't, finding out the constraint name given by the DBMS is an inefficient task; and the constraint name given by the DBMS is not generally a user-friendly name. Thus, the coding technique of method 1 offers greater flexibility and is strongly recommended.

#### 10.1.1.4 Syntax of the DROP TABLE Statement

Just as we can create a base table and make certain changes to it, it is also possible to remove a base table (structure and content) from the database using the **DROP TABLE** SQL/DDL statement:

```
DROP TABLE table_name drop behavior;
```

where the following apply:

- *table\_name* is the name for the base table being deleted.
- The *drop behaviors* possible are CASCADE and RESTRICT.

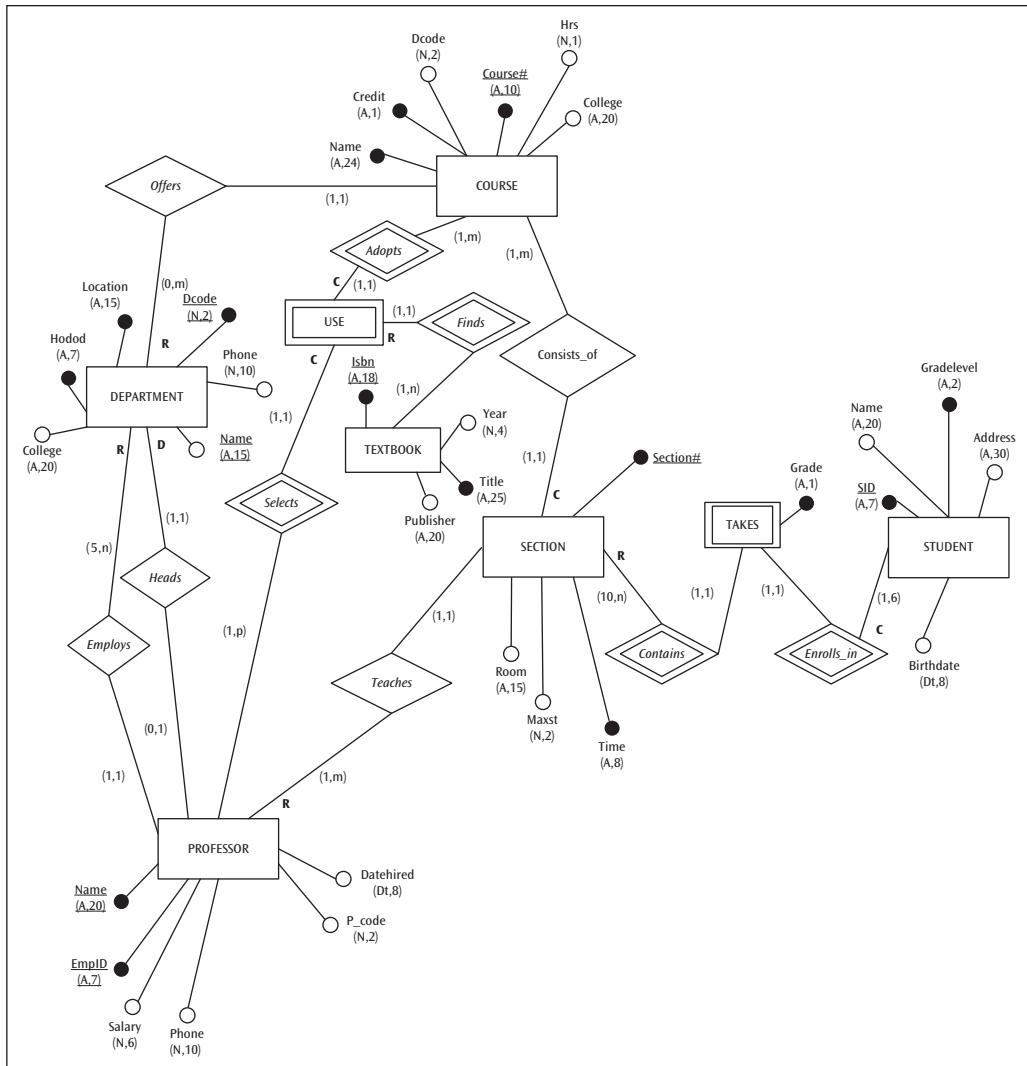
For example, the statement:

```
DROP TABLE medication CASCADE;
```

not only deletes the MEDICATION table (all rows and the table definition) from the database, it also removes all schema elements (constraints and derived tables) that reference any column of the MEDICATION table. For instance, the constraint definition, **fk\_med** in the base table ORDERS, defines the foreign key constraint that references a candidate key of the MEDICATION table. The CASCADE option in the DROP TABLE statement automatically drops the constraint **fk\_med** in ORDERS when the MEDICATION table is dropped. The RESTRICT option, on the other hand, disallows the deletion of the MEDICATION table because the constraint definition, **fk\_med**, exists in the schema.

#### 10.1.1.5 A Comprehensive Example

Figure 10.2 contains a Design-Specific ERD for Madeira College that emerged as a result of using the conceptual modeling process described in Part I of this book. This database is used to keep track of the courses offered by departments, the enrollment of students in courses, and the teaching-related activities of professors.



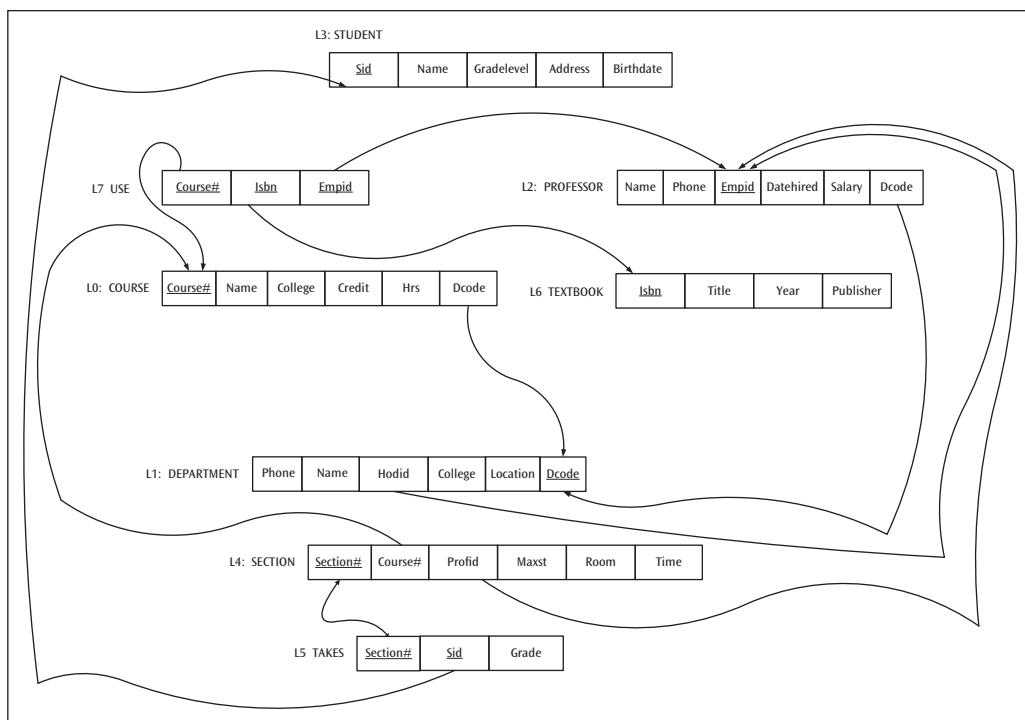
**FIGURE 10.2** A Design-Specific ERD for the Madeira College registration system

A brief description of the various relationships follows:

- Departments can offer a variety of courses, with each offered by a single department.
- Each department must have at least five professors, one of whom serves as the department head.
- Each course consists of at least one section.
- Each student must enroll in (i.e., take) at least one but no more than six courses.
- In order for a section of a course to be offered (i.e., exist), it must have a minimum of 10 students.

- Each professor selects at least one textbook to be used in each course he or she teaches, at least one textbook is adopted for each course offered, and each textbook is used in at least one course.

The relational schema (information-reducing logical schema) shown in Figure 10.3 and the information-preserving logical schema that appears in Figure 10.4 result from the logical data modeling process discussed in Chapter 6.<sup>8</sup> The implementation of this logical schema takes the form of the SQL/DDL script for the Madeira College registration system in Box 4. Observe how the physical schema in Box 4 captures the constraints defined in the information-preserving logical schema.

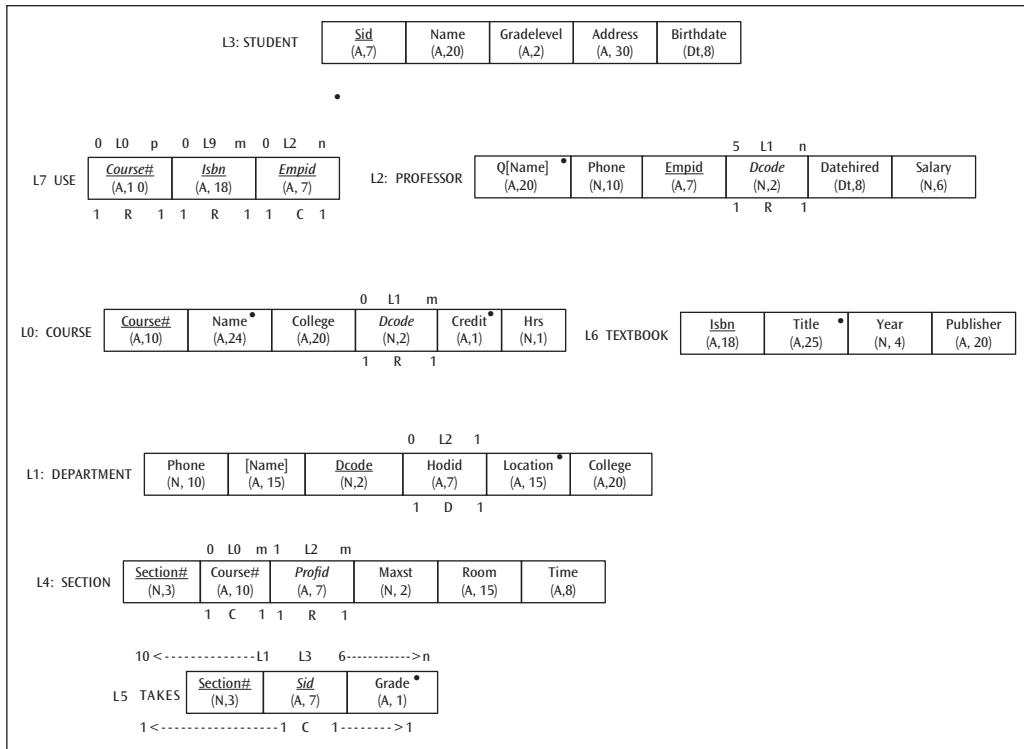


**FIGURE 10.3** Information-reducing logical schema for the Madeira College registration system

<sup>8</sup>In the interest of shortening the column names used throughout this chapter, the guidelines for naming attributes given in Section 6.2 are not used in Figure 10.3, Figure 10.4, and Box 4.

## Chapter 10

522



**FIGURE 10.4** Information-preserving logical schema for the Madeira College registration system

```

CREATE TABLE PROFESSOR
(NAME CHAR(20) CONSTRAINT NN_3NAME UNIQUE,
EMPID VARCHAR(7) CONSTRAINT PK_3COU PRIMARY KEY,
PHONE NUMBER(10),
DATEHIRED DATE,
DCODE NUMBER(2) CONSTRAINT NN_3DCOD NOT NULL,
SALARY NUMBER(6));

CREATE TABLE DEPARTMENT
(NAME CHAR(15),
DCODE NUMBER(2) CONSTRAINT PK_2DEPT PRIMARY KEY,
CONSTRAINT CK_2COD CHECK(DCODE BETWEEN 1 AND 10),
COLLEGE CHAR(20),
PHONE NUMBER(10) CONSTRAINT UNQ_PHONE UNIQUE,
LOCATION CHAR(15) CONSTRAINT NN_2LOC NOT NULL,
HODID VARCHAR(7) CONSTRAINT NN_2HOD NOT NULL,
CONSTRAINT UNQ_2HOD UNIQUE(HODID),
CONSTRAINT FK_2DEPT FOREIGN KEY(HODID) REFERENCES PROFESSOR(EMPID),
CONSTRAINT UNQ_2DNCOL UNIQUE(NAME, COLLEGE));

ALTER TABLE PROFESSOR ADD
CONSTRAINT FK_3DEPT FOREIGN KEY(DCODE) REFERENCES DEPARTMENT(DCODE);

CREATE TABLE COURSE
(NAME CHAR(24) CONSTRAINT NN_1NAME NOT NULL,
COURSE# CHAR(10) CONSTRAINT PK_1COU PRIMARY KEY,
CREDIT CHAR(1) CONSTRAINT NN_1CRED NOT NULL,
CONSTRAINT CK_1CRED CHECK(UPPER(CREDIT) IN ('U','G')),
COLLEGE CHAR(20) CONSTRAINT CK_1COL CHECK(COLLEGE IN ('ARTS AND SCIENCES','EDUCATION',
'ENGINEERING','BUSINESS')),
HRS NUMBER(1),
DCODE NUMBER(2) CONSTRAINT NN_1DCOD NOT NULL,
CONSTRAINT UNQ_1COLNAM UNIQUE(NAME, COLLEGE),
CONSTRAINT FK_1DEP FOREIGN KEY(DCODE) REFERENCES DEPARTMENT(DCODE),
CONSTRAINT CK_1HRS CHECK((UPPER(CREDIT) IN ('U')) AND (HRS<=4)) OR
((UPPER(CREDIT) IN ('G')) AND (HRS=3)));

CREATE TABLE STUDENT
(SID VARCHAR(7) CONSTRAINT PK_4STD PRIMARY KEY,
NAME CHAR(20),
ADDRESS CHAR(30),
BIRTHDATE DATE,
GRADELEVEL CHAR(2));

CREATE TABLE SECTION
(SECTION# VARCHAR2(8),
TIME CHAR(5) CONSTRAINT NN_5TIM NOT NULL,
MAXST NUMBER(2) CONSTRAINT CK_5SIZ CHECK(MAXST <= 35),
ROOM VARCHAR(15),
COURSE# CHAR(10),
CONSTRAINT FK_5COU FOREIGN KEY(COURSE#) REFERENCES COURSE(COURSE#) ON DELETE CASCADE,
PROFID VARCHAR2(9) CONSTRAINT NN_5PROF NOT NULL,
CONSTRAINT PK_5SEC PRIMARY KEY(SECTION#),
CONSTRAINT FK_5DEPT FOREIGN KEY(PROFID) REFERENCES PROFESSOR(EMPID),
CONSTRAINT UNQ_5SECT UNIQUE(SECTION#, TIME, ROOM));

CREATE TABLE TEXTBOOK
(ISBN VARCHAR(14) CONSTRAINT PK_10TB PRIMARY KEY,
TITLE VARCHAR(22) CONSTRAINT NN_10TIT NOT NULL,
YEAR NUMBER(4),
PUBLISHER CHAR(13) CONSTRAINT CK_10PUB CHECK(PUBLISHER IN ('Thomson','Springer', 'Prentice-

```

## Box 4

```
Hall'));
```

```
CREATE TABLE USES
(COURSE#      CHAR(10),
ISBN         VARCHAR(14),
EMPID       VARCHAR(8),
CONSTRAINT PK_12USE PRIMARY KEY(COURSE#,ISBN,EMPID),
CONSTRAINT FK_12AUSE FOREIGN KEY(COURSE#) REFERENCES COURSE(COURSE#),
CONSTRAINT FK_12BUSE FOREIGN KEY(ISBN) REFERENCES TEXTBOOK(ISBN),
CONSTRAINT FK_12CUSE FOREIGN KEY(EMPID) REFERENCES PROFESSOR(EMPID));
```

```
CREATE TABLE TAKES
(SECTION#  VARCHAR2(8),
GRADE     CHAR(5)  CONSTRAINT NN_14GRADE NOT NULL,
              CONSTRAINT CK_14GRADE CHECK(UPPER(GRADE) IN ('A','B','C')),
SID        VARCHAR(7),
              CONSTRAINT FK_14SID FOREIGN KEY(SID) REFERENCES STUDENT(SID) ON DELETE
CASCADE,
              CONSTRAINT FK_14TAK FOREIGN KEY(SECTION#) REFERENCES SECTION(SECTION#),
CONSTRAINT PK_14TAK PRIMARY KEY(SECTION#,SID));
```

524

Box 4 (continued)

## 10.2 DATA POPULATION USING SQL

Every database is subject to continual change. Three SQL statements (INSERT, DELETE, and UPDATE) are used for data insertion, deletion, and modification. This section discusses these three statements in the context of the PATIENT, MEDICATION, and ORDERS tables introduced in Section 10.1.1. For convenience of the reader, the SQL/DDL script from Box 3 in Section 10.1.1.1 is reproduced as Box 5.

```

CREATE TABLE patient
(Pat_p#a      char (2),
Pat_p#n      char (5),
Pat_name      varchar (41) constraint nn_Patnm not null,
Pat_gender    char (1),
Pat_age       smallint constraint nn_Patage not null,
Pat_admit_dt date constraint nn_Patadmt not null,
Pat_wing      char (1),
Pat_room#     integer,
Pat_bed       char (1),
CONSTRAINT pk_pat PRIMARY KEY (Pat_p#a, Pat_p#n),
CONSTRAINT chk_gender CHECK (Pat_gender IN ('M', 'F')),
CONSTRAINT chk_age CHECK (Pat_age IN (1 through 90)),
CONSTRAINT chk_bed CHECK (Pat_bed IN ('A', 'B'))
);

CREATE TABLE medication
(Med_code      char (5) CONSTRAINT nn_medcd not null CONSTRAINT unq_med UNIQUE,
Med_name      varchar (31) CONSTRAINT pk_med PRIMARY KEY,
Med_unitprice decimal (3,2) CONSTRAINT chk_unitprice CHECK (Med_unitprice < 4.50),
Med_qty_onhand integer CONSTRAINT nn_medqty not null,
Med_qty_onorder integer,
CONSTRAINT chk_qty CHECK ((Med_qty_onhand + Med_qty_onorder) BETWEEN 1000 AND 3000)
);

CREATE TABLE orders
(Ord_rx#      char (13) CONSTRAINT pk_ord PRIMARY KEY,
Ord_pat_p#a   char (2) CONSTRAINT nn_ord_pat_p#a not null,
Ord_pat_p#n   char (5) CONSTRAINT nn_ord_pat_p#n not null,
Ord_med_code  char (5) CONSTRAINT fk_med REFERENCES medication (Med_code)
ON DELETE RESTRICT ON UPDATE RESTRICT,
Ord dosage    smallint DEFAULT 2 CONSTRAINT chk_dosage CHECK (Ord dosage BETWEEN 1 AND 3),
Ord_freq      smallint DEFAULT 1 CONSTRAINT chk_freq CHECK (Ord_freq IN (1, 2, 3)),
CONSTRAINT fk_pat FOREIGN KEY (Ord_pat_p#a, Ord_pat_p#n)
REFERENCES patient (Pat_p#a, Pat_p#n) ON DELETE CASCADE ON UPDATE CASCADE
);

```

Box 5

### 10.2.1 The INSERT Statement

The SQL-2003 standard provides two types of **INSERT** statements to add new rows of data to a database:

- Single-row **INSERT**—adds a single row of data to a table
- Multi-row **INSERT**—extracts rows of data from another part of the database and adds them to a table

These statements take the following forms:

```

INSERT INTO <table-name> [(column-name {, column-name})]
VALUES (expression {, expression})

```

```
INSERT INTO <table-name> [ (column-name {, column-name}) ]
<select-statement>9
```

The values should be listed in the same order in which they are specified in the CREATE TABLE statement. The following three INSERT statements add one row to the PATIENT, MEDICATION, and ORDERS tables. Note that SQL allows us to omit the column names from the INSERT statement when assigning a value to each column in the table.

```
INSERT INTO PATIENT VALUES ('DB','77642','Davis, Bill', 'M', 27, '2013-07-07', 'B',
108, 'B') ;10  

1 row created.  

INSERT INTO MEDICATION VALUES ('TAG', 'Tagament', 3.00, 3000, 0);  

1 row created.  

INSERT INTO ORDERS VALUES ('104', 'DB', '77642', 'TAG', 3, 1);  

1 row created.
```

Each of these INSERT statements is successful only because each honors the declarative constraints established in the respective CREATE TABLE statements.

It is also permissible for an INSERT statement to specify explicit column names that correspond to the values provided in the INSERT statement. This is useful if a table has a number of columns but only a few columns are assigned values in a particular new row. Example 1 inserts a row into the PATIENT table that contains only the patient number, patient name, age, and date of admission. In this case, specification of the column names is required.

#### Example 1.

```
INSERT INTO PATIENT (PATIENT.PAT_P#A, PATIENT.PAT_P#N, PATIENT.PAT_NAME,
PATIENT.PAT_AGE, PATIENT.PAT_ADMIT_DT) VALUES ('GD','72222','Grimes, David',
44, '2013-07-12');  

1 row created.
```

This INSERT statement was successful because each of the columns not listed in the INSERT statement permits null values.<sup>11</sup> For example, had an attempt been made to insert a patient without specifying a date of admission, the INSERT statement would have failed.

---

<sup>9</sup>The SQL SELECT statement is used to retrieve (i.e., query) data from tables. In its simplest form, SELECT \* FROM *table\_name*, all columns from the *table\_name* listed are retrieved. The remainder of this section contains several examples that make use of this form of the SQL SELECT statement. Chapter 12 contains an extensive discussion of the SQL SELECT statement.

<sup>10</sup>The '2014-07-07' character string represents the date of admission of the patient. For a date data type, SQL-2003 uses a default date format where the first four digits represent the year component, the next two digits (1-12) represent the month component, and the final two digits (as constrained by the rules of the Gregorian calendar) represent the day of the month (see Table 10.1). Other formats for representing dates are covered in Section 13.1 of Chapter 13.

<sup>11</sup>Though not illustrated here, it is also permissible to omit columns with a DEFAULT value. If a DEFAULT exists for a column not explicitly listed in the INSERT statement, the default value will also be included for this column when the row is inserted.

Thus, every row in the PATIENT table must contain a patient name, age, and date of admission. In addition, since the patient number consisting of the combination of **PATIENT.Pat\_p#a** and **PATIENT.Pat\_p#n** constitutes the primary key, these two columns must be defined as well.

Each order must involve both an existing patient and existing medication. Observe what happens in Example 2 when an attempt is made to insert an order for an existing patient (David Grimes) but nonexistent medication (KEF).

### Example 2.

```
INSERT INTO ORDERS VALUES ('109', 'GD', '72222', 'KEF', 1, 1);
integrity constraint FK_MED violated-parent key not found
```

Note that FK\_MED (see Box 5) is the name of the referential integrity constraint requiring each medication code in the ORDERS table to exist in the MEDICATION table.

The multi-row INSERT statement adds multiple rows of data to a table via the execution of a query. In this form of the INSERT statement, the data values for the new rows appear in a SELECT statement specified as part of the INSERT statement. Suppose, for example, separate patient tables exist for different hospitals within the same hospital system. The INSERT statement in Example 3 inserts all rows in the PATIENT\_SUGARLAND table into the PATIENT table. Since the PATIENT\_SUGARLAND table has only six columns while the PATIENT table has nine columns, the column names are specified in the INSERT statement.

### Example 3.

```
INSERT INTO PATIENT
(PATIENT.PAT_P#a, PATIENT.PAT_P#N, PATIENT.PAT_NAME, PATIENT.PAT_GENDER,
PATIENT.PAT_AGE,
PATIENT.PAT_ADMIT_DT)
SELECT * FROM PATIENT_SUGARLAND;
3 rows created.
SELECT PAT_P#a, PAT_P#N, PAT_NAME, PAT_GENDER, PAT_AGE, PAT_ADMIT_DT
FROM PATIENT;12
```

PAT_P#a	PAT_P#N	PAT_NAME	PAT_GENDER	PAT_AGE	PAT_ADMIT_DT
DB	77642	Davis, Bill	M	27	2013-07-07
GD	72222	Grimes, David		44	2013-07-12
LH	97384	Lisauckis, Hal	M	69	2014-06-06
HJ	99182	Hargrove, Jan	F	21	2014-05-25
RN	31678	Robbins, Nancy	F	57	2014-06-01

Had there been an interest in inserting only those rows in the PATIENT\_SUGARLAND table with a date of admission after June 1, 2014, a WHERE clause referencing the appropriate column name in the PATIENT\_SUGARLAND table could have been added to the SELECT statement in the INSERT statement given in Example 3.

<sup>12</sup>The “SELECT <column list> FROM <table list>” form of the SQL SELECT statement is formally introduced in Chapter 12.

### 10.2.2 The DELETE Statement

The **DELETE** statement removes selected rows of data from a single table and takes the following form:

```
DELETE FROM <table-name> [WHERE <search-condition>]
```

Since the WHERE clause in a DELETE statement is optional, a DELETE statement of the form **DELETE FROM <table-name>** can be used to delete all rows in a table. When used in this manner, while the target table has no rows after execution of the deletion, the table still exists and new rows can still be inserted into the table with the **INSERT** statement. To erase the table definition from the database, the **DROP TABLE** statement (described in Section 10.1.1.4) must be used.

Care must be exercised when using the **DELETE** statement. While rows are deleted from only one table at a time, the deletion may propagate to rows in other tables if actions are specified in the referential integrity constraints. For example, the constraint in the ORDERS table:

```
constraint fk_pat foreign key (Ord_pat_p#a, Ord_pat_p#n)
    references patient (Pat_p#a, Pat_p#n) ON DELETE CASCADE ON UPDATE CASCADE
```

results in the deletion of all orders for a particular patient when that patient is deleted from the PATIENT table. The **DELETE** statement in Example 1 illustrates the propagation of a deletion to another table by deleting the first patient inserted into the PATIENT table, Bill Davis. Observe the content of the PATIENT and ORDERS tables before and after the deletion.

#### Content of Tables Prior to Deletion

```
SELECT PAT_P#a, PAT_P#N, PAT_NAME, PAT_GENDER, PAT_AGE, PAT_ADMIT_DT
FROM PATIENT;
PAT_P#a  PAT_P#N  PAT_NAME          PAT_GENDER  PAT_AGE  PAT_ADMIT_DT
-----
DB        77642   Davis, Bill       M           27        2013-07-07
GD        72222   Grimes, David     M           44        2013-07-12
LH        97384   Lisauckis, Hal    M           69        2014-06-06
HJ        99182   Hargrove, Jan    F           21        2014-05-25
RN        31678   Robbins, Nancy    F           57        2014-06-01

SELECT * FROM MEDICATION;13
MED_CODE  MED_NAME          MED_QTY_ONHAND  MED_QTY_ONORDER  MED_UNITPRICE
-----
TAG       Tagament            0              3000                3

SELECT * FROM ORDERS;
ORD_RX#    ORD_PAT_P#a  ORD_PAT_P#N  ORD_MED_CODE ORD_DOSAGE  ORD_FREQ
-----
104       DB             77642      TAG          3           1
```

---

<sup>13</sup>The requirement that each medication be associated with at least five orders (see Figure 10.1a) has been changed to only one order for this example.

**Example 1.**

```
DELETE FROM PATIENT WHERE PATIENT.PAT_NAME LIKE '%Davis, Bill%';14
```

1 row deleted.

**Content of Tables After Deletion**

```
SELECT PAT_P#A, PAT_P#N, PAT_NAME, PAT_GENDER, PAT_AGE, PAT ADMIT DT
FROM PATIENT;
PAT_P#A  PAT_P#N  PAT_NAME          PAT_GENDER  PAT_AGE  PAT ADMIT DT
-----
GD       72222   Grimes, David      44         2013-07-12
LH       97384   Lisauckis, Hal      M          69         2014-06-06
HJ       99182   Hargrove, Jan      F          21         2014-05-25
RN       31678   Robbins, Nancy     F          57         2014-06-01

SELECT * FROM MEDICATION;
MED_CODE  MED_NAME          MED_QTY_ONHAND  MED_QTY_ONORDER  MED_UNITPRICE
-----
TAG      Tagament            0              3000             3
```

SQL> SELECT \* FROM ORDERS;

no rows selected

On the other hand, observe the effect of the constraint in the ORDERS table:

```
Ord_med_code char(5) constraint fk_med references medication (Med_code)
ON DELETE RESTRICT ON UPDATE RESTRICT
```

when the attempt is made in Example 2 to delete a medication for which one or more orders exist. Assume that the rows previously deleted from the PATIENT and ORDERS tables have been reinserted prior to the execution of the DELETE statement that attempts to delete the Tagament medication from the MEDICATION table.

**Example 2.**

```
DELETE FROM MEDICATION WHERE MEDICATION.MED_CODE = 'TAG';
integrity constraint (FK_MED) violated - child record found
```

Note that FK\_MED is the name of the referential integrity constraint requiring each medication code in the ORDERS table to exist in the MEDICATION table and restricting the deletion of a medication with one or more orders.

---

<sup>14</sup>The LIKE operator and the percent character (%) are used for pattern matching. Pattern matching in SQL is discussed in Section 12.1.6 of Chapter 12.

### 10.2.3 The UPDATE Statement

The UPDATE statement modifies the values of one or more columns in selected rows of a single table. The UPDATE statement takes the following form:

```
UPDATE <table-name>
SET column-name = expression
    {, column-name = expression}
[WHERE <search-condition>]
```

The SET clause specifies which columns are to be updated and calculates the new values for the columns.

As illustrated in Example 1, it is important that an UPDATE statement not violate any existing constraints.

530

#### Example 1.

```
UPDATE MEDICATION SET MEDICATION.MED_UNITPRICE = 5.00
WHERE MEDICATION.MED_CODE = 'TAG';
```

The UPDATE statement in this example violates the check constraint CHK\_UNITPRICE and thus generates the following message:

```
check constraint (CHK_UNITPRICE) violated
```

Several rows can be modified by a single UPDATE statement. As an example, suppose the MEDICATION table now contains the following six rows:

SELECT * FROM MEDICATION;				
MED_CODE	MED_NAME	MED_UNITPRICE	MED_QTY_ONHAND	MED_QTY_ONORDER
TAG	Tagament	3	3000	0
VIB	Vibramycin	1.5	1700	300
KEF	Keflin	2.5	900	410
ASP	Aspirin	.02	3000	0
PCN	Penicillin	.4	2700	0
VAL	Valium	.75	2100	0

Observe the effect of the UPDATE statement in Example 2 designed to add 500 to the quantity on hand for each medication with a unit price greater than 0.50.

**Example 2.**

```
UPDATE MEDICATION
SET MEDICATION.MED_QTY_ONHAND = MEDICATION.MED_QTY_ONHAND + 500
WHERE MEDICATION.MED_UNITPRICE > 0.50;
check constraint (CHK_QTY) violated

SELECT * FROM MEDICATION;
MED_CODE  MED_NAME          MED_UNITPRICE  MED_QTY_ONHAND  MED_QTY_ONORDER
-----  -----
TAG      Tagament           3              3000            0
VIB      Vibramycin         1.5             1700            300
KEF      Keflin             2.5             900             410
ASP      Aspirin            .02             3000            0
PCN      Penicillin          .4              2700            0
VAL      Valium             .75             2100            0
```

While the CHK\_QTY constraint requiring the quantity on hand plus the quantity on order is violated for only one of the four otherwise qualifying rows, none of the four rows is updated. When the WHERE clause excludes Tagament, as shown in Example 3, the UPDATE is successful.

**Example 3.**

```
UPDATE MEDICATION
SET MEDICATION.MED_QTY_ONHAND = MEDICATION.MED_QTY_ONHAND + 500
WHERE MEDICATION.MED_UNITPRICE > 0.50 AND MEDICATION.MED_CODE <> 'TAG';

3 rows updated.
```

```
SELECT * FROM MEDICATION;
MED_CODE  MED_NAME          MED_UNITPRICE  MED_QTY_ONHAND  MED_QTY_ONORDER
-----  -----
TAG      Tagament           3              3000            0
VIB      Vibramycin         1.5             2200            300
KEF      Keflin             2.5             1400            410
ASP      Aspirin            .02             3000            0
PCN      Penicillin          .4              2700            0
VAL      Valium             .75             2600            0
```

## Chapter Summary

---

The SQL-2003 standard contains a data definition language (SQL/DDL) that allows database objects to be created. Section 10.1 focuses primarily on the CREATE TABLE statement as the vehicle for defining required data by specifying the proper data type and using, where appropriate, the NOT NULL clause, domain constraints (defined by either the CHECK clause), entity integrity (via the PRIMARY KEY clause), referential integrity (via the FOREIGN KEY clause along with, as appropriate, update and deletion rules), and row-level constraints (defined by the CHECK and UNIQUE clauses).

The discussion highlights the efficacy of using a logical schema based on the information-preserving grammar in SQL-2003 SQL/DDL to create tables that fully capture all information contained in the Design-Specific ER model.

SQL-2003 DDL includes an ALTER TABLE statement as well as a DROP TABLE statement. The ALTER TABLE statement is used to add or drop a column, add or drop a constraint, or to modify a column definition. The DROP TABLE statement is used to remove a table (structure and content) from the database. Both the ALTER TABLE statement, if it involves some kind of drop action, and the DROP TABLE statement must specify a drop behavior associated with the action (i.e., dropping a column or constraint in the alteration of a table or the dropping of an entire table). The options available in both cases are RESTRICT or CASCADE. RESTRICT implies that the action is rejected if any other object referencing the base table that is the subject of the ALTER TABLE statement or DROP TABLE statement exists. On the other hand, the CASCADE option deletes the object (column, constraint, or base table) along with all references to the object.

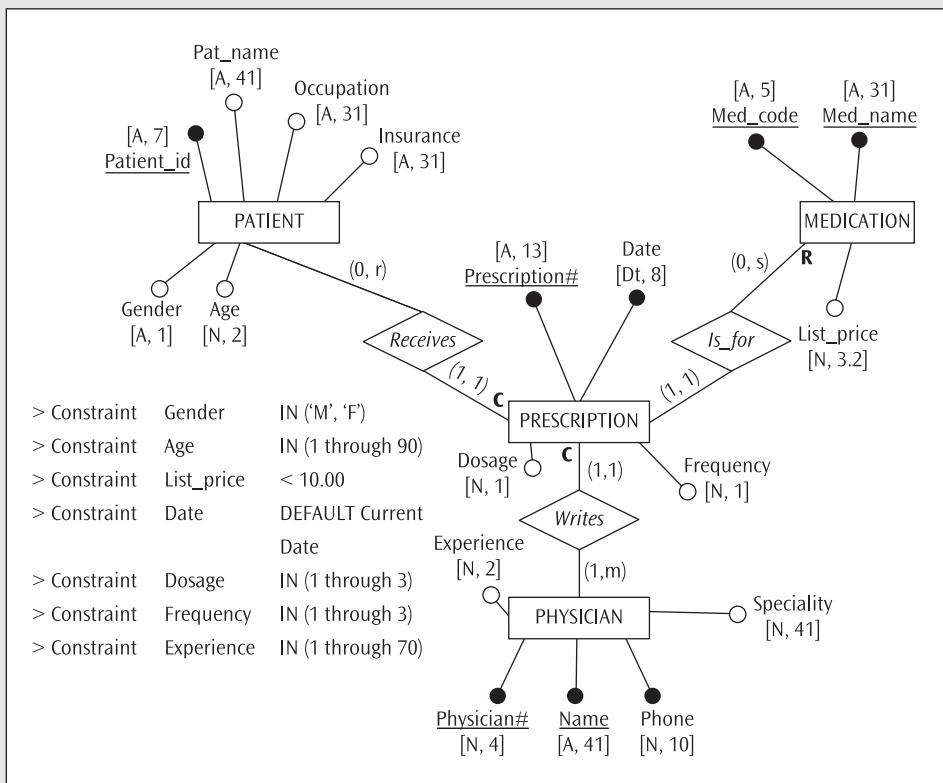
Section 10.2 covers the three statements that can be used to modify the database: INSERT, DELETE, and UPDATE. Two types of INSERT statements exist. One allows for the addition of a single row to a table, while the other allows for multiple rows to be added to a table. Only one type of DELETE and UPDATE statement exists. However, with each statement it is possible to delete (in the case of the DELETE statement) or update (in the case of the UPDATE statement) one or more rows in the table.

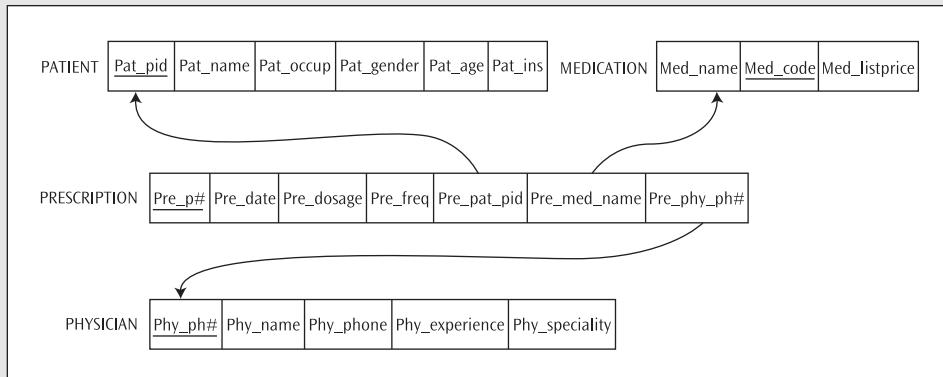
## Exercises

---

1. Discuss the differences between a relation and a table.
2. What are the minimum elements that must be included in the CREATE TABLE statement in defining the structure of a table?
3. What is the difference between a column-level constraint and a row-level constraint?
4. Describe the SQL clauses used in the definition of:
  - a. a primary key constraint
  - b. an alternate key constraint
  - c. a foreign key constraint
  - d. a check constraint
5. With which of the four types of constraints in Exercise 4 is a requirement that an attribute not contain a null value associated?

6. With which type of constraint is the specification of a deletion rule associated?
7. Describe the types of modifications that can be made to a table using the ALTER TABLE statement.
8. Consider a relational schema with three relation schemas: DRIVER (Dr\_license\_no, Dr\_name, Dr\_city, Dr\_state), TICKET\_TYPE (Ttp\_offense, Ttp\_fine), and TICKET (Tic\_ticket\_no, Tic\_ticket\_date, Tic\_dr\_license\_no, Tic\_ttp\_offense). Assume that Dr\_license\_no is a character data type of size 7, Dr\_name, Dr\_city, and Ttp\_offense are character data types of size 20, Dr\_state is a character data type of size 2, Tic\_ticket\_no is a character data type of size 5, and Ttp\_fine is an integer between 15 and 150. Each tuple in the TICKET relation corresponds to a ticket received by a driver for having been issued a specific type of ticket on a given date for committing a given offense. Use SQL/DDL to create a schema of these relations, including the appropriate primary key, integrity (e.g., not null), and foreign key constraints. In addition, include the definition of domains for each of the distinct attribute types.
9. The series of tasks in this exercise is based on the following ER diagram and its associated logical schema.





534

L1: PATIENT	Pat_pid (A, 7)	Pat_name (A, 41)	Pat_occup (A, 31)	Pat_gender (A, 1)	Pat_age (N, 2)	Pat_ins (A, 31)	L2: MEDICATION	Q[Med_name] (A, 31)	Med_code (A, 5)	Med_listprice (N, 3.2)
L3: PRESCRIPTION	Pre_p# (A, 13)	Pre_date (Dt, 8)	Pre_dosage (N, 1)	Pre_freq (N, 1)	Pre_pat_pid (A, 7)	Pre_med_code (A, 5)	Pre_phy_ph# (N, 4)	0 L1 r 0 L2 s 1 L4 m	1 C 1 1 R 1 1 C 1	
L4: PHYSICIAN	Phy_ph# (N, 4)	Q[Phy_name] (A, 41)	Phy_phone (N, 10)	Phy_experience (N, 2)	Phy_speciality (N, 41)					

- Write appropriate CREATE TABLE statements for the logical schema. Be sure to define all appropriate constraints.
- Write an ALTER TABLE statement to add to the MEDICATION table the attribute unit cost that represents the per unit cost of the medication. The unit cost of a medication can range from \$0.50 to \$7.50.
- Write an ALTER TABLE statement that imposes the business rule that the list price of a medication must be at least 20 percent higher than its unit cost.
- Write an ALTER TABLE statement to drop the occupation attribute from the PATIENT table.

10. You must have completed Exercise 9 before beginning this exercise, and thus have used the SQL Data Definition Language to create tables for the three relations DRIVER, TICKET\_TYPE, and TICKET. Use the SQL INSERT statement to populate these tables with the following data.

Dr_license_no	Dr_name	Dr_city	Dr_state
MVX 322	E. Mills	Waller	TX
RVX 287	R. Brooks	Bellaire	TX
TGY 832	L. Silva	Sugarland	TX
KEC 654	R. Lence	Houston	TX
MQA 823	E. Blair	Houston	TX
GRE 720	H. Newman	Pearland	TX

535

Ttp_offense	Ttp_fine
Parking	15
Red Light	50
Speeding	65
Failure To Stop	30

Tic_Ticket_no	Tic_ticket_date	Tic_dr_license_no	Tic_ttp_offense
1023	2014-12-20	MVX 322	Parking
1025	2014-12-21	RVX 287	Red Light
1397	2014-12-03	MVX 322	Parking
1027	2014-12-22	TGY 832	Parking
1225	2014-12-22	KEC 654	Speeding
1212	2014-12-06	MVX 322	Speeding
1024	2014-12-21	RVX 287	Speeding
1037	2014-12-23	MVX 322	Red Light

Note: When entering the date of the ticket, in the INSERT statement enclose the entire date in single quotes.

11. This exercise is based on the data sets associated with Figure 2.25 in Chapter 2.
- Use the SQL Data Definition Language to create a relational schema that consists of the following three relations:
- COMPANY (Co\_name, Co\_size, Co\_headquarters)
- STUDENT (St\_name, St\_major, St\_status)
- INTERNSHIP (In\_co\_name, In\_st\_name, In\_year, In\_qtr, In\_location, In\_stipend)
- When you create a table for each relation, in addition to defining its primary key, define all the appropriate referential integrity constraints. Assume that Co\_name is a character data type of size 5, Co\_size is an integer data type of size 4, Co\_headquarters is a character data type of size 10, St\_name is a Varchar data type of size 10, St\_major is a character data type of size 20, St\_status is a character data type of size 2, In\_co\_name is a character data type of size 5, In\_st\_name is a Varchar data type of size 10, In\_year is an integer data type of size 4, In\_qtr is a character data type of size 10, In\_location is a character data type of size 15, and In\_stipend is an integer data type of size 4. In\_stipend represents the monthly stipend associated with the internship.
- Use the SQL Insert statement to populate the three tables with the following data:

Co_name	Co_size	Co_headquarters
A	1000	Boston
B	500	Chicago
C	1000	Boston
D	400	Houston

St_name	St_major	St_status
Michelle	Communications	SR
Chris	Chemistry	JR
Andy	Finance	SO
Anna	Communications	SR
Amy	Communications	FR

In_co_name	In_st_name	In_year	In_qtr	In_location	In_stipend
A	Chris	2014	Fall	Concord	1000
A	Anna	2014	Fall	Concord	1000
B	Chris	2014	Fall	Concord	600
C	Amy	2014	Fall	South Bend	900
D	Andy	2014	Spring	South Bend	1000
A	Chris	2013	Spring	Concord	1200
D	Anna	2014	Spring	Houston	

12. The purpose of this exercise is to give you an opportunity to create the tables for Bearcat Incorporated. The tables themselves are based on the relations that appear in the following figure:

```
L1: EMPLOYEE (Emp_fname, Emp_minit, Emp_lname, Emp_nameTag, Emp_emp_e#a, Emp_emp_e#n, Emp_address, Emp_pl_name, Emp_gender,
    Emp_dateHired, Emp_e#a, Emp_e#n)

# EMPLOYEE.{Emp_emp_e#a, Emp_emp_e#n} ⊆ EMPLOYEE.{Emp_e#a, Emp_e#n} or ∅
EMPLOYEE.{Emp_pl_name} ⊆ PLANT.{Pl_name}

L2: PLANT (Pl_p#, Pl_budget, Pl_name, Pl_emp_e#a, Pl_emp_e#n, Pl_mgrsId)
# PLANT.{Pl_emp_e#a, Pl_emp_e#n} ⊆ EMPLOYEE.{Emp_e#a, Emp_e#n}

L3: BUILDING (Bld_building, Bld_pl_p#)
# BUILDING.{Bld_pl_p#} ⊆ PLANT.{Pl_p#}

L4: PROJECT (Prj_name, Prj_location, Prj_p#, Prj_pl_p#)
# PROJECT.{Prj_pl_p#} ⊆ PLANT.{Pl_p#} or ∅

L5: ASSIGNMENT (Asq_prj_p#, Asq_emp_e#a, Asq_emp_e#n, Asq_hrs)
# ASSIGNMENT.{Asq_prj_p#} ⊆ PROJECT.{Prj_p#}
ASSIGNMENT.{Asq_emp_e#a, Asq_emp_e#n} ⊆ EMPLOYEE.{Emp_e#a, Emp_e#n}

L6: DEPENDENT (Dep_sex, Dep_bthdte, Dep_name, Dep_relhow, Dep_emp_e#a, Dep_emp_e#n)
# DEPENDENT.{Dep_emp_e#a, Dep_emp_e#n} ⊆ EMPLOYEE.{Emp_e#a, Emp_e#n}

L7: BCU_ACCOUNT (Bcu_dep_name, Bcu_dep_relhow, Bcu_dep_emp_e#a, Bcu_dep_emp_e#n, Bcu_acct_type, Bcu_acct#, Bcu_balance, Bk_emp_e#a,
    Bcu_emp_e#n)

# BCU_ACCOUNT.{Bcu_emp_e#a, Bcu_emp_e#n} ⊆ EMPLOYEE.{Emp_e#a, Emp_e#n} or ∅
BCU_ACCOUNT.{Bcu_dep_name, Bcu_dep_relhow, Bcu_dep_emp_e#a, Bcu_dep_emp_e#n} ⊆ DEPENDENT.{Dep_name, Dep_relhow, Dep_emp_e#a, Dep_emp_e#n} or ∅

L8: PARTICIPATION (Par_dep_name, Par_dep_relhow, Par_dep_emp_e#a, Par_dep_emp_e#n, Par_hob_name, Par_annCost, Par_hrsWeek)
# PARTICIPATION.{Par_hob_name} ⊆ HOBBY.{Hob_name}
PARTICIPATION.{Par_dep_name, Par_dep_relhow, Par_dep_emp_e#a, Par_dep_emp_e#n} ⊆ DEPENDENT.{Dep_name, Dep_relhow, Dep_emp_e#a, Dep_emp_e#n}

L9: HOBBY (Hob_name, Hob_loact, Hob_giact)
```

In addition to the primary key constraints shown in the figure, these tables contain the following constraints (i.e., business rules):

### **PLANT Table**

- No two plants can have the same name.
- Plant numbers are allowed to range between 10 and 20 inclusive.

### **EMPLOYEE Table**

- Each employee must have a first name and a last name.
- Employee salaries can range between \$35,000 and \$90,000 inclusive.
- Valid genders are “M” and “F.”
- Each employee must work in an existing plant.
- The supervisor of an employee must be an existing employee.
- No two employees can have the same first name, middle initial, last name, and name tag combination.

### **BUILDING Table**

- Each building must be part of an existing plant.

**PROJECT Table**

- Projects are located in the following cities: Bellaire, Blue Ash, Mason, Stafford, and Sugarland.
- Each project must be associated with an existing plant.
- Project numbers range from 1 to 40 inclusive.

**ASSIGNMENT Table**

- Each assignment must be associated with an existing employee and an existing project.

**DEPENDENT Table**

- The sex of a dependent can be: "M," "F," "m," or "f."
- A dependent must be a dependent of an existing employee.
- A dependent can be related to an employee in the following ways:
  - A dependent can be the employee's spouse.
  - A dependent who is a mother or daughter must be a female.
  - A dependent who is a father or son must be a male.

**BCU\_ACCOUNT Table**

- A bcu\_account can belong to an employee, a dependent, or (an employee and a dependent).
- Valid account types are "C," "S," or "I."

**HOBBY Table**

- Valid values for the indoor/outdoor attribute are "I" or "O,"
- Valid values for the group/individual attribute are "G" or "I."

**PARTICIPATION Table**

- A participation must involve an existing hobby and an existing dependent.

Once the tables have been created, they must be tested to make sure that the table and column definitions allow for entry of only valid data. For example, it should not be possible to insert two plants with the same name. Data to give the tables you have created a thorough test is stored in the file `insertdata.sql`<sup>15</sup>. If you have defined your constraints properly, some of these insert statements will successfully insert a row into a table. On the other hand, some of the insert statements should fail because the data they contain violate one of the constraints associated with the table.

At the end of running the test data through your table definitions, there should be 4 rows in the PLANT table, 7 rows in the BUILDING table, 13 rows in the EMPLOYEE table, 9 rows in the PROJECT table, 7 rows in the ASSIGNMENT table, 6 rows in the DEPENDENT table, 6 rows in the BCU\_ACCOUNT table, 7 rows in the HOBBY table, and 4 rows in the PARTICIPATION table.

*Note:* You need not make any changes to the `insertdata.sql` file. This file was "seeded" with errors in order to test whether your CREATE TABLE statements handle all of the necessary constraints.

<sup>15</sup>Insertdata.sql can be downloaded from [www.course.com](http://www.course.com) (search on the ISBN for this book) or obtained from your instructor.

# CHAPTER 11

## RELATIONAL ALGEBRA

The relational data model includes a group of basic data manipulation operations. As a result of its theoretical foundation in set theory, the relational data model's operations include Union, Intersection, and Difference. Five other relational operators also exist: Select, Project, Cartesian Product, Join, and Divide. Collectively, these eight operators comprise relational algebra.<sup>1</sup> This section discusses each of these eight operators and gives examples of their use in the formulation of queries. The examples are based on the Madeira College registration system introduced in Section 10.1.1.5. The Design-Specific ERD for Madeira College appears in Figure 10.2, and its information-reducing and information-preserving logical schema are shown in Figures 10.3 and 10.4, respectively. Figure 11.1 contains representative data for the DEPARTMENT, PROFESSOR, COURSE, and SECTION relations used in the relational algebra examples in this chapter along with representative data for other Madeira College relations used in conjunction with the SQL examples that begin in Chapter 12. In an effort to keep the amount of sample data used in the examples in Chapters 11, 12, and 13 reasonably small, not all cardinality ratio and participation constraints defined in the Design-Specific ERD in Figure 10.2 are reflected in the relations in Figure 11.1. In addition, in order to reduce the number of characters in various column names, the attribute naming convention introduced in Section 6.2 of Chapter 6 has been dropped. For example, the column names PR\_NAME, PR\_EMPID, PR\_PHONE, PR\_DATEHIRED, PR\_DPT\_CODE, and PR\_SALARY used in the PROFESSOR relation in Box 4 have been changed to NAME, EMPID, PHONE, DATEHIRED, DCODE, and SALARY in Chapter 11 and all of Chapters 12 and 13.<sup>2</sup>

The discussion of these relational algebra operators begins with the two that operate on a single relation (**unary operators**) followed by those that operate on two relations (**binary operators**). Figure 11.2 shows the fundamental relational algebra operators, along with their symbolic representations. In Figure 11.2, the fundamental operators are indicated by a double asterisk (\*\*). The rest can be defined from the fundamental operators. Nonetheless, these operators are usually included in relational algebra as a matter of convenience.

---

<sup>1</sup>The discussion of and notation used for various relational algebra operations in this chapter is based on R. Elmasri and S.B. Navathe (2010), *Fundamentals of Database Systems*, Pearson Education. The reader is encouraged to refer to Elmasri and Navathe (2010) and C. J. Date (2006), *An Introduction to Database Systems*, Pearson Education for a more in-depth discussion of relational algebra.

<sup>2</sup>The Madeira College relations contain a NAME column in the DEPARTMENT, COURSE, STUDENT and PROFESSOR tables. When referenced in either a relational algebra operation or an SQL query, the NAME column is always prefaced by the name of the relation or table. DCODE, COLLEGE, COURSE#, and SECTION# are other column names that appear in multiple relations/tables.

## Chapter 11

540

DEPARTMENT Relation					
NAME	DCODE	COLLEGE	PHONE	LOCATION	HODID
Economics	1	Arts and Sciences	5235567654	123 McMicken	FM49276
QA/QM	3	Business	5235566656	333 Lindner	BA54325
Economics	4	Education	5235569978	336 Dyer	CM65436
Mathematics	6	Engineering	5235564379	728 Old Chem	RR79345
IS	7	Business	5235567489	333 Lindner	CC49234
Philosophy	9	Arts and Sciences	5235565546	272 McMicken	CM87659
COURSE Relation					
NAME	COURSE#	CREDIT	COLLEGE	HRS	DCODE
Intro to Economics	15ECON112	U	Arts and Sciences	3	1
Operations Research	22QA375	U	Business	2	3
Intro to Economics	18ECON123	U	Education	4	4
Supply Chain Analysis	22QA411	U	Business	3	3
Principles of IS	22IS270	G	Business	3	7
Programming in C++	20ECE5212	G	Engineering	3	6
Optimization	22QA888	G	Business	3	3
Financial Accounting	18ACCT801	G	Education	3	4
Database Concepts	22IS330	U	Business	4	7
Database Principles	22IS832	G	Business	3	7
Systems Analysis	22IS430	G	Business	3	7
Architectural History	05ARCH101	U		3	
STUDENT Relation					
SID	NAME	ADDRESS	BIRTHDATE	GRADELEVEL	
BE76598	Elijah Baley	2920 Scioto Street		JR	
OD76578	Daniel Olive	338 Bishop Street	12-MAY-82	JR	
SW56547	Wanda Seldon	3138 Probasco	03-MAR-70	FR	
BG66765	Gladis Bale	356 Vine Street	23-OCT-77	FR	
GS76775	Shweta Gupta	356 Probasco	21-MAY-79	SO	
HT67657	Troy Hudson			JR	
FR45545	Rick Fox	314 Clifton	09-OCT-83	GR	
FV67733	Vanessa Fox	314 Clifton	20-OCT-83	SO	
HJ45633	Jenna Hopp	2930 Scioto Street	03-MAR-70	SO	
SD23556	David Sane	245 University Avenue	14-JUL-84	SR	
DT87656	Tim Duncan		21-MAY-75	SR	
KJ56656	Joumana Kidd	2920 Scioto Street		FR	
AJ76998	Jenny Aniston	88 MLK		SR	
KP78924	Poppy Kramer	437 Love Lane	11-NOV-80	JR	
KS39874	Sweety Kramer	748 Hope Avenue	11-NOV-80	JR	
JD35477	Diana Jackson	2920 Scioto Street	20-FEB-76	GR	
SECTION Relation					
SECTION#	TIME	MAXST	ROOM	COURSE#	PROFID
101A2014	T1015	25		22QA375	HT54347
901A2013	W1800	35	Rhodes 611	22IS270	SK85977
902A2013	H1700	25	Lindner 108	22IS270	SK85977
201S2013	T1045	29	Lindner 110	22IS330	SK85977
301S2013	H1045	29	Lindner 110	22IS330	CC49234
401W2014	M1000	33	Braunstien 211	22IS832	CC49234
102A2014	W1800		Baldwin 437	20ECE5212	RR79345
103U2014	T1015	33		22QA375	HT54347
104A2014	H1700	29	Lindner 108	22IS330	SK85977
105S2014	T1015	30		22QA375	HT54347
106W2014	T1015	20		22QA375	HT54347

© 2015 Cengage Learning®

**FIGURE 11.1** Madeira College relations

<b>PROFESSOR Relation</b>					
NAME	EMPID	PHONE	DATEHIRED	DCODE	SALARY
John Smith	SJ89324	5235567645	23-JUN-01	1	45000
Mike Faraday	FM49276	5235568492	01-MAY-96	1	92000
Kobe Bryant	BK68765	5235568522	01-MAY-98	1	66000
Ram Raj	RR79345	5235567244	23-JUN-01	6	44000
John B Smith	SJ65436	5235567556		6	
Prester John	JP77869	5235567244	25-AUG-95	6	44000
Chelsea Bush	BC65437	5235567777	01-MAY-93	3	77000
Tony Hopkins	HT54347	5235569977	20-JAN-97	3	77000
Alan Brodie	BA54325	5235569876	16-MAY-00	3	76000
Jessica Simpson	SJ67543	5235565567	25-AUG-95	3	67000
Laura Jackson	JL65436	5235565436	23-SEP-00	3	43000
Marie Curie	CM65436	5235569899	22-OCT-99	4	99000
Jack Nicklaus	NJ33533	5235566767	31-DEC-99	4	67000
John Nicholson	NJ43728	5235569999	22-JUN-03	4	99000
Sunil Shetty	SS43278	5235566764	28-JUN-93	7	64000
Katie Shef	SK85977	5235568765	06-JUN-97	7	65000
Cathy Cobal	CC49234	5235565345	23-JAN-01	7	45000
Jeanine Troy	TJ76546	5235565545		9	45000
Tiger Woods	WT65487	5235565563	14-NOV-03	9	
Mike Crick	CM87659	5235565569	30-MAY-02	9	69000

<b>TEXTBOOK Relation</b>					
ISBN	TITLE	YEAR	PUBLISHER		
000-66574998	Database Management	2007	Thomson		
003-6679233	Linear Programming	2005	Prentice-Hall		
001-55-435	Simulation Modeling	2009	Springer		
118-99898-67	Systems Analysis	2008	Thomson		
77898-8769	Principles of IS	2010	Prentice-Hall		
0296748-99	Economics For Managers	2009			
0296437-1118	Programming in C++	2010	Thomson		
012-54765-32	Fundamentals of SQL	2012			
111-11111111	Data Modeling	2013			

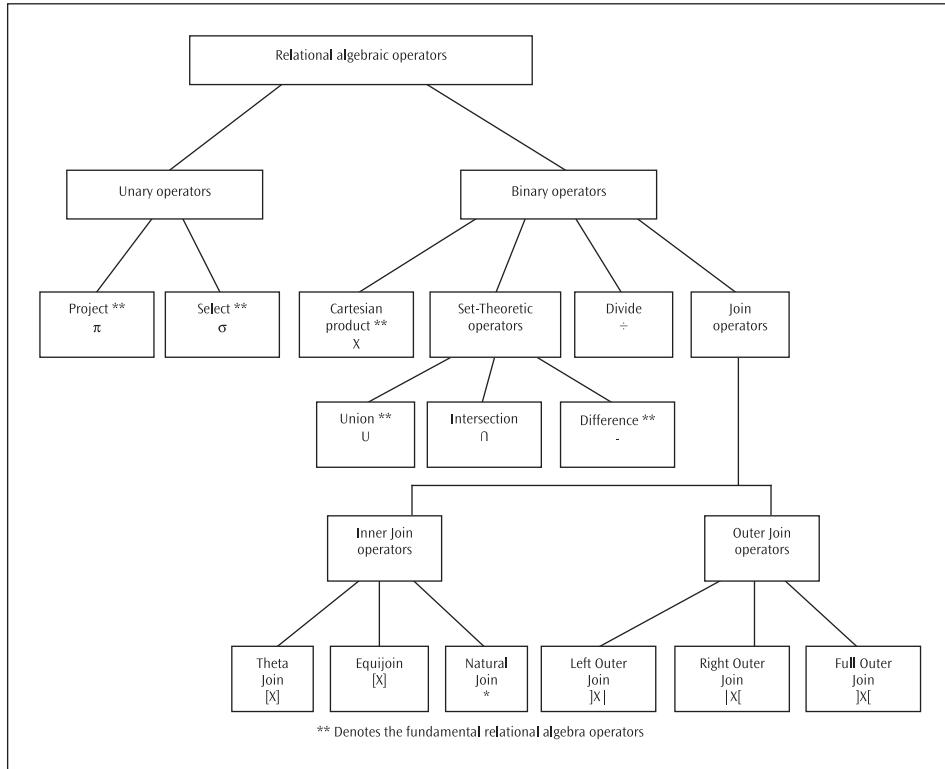
  

<b>TAKES Relation</b>					
SECTION#	GRADE	SID			
101A2014 A	KP78924				
101A2014 A	KS39874				
101A2014 B	BE66765				
201S2013 C	BE76598				
104A2014 B	KJ56656				
104A2014 A	KP78924				
104A2014 A	KS39874				
401W2014 A	KS39874				
104A2014 A	BE76598				
401W2014 B	BG66765				
104A2014 C	GS76775				

<b>USES Relation</b>					
COURSE#	ISBN	EMPID			
18ECON123	0296748-99	CM65436			
20ECE5212	0296437-1118	CC49234			
22IS270	000-66574998	SS43278			
22IS270	77898-8769	CC49234			
22IS270	77898-8769	SK85977			
22IS270	77898-8769	SS43278			
22IS330	003-6679233	BC65437			
22IS330	118-99898-67	SS43278			
22IS832	000-66574998	SS43278			
22IS832	118-99898-67	SK85977			
22QA375	0296437-1118	SJ65436			
22QA888	001-55-435	HT54347			

FIGURE 11.1 Madeira College relations (continued)



**FIGURE 11.2** Classification of relational algebra operators

## 11.1 UNARY OPERATORS

Unary operators “operate” on a single relation. There are two relational algebra unary operators: the Select operator and the Project operator.

### 11.1.1 The Select Operator

The **Select operator** is used to select a horizontal subset of the tuples that satisfy a selection condition from a relation. The general form of the Selection operation is:

$\sigma_{\text{selection condition}}(R)$

where the following apply:

- The symbol  $\sigma$  (sigma) designates the Select operator.
- The selection condition is a Boolean expression specified on the attributes of relation schema R.

R is generally a *relational algebra expression* whose result is a relation; the simplest expression is the name of a single relation. The relation resulting from the Selection operation

has the same attributes as R. The Boolean expression specified as <selection condition> is composed of a number of clauses of the following forms:

<attribute name><comparison operator><constant value>

or:

<attribute name><comparison operator><attribute name>

where the following apply:

- <attribute name> is the name of an attribute of R.
- <comparison operator> is normally one of the operators  $\{=, \neq, <, \leq, >, \geq\}$ .
- <constant value> is a constant value from the domain of the attribute.

Following are three examples of Selection operations performed on the COURSE relation shown in Figure 11.1.

**Selection Example 1.** Which courses are three-hour courses?

**Relational Algebra Syntax:**

$\sigma_{(\text{HRS} = 3)} (\text{COURSE})$

Observe that the result is an unnamed relation that contains a subset of the tuples in the COURSE relation.

**Result:**

NAME	COURSE#	CREDIT	COLLEGE	HRS	DCODE
Intro to Economics	15ECON112	U	Arts and Sciences	3	1
Supply Chain Analysis	22QA411	U	Business	3	3
Principles of IS	22IS270	G	Business	3	7
Programming in C++	20ECE5212	G	Engineering	3	6
Optimization	22QA888	G	Business	3	3
Financial Accounting	18ACCT801	G	Education	3	4
Database Principles	22IS832	G	Business	3	7
Systems Analysis	22IS430	G	Business	3	7
Architectural History	05ARCH101	U		3	

The Boolean operators AND, OR, and NOT can be used to form a general selection condition.

**Selection Example 2.** Which courses offered by Department 7 are three-hour courses?

**Relational Algebra Syntax:**

$\sigma_{(\text{DCODE} = 7 \text{ and } \text{HRS} = 3)} (\text{COURSE})$

Use of the logical operator AND requires that *both* conditions (**DCODE** = 7 and **HRS** = 3) be satisfied.

**Result:**

NAME	COURSE#	CREDIT	COLLEGE	HRS	DCODE
Principles of IS	22IS270	G	Business	3	7
Database Principles	22IS832	G	Business	3	7
Systems Analysis	22IS430	G	Business	3	7

**Selection Example 3.** Which courses are offered in either the College of Arts and Sciences or the College of Education?

**Relational Algebra Syntax:**

$\sigma_{(\text{COLLEGE} = \text{'Arts and Sciences'} \text{ or } \text{COLLEGE} = \text{'Education'})} (\text{COURSE})$

Use of the logical operator OR allows either condition (**COLLEGE** = ‘Arts and Sciences’ or **COLLEGE** = ‘Education’) to be satisfied.

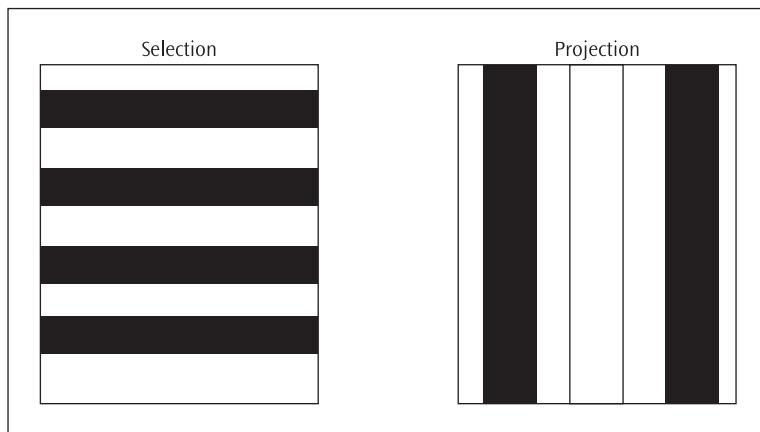
**Result:**

NAME	COURSE#	CREDIT	COLLEGE	HRS	DCODE
Intro to Economics	15ECON112	U	Arts and Sciences	3	1
Intro to Economics	18ECON123	U	Education	4	4
Financial Accounting	18ACCT801	G	Education	3	4

### 11.1.2 The Project Operator

544

While the Select operator selects some of the *tuples* from the relation while eliminating unwanted tuples, the **Project operator** selects certain *attributes* from the relation and eliminates unwanted attributes. In other words, a Selection operation forms a new relation by taking a *horizontal* subset of an existing relation, whereas a Projection operation forms a new relation by taking a *vertical* subset of an existing relation. The difference between Selection and Projection is shown pictorially in Figure 11.3.



**FIGURE 11.3** Selection compared to Projection

The general form of the Projection operation is:

$\pi_{<\text{attribute list}>} (R)$

where the following apply:

- The symbol  $\pi$  (pi) is used to represent the Project operator.
- $<\text{attribute list}>$  is a subset of the attributes of the relation schema R.

As was the case in the Selection operation, R, in general, is a *relational algebra expression* whose result is a relation, which in the simplest case is just the name of a single relation. The result of the Projection operation contains only the attributes specified in the <attribute list> in the same order as they appear in the list.

In cases where the attribute list produced as a result of the Projection operation is not a superkey of R, duplicate tuples are likely to occur in the result. Since relations are sets and do not allow duplicate tuples, only one copy of each group of identical tuples is included in the result of a Projection. A couple of examples of Projection operations based on the COLLEGE and DEPARTMENT relations in Figure 11.1 follow.

**Projection Example 1.** Which colleges offer courses?

**Relational Algebra Syntax:**

$$\pi_{(\text{COLLEGE})} (\text{COURSE})$$

Since **COLLEGE** is not a superkey of COURSE, the number of tuples in the result is less than the number of tuples in COURSE. There are five distinct values for the COLLEGE attribute: Engineering, Education, Arts and Sciences, Business, and the null value associated with the Architectural History course. This null value appears as the blank line in the following result.

**Result:**

COLLEGE
-----

Engineering
Education
Arts and Sciences
Business

**Projection Example 2.** What is the name and college of each department?

**Relational Algebra Syntax:**

$$\pi_{(\text{NAME}, \text{COLLEGE})} (\text{DEPARTMENT})$$

Since **[NAME, COLLEGE]** is a superkey of DEPARTMENT, the number of tuples resulting from a Projection operation on DEPARTMENT is equal to the number of tuples in DEPARTMENT.

**Result:**

NAME	COLLEGE
-----	-----
Economics	Arts and Sciences
QA/QM	Business
Economics	Education
Mathematics	Engineering
IS	Business
Philosophy	Arts and Sciences

## 11.2 BINARY OPERATORS

The relational algebra binary operators “operate” on two relations. There are four binary operators:

- The Cartesian Product operator
- Set theoretic operators
- Join operators
- The Divide operator

### 11.2.1 The Cartesian Product Operator

The **Cartesian Product operator** (often referred to as the Product or Cross-Product operator), denoted by  $\times$ , is used to combine tuples from any two relations in a combinatorial fashion. The Cartesian Product of relations R and S is created by (a) concatenating the attributes of R and S together, and (b) attaching to each tuple in R each of the tuples in S. Thus, if R has  $n_R$  tuples and S has  $n_S$  tuples, then the Cartesian Product Q will have  $n_R$  times  $n_S$  tuples and take the form shown in Figure 11.4.

546

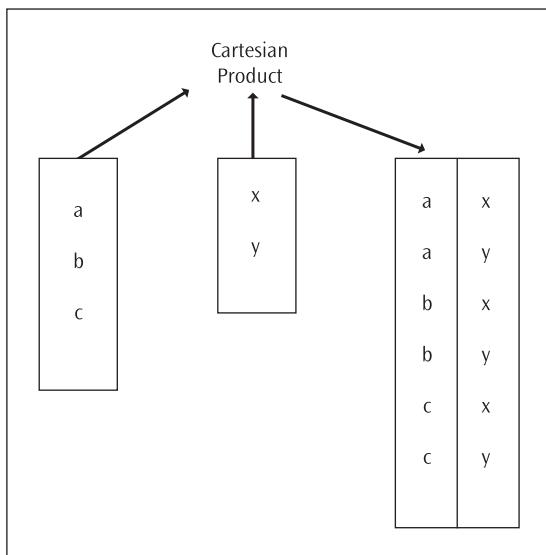


FIGURE 11.4    Cartesian Product operation

**Cartesian Product Example 1.** What is the product of the DEPARTMENT and COURSE relations?

### Relational Algebra Syntax:

COURSE **X** DEPARTMENT

Since there are six tuples in DEPARTMENT and 12 tuples in COURSE, a total of 72 tuples that contain 12 attributes per tuple is produced when the Cartesian Product of DEPARTMENT and COURSE is formed.

In order to illustrate the result obtained by a Cartesian Product operation, consider two relations—D and C derived from the DEPARTMENT and COURSE relations. Relation D contains six tuples with the attributes **NAME**, **DCODE**, and **COLLEGE**; relation C contains 12 tuples with the attributes **NAME**, **COURSE#**, **CREDIT**, and **DCODE**. The content of relations C and D is shown in Figure 11.5.

<b>C Relation</b>			
NAME	COURSE#	CREDIT	DCODE
Intro to Economics	15ECON112	U	1
Operations Research	22QA375	U	3
Intro to Economics	18ECON123	U	4
Supply Chain Analysis	22QA411	U	3
Principles of IS	22IS270	G	7
Programming in C++	20ECE5212	G	6
Optimization	22QA888	G	3
Financial Accounting	18ACCT801	G	4
Database Concepts	22IS330	U	7
Database Principles	22IS832	G	7
Systems Analysis	22IS430	G	7
Architectural History	05ARCH101	U	

<b>D Relation</b>		
NAME	DCODE	COLLEGE
Economics	1	Arts and Sciences
QA/QM	3	Business
Economics	4	Education
Mathematics	6	Engineering
IS	7	Business
Philosophy	9	Arts and Sciences

**FIGURE 11.5** The C and D relations

**Relational Algebra Syntax:**C  $\times$  D

The first 24 of the 72 tuples produced by the Cartesian Product of relations C and D are as follows:

**Result:**

NAME	COURSE#	CREDIT	DCODE	NAME	DCODE	COLLEGE
Intro to Economics	15ECON112	U	1	Economics	1	Arts and Sciences
Operations Research	22QA375	U	3	Economics	1	Arts and Sciences
Intro to Economics	18ECON123	U	4	Economics	1	Arts and Sciences
Supply Chain Analysis	22QA411	U	3	Economics	1	Arts and Sciences
Principles of IS	22IS270	G	7	Economics	1	Arts and Sciences
Programming in C++	20ECES212	G	6	Economics	1	Arts and Sciences
Optimization	22QA888	G	3	Economics	1	Arts and Sciences
Financial Accounting	18ACCT801	G	4	Economics	1	Arts and Sciences
Database Concepts	22IS330	U	7	Economics	1	Arts and Sciences
Database Principles	22IS832	G	7	Economics	1	Arts and Sciences
Systems Analysis	22IS430	G	7	Economics	1	Arts and Sciences
Architectural History	05ARCH101	U		Economics	1	Arts and Sciences
Intro to Economics	15ECON112	U	1	QA/QM	3	Business
Operations Research	22QA375	U	3	QA/QM	3	Business
Intro to Economics	18ECON123	U	4	QA/QM	3	Business
Supply Chain Analysis	22QA411	U	3	QA/QM	3	Business
Principles of IS	22IS270	G	7	QA/QM	3	Business
Programming in C++	20ECES212	G	6	QA/QM	3	Business
Optimization	22QA888	G	3	QA/QM	3	Business
Financial Accounting	18ACCT801	G	4	QA/QM	3	Business
Database Concepts	22IS330	U	7	QA/QM	3	Business
Database Principles	22IS832	G	7	QA/QM	3	Business
Systems Analysis	22IS430	G	7	QA/QM	3	Business
Architectural History	05ARCH101	U		QA/QM	3	Business

Note that the result shown here is the concatenation of the 12 tuples of relation C (see columns 1–3) with the first two tuples of relation D (see columns 4–7).

The Cartesian Product operation by itself is generally of little value. It is useful when followed first by a Selection operation that matches values of attributes coming from the component relations (technically, a Cartesian Product operation followed by a Selection operation is equivalent to a Join operation) and sometimes by a Projection operation that selects certain columns from the selected set of tuples.

**Cartesian Product Example 2.** What are the names of the departments and associated colleges that offer a four-hour course?

$\pi_{(\text{DEPARTMENT.NAME}, \text{COLLEGE})} (\sigma_{(\text{HRS} = 4 \text{ and } \text{COLLEGE.DCODE} = \text{DEPARTMENT.DCODE})} (\text{COURSE} \times \text{DEPARTMENT}))$

As we will see in Section 11.2.3,

$\sigma_{(\text{HRS} = 4 \text{ and } \text{COLLEGE.DCODE} = \text{DEPARTMENT.DCODE})} (\text{COURSE} \times \text{DEPARTMENT})$

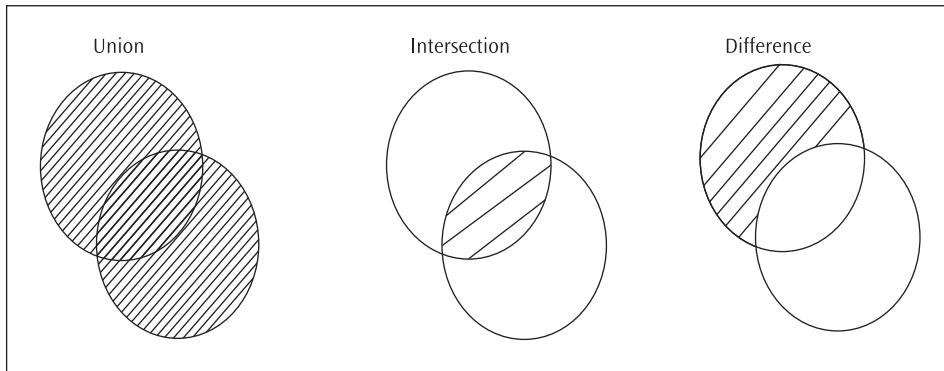
represents a JOIN operation, with COURSE X DEPARTMENT serving as its fundamental building block.

**Result:**

NAME	COLLEGE
Economics	Education
IS	Business

### 11.2.2 Set Theoretic Operators

Three **set theoretic operators**—Union, Intersection, and Difference—are used to combine the tuples from two relations. These are binary operations, as each is applied to two sets. When adapted to relational databases, the two relations on which any of these three operations are applied must be union compatible. Two relations R ( $A_1, A_2, \dots, A_n$ ) and S ( $B_1, B_2, \dots, B_n$ ) are said to be **union compatible** if (a) they have the same degree (i.e., have the same number of attributes), and (b) each pair of corresponding attributes in R and S share the same domain. Venn diagrams illustrating Union, Intersection, and Difference are shown in Figure 11.6.



**FIGURE 11.6** The Union, Intersection, and Difference operations

The remainder of this section consists of the definition and examples of the three set theoretic operators.

**Union:** The result of this operation, denoted by  $R \cup S$ , is a relation that includes all tuples that belong to either R or S or to both R and S. Duplicate tuples are eliminated.

**Union Example.** Let relations R and S be derived from the SECTION relation.

R contains tuples indicating Fall quarter sections (the fourth character in the SECTION# is an A) and S contains tuples listing sections offered in a Lindner classroom (**ROOM** = ‘Lindner’).

**RELATION R**

SECTION#	TIME	MAXST	ROOM	COURSE#	PROFID
101A2014	T1015	25		22QA375	HT54347
901A2013	W1800	35	Rhodes 611	22IS270	SK85977
902A2013	H1700	25	Lindner 108	22IS270	SK85977
102A2014	W1800		Baldwin 437	20ECE212	RR79345
104A2014	H1700	29	Lindner 108	22IS330	SK85977

**RELATION S**

SECTION#	TIME	MAXST	ROOM	COURSE#	PROFID
902A2013	H1700	25	Lindner 108	22IS270	SK85977
201S2013	T1045	29	Lindner 110	22IS330	SK85977
301S2013	H1045	29	Lindner 110	22IS330	CC49234
104A2014	H1700	29	Lindner 108	22IS330	SK85977

**Relational Algebra Syntax and Result:  $R \cup S$** 

SECTION#	TIME	MAXST	ROOM	COURSE#	PROFID
101A2014	T1015	25		22QA375	HT54347
102A2014	W1800		Baldwin 437	20ECE212	RR79345
104A2014	H1700	29	Lindner 108	22IS330	SK85977
201S2013	T1045	29	Lindner 110	22IS330	SK85977
301S2013	H1045	29	Lindner 110	22IS330	CC49234
901A2013	W1800	35	Rhodes 611	22IS270	SK85977
902A2013	H1700	25	Lindner 108	22IS270	SK85977

The Union  $R \cup S$  contains the sections that are offered either exclusively in a Fall quarter or exclusively in a Lindner classroom, or offered in a Lindner classroom during a Fall quarter(see the third and seventh tuples).

**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.

**Intersection Example.** Using the data in the relations R and S given previously, form the intersection of R and S.

**Relational Algebra Syntax and Result:  $R \cap S$** 

SECTION#	TIME	MAXST	ROOM	COURSE#	PROFID
104A2014	H1700	29	Lindner 108	22IS330	SK85977
902A2013	H1700	25	Lindner 108	22IS270	SK85977

Observe that the sections shown here are the only sections offered in a Lindner classroom during a Fall quarter.

**Difference:** The result of this operation, denoted by  $R - S$ , is a relation that includes all tuples that are in  $R$  but not in  $S$ .

**Difference Example 1.** Using the data in the relations  $R$  and  $S$  given previously, form the difference  $R$  minus  $S$ .

#### Relational Algebra Syntax and Result: $R - S$

SECTION#	TIME	MAXST	ROOM	COURSE#	PROFID
101A2014	T1015	25		22QA375	HT54347
102A2014	W1800		Baldwin 437	20ECES212	RR79345
901A2013	W1800	35	Rhodes 611	22IS270	SK85977

Note that the result obtained by subtracting  $S$  from  $R$  is equal to all sections offered during a Fall quarter in a classroom other than Lindner. The classroom associated with the section in the first tuple is not available (i.e., is a null value) and satisfies the condition that the section be offered in a classroom other than Lindner.

**Difference Example 2.** Using the data in the relations  $R$  and  $S$  given previously, form the difference  $S$  minus  $R$ .

#### Relational Algebra Syntax and Result: $S - R$

SECTION#	TIME	MAXST	ROOM	COURSE#	PROFID
201S2013	T1045	29	Lindner 110	22IS330	SK85977
301S2013	H1045	29	Lindner 110	22IS330	CC49234

Observe that the result obtained by subtracting  $R$  from  $S$  is equal to all sections offered in a classroom located in Lindner during a quarter other than the Fall quarter.

### 11.2.3 Join Operators

Joins come in several varieties. Basically, in each, the **Join operation**, denoted by  $[X]$ , is used to combine related tuples from two relations into single tuples. The example given in Cartesian Product Example 2, where the names of the departments and their associated colleges that offer a four-hour course is requested, can be specified using the Join operation by replacing:

$\pi_{(\text{DEPARTMENT.NAME}, \text{COLLEGE})} (\sigma_{(\text{HRS} = 4 \text{ and } \text{COLLEGE.DCODE} = \text{DEPARTMENT.DCODE})} (\text{COURSE} \times \text{DEPARTMENT}))$

with a Join operation followed by a Projection operation, as follows:

$\pi_{(\text{DEPARTMENT.NAME}, \text{COLLEGE})} (\text{COURSE} [x] \text{ HRS} = 4 \text{ and } \text{COURSE.DCODE} = \text{DEPARTMENT.DCODE} \text{ DEPARTMENT})$

The general form of a Join operation on two relations  $R$  ( $A_1, A_2, \dots, A_n$ ) and  $S$  ( $B_1, B_2, \dots, B_m$ ) is:

$R [x] \text{ <join condition>} S$

The result of the Join operation is a relation  $Q$  with  $n + m$  attributes  $Q (A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m)$ , 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. In order to join the two relations R and S, they must be **join compatible**—that is, the join condition must involve attributes from R and S that share the same domain.

A general join condition is of the form:

```
<condition> AND <condition> AND ... AND <condition>
```

where each condition is of the form  $A_i \theta B_j$ , where  $A_i$  is an attribute of R, where  $B_j$  is an attribute of S, where  $A_i$  and  $B_j$  have the same domain, and where  $\theta$  (theta) is one of the comparison operators  $\{=, \neq, <, \leq, >, \geq\}$ . 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.

### 11.2.3.1 The Equijoin Operator

The most common join involves join conditions where the comparison operator is “ $=$ .” This type of join is called the **Equijoin**. The result of an Equijoin includes all attributes from both relations participating in the Join operation. This implies duplication of the joining attributes in the result.

**Equijoin Example.** Join the C and D relations (derived from the COURSE and DEPARTMENT relations) over their common attribute department code (**D.DCODE** in relation D and **C.DCODE** in relation C).

**Relational Algebra Syntax:**

```
C [x] C.DCODE = D.DCODE D
```

**Result:**

NAME	COURSE#	CREDIT	DCODE	NAME	DCODE	COLLEGE
Intro to Economics	15ECON112	U	1	Economics	1	Arts and Sciences
Operations Research	22QA375	U	3	QA/QM	3	Business
Intro to Economics	18ECON123	U	4	Economics	4	Education
Supply Chain Analysis	22QA411	U	3	QA/QM	3	Business
Principles of IS	22IS270	G	7	IS	7	Business
Programming in C++	20ECES212	G	6	Mathematics	6	Engineering
Optimization	22QA888	G	3	QA/QM	3	Business
Financial Accounting	18ACCT801	G	4	Economics	4	Education
Database Concepts	22IS330	U	7	IS	7	Business
Database Principles	22IS832	G	7	IS	7	Business
Systems Analysis	22IS430	G	7	IS	7	Business

Observe that the name (**NAME - see column 5**) and (**COLLEGE - see column 7**) of the department associated with each course appears in the result just shown.

Had the Equijoin involved the complete COURSE and DEPARTMENT relations joined on the **COURSE.DCODE** and **DEPARTMENT.DCODE** attributes instead of just relations C and D, the result would have also included 11 tuples, with each tuple containing 12 attributes.

While it is possible to join COURSE and DEPARTMENT over the **COLLEGE** attribute from COURSE and the **COLLEGE** attribute from DEPARTMENT, the result of such a join would be meaningless; each tuple in COURSE would be concatenated not only with the tuple from DEPARTMENT that offers the course but also with the tuples from DEPARTMENT with the same **DEPARTMENT.COLLEGE** but associated with a different department code than that associated with the course. Using the data in the COURSE and DEPARTMENT relations in Figure 11.1, it is left as an exercise for the reader to demonstrate that an Equijoin of COURSE and DEPARTMENT on the attributes **COURSE.COLLEGE** and **DEPARTMENT.COLLEGE** yields a result that contains 19 tuples.

### 11.2.3.2 The Natural Join Operator

Because the result of an Equijoin results in pairs of attributes with identical values in all the tuples (see columns 4 and 6 in the result shown previously), the relational algebra operation called a **Natural Join**, denoted by  $*$ , was created to omit the second (and unnecessary) attribute in an Equijoin condition. The Natural Join of two relations with the common attribute  $b$  is illustrated in Figure 11.7.

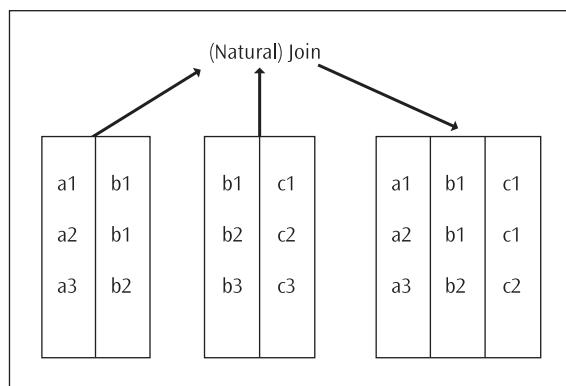


FIGURE 11.7 The Natural Join operation

**Natural Join Example.** Join the C and D relations over their common attribute department code (**DCODE** in relation D and **DCODE** in relation C).

**Relational Algebra Syntax:**<sup>3</sup>

$$C *_{C.DCODE = D.DCODE} D$$

<sup>3</sup>Contrary to the requirement that attributes have unique names over the entire relational schema, the standard definition of “Natural Join” requires that the two join attributes (or each pair of join attributes) have the same name. If this is not the case, a renaming operation must be applied first. See Elmasri and Navathe (2010) for a discussion of the use of the renaming operation in representing a Natural Join.

**Result:**

NAME	COURSE#	CREDIT	DCODE	NAME	COLLEGE
Intro to Economics	15ECON112	U	1	Economics	Arts and Sciences
Operations Research	22QA375	U	3	QA/QM	Business
Intro to Economics	18ECON123	U	4	Economics	Education
Supply Chain Analysis	22QA411	U	3	QA/QM	Business
Principles of IS	22IS270	G	7	IS	Business
Programming in C++	20ECES212	G	6	Mathematics	Engineering
Optimization	22QA888	G	3	QA/QM	Business
Financial Accounting	18ACCT801	G	4	Economics	Education
Database Concepts	22IS330	U	7	IS	Business
Database Principles	22IS832	G	7	IS	Business
Systems Analysis	22IS430	G	7	IS	Business

554

Observe that only one (**DCODE**) of the two join attributes common to both relations (**D.DCODE** and **C.DCODE**) appears in the result just shown.

The Join operation is used to combine data from multiple relations so that related information can be presented in a single relation. As illustrated in Cartesian Product Example 2, Join operations are typically followed by a Projection operation.

### 11.2.3.3 The Theta Join Operator

While occurring infrequently in practical applications, **Theta Joins** that do not involve equality conditions are possible as long as the join condition involves attributes that share the same domain.

**Theta Join Example.** Instead of doing an Equijoin of C and D when an equality condition involving their two common attributes (**C.DCODE** and **D.DCODE**) exists, do a join of C and D when an inequality condition exists for these common attributes.

#### Relational Algebra Syntax:

**C [x] c.DCODE < > d.DCODE D**

In such a Theta Join, a tuple from D is concatenated with a tuple from C only when an inequality exists for each of the join conditions. Thus, the first tuple in D is not concatenated with the first tuple in C because the join condition is not satisfied. Observe, however, that the join condition is satisfied when the first tuple of D is evaluated against all other tuples in C, thus resulting in the first 10 tuples of the 55 tuples in the result.<sup>4</sup> Using the data for relations C and D shown in Figure 11.5, the reader is encouraged to verify why this Theta Join produces a total of 55 tuples.

---

<sup>4</sup>The reason why the join condition is not satisfied for the 12th tuple in COURSE where C.DCODE is a null value is discussed in Section 12.1.5 of Chapter 12.

### 11.2.3.4 Outer Join Operators

The Join operations discussed to this point are **Inner Join** operations, meaning that for the relations R and S, only tuples from R that have matching tuples in S (and vice versa) appear in the result. In other words, tuples without a matching (or related) tuple are eliminated from the join result. Tuples with null values in the join attributes are also eliminated (e.g., the Architectural History course in the COURSE relation). 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.

The **Left Outer Join** operation, denoted by  $[X]$ , keeps every tuple in the *first or left* relation R in  $R [X] S$ . If no matching tuple is found in S, then the attributes of S in the join result are filled or “padded” with null values. Thus, in effect, a tuple of null values is added to relation S, and any tuple in relation R without a matching tuple in relation S is concatenated with the tuple of null values in relation S. On the other hand, a **Right Outer Join**, denoted by  $[X]$ , keeps every tuple in the *second or right* relation S in the result  $R [X] S$ . If for a particular tuple in relation S there is no matching tuple in relation R, then for that particular tuple of relation S, attributes from relation R are “padded” with null values. A third operation, **Full Outer Join**, denoted by  $[X]$ , keeps all tuples in both the left and right relations when no matching tuples are found, padding them with null values as needed.

**Left Outer Join Example.** Do a Left Outer Join of relations D and C over their common attributes (**D.CODE** in relation D and **C.CODE** in relation C).

**Relational Algebra Syntax:**

$D [X]_{D.CODE = C.CODE} C$

**Result:**

NAME	DCODE	COLLEGE	NAME	COURSE#	CREDIT	DCODE
Economics	1	Arts and Sciences	Intro to Economics	15ECON112	U	1
QA/QM	3	Business	Operations Research	22QA375	U	3
Economics	4	Education	Intro to Economics	18ECON123	U	4
QA/QM	3	Business	Supply Chain Analysis	22QA411	U	3
IS	7	Business	Principles of IS	22IS270	G	7
Mathematics	6	Engineering	Programming in C++	20ECES212	G	6
QA/QM	3	Business	Optimization	22QA888	G	3
Economics	4	Education	Financial Accounting	18ACCT801	G	4
IS	7	Business	Database Concepts	22IS330	U	7
IS	7	Business	Database Principles	22IS832	G	7
IS	7	Business	Systems Analysis	22IS430	G	7
Philosophy	9	Arts and Sciences				

Observe how the sixth tuple of D is concatenated with the tuple of null values from the C relation to produce the twelfth tuple in the result, revealing that the Philosophy Department has yet to offer a course.

**Right Outer Join Example.** Do a Right Outer Join of relations D and C over their common attributes (**D.CODE** in relation D and **C.CODE** in relation C).

**Relational Algebra Syntax:**

$$D \mid X [ D.DCODE = C.DCODE ] C$$
**Result:**

NAME	DCODE	COLLEGE	NAME	COURSE#	CREDIT	DCODE
Economics	1	Arts and Sciences	Intro to Economics	15ECON112	U	1
QA/QM	3	Business	Optimization	22QA888	G	3
QA/QM	3	Business	Supply Chain Analysis	22QA411	U	3
QA/QM	3	Business	Operations Research	22QA375	U	3
Economics	4	Education	Financial Accounting	18ACCT801	G	4
Economics	4	Education	Intro to Economics	18ECON123	U	4
Mathematics	6	Engineering	Programming in C++	20ECES212	G	6
IS	7	Business	Systems Analysis	22IS430	G	7
IS	7	Business	Database Principles	22IS832	G	7
IS	7	Business	Database Concepts	22IS330	U	7
IS	7	Business	Principles of IS	22IS270	G	7
			Architectural History	05ARCH101	U	

Observe how each tuple of C (including the course in Architectural History that is not affiliated with a department) appears in the result just shown.

**Full Outer Join Example.** Join the D and C relations, making sure that each tuple from each relation appears in the result.

A Full Outer Join of D and C, expressed as:

**Relational Algebra Syntax:**

$$D \mid X [ D.DCODE = C.DCODE ] C$$

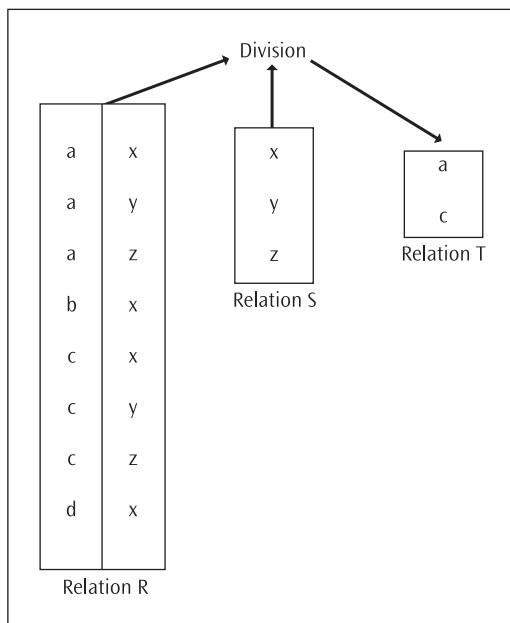
adds a blank tuple to both relation D and relation C to ensure that each tuple in each relation is reflected in the result. Observe how each tuple in each relation appears in the result shown here:

NAME	DCODE	COLLEGE	NAME	COURSE#	CREDIT	DCODE
Economics	1	Arts and Sciences	Intro to Economics	15ECON112	U	1
QA/QM	3	Business	Operations Research	22QA375	U	3
Economics	4	Education	Intro to Economics	18ECON123	U	4
QA/QM	3	Business	Supply Chain Analysis	22QA411	U	3
IS	7	Business	Principles of IS	22IS270	G	7
Mathematics	6	Engineering	Programming in C++	20ECES212	G	6
QA/QM	3	Business	Optimization	22QA888	G	3
Economics	4	Education	Financial Accounting	18ACCT801	G	4
IS	7	Business	Database Concepts	22IS330	U	7
IS	7	Business	Database Principles	22IS832	G	7
IS	7	Business	Systems Analysis	22IS430	G	7
Philosophy	9	Arts and Sciences	Architectural History	05ARCH101	U	

It is left as an exercise for the reader to verify that the union of the results of a Left Outer Join and a Right Outer Join is equal to the result of a Full Outer Join.

#### 11.2.4 The Divide Operator

The **Divide operator** is useful when there is a need to identify tuples in one relation that match *all* tuples in another relation. For example, let R be a relation schema with attributes  $A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m$  and let S be a relation schema with attributes  $B_1, B_2, \dots, B_m$ . In other words, the set of attributes of S (*the divisor*) is a subset of the attributes of R (*the dividend*). The division of R by S can be expressed as  $R/S$ . Figure 11.8 contains a schematic view of the Division operation. Observe how the two tuples in relation T represent a subset of the tuples in relation R that match all three tuples in relation S. In order to divide R by S, relations R and S must be division compatible in that the set of attributes of S must be a subset of the attributes of R. In other words, if relation S in Figure 11.8 were also to contain the attribute p, then division compatibility would not exist.



**FIGURE 11.8** The Division operation

For ease of understanding, Kifer, Bernstein, and Lewis (2005) illustrate how the Division operation can be decomposed into a sequence of Projection, Cartesian Product, and Difference operations, as shown here:

$T_1 = \pi_A(R) \times S$	All possible associations between the A values in R and B values in S.
$T_2 = \pi_A(T_1 - R)$	All those A values in R that are not associated in R with every B value in S. These are those A values that should not be in the answer.
$T_3 = \pi_A(R) - T_2$	The quotient: all those A values in R that are associated in R with all B values in S.

**Divide Example.** List the course numbers of courses that are offered in all quarters during which course sections are offered.<sup>5</sup>

#### Relational Algebra Syntax:

```
R = π (COURSE#, SUBSTR(SECTION#, 4, 1)) (SECTION)
S = π (SUBSTR(SECTION#, 4, 1)) (SECTION)
R ÷ S
```

558

#### Result:

```
COURSE#
-----
22QA375
```

Expressing the result as a sequence of Projection, Cartesian Product, and Difference operations yields the following:

Relation R		Relation S
COURSE#	SUBSTR(SECTION#, 4, 1) <sup>6</sup>	SUBSTR(SECTION#, 4, 1)
-----	-----	-----
22QA375	A	A
22IS270	A	S
22IS330	S	W
22IS832	W	U
20ECES212	A	
22QA375	U	
22IS330	A	
22QA375	S	
22QA375	W	

<sup>5</sup>The fourth character position of the SECTION# represents the quarter in which the section is offered. The letters “A,” “S,” “W,” and “U” represent the Fall, Spring, Winter, and Summer quarters, respectively.

<sup>6</sup>SUBSTR(SECTION#,4,1) is used to extract the fourth character containing the quarter in which the section is offered from the 8-character section number. The SQL SUBSTR(char, m, [n]) function is discussed in Section 13.1.1 of Chapter 13.

$T_1 = \pi_{(Se\_co\_course\#, Se\_qtr)}(R) \times S$ —All possible associations between courses and quarters during which sections of courses are offered:

$\pi_{COURSE\#}(R)$	S	$T_1$	
22QA375	A	22QA375	A
22IS270	S	22QA375	S
20IS330	W	22QA375	W
22IS832	U	22QA375	U
20ECES212		22IS270	A
		22IS270	S
		22IS270	W
		22IS270	U
		20IS330	A
		20IS330	S
		20IS330	W
		20IS330	U
		22IS832	A
		22IS832	S
		22IS832	W
		22IS832	U
		20ECES212	A
		20ECES212	S
		20ECES212	W
		20ECES212	U

559

$T_2 = \pi_{(COURSE\#)}(T_1 - R)$ —All courses that are not offered during each quarter:

$(T_1 - R)$		$T_2$
22IS270	S	22IS270
22IS270	W	20IS330
22IS270	U	20IS832
22IS832	A	20ECES212
22IS832	S	
22IS832	U	
20ECES21	S	
20ECES212	W	
20ECES212	U	

$T_3 = \pi_{(COURSE\#)}(R) - T_2$ —All courses offered during each quarter:

$\pi_{(COURSE\#)}(R)$	$T_2$	$T_3$
22QA375	20IS270	
22IS270	20IS330	22QA375
20IS330	22IS832	
22IS832	20ECES212	
20ECES212		

Table 11.1 summarizes the basic relational algebra operators and notation.

Relational Algebra Operator	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.	$\pi_{<\text{attribute list}>} (R)$
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 $R_2$ .	$R_1 \times R_2$
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.	$R_1 - R_2$
Equijoin	Produces all the combinations of tuples from $R_1$ and $R_2$ that satisfy a join condition with only equality comparisons. $R_1$ and $R_2$ must be join compatible.	$R_1[X]_{<\text{join condition}>} R_2$
Natural Join	Produces all the combinations of tuples from $R_1$ and $R_2$ that satisfy a join condition with only equality comparisons, except that one of the join attributes is not included in the resulting relation. $R_1$ and $R_2$ must be join compatible.	$R_1 *_{<\text{join condition}>} R_2$
Theta Join	Produces all the combinations of tuples from $R_1$ and $R_2$ that satisfy a join condition which does not have to involve equality comparisons. $R_1$ and $R_2$ must be join compatible.	$R_1[X]_{<\text{join condition}>} R_2$
Divide	Returns every tuple from $R_1$ that match all tuples in $R_2$ ; $R_1$ and $R_2$ must be division compatible.	$R_1 \div R_2$

\*Source: R. Elmasri and S. B. Navathe (2010), *Fundamentals of Database Systems*, Pearson Education.

**TABLE 11.1** Basic relation algebra operators\*

### 11.2.5 Additional Relational Operators

Some common database requests cannot be performed with the basic relational algebra operations described previously. This section defines three additional operations to express these requests.

#### 11.2.5.1 The Semi-Join Operation

The **Semi-Join operation** defines a relation that contains the tuples of R that participate in the join of R with S. In other words, a Semi-Join is defined to be equal to the join of R and S,

projected back on the attributes of R. The Semi-Join operation can be expressed using the Projection and Join operations, as follows:

$$\Pi_R (R \mid X_{A=B} S)$$

where A and B are domain-compatible attributes of R and S, respectively.

**Semi-Join Example.** List the complete details of all courses for which sections are being offered in the fall 2014 quarter.

**Relational Algebra Syntax:**

$$\pi_{\text{COURSE}} (\text{COURSE} \mid X_{(\text{COURSE.COURSE\#} = \text{SECTION.COURSE\#} \text{ and } \text{SUBSTR}(\text{SECTION\#}, 4, 5) = 'A2014') \text{ SECTION})$$

**Result:**

NAME	COURSE#	CREDIT	COLLEGE	HRS	DCODE
Operations Research	22QA375	U	Business	2	3
Programming in C++	20ECE212	G	Engineering	3	6
Database Concepts	22IS330	U	Business	4	7

561

### 11.2.5.2 The Semi-Minus Operation

The semi-difference between R and S (in this order) is defined to be equivalent to:

$$R - \pi_R (R \mid X_{A=B} S)$$

The result of the **Semi-Minus operation** is thus the tuples of R that have no counterpart in S.

**Semi-Minus Example.** List complete details of all courses for which sections are not offered in the fall 2014 quarter.

**Relational Algebra Syntax:**

$$\text{COURSE} - \pi_{\text{COURSE}} (\text{COURSE} \mid X_{(\text{COURSE.COURSE\#} = \text{SECTION.COURSE\#} \text{ and } \text{SUBSTR}(\text{SECTION\#}, 4, 5) = 'A2014') \text{ SECTION})$$

**Result:**

NAME	COURSE#	CREDIT	COLLEGE	HRS	DCODE
Architectural History	05ARCH101	U		3	
Database Principles	22IS832	G	Business	3	7
Financial Accounting	18ACCT801	G	Education	3	4
Intro to Economics	15ECON112	U	Arts and Sciences	3	1
Intro to Economics	18ECON123	U	Education	4	4
Optimization	22QA888	G	Business	3	3
Principles of IS	22IS270	G	Business	3	7
Supply Chain Analysis	22QA411	U	Business	3	3
Systems Analysis	22IS430	G	Business	3	7

### 11.2.5.3 Aggregate Functions and Grouping

One type of request that cannot be expressed in the basic relational algebra involves mathematical aggregate functions on collections of values from the database. Examples of such functions include retrieving the sum, average, maximum, and minimum of a series of numeric values. Another type of request involves the grouping of 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 the courses taken by course number and then count the number of students taking each course during the year 2014.

The **Aggregate Function** operation can be defined using the symbol  $\mathcal{F}$  to specify these types of requests, as follows:

`<grouping attributes>  $\mathcal{F}$  <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 where the following applies:

- **<function>** is one of the allowed functions, such as SUM, AVERAGE, MAXIMUM, MINIMUM, COUNT.
- **<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.

**Aggregate Functions and Grouping Example.** Compute the average number of hours for the courses offered by each college.

#### Relational Algebra Syntax:

`COLLEGE  $\mathcal{F}$  AVERAGE HRS (COURSE)`

#### Result:

COLLEGE	AVG (HRS)
	-----
	3
Engineering	3
Education	3 . 5
Arts and Sciences	3
Business	3

As mentioned in Section 11.1.2 in the context of Projection Example 1, there are five distinct values for the COLLEGE attribute: Engineering, Education, Arts and Sciences, Business, and the null value associated with the Architectural History course. From a relational algebra standpoint, the null value is considered one of the five values of the COLLEGE grouping attribute.

## Chapter Summary

---

Chapter 11 introduces relational algebra, a mathematical expression of data retrieval methods prescribed by E. F. Codd, as a means to specify the logic for data retrieval from a relational database. The tuples of a relation can be considered elements of a set and thus can be involved in operations. In the same way that algebra is a system of operations on numbers, relational algebra is a system of operations on relations. Expressed in terms of the relations R and S, the basic operations of relational algebra are Union ( $R \cup S$ ), Difference ( $R - S$ ), Selection ( $\langle \text{selection condition} \rangle(R)$ ), Projection ( $\langle \text{attribute list} \rangle(R)$ ), and Cartesian Product ( $(R(A_1, A_2, \dots, A_n) \times S(B_1, B_2, \dots, B_m))$ ), where  $A_i$  and  $B_j$  are attributes of R and S, respectively. Certain combinations of these five operations can be used to define three other basic operations. When the Union of R and S is formed, an Intersection operation identifies those tuples common to both R and S. A Join operation consists of a Cartesian Product followed by a Selection. A Division operation can be expressed as a sequence of Projection, Cartesian Product, and Difference operations. A summary of the basic relational algebra operations discussed in this chapter appears in Figure 11.2.

563

## Exercises

---

1. What constitutes union compatibility?
2. What are the purposes of the Union, Intersection, and Difference operations?
3. What is a Cartesian Product operation?
4. What is the difference between the result obtained from a Selection operation versus a Projection operation?
5. Why does the relation created as result of a Projection operation on a relation that includes the primary key not contain fewer tuples than those in the source relation?
6. What is a Join operation? What is meant when it is said that two relations are join compatible? What is the difference between an Inner Join operation and an Outer Join operation?
7. What is meant by the term “division compatibility”?
8. This question is based on the four relations shown below. Show the results along with the relational algebra expressions for the following four retrieval requests.
  - a. Display the products and their associated prices for stores located in Houston.
  - b. Which products are not in the inventory of any store?
  - c. Which products are offered at a 10% discount?
  - d. Which stores stock products with a price greater than \$1000?

STORE				PRODUCT	
Store	Location	Sq_ft	Manager	Product	Price
15	Houston	2300	Metzger	Refrigerator	1850
13	Tulsa		Metzger	Dish washer	600
14	Dallas	1900	Schott	Television	1400
17	Memphis	2300	Creech	Humidifier	55
11	Houston	1900	Parrish	Vacuum Cleaner	300
				Computer	
				Lawn Mower	300
				Washing Machine	750

564

INVENTORY			DISC_STRUCTURE	
Store	Product	Quantity	Quantity	Discount
15	Refrigerator	120	120	5%
15	Dish Washer	150	150	5%
13	Dish Washer	180	180	10%
14	Refrigerator	150	280	10%
14	Television	280	30	
14	Humidifier	30	10	
17	Television	10		
17	Vacuum Cleaner	150		
17	Dish washer	150		
11	Computer	120		
11	Refrigerator	180		
11	Lawn Mower			

9. Consider the TUTOR, TUTOR\_ASSIGNMENT, and COURSE relations that appear below. Show the results and the relational algebra expressions for the following four retrieval requests.
- What are the Course IDs for those courses that do not have an assigned tutor?
  - What are the names of the tutors for MIS 4372?
  - What are the names of the tutors for those courses with no prerequisites?
  - What are the course names and number of prerequisites for those tutors hired in 2009.

TUTOR

TutorID	Tutor Name	Year Hired	Tutor Status
10000	Kubiak	2007	Active
10001	Johnson	2003	Active
10002	Foster	2009	Active
10003	Joseph	2011	Inactive
10004	Brown	2008	Active
10005	Watt	2011	Active
10006	Cushing	2009	Inactive

565

TUTOR\_ASSIGNMENT

TutorID	CourseID
10002	MIS 3300
10002	MIS 3360
10004	MIS 3370
10000	MIS 3371
10001	MIS 3376
10001	MIS 4386
10003	MIS 4372
10004	MIS 4372
10005	MIS 4374

## Chapter 11

### COURSE

CourseID	CourseName	Number of Prerequisites
MIS 3300	Introduction to Computers and Management Information Systems	0
MIS 3360	System Analysis and Design	1
MIS 3370	Information Systems Development Tools	2
MIS 3371	Transactions Processing Systems I	2
MIS 3376	Business Applications of Database Management Systems I	2
MIS 4374	Information Technology Project Management	2
MIS 4478	Administration of Management Information Systems	2
MIS 4372	Transactions Processing Systems II	3
MIS 4477	Network & Security Infrastructure	2
MIS 4381	Management of Information Security	0
MIS 4386	Business Applications of Database Management Systems II	3
MIS 4390	Energy Trading Systems	0

566

# CHAPTER 12

## STRUCTURED QUERY LANGUAGE (SQL)

SQL is the standard language for manipulating relational databases. The language was created to facilitate implementation of relational algebra in a database. Like relational algebra, SQL uses one or more relations as input and produces a single relation as output.<sup>1</sup> This chapter and Chapter 13 contain an informal overview of the use of the SQL SELECT statement for information retrieval. Through the use of numerous example queries, the discussion introduces the SELECT statement's features, beginning gradually with queries based on a single table followed by queries that retrieve data from several tables. As was the case in Chapter 10, except where indicated, the examples in this chapter as well as in Chapter 13 are based on the syntax associated with the SQL-2003 standard.

Before proceeding further, it is important to note that space does not permit a thorough and complete discussion of all features of the SQL SELECT statement as well as other features associated with SQL's data definition language and data manipulation language. Hundreds of books and reference manuals are available on SQL, and the interested reader is encouraged to consider such sources for more comprehensive syntax than that shown in this book. In addition, many Web sites have information about standard SQL and its implementation under various database platforms.

As mentioned previously, the SQL SELECT statement is employed to retrieve (i.e., query) data from tables (i.e., relations) and is used in conjunction with all relational algebra operations. For example, using a SELECT statement, it is possible to view all the columns and rows within a table (i.e., to execute a relational algebra Selection operation) or specify that only certain columns and rows be viewed (execute a Selection operation followed by a Projection operation).

The syntax for an SQL statement gives the basic structure, or rules, required to execute the statement. The basic form of the SQL SELECT statement is called a

---

<sup>1</sup>SQL uses the terms "table," "row," and "column" for "relation," "tuple," and "attribute," respectively. Thus, when the discussion focuses on SQL, the terms "table," "row," and "column" will be used. When the discussion focuses on relational algebra operations, the terms "relation," "tuple," and "attribute" will be used.

**select-from-where block** and contains the three clauses SELECT, FROM, and WHERE in the following form:

```
SELECT <column list>
FROM <table list>
WHERE <condition>
```

where the following applies:

- <column list> is a list of column names whose values are to be retrieved by the query,
- <table list> is a list of the table names required to process the query.<sup>2</sup>
- <condition> is a conditional (Boolean) expression that identifies the rows to be retrieved by the query.

Kifer, Bernstein, and Lewis (2005) provide a useful description of the basic algorithm for evaluating an SQL query<sup>3</sup>:

1. The FROM clause is evaluated. It produces a table that is the Cartesian Product of the tables listed as arguments. If a table occurs more than once in the FROM clause, then this table occurs as many times in the Cartesian Product.
2. The WHERE clause is evaluated by taking the table produced in Step 1 and processing each row individually. Values from the row are substituted for the column names in the condition and the condition is evaluated. The table produced by the WHERE clause contains exactly those rows for which the condition evaluates to true.
3. The SELECT clause is evaluated. It takes the table produced in Step 2 and retains only those columns that are listed as arguments. The resulting table is output by the SELECT statement.

Additional features of the language result in the addition of steps to this algorithm. Also, certain steps need not be present in a particular evaluation. For example, Step 2 above is not evaluated if there is no WHERE clause.<sup>4</sup> Likewise, if the FROM clause refers to only a single table, the Cartesian Product is not formed.

Three other clauses can also be used in an SQL SELECT statement:

```
GROUP BY group_by_expression
HAVING group_condition
ORDER BY column name(s)
```

<sup>2</sup>As we will see, the table list may include both database views and inline views.

<sup>3</sup>Kifer, M., A. Bernstein, and P. M. Lewis. *Databases and Transactions Processing: An Application-Oriented Approach*, Second Edition. Addison-Wesley, 2005b.

<sup>4</sup>While the WHERE clause is optional, the SELECT and FROM clauses are not.

where the following applies:

- *group\_by\_expression* forms groups of rows with the same value.
- *group\_condition* filters the groups subject to some condition.
- *column name(s)* specifies the order of the output.

In this book, a semicolon (;) follows the last clause in an SQL SELECT statement.<sup>5</sup> In addition, while the keywords and clauses (e.g., SELECT, FROM, WHERE, GROUP BY, etc.) in an SQL SELECT statement appear capitalized here, only conditions in the WHERE clause that involve non-numeric (i.e., character or string) literals are actually case sensitive.

Many of the illustrations in this chapter make use of the DEPARTMENT, COURSE, STUDENT, SECTION, TAKES, PROFESSOR, TEXTBOOK, and USES tables from the Madeira College registration system described in Section 10.1.1.5 of Chapter 10. The initial form of these eight tables is shown in Figure 11.1 of Chapter 11. Each illustration has been tested using Oracle. Thus, please note that not all syntax shown will work on other platforms, such as IBM DB2, Microsoft SQL Server, and MySQL. For more comprehensive syntax than that shown in conjunction with the examples in this book, refer to the SQL reference material of the respective database platform.

## 12.1 SQL QUERIES BASED ON A SINGLE TABLE

This section considers several SQL features in the context of queries that involve a single table. Included in the discussion are illustrations of (a) the Select and Project operators, the two unary relational algebra operators based on a single relation, (b) the hierarchy of operations, (c) the handling of null values, and (d) pattern matching.

The examples in the remainder of this chapter are labeled in accordance with the following scheme:

- The section number is ignored (e.g., in Section 12.1.1, the number 12 is ignored)
- The examples within the section are numbered from 1 to n, beginning with the third element of the section number.

Thus, you will find that the three examples in Section 12.1.1 are labeled 1.1.1 through 1.1.3; the 10 examples in Section 12.1.6 are labeled 1.6.1 through 1.6.10; and the six examples in Section 12.2.3 are labeled 2.3.1 through 2.3.6.

### 12.1.1 Examples of the Selection Operation

Three examples of the use of the relational algebra Select operator in a Selection operation appear in Section 11.1.1 of Chapter 11. SQL SELECT statements that correspond to each of these examples follow:

**Example 1.1.1 (Corresponds to Selection Example 1 in Section 11.1.1).** Which courses are three-hour courses?

---

<sup>5</sup>The SQL-2003 standard actually prescribes the semicolon as a statement terminator only in the case of embedded SQL.

**SQL SELECT Statements:**

```
SELECT COURSE.NAME, COURSE.COURSE#, COURSE.CREDIT,
COURSE.COLLEGE, COURSE.HRS, COURSE.DCODE
FROM COURSE
WHERE COURSE.HRS = 3;
```

or:

```
SELECT *
FROM COURSE
WHERE COURSE.HRS = 3;
```

**Result:**

NAME	COURSE#	CREDIT	COLLEGE	HRS	DCODE
Intro to Economics	15ECON112	U	Arts and Sciences	3	1
Supply Chain Analysis	22QA411	U	Business	3	3
Principles of IS	22IS270	G	Business	3	7
Programming in C++	20ECES212	G	Engineering	3	6
Optimization	22QA888	G	Business	3	3
Financial Accounting	18ACCT801	G	Education	3	4
Database Principles	22IS832	G	Business	3	7
Systems Analysis	22IS430	G	Business	3	7
Architectural History	05ARCH101	U		3	

570

The asterisk (\*) means that all columns from the table are to be selected. The WHERE clause tells SQL to search the rows in the COURSE table and to return (i.e., display) only those rows where the value of **COURSE.HRS** is equal to exactly 3. In addition, while not required, in an SQL SELECT statement it is a good idea to prefix each column name with its table name in order to minimize ambiguity and confusion.<sup>6</sup>

Since SQL SELECT statements are not case sensitive, the two SQL statements shown here could also have been written as follows:

```
select name, course#, credit, college, hrs, dcode
from course
where hrs = 3;

select *
from course
where hrs = 3;
```

---

<sup>6</sup>The convention of prefixing each column name with its table name is used throughout this chapter. SQL, however, permits duplicate column names as long as they appear in different tables. When two tables involved in a SELECT statement share a common column name or names, the qualification of the column name(s) with the appropriate table name is required if the column name appears in the <attribute list> or in the WHERE condition. Otherwise, ambiguity will exist and an error message of the form “column ambiguously defined” will appear.

While each of these statements is semantically correct, once again we recommend that each column name referenced be preceded (i.e., prefixed) by the table name COURSE in either lowercase or uppercase.

The ORDER BY clause is used to specify how the results of a query are to be sorted. The default sort is an ascending sort. The keywords, ASCENDING and DESCENDING (abbreviated ASC and DESC), are used to control the sorting of each column. Thus, the query:

```
SELECT *
FROM COURSE
WHERE COURSE.HRS = 3
ORDER BY COURSE.DCODE DESC;
```

lists the three-hour courses in descending order by the department number in which the course is offered, whereas the query:

```
SELECT *
FROM COURSE
WHERE COURSE.HRS = 3
ORDER BY COURSE.DCODE DESC, COURSE.NAME;
```

lists the three-hour courses in descending order by the department number in which the course is offered and in ascending order by course name within each department.

**Example 1.1.2 (Corresponds to Selection Example 2 in Section 11.1.1).** Which courses offered by department 7 are three-hour courses?

#### SQL SELECT Statement:

```
SELECT *
FROM COURSE
WHERE COURSE.DCODE = 7 AND COURSE.HRS = 3;
```

#### Result:

NAME	COURSE#	CREDIT	COLLEGE	HRS	DCODE
Principles of IS	22IS270	G	Business	3	7
Database Principles	22IS832	G	Business	3	7
Systems Analysis	22IS430	G	Business	3	7

Use of the logical operator AND requires that *both* conditions (i.e., COURSE.DCODE = 7 and COURSE.HRS = 3) be satisfied.

**Example 1.1.3.** Which sections have a maximum number of students greater than 30 or are offered in Lindner 110?

#### SQL SELECT Statement:

```
SELECT *
FROM SECTION
WHERE SECTION.MAXST > 30 OR SECTION.ROOM = 'Lindner 110';
```

**Result:**

SECTION#	TIME	MAXST	ROOM	COURSE#	PROFID
901A2013	W1800	35	Rhodes 611	22IS270	SK85977
201S2013	T1045	29	Lindner 110	22IS330	SK85977
301S2013	H1045	29	Lindner 110	22IS330	CC49234
401W2014	M1000	33	Braunstien 211	22IS832	CC49234
103U2014	T1015	33		22QA375	HT54347

Use of the logical operator OR requires that either one or both conditions (i.e., **SECTION.MAXST > 30 OR SECTION.ROOM = 'Lindner 110'**) be satisfied. Observe that in Example 1.1.3, none of the rows in the SECTION table satisfy both conditions of the WHERE clause. Also, whenever a character or string literal (e.g., Lindner 110) is used as part of a condition, the value must be enclosed within *single quotation marks*. Character or string literals enclosed in single quotation marks are case sensitive. Thus, the WHERE condition will not be true if anything other than Lindner 110 appears inside the single quotation marks.

572

### 12.1.2 Use of Comparison and Logical Operators

Combining AND or OR in the same logical expression must be done with care. When AND and OR appear in the same WHERE clause, all the ANDs are performed first, then all the ORs are performed. In this way, AND is said to have a higher precedence than OR.

For example, the WHERE clause:

```
WHERE COURSE.CREDIT = 'U' AND COURSE.COLLEGE = 'Business' OR COURSE.COLLEGE
= 'Engineering'
```

is evaluated in the following manner:

1. Each of the component expressions is evaluated yielding a value that is either “true” or “false.”
2. The results of **COURSE.CREDIT = 'U'** AND **COURSE.COLLEGE = 'Business'** are combined, yielding a result that is “true” if both expressions are true and “false” otherwise.
3. The result of **COURSE.COLLEGE = 'Engineering'** is evaluated as being either “true” or “false.”
4. The results of Steps 2 and 3 are combined, yielding a result that is “true” if either result is “true” and “false” if both results are “false.”

Applying this evaluation scheme to the first row of the COURSE table yields the following results:

1. **COURSE.CREDIT = 'U'** — True  
**COURSE.COLLEGE = 'Business'** — False  
**COURSE.COLLEGE = 'Engineering'** — False
2. **COURSE.CREDIT = 'U'** AND **COURSE.COLLEGE = 'Business'** — False
3. **COURSE.COLLEGE = 'Engineering'** — False
4. Since both Steps 2 and 3 are false, the overall result is *false* and the first row fails to satisfy the WHERE clause.

In SQL, all operators are arranged in a hierarchy that determines their precedence. In any expression, operations are performed in order of their precedence, from highest to lowest. When operators of equal precedence are used next to each other, they are performed from left to right. The precedence of common logical operators in SQL is:

1. All of the comparison operators ( $=$ ,  $<>$ ,  $\leq$ ,  $<$ ,  $\geq$ ,  $>$ ) have equal precedence.
2. NOT
3. AND
4. OR

When the normal rules of operator precedence do not fit the needs, one can override them by placing part of an expression in parentheses. That part of the expression will be evaluated first, then the rest of the expression will be evaluated.

The following examples illustrate the incorporation of logical operators in a SELECT statement. Note that Example 1.2.1 is based on the WHERE clause discussed earlier.

### Example 1.2.1

```
SELECT *
FROM COURSE
WHERE COURSE.CREDIT = 'U'
AND COURSE.COLLEGE = 'Business'
OR COURSE.COLLEGE = 'Engineering';
```

573

#### Result:

NAME	COURSE#	CREDIT	COLLEGE	HRS	DCODE
Operations Research	22QA375	U	Business	2	3
Supply Chain Analysis	22QA411	U	Business	3	3
Programming in C++	20ECES212	G	Engineering	3	6
Database Concepts	22IS330	U	Business	4	7

In Example 1.2.2, the rows with **COURSE.COLLEGE** equal to Business or Engineering are selected first as a result of the presence of the parentheses. However, in order to be included in the result, **COURSE.CREDIT** must be equal to U. This eliminates the graduate Programming in C++ course offered in the College of Engineering.

### Example 1.2.2

```
SELECT *
FROM COURSE
WHERE COURSE.CREDIT = 'U'
AND (COURSE.COLLEGE = 'Business'
OR COURSE.COLLEGE = 'Engineering');
```

#### Result:

NAME	COURSE#	CREDIT	COLLEGE	HRS	DCODE
Operations Research	22QA375	U	Business	2	3
Supply Chain Analysis	22QA411	U	Business	3	3
Database Concepts	22IS330	U	Business	4	7

In Example 1.2.3, each of the first 11 rows satisfies at least one of the conditions in parentheses (**COURSE.COLLEGE** <> ‘Business’ OR **COURSE.COLLEGE** <> ‘Engineering’)<sup>7</sup>. Five of these rows are also associated with a course that is offered for undergraduate credit.

### Example 1.2.3

```
SELECT *
FROM COURSE
WHERE (COURSE.COLLEGE <> 'Business'
OR COURSE.COLLEGE <> 'Engineering')
AND COURSE.CREDIT = 'U' ;
```

#### Result:

NAME	COURSE#	CREDIT	COLLEGE	HRS	DCODE
Intro to Economics	15ECON112	U	Arts and Sciences	3	1
Operations Research	22QA375	U	Business	2	3
Intro to Economics	18ECON123	U	Education	4	4
Supply Chain Analysis	22QA411	U	Business	3	3
Database Concepts	22IS330	U	Business	4	7

In Example 1.2.4, the condition **COURSE.COLLEGE** <> ‘Engineering’ AND **COURSE.CREDIT** = ‘U’ is evaluated first and is satisfied by the rows for the five undergraduate courses. The condition **COURSE.COLLEGE** <> ‘Business’ is evaluated next and satisfied by the rows for the two Intro to Economics courses, the graduate Programming in C++ course, and the graduate Financial Accounting course. The rows for the two Intro to Economics courses satisfy both conditions, while each of the other five rows satisfies one condition. Section 12.1.5 discusses handling null values and points out that the only comparison operations that can produce a true condition with a null value are IS NULL and IS NOT NULL. Hence, the Architectural History course does not satisfy the condition **COURSE.COLLEGE** <> ‘Engineering’ AND **COURSE.CREDIT** = ‘U’.

### Example 1.2.4

```
SELECT *
FROM COURSE
WHERE COURSE.COLLEGE <> 'Business'
OR COURSE.COLLEGE <> 'Engineering'
AND COURSE.CREDIT = 'U' ;
```

---

<sup>7</sup>Section 12.1.5 explains why the Architectural History course in row 12 of COURSE fails to satisfy either condition.

**Result:**

NAME	COURSE#	CREDIT	COLLEGE	HRS	DCODE
Intro to Economics	15ECON112	U	Arts and Sciences	3	1
Operations Research	22QA375	U	Business	2	3
Intro to Economics	18ECON123	U	Education	4	4
Supply Chain Analysis	22QA411	U	Business	3	3
Programming in C++	20ECES212	G	Engineering	3	6
Financial Accounting	18ACCT801	G	Education	3	4
Database Concepts	22IS330	U	Business	4	7

The keywords IN or NOT IN can also be used as comparison operators. IN is evaluated in the context of being “equal to any member of” a set of values. Thus, Example 1.2.5 is equivalent to Example 1.2.2.

**Example 1.2.5**

```
SELECT *
FROM COURSE
WHERE COURSE.CREDIT = 'U'
AND COURSE.COLLEGE IN ('Business', 'Engineering');
```

575

**Result:**

NAME	COURSE#	CREDIT	COLLEGE	HRS	DCODE
Operations Research	22QA375	U	Business	2	3
Supply Chain Analysis	22QA411	U	Business	3	3
Database Concepts	22IS330	U	Business	4	7

The logical operator NOT reverses the result of a logical expression. NOT can be used to precede any of the comparison operators (=, <>, <=, <, >=, >) as well as the word IN. Thus, Example 1.2.6 displays all courses offered for undergraduate credit that are not offered at either the College of Business or the College of Engineering. Once again, the Architectural History course does not appear in the result because null values only satisfy a WHERE clause that uses IS NULL or IS NOT NULL.

**Example 1.2.6**

```
SELECT *
FROM COURSE
WHERE COURSE.CREDIT = 'U'
AND COURSE.COLLEGE NOT IN ('Business', 'Engineering');
```

**Result:**

NAME	COURSE#	CREDIT	COLLEGE	HRS	DCODE
Intro to Economics	15ECON112	U	Arts and Sciences	3	1
Intro to Economics	18ECON123	U	Education	4	4

So far, the conditions in the WHERE clause have involved comparisons where the operands were of the form:

```
<column name><comparison operator><constant value>
```

and the constant value was either a numeric constant or a character constant (i.e., character string). However, as we will see, the operands of a comparison operator can be expressions, not just a simple column name or constant. For example, suppose we are interested in identifying all professors with a monthly salary that exceeds \$6,000.

### Example 1.2.7

```
SELECT *
FROM PROFESSOR
WHERE PROFESSOR.SALARY/12 > 6000;
```

#### Result:

NAME	EMPID	PHONE	DATEHIRED	DCODE	SALARY
Mike Faraday	FM49276	5235568492	01-MAY-96	1	92000
Chelsea Bush	BC65437	5235567777	01-MAY-93	3	77000
Tony Hopkins	HT54347	5235569977	20-JAN-97	3	77000
Alan Brodie	BA54325	5235569876	16-MAY-00	3	76000
Marie Curie	CM65436	5235569899	22-OCT-99	4	99000
John Nicholson	NJ43728	5235569999	22-JUN-03	4	99000

576

If we follow this Selection operation with a Projection operation, the query in Example 1.2.7 could be rewritten to display just the name and monthly salary of each qualifying professor, as follows:

```
SELECT NAME, PROFESSOR.SALARY/12 AS "Monthly Salary"
FROM PROFESSOR
WHERE PROFESSOR.SALARY/12 > 6000;
```

#### Result:

NAME	Monthly Salary
Mike Faraday	7666.66667
Chelsea Bush	6416.66667
Tony Hopkins	6416.66667
Alan Brodie	6333.33333
Marie Curie	8250
John Nicholson	8250

Rather than displaying the heading of the second column as **PROFESSOR.SALARY/12**, SQL allows the column name to be changed with the use of the keyword AS<sup>8</sup> and thus

---

<sup>8</sup>The keyword AS is optional and is often used in the column list to distinguish between the column name and column alias.

provide a more descriptive column heading as a column alias. It should be noted that this column alias cannot be used in other clauses associated with the SELECT statement. The TRUNC function of the form TRUNC (**PROFESSOR.SALARY**/12,2) or the ROUND function of the form ROUND (**PROFESSOR.SALARY**/12,2) could be used to either truncate or round each monthly salary to two places to the right of the decimal point.

WHERE clauses can also refer to a range of values through use of the comparison operator BETWEEN, which searches for rows in a specific range of values. For example, suppose we are interested in identifying the name and monthly salary of all professors whose monthly salary is between \$6,000 and \$7,000.

#### Example 1.2.8

```
SELECT PROFESSOR.NAME, PROFESSOR.SALARY/12 AS "Monthly Salary"
FROM PROFESSOR
WHERE PROFESSOR.SALARY/12 BETWEEN 6000 AND 7000;
```

#### Result:

NAME	Monthly Salary
Chelsea Bush	6416.66667
Tony Hopkins	6416.66667
Alan Brodie	6333.33333

577

The BETWEEN operator is inclusive (i.e., professors with a monthly salary of exactly \$6,000 or \$7,000 would also be included in the result) and can be applied to all data types. Further, NOT can also be used with the BETWEEN operator. For example, the query in Example 1.2.9 identifies all professors whose monthly salary is outside the range of \$6,000 to \$7,000. A close inspection of the results indicates that the two professors without a salary (John B. Smith and Tiger Woods) do not appear. Note: Professor John Smith (without a middle initial) has a \$45,000 annual salary.

#### Example 1.2.9

```
SELECT PROFESSOR.NAME, ROUND(PROFESSOR.SALARY/12, 0) AS
"Monthly Salary"
FROM PROFESSOR
WHERE PROFESSOR.SALARY/12 NOT BETWEEN 6000 AND 7000;
```

**Result:**

NAME	Monthly Salary
John Smith	3750
Mike Faraday	7667
Kobe Bryant	5500
Ram Raj	3667
Prester John	3667
Jessica Simpson	5583
Laura Jackson	3583
Marie Curie	8250
Jack Nicklaus	5583
John Nicholson	8250
Sunil Shetty	5333
Katie Shef	5417
Cathy Cobal	3750
Jeanine Troy	3750
Mike Crick	5750

Section 12.1.5 explains why John B. Smith and Tiger Woods are not included in the results for either Example 1.2.8 or Example 1.2.9. Section 12.2 illustrates how the WHERE and ON clauses have comparisons where operands of the form:

```
<column name><comparison operator><column name>
```

are used to express join conditions.

### 12.1.3 Examples of the Projection Operation

Examples illustrating use of the relational algebra Project operator in a Projection operation appear in Section 11.1.2 of Chapter 11. An SQL SELECT statement that corresponds to one of those examples is shown next, along with other examples illustrating how a Projection operation typically follows a Selection operation.

**Example 1.3.1 (Corresponds to Projection Example 1 in Section 11.1.2).** Which colleges offer courses?

#### SQL SELECT Statement:

```
SELECT COURSE.COLLEGE
FROM COURSE;
```

Execution of the SELECT statement given above, since it refers to only one of the six columns in the COURSE table, actually generates duplicate rows, and thus the result shown next does not constitute a relation. Since the Projection operation displays the content of the COLLEGE column from each row of the COURSE table, the null value in the COLLEGE column for the Architectural History course is displayed in the first row.

**Result:**<sup>9</sup>

```
COLLEGE
-----
```

```
Business
Business
Education
Arts and Sciences
Education
Business
Business
Business
Engineering
Business
Business
```

However, if the qualifier **DISTINCT** is specified as part of the **SELECT** statement, duplicate rows are removed. In other words, in SQL, without use of the qualifier **DISTINCT**, the general form of the **SELECT** statement is:

```
SELECT ALL <column list>
FROM <table list>
WHERE <condition>
```

where **ALL** retains duplicate values in queries (note that **ALL** is the default as compared to **DISTINCT**).

Thus, the revised **SELECT** statement with the **DISTINCT** qualifier produces the following result:

```
SELECT DISTINCT COURSE.COLLEGE
FROM COURSE;
```

**Result:**

```
COLLEGE
-----
```

```
Engineering
Education
Arts and Sciences
Business
```

Observe that a null value for an attribute is distinctly different from non-null values such as Engineering, Education, etc.

---

<sup>9</sup>Different implementations of SQL can result in the order of the rows displayed as a result of a Projection operation to be unpredictable. Use of an **ORDER BY** clause is the best way to control the order of the rows displayed.

**Example 1.3.2.** What is the name and address of each student?

**SQL SELECT Statement:**

```
SELECT STUDENT.NAME, STUDENT.ADDRESS
FROM STUDENT;
```

**Result (not all 16 students are shown):**

NAME	ADDRESS
Elijah Baley	2920 Scioto Street
Daniel Olive	338 Bishop Street
Wanda Seldon	3138 Probasco
.....	
.....	
.....	
Poppy Kramer	437 Love Lane
Sweety Kramer	748 Hope Avenue
Diana Jackson	2920 Scioto Street

In most cases, several relational algebra operations are applied one after the other (e.g., a Selection operation is followed by a Projection operation).

**Example 1.3.3.** What are the names of the courses offered in the College of Business?

**SQL SELECT Statement:**

```
SELECT COURSE.NAME
FROM COURSE
WHERE COURSE.COLLEGE = 'Business';
```

**Result:**

NAME
Database Concepts
Database Principles
Operations Research
Optimization
Principles of IS
Supply Chain Analysis
Systems Analysis

### 12.1.4 Grouping and Summarizing

SQL allows the grouping of rows into sets; it then can summarize data in such a way that one row is returned for each set. The GROUP BY and HAVING clauses facilitate the grouping, while built-in aggregate functions are used in conjunction with grouping.

Aggregate functions take as input a set of values, one from each row in a group of rows, and return one value as output. Common aggregate functions include:

- COUNT (x)—Counts the number of non-null values in a set of values
- SUM (x)—Sums all numbers in a set of values

- AVG (x)—Computes the average of a set of numbers in a set of values
- MAX (x)—Computes the maximum of a set of numbers in a set of values
- MIN (x)—Computes the minimum of a set of numbers in a set of values

*Note:* x represents a numeric column name.

**Example 1.4.1.** Count the number of salaries in the PROFESSOR table and, at the same time, display the sum of the salaries, the average salary, the maximum salary, and the minimum salary.

#### SQL SELECT Statement:

```
SELECT COUNT(SALARY), SUM(SALARY),
AVG(SALARY), MIN(SALARY), MAX(SALARY)
FROM PROFESSOR;
```

#### Result:

COUNT(SALARY)	SUM(SALARY)	AVG(SALARY)	MIN(SALARY)	MAX(SALARY)
18	1184000	65777.7778	43000	99000

581

Note that while there are 20 rows in the PROFESSOR table, the salaries of two professors (John B. Smith and Tiger Woods) are unknown, and thus a null value is stored in the respective **SALARY** column for these professors. Section 12.1.5 discusses the impact of null values on aggregate functions.

Aggregate functions are often used in conjunction with groups of rows rather than with all rows in a table. The column or columns on which the grouping takes place are called the grouping column(s). Doing this requires the application of the GROUP BY clause. The GROUP BY clause divides data into sets (i.e., groups) based on the contents of specified columns. The general form of the GROUP BY clause is:

GROUP BY column name, [,column name,...]

The grouping column(s) must also appear in the SELECT clause so that the value from applying each function to a group of rows appears along with the value of the grouping column(s).

**Example 1.4.2.** Count the number of students at each grade level.

#### SQL SELECT Statement:

```
SELECT STUDENT.GRADELEVEL, COUNT(*)
FROM STUDENT
GROUP BY STUDENT.GRADELEVEL;
```

#### Result:

GRADELEVEL	COUNT (*)
SR	3
FR	3
SO	3
GR	2
JR	5

**Example 1.4.3.** Calculate the average salary of the professors in each department.

**SQL SELECT Statement:**

```
SELECT PROFESSOR.DCODE, AVG(PROFESSOR.SALARY)
FROM PROFESSOR
GROUP BY PROFESSOR.DCODE;
```

**Result:**

DCODE	AVG (PROFESSOR.SALARY)
1	67666.6667
6	44000
4	88333.3333
3	68000
7	58000
9	57000

As mentioned above, grouping can be done on a combination of columns.

582

**Example 1.4.4.** Count the number of undergraduate and graduate courses offered by each department.

**SQL SELECT Statement:**

```
SELECT COURSE.DCODE, COURSE.CREDIT, COUNT(*)
FROM COURSE
GROUP BY COURSE.DCODE, COURSE.CREDIT;
```

**Result:**

DCODE	CREDIT	COUNT (*)
	U	1
1	U	1
4	U	1
7	G	3
4	G	1
7	U	1
3	U	2
6	G	1
3	G	1

Observe that a group is created for those courses for which the department code is a null value (see row 1 of the result). Use of a GROUP BY clause is often combined with an ORDER BY clause to display the result in a more meaningful manner.

The HAVING clause is used in conjunction with the GROUP BY clause to place restrictions on the rows returned by the GROUP BY clause in a query. A condition in a HAVING clause must always involve an aggregation. In addition, a HAVING clause cannot be used apart from an associated GROUP BY clause.

**Example 1.4.5.** Calculate the average salary of the professors in each department for those departments where the minimum salary of a professor is \$45,000.

**SQL SELECT Statement:**

```
SELECT PROFESSOR.DCODE, AVG(PROFESSOR.SALARY)
FROM PROFESSOR
GROUP BY PROFESSOR.DCODE
HAVING MIN(PROFESSOR.SALARY) >= 45000;
```

**Result:**

DCODE	Avg (PROFESSOR.SALARY)
1	67666.6667
4	88333.3333
7	58000
9	57000

Note that departments 3 and 6 do not appear in the results because there is at least one professor in each department with a salary less than \$45,000. In addition, observe that the calculation of the average salary of the professors in department 9 ignores Tiger Woods since his salary is a null value (i.e., is not available). The following section goes into more detail on the handling of null values. In addition, Section 12.2 describes how to join tables in SQL, which would represent one way to revise the query in Example 1.4.5 in order to display department names instead of department numbers.

583

### 12.1.5 Handling Null Values

It is important to understand how null values are handled in SQL. A data field without a value in it is said to contain a **null value**. A null value can occur in two situations: (a) where a value is unknown (e.g., the salary of an employee is unknown or unavailable), and (b) where a value is not meaningful (e.g., in a column representing “commission” for an employee who is not a salesperson and thus is not eligible for a commission).

A number field containing a null value is different from one that contains a value of zero. Null values are displayed as blanks, while zero values are displayed as numeric zeros. A null value will evaluate to null in any expression. For example, a null value added to 15 results in a null value; a null value minus a null value yields a null value, not zero. The aggregate function COUNT(\*) counts all null and not null rows in a table; COUNT (attribute) counts all rows whose attribute value is not null. Other SQL aggregate functions ignore null values in their computation.

This section consists of examples that illustrate the impact of null values in SQL. Each of the examples is based on the TEXTBOOK table whose structure and initial contents follow:

```
CREATE TABLE TEXTBOOK
  (ISBN      VARCHAR(14)  CONSTRAINT PK_10TB PRIMARY KEY,
   TITLE     VARCHAR(22)  CONSTRAINT NN_10TIT NOT NULL,
   YEAR      INTEGER(4),
   PUBLISHER CHAR(13));
```

```
SELECT *
FROM TEXTBOOK;
```

ISBN	TITLE	YEAR	PUBLISHER
000-66574998	Database Management	2007	Thomson
003-6679233	Linear Programming	2005	Prentice-Hall
001-55-435	Simulation Modeling	2009	Springer
118-99898-67	Systems Analysis	2008	Thomson
77898-8769	Principles of IS	2010	Prentice-Hall
0296748-99	Economics For Managers	2009	
0296437-1118	Programming in C++	2010	Thomson
012-54765-32	Fundamentals of SQL	2012	
111-11111111	Data Modeling	2013	

The behavior of null values in SQL can on occasion be surprising or unintuitive. In order to illustrate differences between how SQL handles a null value (defined as a character string zero characters long) versus a character string that consists of a single blank space, a null value appears in the **TEXTBOOK.PUBLISHER** column for both the titles *Economics For Managers* and *Fundamentals of SQL*<sup>10</sup>, while the value in the **TEXTBOOK.PUBLISHER** column for the title *Data Modeling* consists of a single blank space.

As expected, both the previous query and the following query (labeled Example 1.5.1) display the **TEXTBOOK.PUBLISHER** column for *Economics For Managers*, *Fundamentals of SQL*, and *Data Modeling* as blank values in the **TEXTBOOK.PUBLISHER** column.

### Example 1.5.1<sup>11</sup>

```
SELECT TEXTBOOK.PUBLISHER
FROM TEXTBOOK;
```

#### Result:

```
PUBLISHER
-----
Thomson
Prentice-Hall
Springer
Thomson
Prentice-Hall

Thomson
```

9 rows selected.

<sup>10</sup>In SQL-2003, a character value that is zero characters long is treated as a null value.

<sup>11</sup>Most versions of SQL contain a system variable that allows for the number of rows retrieved by the execution of a query to be displayed. The value of this variable will be shown in conjunction with various examples when appropriate. It is useful here as a way of showing that all rows from the **TEXTBOOK** table were selected.

A query (see Example 1.5.2) that displays the textbooks where the **TEXTBOOK.PUBLISHER** column contains a not null value excludes the rows associated with the titles *Economics For Managers* and *Fundamentals of SQL*, since the **TEXTBOOK.PUBLISHER** column for each of these textbooks contains a null value. The row associated with the title *Data Modeling* is not excluded since the value in the **TEXTBOOK.PUBLISHER** column for this title is not null (i.e., consists of a single blank space).

### Example 1.5.2

```
SELECT TEXTBOOK.TITLE, TEXTBOOK.PUBLISHER
FROM TEXTBOOK
WHERE TEXTBOOK.PUBLISHER IS NOT NULL;
```

#### Result:

TITLE	PUBLISHER
Database Management	Thomson
Linear Programming	Prentice-Hall
Simulation Modeling	Springer
Systems Analysis	Thomson
Principles of IS	Prentice-Hall
Programming in C++	Thomson
Data Modeling	

585

7 rows selected.

The only comparison operators that can be used with null values are IS NULL and IS NOT NULL. If any other operator (=, >, <>, etc.) is used with a null value, the result is always unknown.<sup>12</sup> In addition, since a NULL represents a lack of data, a null value cannot be equal or unequal to any other value, even another NULL. Examples 1.5.3, 1.5.4, and 1.5.5 illustrate the fact that only IS NULL and IS NOT NULL can be used as comparison operators with null values.

### Example 1.5.3

```
SELECT *
FROM TEXTBOOK
WHERE TEXTBOOK.PUBLISHER = NULL;
```

#### Result:

no rows selected

### Example 1.5.4

```
SELECT *
FROM TEXTBOOK
WHERE TEXTBOOK.PUBLISHER <> NULL;
```

---

<sup>12</sup>SQL-2003 treats conditions evaluating to unknown as FALSE.

**Result:**

```
no rows selected
```

While conditional expressions of the form “WHERE X = NULL” and “WHERE X <> NULL” are illegal, SQL does not generate a syntax error. This can create serious problems in cases where “WHERE X IS NULL” would otherwise cause one or more rows to be selected.

**Example 1.5.5**

```
SELECT *
FROM TEXTBOOK
WHERE TEXTBOOK.PUBLISHER IS NULL;
```

**Result:**

ISBN	TITLE	YEAR	PUBLISHER
0296748-99	Economics For Managers	2009	
012-54765-32	Fundamentals of SQL	2012	

586

2 rows selected.

The SELECT statement in Example 1.5.6 demonstrates that there are five distinct publishers in the TEXTBOOK table and reflects the fact that a null value in a column can be distinguished from a column that contains a single blank space. The SELECT statement in Example 1.5.7 (with the IS NOT NULL condition) indicates that the publisher whose name consists of a single blank space can be distinguished from the publisher with a null value for its name.

**Example 1.5.6**

```
SELECT DISTINCT TEXTBOOK.PUBLISHER
FROM TEXTBOOK;
```

**Result:**

PUBLISHER
Springer

Thomson  
Prentice-Hall

5 rows selected.

**Example 1.5.7**

```
SELECT DISTINCT TEXTBOOK.PUBLISHER
FROM TEXTBOOK
WHERE TEXTBOOK.PUBLISHER IS NOT NULL;
```

**Result:**

```
PUBLISHER
-----
Springer
Thomson
Prentice-Hall
```

4 rows selected.

Example 1.5.8 uses the COUNT function to count the number of rows in the TEXTBOOK table and returns the result in a single table with a single column. Recall that when COUNT(\*) is used, SQL focuses on the presence of rows rather than values appearing in a column.

**Example 1.5.8**

```
SELECT COUNT (*)
FROM TEXTBOOK;
```

**Result:**

587

```
COUNT (*)
-----
9
```

1 row selected.

When the COUNT function refers to a column, it behaves like the other aggregate functions and ignores null values (see Section 12.1.4). Thus the query in Example 1.5.9 displays the number of rows in the TEXTBOOK table with something other than a null value in the TEXTBOOK.PUBLISHER column.

**Example 1.5.9**

```
SELECT COUNT (TEXTBOOK.PUBLISHER)
FROM TEXTBOOK;
```

**Result:**

```
COUNT (TEXTBOOK.PUBLISHER)
-----
7
```

1 row selected.

Numeric functions, such as the COUNT function, are associated with columns of output whose width is equal to the number of characters required to display the name of the function plus its argument(s). Often, a column alias, such as the one used in Example 1.5.10, is used to provide a more descriptive column heading. Observe how the width of the column has been adjusted to display the entire column alias (i.e., heading).

**Example 1.5.10**

```
SELECT COUNT(TEXTBOOK.PUBLISHER) "Number of Publishers"
FROM TEXTBOOK;
```

**Result:**

```
Number of Publishers
-----
7
```

1 row selected.

In Example 1.5.10, the two rows in the TEXTBOOK table with null publishers are ignored by the COUNT function. This can be verified by the query in Example 1.5.11, which counts the number of distinct not null values in the **TEXTBOOK.PUBLISHER** column.

**Example 1.5.11**

**588**

```
SELECT COUNT(DISTINCT TEXTBOOK.PUBLISHER) "Number of Distinct Publishers"
FROM TEXTBOOK;
```

**Result:**

```
Number of Distinct Publishers
-----
4
```

1 row selected.

Aggregate functions are frequently applied to groups of rows in a table rather than to all rows in a table. The SELECT statement in Example 1.5.12 reflects the fact that there are five distinct publishers in the TEXTBOOK table and counts the number of rows (i.e., the COUNT (\*) function is focusing on the presence of a row) associated with each distinct publisher.

**Example 1.5.12**

```
SELECT TEXTBOOK.PUBLISHER, COUNT(*)
FROM TEXTBOOK
GROUP BY TEXTBOOK.PUBLISHER;
```

**Result:**

PUBLISHER	COUNT(*)
Springer	1
	2
Thomson	3
Prentice-Hall	2
	1

5 rows selected.

Observe that the second row displayed is associated with the two textbooks with a null value in the **TEXTBOOK.PUBLISHER** column.

The SELECT statement in Example 1.5.13 also recognizes that there are five distinct publishers, but since group functions ignore the presence of null values in the **TEXTBOOK.PUBLISHER** column, the accumulator set up for the publisher whose value is a null value never has its initial value of zero incremented.

### Example 1.5.13

```
SELECT TEXTBOOK.PUBLISHER, COUNT(TEXTBOOK.PUBLISHER)
FROM TEXTBOOK
GROUP BY TEXTBOOK.PUBLISHER;
```

#### Result:

PUBLISHER	COUNT (TEXTBOOK.PUBLISHER)
Springer	1
	0
Thomson	3
Prentice-Hall	2
	1

5 rows selected.

The SELECT statement in Example 1.5.14 works as expected, since the WHERE clause places the condition that the value in the **TEXTBOOK.PUBLISHER** column must be not null before that publisher can be used to form a group. The SELECT statement in Example 1.5.15 also works as expected since the WHERE clause focuses only on the publisher whose name consists of a single blank space. Finally, the SELECT statement in Example 1.5.16 serves as a reminder that the only publishers whose name consists of something other than a single blank space are Thomson, Prentice-Hall, and Springer. Recall, using a `<>` comparison operator in the evaluation of whether a null value differs from a single blank space produces an unknown result (which is treated as false).

### Example 1.5.14

```
SELECT TEXTBOOK.PUBLISHER, COUNT(*)
FROM TEXTBOOK
WHERE TEXTBOOK.PUBLISHER IS NOT NULL
GROUP BY TEXTBOOK.PUBLISHER;
```

#### Result:

PUBLISHER	COUNT (*)
Springer	1
Thomson	3
Prentice-Hall	2
	1

4 rows selected.

**Example 1.5.15**

```
SELECT TEXTBOOK.PUBLISHER, COUNT(*)
FROM TEXTBOOK
WHERE TEXTBOOK.PUBLISHER = ''
GROUP BY TEXTBOOK.PUBLISHER;
```

**Result:**

PUBLISHER	COUNT (*)
	1

1 row selected.

**Example 1.5.16**

```
SELECT TEXTBOOK.PUBLISHER, COUNT(*)
FROM TEXTBOOK
WHERE TEXTBOOK.PUBLISHER <> ''
GROUP BY TEXTBOOK.PUBLISHER;
```

590

**Result:**

PUBLISHER	COUNT (*)
Springer	1
Thomson	3
Prentice-Hall	2

3 rows selected.

The SELECT statements in Examples 1.5.17, 1.5.18, and 1.5.19 illustrate the impact of a null value in queries that involve multiple conditions. In Example 1.5.17, the first condition selects the rows associated with the four not null publishers, while the second condition selects the rows associated with the three publishers Thomson, Prentice-Hall, and Springer. Since OR is used to connect the two conditions, groups are formed and counts accumulated for the four not null publishers.

**Example 1.5.17**

```
SELECT TEXTBOOK.PUBLISHER, COUNT(*)
FROM TEXTBOOK
WHERE TEXTBOOK.PUBLISHER IS NOT NULL
OR TEXTBOOK.PUBLISHER <> ''
GROUP BY TEXTBOOK.PUBLISHER;
```

**Result:**

PUBLISHER	COUNT (*)
Springer	1
Thomson	3
Prentice-Hall	2
	1

4 rows selected.

In Example 1.5.18, since AND is used to connect the two conditions, groups are formed only for those publishers that satisfy both conditions (i.e., Thomson, Prentice-Hall, and Springer).

**Example 1.5.18**

```
SELECT TEXTBOOK.PUBLISHER, COUNT(*)
FROM TEXTBOOK
WHERE TEXTBOOK.PUBLISHER IS NOT NULL
AND TEXTBOOK.PUBLISHER <> ''
GROUP BY TEXTBOOK.PUBLISHER;
```

591

**Result:**

PUBLISHER	COUNT (*)
Springer	1
Thomson	3
Prentice-Hall	2

3 rows selected.

The SELECT statement in Example 1.5.19 illustrates a situation where the first condition selects rows associated with the **TEXTBOOK.PUBLISHER** column containing a null value and the second condition selects the rows associated with the three publishers other than the single-space publisher. Since OR is used to connect the two conditions, groups are formed and counts accumulated for Thomson, Prentice-Hall, and Springer, and the rows associated with the **TEXTBOOK.PUBLISHER** column containing a null value.

**Example 1.5.19**

```
SELECT TEXTBOOK.PUBLISHER, COUNT(*)
FROM TEXTBOOK
WHERE TEXTBOOK.PUBLISHER IS NULL
OR TEXTBOOK.PUBLISHER <> ''
GROUP BY TEXTBOOK.PUBLISHER;
```

**Result:**

PUBLISHER	COUNT (*)
Springer	1
	2
Thomson	3
Prentice-Hall	2

4 rows selected.

The SELECT statement in Example 1.5.20 introduces the use of the operator IN to test whether a value is contained within a set of values.

**Example 1.5.20**

```
SELECT *
FROM TEXTBOOK
WHERE TEXTBOOK.PUBLISHER IN ('Thomson', 'Springer');
```

592

**Result:**

ISBN	TITLE	YEAR	PUBLISHER
000-66574998	Database Management	2007	Thomson
001-55-435	Simulation Modeling	2009	Springer
118-99898-67	Systems Analysis	2008	Thomson
0296437-1118	Programming in C++	2010	Thomson

4 rows selected.

NOT IN tests for whether a value does not appear within a set of values. Example 1.5.21 illustrates how null values only satisfy a WHERE clause that uses IS NULL or IS NOT NULL since the titles *Economics For Managers* and *Fundamentals of SQL* do not appear in the results but Prentice-Hall and the single-space publisher do.

**Example 1.5.21**

```
SELECT *
FROM TEXTBOOK
WHERE TEXTBOOK.PUBLISHER NOT IN ('Thomson', 'Springer');
```

**Result:**

ISBN	TITLE	YEAR	PUBLISHER
003-6679233	Linear Programming	2005	Prentice-Hall
77898-8769	Principles of IS	2010	Prentice-Hall
111-11111111	Data Modeling	2013	

3 rows selected.

### 12.1.6 Pattern Matching in SQL

In the context of the TEXTBOOK table, pattern matching would be useful if we were trying to find all textbooks used in Madeira College with a specific word or phrase in their titles. Examples include (a) all textbooks with the word “Introduction” in the title, (b) all textbooks whose title includes the words “Information System” or “Accounting System,” and (c) all textbooks with titles that include “Programming.” SQL-2003 supports pattern matching through the use of the LIKE operator in conjunction with the two wildcard characters—the percent character (%) and the underscore character (\_).<sup>13</sup> The percent character represents *a series of one or more unspecified characters*, while the underscore character represents *exactly one character*. Let us continue to use the data associated with the TEXTBOOK table as a vehicle to explore use of the LIKE operator. Given the number of rows and columns associated with the TEXTBOOK table, the examples that follow have limited practical application but instead are intended to simply illustrate how the LIKE operator works.

The SELECT statement in Example 1.6.1 searches for all textbooks such that the first two characters of their ISBN begin with the digit 1 and are followed by any alphanumeric character. Observe that while the textbooks with the titles *Systems Analysis* and *Data Modeling* seem to satisfy this request, no rows are selected. Likewise, the SELECT statement in Example 1.6.2 searches for all textbooks whose title contains the letter “i” in the second character position. While two textbooks would appear to satisfy this request, no rows are selected.

#### Example 1.6.1

```
SELECT TEXTBOOK.TITLE, TEXTBOOK.YEAR
FROM TEXTBOOK
WHERE TEXTBOOK.TITLE LIKE '1_';
```

#### Result:

no rows selected.

#### Example 1.6.2

```
SELECT TEXTBOOK.TITLE
FROM TEXTBOOK
WHERE TEXTBOOK.TITLE LIKE '_i';
```

#### Result:

no rows selected.

The SELECT statements in Examples 1.6.1 and 1.6.2 return no rows because no textbook has an ISBN number that is exactly two characters long, nor is there any textbook title that has the letter “i” in the second character position and is exactly two characters long. As illustrated in Example 1.6.3, revising Example 1.6.2 by adding the % sign as a

---

<sup>13</sup>Although the SQL-2003 standard specifies use of the underscore character (\_) and percent character (%), some implementations of SQL use other characters to represent a single character or a series of characters.

wildcard character searches for textbook titles with the letter “i” in the second character position but allows the title to have as many as 22 characters (the width of the **TEXTBOOK.TITLE** column).

### Example 1.6.3

```
SELECT TEXTBOOK.TITLE  
FROM TEXTBOOK  
WHERE TEXTBOOK.TITLE LIKE '_i%';
```

#### Result:

```
TITLE  
-----  
Linear Programming  
Simulation Modeling
```

2 rows selected.

Examples 1.6.4 and 1.6.5 illustrate that the LIKE operator is case sensitive.

594

### Example 1.6.4

```
SELECT TEXTBOOK.TITLE  
FROM TEXTBOOK  
WHERE TEXTBOOK.TITLE LIKE 'P%';
```

#### Result:

```
TITLE  
-----  
Principles of IS  
Programming in C++
```

2 rows selected.

### Example 1.6.5

```
SELECT TEXTBOOK.TITLE  
FROM TEXTBOOK  
WHERE TEXTBOOK.TITLE LIKE 'p%';
```

#### Result:

no rows selected.

The SELECT statements in Examples 1.6.6 and 1.6.7 represent rather unusual uses of the LIKE operator. Example 1.6.6 searches for the titles of all textbooks that contain the letter “e,” while Example 1.6.7 displays the titles of all textbooks.

### Example 1.6.6

```
SELECT TEXTBOOK.TITLE  
FROM TEXTBOOK  
WHERE TEXTBOOK.TITLE LIKE '%e%';
```

**Result:**

```
TITLE
-----
Database Management
Linear Programming
Simulation Modeling
Systems Analysis
Principles of IS
Economics For Managers
Fundamentals of SQL
Data Modeling
```

8 rows selected.

**Example 1.6.7**

```
SELECT TEXTBOOK.TITLE
FROM TEXTBOOK
WHERE TEXTBOOK.TITLE LIKE '%' ;
```

595

**Result:**

```
TITLE
-----
Database Management
Linear Programming
Simulation Modeling
Systems Analysis
Principles of IS
Economics For Managers
Programming in C++
Fundamentals of SQL
Data Modeling
```

9 rows selected.

As shown in the SELECT statement in Example 1.6.8, the string '%' cannot match a null value, as the two publishers with a null value in the **TEXTBOOK.PUBLISHER** column do not appear in the results.

**Example 1.6.8**

```
SELECT *
FROM TEXTBOOK
WHERE TEXTBOOK.PUBLISHER LIKE '%' ;
```

**Result:**

ISBN	TITLE	YEAR	PUBLISHER
000-66574998	Database Management	2007	Thomson
003-6679233	Linear Programming	2005	Prentice-Hall
001-55-435	Simulation Modeling	2009	Springer
118-99898-67	Systems Analysis	2008	Thomson
77898-8769	Principles of IS	2010	Prentice-Hall
0296437-1118	Programming in C++	2010	Thomson
111-11111111	Data Modeling	2013	

7 rows selected.

CHAR( $\ell$ ) and VARCHAR( $\ell$ ) are string data types (see Table 10.1) commonly used to define fixed-length and variable-length character data. A CHAR( $\ell$ ) data type, where  $\ell$  represents the length of the column, has a maximum size of 255 characters in most DBMSs; blank characters are added to the data should the number of characters be less than  $\ell$ . A VARCHAR( $\ell$ ) data type, where  $\ell$  also represents the length of the column, has a maximum size of 2,000 characters in most DBMSs. A VARCHAR( $\ell$ ) data type does not append blank characters to the data if the number of characters is less than  $\ell$ . CHAR( $\ell$ ) and VARCHAR( $\ell$ ) data types can produce different results in some comparisons that involve the LIKE operator. For example, the query in Example 1.6.9 searches for and locates all titles that end with the letter “s.”

**Example 1.6.9**

```
SELECT *
FROM TEXTBOOK
WHERE TEXTBOOK.TITLE LIKE '%s';
```

**Result:**

ISBN	TITLE	YEAR	PUBLISHER
118-99898-67	Systems Analysis	2008	Thomson
0296748-99	Economics For Managers	2009	

2 rows selected.

However, had the **TEXTBOOK.TITLE** column been defined as a CHAR(22) data type instead of as a VARCHAR(22) data type, only one of the titles, *Economics For Managers* (a title with an “s” in the 22nd character position) would have been located since the 22nd character position in the title, *Systems Analysis*, would have contained a single blank space character (i.e., the rightmost “s” in *Systems Analysis* would have been in the 16th character position, with character positions 17–22 containing blank spaces).

SQL allows for the definition of an escape character in cases where either a percent character (%) or an underscore character (\_) stand for themselves as part of the search. For example, suppose you need to identify the name of each table in the Madeira College

database with an underscore character as part of its table name. This would be possible with the following SELECT statement:

**Example 1.6.10<sup>14</sup>**

```
SELECT TABLE_NAME
FROM USER_TABLES
WHERE TABLE_NAME LIKE '%/_%'
ESCAPE '/';
```

ESCAPE is used here to declare the slash (/) as an escape character so that it can be prefixed to the underscore character in the character string expression used in the LIKE operator.

## 12.2 SQL QUERIES BASED ON BINARY OPERATORS

Recall that binary operators “operate” on two relations and are of four types: (a) the Cartesian Product operator, (b) set theoretic operators, (c) join operators, and (d) the divide operator. The examples in this section are based on the Madeira College tables shown in Figure 11.1 of Chapter 11.

597

### 12.2.1 The Cartesian Product Operation

An SQL SELECT statement that references two or more tables and does not include a WHERE clause always involves a Cartesian Product operation. The Cartesian Product operation by itself is generally meaningless. However, all joins actually begin as a Cartesian Product followed by a WHERE condition that selects only the rows that make sense in the context of the question. In other words, a Cartesian Product is useful when followed first by a Selection operation that matches values of attributes coming from the component relations (technically, a Cartesian Product operation followed by a Selection operation is a Join operation) and then by a Projection operation that selects certain columns from the combined result.

**Example 2.1.1.** What is the product of the relations D and C derived from the DEPARTMENT and COURSE relations? See Figure 11.5 of Chapter 11 for the content of the D and C relations.

**SQL SELECT Statement:**

```
SELECT * FROM C CROSS JOIN D;
```

In the SQL-2003 standard, the CROSS keyword, combined with the JOIN keyword, is used in the FROM clause to create a Cartesian Product. Sometimes, a Cartesian Product is referred to as a Cross Join. The first 24 rows of the result of this Cartesian Product appear next. Since there are 12 rows in C and six rows in D, a total of 72 rows are produced as a result of the concatenation of C and D. The first 12 rows of the result constitute the concatenation of the first row of D (observe how the final three columns remain the same)

---

<sup>14</sup>USER\_TABLES is the name of a data dictionary table comprised of columns that contain data about various tables that comprise the Madeira College database. Included in this table is a column that records the name of each table.

with the 12 rows in C. Observe that the next 12 rows of the result shows the second row of D with each of the 12 rows of C.

**Result:**

NAME	COURSE#	CREDIT	DCODE	NAME	DCODE	COLLEGE
Intro to Economics	15ECON112	U		1 Economics	1	Arts and Sciences
Operations Research	22QA375	U		3 Economics	1	Arts and Sciences
Intro to Economics	18ECON123	U		4 Economics	1	Arts and Sciences
Supply Chain Analysis	22QA411	U		3 Economics	1	Arts and Sciences
Principles of IS	22IS270	G		7 Economics	1	Arts and Sciences
Programming in C++	20ECES212	G		6 Economics	1	Arts and Sciences
Optimization	22QA888	G		3 Economics	1	Arts and Sciences
Financial Accounting	18ACCT801	G		4 Economics	1	Arts and Sciences
Database Concepts	22IS330	U		7 Economics	1	Arts and Sciences
Database Principles	22IS832	G		7 Economics	1	Arts and Sciences
Systems Analysis	22IS430	G		7 Economics	1	Arts and Sciences
Architectural History	05ARCH101	U		Economics	1	Arts and Sciences
Intro to Economics	15ECON112	U	1	QA/QM	3	Business
Operations Research	22QA375	U	3	QA/QM	3	Business
Intro to Economics	18ECON123	U	4	QA/QM	3	Business
Supply Chain Analysis	22QA411	U	3	QA/QM	3	Business
Principles of IS	22IS270	G	7	QA/QM	3	Business
Programming in C++	20ECES212	G	6	QA/QM	3	Business
Optimization	22QA888	G	3	QA/QM	3	Business
Financial Accounting	18ACCT801	G	4	QA/QM	3	Business
Database Concepts	22IS330	U	7	QA/QM	3	Business
Database Principles	22IS832	G	7	QA/QM	3	Business
Systems Analysis	22IS430	G	7	QA/QM	3	Business
Architectural History	05ARCH101	U		QA/QM	3	Business

598

When a Cartesian Product operation is accompanied by a Selection operation, an Inner Join results. As shown next, using the SQL-2003 standard, the words INNER JOIN<sup>15</sup> are used in the FROM clause instead of the words CROSS JOIN, and the ON clause is used to specify the join condition(s).

**Example 2.1.2.** What are the student IDs of those students who took a section of a course that meets on a Tuesday?<sup>16</sup>

<sup>15</sup>Use of the word “INNER” is optional.

<sup>16</sup>Recall that the letter “A” in the fourth character position of the SECTION# represents the Fall quarter. The letters “S,” “W,” and “U” represent the Spring, Winter, and Summer quarters, respectively.

**SQL SELECT Statement:**

```
SELECT TAKES.SID, TIME
FROM TAKES INNER JOIN SECTION
ON SECTION.SECTION# = TAKES.SECTION#
AND SECTION.TIME LIKE 'T%';
```

**Result:**

SID	TIME
BG66765	T1015
KP78924	T1015
KS39874	T1015
BE76598	T1045

4 rows selected.

Observe that the ON clause given above effectively constrains the concatenation of a row from TAKES with a row from SECTION to the condition where the SECTION# in TAKES matches a SECTION# in SECTION and the day of the week when the section meets recorded in fourth character position in the SECTION# column contains the letter “T.”

599

Reviewing the content of the SECTION and TAKES tables provides an explanation of this result. Since there are 11 rows in the SECTION table and 11 rows in the TAKES tables, the Cartesian Product operation results in an unnamed table with 121 rows and nine columns (there are six columns in SECTION and three columns in TAKES). Since only five rows in SECTION involve sections offered on a Tuesday, this Cartesian Product operation yields a result where 55 of these 121 rows contain a value in the TIME column that begins with the letter “T.” However, the portion of the Selection operation that requires SECTION.SECTION# = TAKES.SECTION# selects only four of these 55 rows. Finally, the SQL Projection operation displays the Student ID and time that appear on these four rows.<sup>17</sup>

### **12.2.2 SQL Queries Involving Set Theoretic Operations**

Union, Intersection, and Difference are three set theoretic operators used to merge elements of two union-compatible sets. The examples shown next illustrate the incorporation of these operators in SQL statements that involve the tables R and S derived from the SECTION table, where R contains those sections offered during a Fall quarter (the fourth character in the SECTION# is an A) and S contains those sections offered in a room located in Lindner Hall.<sup>18</sup>

---

<sup>17</sup>The actual execution of this SELECT statement would differ from this description. For example, a Selection operation on the SECTION table would occur first and result in “selecting” the five rows where SECTION.TIME LIKE ‘T%’. A Cartesian Product operation would follow, concatenating the five rows with the 11 rows in the TAKES table. This would be followed by a second Selection operation that would “select” the four rows in the concatenated result where SECTION.TIME LIKE ‘T%’ and where TAKES.SECTION# = SECTION.SECTION#. The Projection operation on the TAKES.SID and SECTION.TIME columns is the final operation executed.

<sup>18</sup>The tables R and S were created simply as a means to illustrate use of the UNION, INTERSECT, and DIFFERENCE operators in an SQL query. Each of the examples in this section can be expressed by a query that refers to just the SECTION table.

**RELATION R**

SECTION#	TIME	MAXST	ROOM	COURSE#	PROFID
101A2014	T1015	25		22QA375	HT54347
901A2013	W1800	35	Rhodes 611	22IS270	SK85977
902A2013	H1700	25	Lindner 108	22IS270	SK85977
102A2014	W1800		Baldwin 437	20ECE212	RR79345
104A2014	H1700	29	Lindner 108	22IS330	SK85977

**RELATION S**

SECTION#	TIME	MAXST	ROOM	COURSE#	PROFID
902A2013	H1700	25	Lindner 108	22IS270	SK85977
201S2013	T1045	29	Lindner 110	22IS330	SK85977
301S2013	H1045	29	Lindner 110	22IS330	CC49234
104A2014	H1700	29	Lindner 108	22IS330	SK85977

600

**Example 2.2.1 (Corresponds to Union Example in Section 11.2.2 of Chapter 11).** Display the union of R and S (i.e., those sections offered in either the Fall quarter or in a room located in Lindner Hall, or offered in both the Fall quarter and in a room located in Lindner Hall).

**SQL SELECT Statement:**

```
SELECT *
FROM R
UNION
SELECT *
FROM S;
```

Note that when duplicate rows exist in tables, only one row per set of duplicates is displayed in the result when using the UNION operator unless UNION ALL has been used.

**Result:**

SECTION#	TIME	MAXST	ROOM	COURSE#	PROFID
101A2014	T1015	25		22QA375	HT54347
102A2014	W1800		Baldwin 437	20ECE212	RR79345
104A2014	H1700	29	Lindner 108	22IS330	SK85977
201S2013	T1045	29	Lindner 110	22IS330	SK85977
301S2013	H1045	29	Lindner 110	22IS330	CC49234
901A2013	W1800	35	Rhodes 611	22IS270	SK85977
902A2013	H1700	25	Lindner 108	22IS270	SK85977

**Example 2.2.2 (Corresponds to Intersection Example in Section 11.2.2).** Display the intersection of R and S (i.e., those sections offered in both the Fall quarter and in a room located in Lindner Hall).

**SQL SELECT Statement:**

```
SELECT *
FROM R
    INTERSECT
SELECT *
FROM S;
```

**Result:**

SECTION#	TIME	MAXST	ROOM	COURSE#	PROFID
104A2014	H1700	29	Lindner 108	22IS330	SK85977
902A2013	H1700	25	Lindner 108	22IS270	SK85977

**Example 2.2.3 (Corresponds to Difference Example 1 in Section 11.2.2).** Display the difference R minus S (i.e., those sections offered in the Fall quarter but not in a room located in Lindner Hall).

**SQL SELECT Statement:**

```
SELECT *
FROM R
    MINUS
SELECT *
FROM S;
```

601

**Result:**

SECTION#	TIME	MAXST	ROOM	COURSE#	PROFID
101A2014	T1015	25		22QA375	HT54347
102A2014	W1800		Baldwin 437	20ECES212	RR79345
901A2013	W1800	35	Rhodes 611	22IS270	SK85977

The word MINUS is used in Oracle's SQL. The word EXCEPT is part of the SQL-2003 standard.

**Example 2.2.4 (Corresponds to Difference Example 2 in Section 11.2.2).** Using the data in the relations R and S given previously, form the difference S minus R (i.e., those sections offered in a room located in Lindner Hall but not in the Fall quarter).

**SQL SELECT Statement:**

```
SELECT *
FROM S
    MINUS
SELECT *
FROM R;
```

**Result:**

SECTION#	TIME	MAXST	ROOM	COURSE#	PROFID
201S2013	T1045	29	Lindner 110	22IS330	SK85977
301S2013	H1045	29	Lindner 110	22IS330	CC49234

### 12.2.3 Join Operations

Four types of relational algebra joins were discussed earlier in Section 11.2.3 of Chapter 11. The Equijoin involves join conditions with equality comparisons only and produces a new relation that contains all columns associated with the join condition. On the other hand, a Natural Join omits one of each pair of columns associated with the join condition, thus eliminating the possibility of obtaining a result that contains columns with the same set of values. A Theta Join, the third type of join, is based on a join condition that does not involve an equality comparison. A fourth type of join is the Outer Join. An Outer Join is used when it is desired to include each row in a relation in the result even if the row does not contain an attribute value satisfying the join condition. Examples of these four types of joins using SQL are illustrated here in the context of the SECTION and TAKES tables.

In the SECTION table, a Section# is assigned to each course offered during a particular quarter and year. This allows Section# 101A2014 to be assigned to the course 22QA375 offered during the Fall quarter in 2014.

The TAKES table records the sections of the courses taken by various students. Observe that the concatenated primary key of TAKES is (TAKES.SECTION#, TAKES.SID).

602

#### 12.2.3.1 An Example of an Equijoin Operation

The Equijoin of TAKES and SECTION involves all the attributes that share the same domains in TAKES and SECTION. This allows the result of the join to include only rows that match information on a section taken by a student (recorded in TAKES) with the information about that section (recorded in SECTION).

**Example 2.3.1.** Join the SECTION and TAKES tables over their common attributes (in our case, there is just one common attribute, SECTION#).

##### SQL SELECT Statement:

```
SELECT *
FROM SECTION JOIN TAKES
ON SECTION.SECTION# = TAKES.SECTION#;
```

##### Result:

SECTION#	TIME	MAXST	ROOM	COURSE#	PROFID	SECTION#	GRADE	SID
101A2014	T1015	25		22QA375	HT54347	101A2014 A	KP78924	
101A2014	T1015	25		22QA375	HT54347	101A2014 A	KS39874	
101A2014	T1015	25		22QA375	HT54347	101A2014 B	BG66765	
201S2013	T1045	29	Lindner 110	22IS330	SK85977	201S2013 C	BE76598	
104A2014	H1700	29	Lindner 108	22IS330	SK85977	104A2014 B	KJ56656	
104A2014	H1700	29	Lindner 108	22IS330	SK85977	104A2014 A	KP78924	
104A2014	H1700	29	Lindner 108	22IS330	SK85977	104A2014 A	KS39874	
401W2014	M1000	33	Braunstien 211	22IS832	CC49234	401W2014 A	KS39874	
104A2014	H1700	29	Lindner 108	22IS330	SK85977	104A2014 A	BE76598	
401W2014	M1000	33	Braunstien 211	22IS832	CC49234	401W2014 B	BG66765	
104A2014	H1700	29	Lindner 108	22IS330	SK85977	104A2014 C	GS76775	

Note that the result of this Equijoin displays for each section taken by a student (see columns 7–9) a complete description of the section (see columns 1–6). The following example illustrates how the combination of a Natural Join operation and a Projection operation allows a less cluttered result to be displayed.

### 12.2.3.2 Examples of Natural Join Operations

Since the result of an Equijoin results in pairs of attributes with identical values, a Natural Join operation was created to omit the second (and superfluous) attribute(s) in an Equijoin condition. In direct violation of the requirement of the relational data model that attributes have unique names over the entire relational schema, the standard definition of NATURAL JOIN requires that the two join attributes (or each pair of join attributes) have the same name in both relations.

**Example 2.3.2.** Join the SECTION and TAKES tables over their common attributes.

SQL-2003 supports three approaches for representing a Natural Join. One uses the NATURAL JOIN keyword, another uses JOIN ... USING, and the third uses JOIN ... ON. The first two approaches can only be used if the requirement that attributes have unique names over the entire relational schema is not enforced and tables that contain columns with the same name are allowed to exist. In addition, in order to display just the distinct column names, the third approach based on the SQL-2003 syntax requires that each column name appear in the <column list>.

#### Approach 1

```
SELECT * FROM tablename1 NATURAL JOIN tablename2

SELECT *
FROM SECTION NATURAL JOIN TAKES;
```

#### Result:

SECTION#	TIME	MAXST	ROOM	COURSE#	PROFID	GRADE	SID
101A2014	T1015	25		22QA375	HT54347	A	KP78924
101A2014	T1015	25		22QA375	HT54347	A	KS39874
101A2014	T1015	25		22QA375	HT54347	B	BG66765
201S2013	T1045	29	Lindner 110	22IS330	SK85977	C	BE76598
104A2014	H1700	29	Lindner 108	22IS330	SK85977	B	KJ56656
104A2014	H1700	29	Lindner 108	22IS330	SK85977	A	KP78924
104A2014	H1700	29	Lindner 108	22IS330	SK85977	A	KS39874
401W2014	M1000	33	Braunstien 211	22IS832	CC49234	A	KS39874
104A2014	H1700	29	Lindner 108	22IS330	SK85977	A	BE76598
401W2014	M1000	33	Braunstien 211	22IS832	CC49234	B	BG66765
104A2014	H1700	29	Lindner 108	22IS330	SK85977	C	GS76775

## Chapter 12

### Approach 2

```
SELECT * FROM tablename1 JOIN tablename2 USING (columnname_a, columnname_b, ..., columnname_n)
```

```
SELECT *
FROM SECTION JOIN TAKES
USING (SECTION#);
```

### Result:

SECTION#	TIME	MAXST	ROOM	COURSE#	PROFID	GRADE	SID
101A2014	T1015	25		22QA375	HT54347	A	KP78924
101A2014	T1015	25		22QA375	HT54347	A	KS39874
101A2014	T1015	25		22QA375	HT54347	B	BG66765
201S2013	T1045	29	Lindner 110	22IS330	SK85977	C	BE76598
104A2014	H1700	29	Lindner 108	22IS330	SK85977	B	KJ56656
104A2014	H1700	29	Lindner 108	22IS330	SK85977	A	KP78924
104A2014	H1700	29	Lindner 108	22IS330	SK85977	A	KS39874
401W2014	M1000	33	Braunstien 211	22IS832	CC49234	A	KS39874
104A2014	H1700	29	Lindner 108	22IS330	SK85977	A	BE76598
401W2014	M1000	33	Braunstien 211	22IS832	CC49234	B	BG66765
104A2014	H1700	29	Lindner 108	22IS330	SK85977	C	GS76775

604

### Approach 3

```
SELECT SECTION.* , TAKES.GRADE, TAKES.SID
FROM SECTION JOIN TAKES ON
SECTION.SECTION# = TAKES.SECTION#;
```

### Result:

SECTION#	TIME	MAXST	ROOM	COURSE#	PROFID	GRADE	SID
101A2014	T1015	25		22QA375	HT54347	A	KP78924
101A2014	T1015	25		22QA375	HT54347	A	KS39874
101A2014	T1015	25		22QA375	HT54347	B	BG66765
201S2013	T1045	29	Lindner 110	22IS330	SK85977	C	BE76598
104A2014	H1700	29	Lindner 108	22IS330	SK85977	B	KJ56656
104A2014	H1700	29	Lindner 108	22IS330	SK85977	A	KP78924
104A2014	H1700	29	Lindner 108	22IS330	SK85977	A	KS39874
401W2014	M1000	33	Braunstien 211	22IS832	CC49234	A	KS39874
104A2014	H1700	29	Lindner 108	22IS330	SK85977	A	BE76598
401W2014	M1000	33	Braunstien 211	22IS832	CC49234	B	BG66765
104A2014	H1700	29	Lindner 108	22IS330	SK85977	C	GS76775

Since most queries that involve one or more Join operations also include some sort of Projection operation, in effect virtually all joins take the form of this “variation” of a Natural Join.

Sometimes, a join may be specified between a table and itself. This type of join is often referred to as a Self Join.

**Example 2.3.3.** List the student IDs of those students recorded as having taken more than one course.

```
SELECT X.SID
FROM TAKES X JOIN TAKES Y
ON X.SID = Y.SID
AND X.SECTION# <> Y.SECTION#;
```

In this SELECT statement, the TAKES table is referenced twice. To prevent ambiguity, each use of TAKES in the FROM clause has been assigned a temporary name (called a table alias). X and Y serve as the table aliases in this SELECT statement. Whenever a table alias is associated with a table name in the FROM clause, it must also be used any time the table is referenced in the SELECT statement (i.e., in this example, when referencing the table in the ON condition and also in the column list that follows use of the word SELECT). Whenever a table alias has been introduced in the FROM clause, we cannot use the full table name anywhere else in the SELECT statement.

Since, as we have seen, duplicate rows are not automatically removed in SQL, the execution of the SELECT statement given previously generates 12 rows (six with an **X.SID** value of KS39874, two with an **X.SID** value of KP78924, two with an **X.SID** value of BE76598, and two with an **X.SID** value of BG66765):

```
SID
-----
BG66765
KP78924
KS39874
KS39874
BE76598
KP78924
KS39874
KS39874
BE76598
BG66765
KS39874
KS39874
```

This is due to the fact that the join condition is satisfied two times for each of the three rows in TAKES that involve **X.SID KS39874**, one time for each of the two rows in TAKES that involve **X.SID KP78924**, one time for each of the two rows in TAKES that involve **X.SID BE76598**, and one time for each of the two rows in TAKES that involve **X.SID BG66765**.

Use of the qualifier DISTINCT in the SELECT statement eliminates the duplicate rows and produces a result that corresponds to the relational algebra result, as shown here:

**SQL SELECT Statement Revised:**

```
SELECT DISTINCT X.SID
FROM TAKES X JOIN TAKES Y
ON X.SID = Y.SID
AND X.SECTION# <> Y.SECTION#;
```

**Result:**

```
SID
-----
KP78924
KS39874
BE76598
BG66765
```

The Natural Join or Equijoin operation can also be specified among multiple tables, leading to what is sometimes referred to as an “n-way join.”

606

**Example 2.3.4.** Instead of listing the student IDs of those students having taken more than one course, list the names of those students having taken more than one course.

**SQL SELECT Statement:**

```
SELECT DISTINCT STUDENT.NAME
FROM (TAKES X JOIN TAKES Y
ON X.SID = Y.SID
AND X.SECTION# <> Y.SECTION#)
JOIN STUDENT ON X.SID = STUDENT.SID;
```

This SELECT statement uses the result of the Self Join from Example 2.3.2.2 and joins it with the STUDENT table. Without the use of the qualifier DISTINCT to eliminate duplicate rows, the four names shown next would be displayed a total of 12 times (i.e., Gladis Bale twice, Poppy Kramer twice, Elijah Baley twice, and Sweety Kramer six times).

**Result:**

```
NAME
-----
Poppy Kramer
Sweety Kramer
Gladis Bale
Elijah Baley
```

**Example 2.3.5.** For each student taking a section where the maximum number of students is greater than 25, list the student’s name, classroom where the course is offered, course number, and section number.

**SQL SELECT Statement:**

```
SELECT STUDENT.NAME, SECTION.ROOM, SECTION.COURSE#, SECTION.SECTION#
FROM (STUDENT JOIN TAKES
ON STUDENT.SID = TAKES.SID)
JOIN SECTION ON TAKES.SECTION# = SECTION.SECTION#
AND SECTION.MAXST > 25;
```

**Result:**

NAME	ROOM	COURSE#	SECTION#
Elijah Baley	Lindner 110	22IS330	201S2013
Elijah Baley	Lindner 108	22IS330	104A2014
Gladis Bale	Braunstien 211	22IS832	401W2014
Shweta Gupta	Lindner 108	22IS330	104A2014
Joumana Kidd	Lindner 108	22IS330	104A2014
Poppy Kramer	Lindner 108	22IS330	104A2014
Sweety Kramer	Braunstien 211	22IS832	401W2014
Sweety Kramer	Lindner 108	22IS330	104A2014

In effect, prior to the execution of the Projection operation, the SQL SELECT statement first links (i.e., concatenates) each row of TAKES to the corresponding row in STUDENT and then links each row in the combined result to the corresponding row in SECTION. Should both the course number and course name need to be displayed, the COURSE table must be included in the join. As shown next, the SELECT statement required to generate this result takes the result of joining the STUDENT, TAKES, and SECTION tables and joins it with the COURSE table.

**SQL SELECT Statement:**

```
SELECT STUDENT.NAME, SECTION.ROOM, SECTION.COURSE#, COURSE.NAME,
SECTION.SECTION#
FROM ((STUDENT JOIN TAKES
ON STUDENT.SID = TAKES.SID)
JOIN SECTION ON TAKES.SECTION# = SECTION.SECTION#
AND SECTION.MAXST > 25)
JOIN COURSE ON SECTION.COURSE# = COURSE.COURSE#;
```

**Result:**

NAME	ROOM	COURSE#	NAME	SECTION#
Elijah Baley	Lindner 110	22IS330	Database Concepts	201S2013
Elijah Baley	Lindner 108	22IS330	Database Concepts	104A2014
Gladis Bale	Braunstien 211	22IS832	Database Principles	401W2014
Shweta Gupta	Lindner 108	22IS330	Database Concepts	104A2014
Joumana Kidd	Lindner 108	22IS330	Database Concepts	104A2014
Poppy Kramer	Lindner 108	22IS330	Database Concepts	104A2014
Sweety Kramer	Braunstien 211	22IS832	Database Principles	401W2014
Sweety Kramer	Lindner 108	22IS330	Database Concepts	104A2014

### 12.2.3.3 The Theta Join Operation

While occurring infrequently in practical applications, Theta Joins (or Non-Equiijoins) that do not involve equality conditions are possible as long as the join condition involves attributes that share the same domain.

**Example 2.3.6.** Instead of doing an Equijoin of SECTION and TAKES when an equality condition involving each of their common attributes exists, do a Join of SECTION and TAKES when an inequality condition exists for each of their common attributes. It is left as an exercise for the reader to demonstrate that this Theta Join yields a result that contains 110 rows.

#### SQL SELECT Statement:

```
SELECT *
FROM SECTION JOIN TAKES
ON SECTION.SECTION# <> TAKES.SECTION#;
```

## 12.2.4 Outer Join Operations

608

In relational algebra, Outer Joins can be used when there is a need to keep all the tuples in R (or all those in S or all those in both relations) in the result of the Join, whether or not they have matching tuples in the other relation.

### 12.2.4.1 Left Outer Join

The Left Outer Join operation is used when we want to retain all rows in the leftmost table (i.e., the first table to be listed), regardless of whether corresponding rows exist in the other table. SQL uses the LEFT OUTER JOIN<sup>19</sup> keywords to create a Left Outer Join.

**Example 2.4.1.** For each section taken by a student, display the section number, grade, student ID, and student name. Include the names of all students (i.e., even those who have never taken a course).

```
SELECT TAKES.* , STUDENT.NAME
FROM STUDENT LEFT OUTER JOIN TAKES
ON STUDENT.SID = TAKES.SID;
```

---

<sup>19</sup>Use of the word “OUTER” is not required to create a LEFT, RIGHT, or FULL Outer Join.

**Result:**

SECTION#	GRADE	SID	NAME
			Jenny Aniston
104A2007	A	BE76598	Elijah Baley
201S2006	C	BE76598	Elijah Baley
401W2007	B	BG66765	Gladis Bale
101A2007	B	BG66765	Gladis Bale
			Tim Duncan
			Rick Fox
			Vanessa Fox
104A2007	C	GS76775	Shweta Gupta
			Jenna Hopp
			Troy Hudson
			Diana Jackson
104A2007	B	KJ56656	Joumana Kidd
101A2007	A	KP78924	Poppy Kramer
104A2007	A	KP78924	Poppy Kramer
104A2007	A	KS39874	Sweety Kramer
101A2007	A	KS39874	Sweety Kramer
401W2007	A	KS39874	Sweety Kramer
			Daniel Olive
			David Sane
			Wanda Seldon

609

In SQL-2003, the words “LEFT OUTER JOIN” are used to designate a Left Outer Join operation. The use of the LEFT JOIN keywords means that if the table listed on the left side of the join condition given in the ON clause has an unmatched row, it should be matched with a null row and displayed in the results.

One way to verify that the 10 rows with null values for the attributes in TAKES represent the 10 students who have not taken a course is to take the difference between the STUDENT and the TAKES relations over those attributes that have the same domain:

```
SELECT STUDENT.SID
FROM STUDENT
MINUS
SELECT TAKES.SID
FROM TAKES;
```

**Result:**

```
SID
-----
AJ76998
DT87656
FR45545
FV67733
HJ45633
HT67657
JD35477
OD76578
SD23556
SW56547
```

The student IDs shown here can be replaced by a list of student names through the use of the following nested subquery (subqueries are discussed in Section 12.3):

610

```
SELECT STUDENT.NAME FROM STUDENT
WHERE STUDENT.SID IN
  (SELECT STUDENT.SID
   FROM STUDENT
   MINUS
   SELECT TAKES.SID
   FROM TAKES) ;
```

**Result:**

```
NAME
-----
Jenny Aniston
Tim Duncan
Rick Fox
Vanessa Fox
Jenna Hopp
Troy Hudson
Diana Jackson
Daniel Olive
David Sane
Wanda Seldon
```

**12.2.4.2 Right Outer Join**

Semantically, a Right Outer Join is the same as a Left Outer Join. The difference is that the required table is the rightmost table listed.

**Example 2.4.2.** For each textbook, display all available information about the book and its usage in courses. Include textbooks that have never been used in a course.

**SQL SELECT Statement:**

```
SELECT *
FROM USES RIGHT OUTER JOIN TEXTBOOK
ON USES.ISBN = TEXTBOOK.ISBN;
```

**Result:**

COURSE#	ISBN	EMPID	ISBN	TITLE	YEAR	PUBLISHER
18ECON123	0296748-99	CM65436	0296748-99	Economics For Managers	2009	
20ECES212	0296437-1118	CC49234	0296437-1118	Programming in C++	2010	Thomson
22IS270	000-66574998	SS43278	000-66574998	Database Management	2007	Thomson
22IS270	77898-8769	CC49234	77898-8769	Principles of IS	2010	Prentice-Hall
22IS270	77898-8769	SK85977	77898-8769	Principles of IS	2010	Prentice-Hall
22IS270	77898-8769	SS43278	77898-8769	Principles of IS	2010	Prentice-Hall
22IS330	003-6679233	BC65437	003-6679233	Linear Programming	2005	Prentice-Hall
22IS330	118-99898-67	SS43278	118-99898-67	Systems Analysis	2008	Thomson
22IS832	000-66574998	SS43278	000-66574998	Database Management	2007	Thomson
22IS832	118-99898-67	SK85977	118-99898-67	Systems Analysis	2008	Thomson
22QA375	0296437-1118	SJ65436	0296437-1118	Programming in C++	2010	Thomson
22QA888	001-55-435	HT54347	001-55-435	Simulation Modeling	2009	Springer
			111-11111111	Data Modeling	2013	
			012-54765-32	Fundamentals of SQL	2012	

In SQL-2003, the words “RIGHT OUTER JOIN” are used to designate a Right Outer Join operation. The use of the RIGHT JOIN keywords means that if the table listed on the right side of the join condition given in the ON clause has an unmatched row, it should be matched with a null row and displayed in the results.

**12.2.4.3 Full Outer Join**

A Full Outer Join operation is used when we want all rows in both the left and right tables included, even though no corresponding rows exist in the other table.

**Example 2.4.3.** The SQL Syntax for the Full Outer Join Example that appears in Section 11.2.3 of Chapter 11 is given next. Only syntaxes beginning with SQL-92 support the Full Outer Join operation directly. This is accomplished through use of the FULL OUTER JOIN keywords.

**SQL SELECT Statement:**

```
SELECT *
FROM C FULL OUTER JOIN D
ON C.DCODE = D.DCODE;
```

**Result:**

NAME	COURSE#	CREDIT	DCODE	NAME	DCODE	COLLEGE
Intro to Economics	15ECON112	U		1 Economics	1	Arts and Sciences
Optimization	22QA888	G		3 QA/QM	3	Business
Supply Chain Analysis	22QA411	U		3 QA/QM	3	Business
Operations Research	22QA375	U		3 QA/QM	3	Business
Financial Accounting	18ACCT801	G		4 Economics	4	Education
Intro to Economics	18ECON123	U		4 Economics	4	Education
Programming in C++	20ECES212	G		6 Mathematics	6	Engineering
Systems Analysis	22IS430	G		7 IS	7	Business
Database Principles	22IS832	G		7 IS	7	Business
Database Concepts	22IS330	U		7 IS	7	Business
Principles of IS	22IS270	G		7 IS	7	Business
Architectural History	05ARCH101	U		Philosophy	9	Arts and Sciences

612

Note that the union of a Left Outer Join and a Right Outer Join is equal to the results of a Full Outer Join.

### 12.2.5 SQL and the Semi-Join and Semi-Minus Operations

Recall that the Semi-Join operation defines a relation that contains the tuples of R that participate in the Join of R with S. In other words, a Semi-Join is defined to be equal to the Join of R and S, projected back on the attributes of R. As such, a Semi-Join can be handled in SQL by using the Projection and Join operations.

**Example 2.5.1 (Corresponds to Semi-Join Example 1 in Section 11.2.5 of Chapter 11).** List the complete details of all courses for which sections are being offered in the Fall 2014 quarter.

**SQL SELECT Statement:**

```
SELECT COURSE.*
FROM COURSE JOIN SECTION
ON COURSE.COURSE# = SECTION.COURSE#
AND SECTION# LIKE '%2014%' AND SECTION# LIKE '%A%';
```

**Result:**

NAME	COURSE#	CREDIT	COLLEGE	HRS	DCODE
Operations Research	22QA375	U	Business	2	3
Programming in C++	20ECES212	G	Engineering	3	6
Database Concepts	22IS330	U	Business	4	7

A Semi-Minus operation occurs in cases where there are tuples in relation R that have no counterpart in relation S. A Semi-Minus operation can be handled in SQL through use of a combination of Join, Projection, and Minus operations.

**Example 2.5.2 (Corresponds to Semi-Minus Example 1 in Section 11.2.5 of Chapter 11).**  
List complete details of all courses for which sections are not offered in the Fall 2014 quarter.

**SQL SELECT Statement:**

```
SELECT * FROM COURSE
MINUS
SELECT COURSE.*
FROM COURSE JOIN SECTION
ON COURSE.COURSE# = SECTION.COURSE#
AND SECTION# LIKE '%2014%' AND SECTION# LIKE '%A%';
```

**Result:**

NAME	COURSE#	CREDIT	COLLEGE	HRS	DCODE
Architectural History	05ARCH101	U		3	
Database Principles	22IS832	G	Business	3	7
Financial Accounting	18ACCT801	G	Education	3	4
Intro to Economics	15ECON112	U	Arts and Sciences	3	1
Intro to Economics	18ECON123	U	Education	4	4
Optimization	22QA888	G	Business	3	3
Principles of IS	22IS270	G	Business	3	7
Supply Chain Analysis	22QA411	U	Business	3	3
Systems Analysis	22IS430	G	Business	3	7

613

## 12.3 SUBQUERIES

A complete SELECT statement embedded within another SELECT statement is called a subquery. In data retrieval, subqueries may be used (a) in the SELECT list of a SELECT statement, (b) in the FROM clause of a SELECT statement, (c) in the WHERE clause of a SELECT statement, and (d) in the ORDER BY clause of a SELECT statement. As illustrated in Section 10.2.1, subqueries can also be used in an INSERT ... SELECT ... FROM statement as well as in the SET clause of an UPDATE statement. The output of a subquery can consist of a single value (a single-row subquery) or several rows of values (a multiple-row subquery). There are two types of subqueries: (a) uncorrelated subqueries, where the subquery is executed first and passes one or more values to the outer query, and (b) correlated subqueries, where the subquery is executed once for every row retrieved by the outer query.

### 12.3.1 Multiple-Row Uncorrelated Subqueries

Multiple-row subqueries are nested queries that can return more than one row of results to the parent query. Three multiple-row operators (IN, ALL, and ANY) are used with multiple-row queries.

### 12.3.1.1 The IN and NOT IN Operators

When used in conjunction with a subquery, the IN operator evaluates if rows processed by the outer query are equal to any of the values returned by the subquery (i.e., it creates an OR condition). Example 3.1.1 illustrates how the IN operator can be used to display the course number, course name, and college of those courses for which sections have been offered. In this query, the subquery is executed first and returns the set of values (22QA375, 22IS270, 22IS330, 22IS832, and 20ECES212). The main query then displays the course number, name, and college for these courses. Note that a subquery of the form SELECT DISTINCT SECTION.COURSE# FROM SECTION would have produced exactly the same result.

#### Example 3.1.1

```
SELECT COURSE.COURSE#, COURSE.NAME, COURSE.COLLEGE
FROM COURSE
WHERE COURSE.COURSE# IN
    (SELECT SECTION.COURSE#
     FROM SECTION) ;
```

#### Result:

614

COURSE#	NAME	COLLEGE
22QA375	Operations Research	Business
22IS270	Principles of IS	Business
22IS330	Database Concepts	Business
22IS832	Database Principles	Business
20ECES212	Programming in C++	Engineering

The NOT IN operator is the opposite of the IN operator and indicates that the rows processed by the outer query are not equal to any of the values returned by the subquery. Example 3.1.2 displays the course number, course name, and college of those courses for which sections have not been offered.

#### Example 3.1.2

```
SELECT COURSE.COURSE#, COURSE.NAME, COURSE.COLLEGE
FROM COURSE
WHERE COURSE.COURSE# NOT IN
    (SELECT SECTION.COURSE#
     FROM SECTION) ;
```

#### Result:

COURSE#	NAME	COLLEGE
05ARCH101	Architectural History	
18ACCT801	Financial Accounting	Education
15ECON112	Intro to Economics	Arts and Sciences
18ECON123	Intro to Economics	Education
22QA888	Optimization	Business
22QA411	Supply Chain Analysis	Business
22IS430	Systems Analysis	Business

The comparison operators `=`, `<>`, `>`, `>=`, `<`, and `<=` are *single-row operators*. Observe the error message generated when the multiple-row operator `IN` is replaced by the single-row operator `=` (equals sign) in Example 3.1.3. This error is caused by the fact that there are several section numbers in the `TAKES` table associated with a grade of “A.” Observe what happens, however, when the single-row operator `=` is replaced by `IN`.

The purpose of Example 3.1.3 is to display the section number and course number for which at least one grade of “A” has been assigned.

### Example 3.1.3

```
SELECT DISTINCT SECTION.SECTION#, SECTION.COURSE#
FROM SECTION
WHERE SECTION.SECTION# =
  (SELECT TAKES.SECTION#
   FROM TAKES
   WHERE TAKES.GRADE = 'A') ;

  (SELECT TAKES.SECTION#
   *
ERROR at line 4:
ORA-01427: single-row subquery returns more than one row
```

615

### Example 3.1.3 (Corrected)

```
SELECT DISTINCT SECTION.SECTION#, SECTION.COURSE#
FROM SECTION
WHERE SECTION.SECTION# IN
  (SELECT TAKES.SECTION#
   FROM TAKES
   WHERE TAKES.GRADE = 'A') ;
```

#### Result:

SECTION#	COURSE#
101A2014	22QA375
401W2014	22IS832
104A2014	22IS330

The remaining examples in this section illustrate the use of the `ANY` and `ALL` operators in the context of the `PROFESSOR` table.

#### 12.3.1.2 The ALL and ANY Operators

The `ALL` and `ANY` operators can be combined with the comparison operators `=`, `<>`, `>`, `>=`, `<`, and `<=` to treat the results of a subquery as a set of values rather than as individual values. `ANY` specifies that the condition be true for *at least one value* from the set of values. `ALL`, on the other hand, specifies that the condition be true for *all values* in the set of values. Table 12.1 summarizes the use of the `ALL` and `ANY` operators in conjunction with other comparison operators.

Operator	Description
> ALL	Greater than the highest value returned by the subquery
>= ALL	Greater than or equal to the highest value returned by the subquery
< ALL	Less than the lowest value returned by the subquery
<= ALL	Less than or equal to the lowest value returned by the subquery
> ANY	Greater than the lowest value returned by the subquery
>= ANY	Greater than or equal to the lowest value returned by the subquery
< ANY	Less than the highest value returned by the subquery
<= ANY	Less than or equal to the highest value returned by the subquery
= ANY	Equal to any value returned by the subquery (same as the IN operator)

**TABLE 12.1** Use of ANY and ALL operators in subqueries

616

**Example 3.1.4.** Display the names and salaries of those professors who earn more than all professors in department number 3.

**SQL SELECT Statement:**

```
SELECT PROFESSOR.NAME, PROFESSOR.SALARY
FROM PROFESSOR
WHERE PROFESSOR.SALARY > ALL
  (SELECT PROFESSOR.SALARY
   FROM PROFESSOR
   WHERE PROFESSOR.DCODE = 3) ;
```

**Result:**

NAME	SALARY
Mike Faraday	92000
Marie Curie	99000
John Nicholson	99000

The following query includes Chelsea Bush and Tony Hopkins in the result since their salary is equal to the highest value returned by the subquery:

```
SELECT PROFESSOR.NAME, PROFESSOR.SALARY
FROM PROFESSOR
WHERE PROFESSOR.SALARY >= ALL
  (SELECT PROFESSOR.SALARY
   FROM PROFESSOR
   WHERE PROFESSOR.DCODE = 3) ;
```

**Result:**

NAME	SALARY
-----	-----
Mike Faraday	92000
Chelsea Bush	77000
Tony Hopkins	77000
Marie Curie	99000
John Nicholson	99000

**Example 3.1.5.** Display the names and salaries of those professors who earn less than all professors in department number 7.

```
SELECT PROFESSOR.NAME, PROFESSOR.SALARY
FROM PROFESSOR
WHERE PROFESSOR.SALARY < ALL
  (SELECT PROFESSOR.SALARY
   FROM PROFESSOR
   WHERE PROFESSOR.DCODE = 7) ;
```

617

**Result:**

NAME	SALARY
-----	-----
Ram Raj	44000
Prester John	44000
Laura Jackson	43000

**Example 3.1.6.** Revise the query in Example 3.1.5 and display the names and salaries of those professors with a salary that is less than or equal to that of the lowest paid professor in department number 7.

```
SELECT PROFESSOR.NAME, PROFESSOR.SALARY
FROM PROFESSOR
WHERE PROFESSOR.SALARY <= ALL
  (SELECT PROFESSOR.SALARY
   FROM PROFESSOR
   WHERE PROFESSOR.DCODE = 7) ;
```

**Result:**

NAME	SALARY
-----	-----
John Smith	45000
Ram Raj	44000
Prester John	44000
Laura Jackson	43000
Cathy Cobal	45000
Jeanine Troy	45000

**Example 3.1.7.** Revise the query in Example 3.1.6 to exclude display of any employees in department number 7.

```
SELECT PROFESSOR.NAME, PROFESSOR.SALARY
FROM PROFESSOR
WHERE PROFESSOR.DCODE <> 7
AND PROFESSOR.SALARY <= ALL
    (SELECT PROFESSOR.SALARY
     FROM PROFESSOR
     WHERE PROFESSOR.DCODE = 7) ;
```

**Result:**

NAME	SALARY
John Smith	45000
Ram Raj	44000
Prester John	44000
Laura Jackson	43000
Jeanine Troy	45000

618

Since `< ANY` returns all rows with a salary less than the highest salary associated with department 3, the query in Example 3.1.8 displays the rows for all professors with a salary less than \$77,000.

**Example 3.1.8**

```
SELECT PROFESSOR.NAME, PROFESSOR.SALARY
FROM PROFESSOR
WHERE PROFESSOR.SALARY < ANY
    (SELECT PROFESSOR.SALARY
     FROM PROFESSOR
     WHERE PROFESSOR.DCODE = 3) ;
```

**Result:**

NAME	SALARY
Laura Jackson	43000
Prester John	44000
Ram Raj	44000
Jeanine Troy	45000
John Smith	45000
Cathy Cobal	45000
Sunil Shetty	64000
Katie Shef	65000
Kobe Bryant	66000
Jessica Simpson	67000
Jack Nicklaus	67000
Mike Crick	69000
Alan Brodie	76000

On the other hand, since `<= ANY` returns all rows with a salary less than or equal to the highest salary associated with department 3, the query in Example 3.1.9 also displays the rows for both Chelsea Bush and Tony Hopkins.

### Example 3.1.9

```
SELECT PROFESSOR.NAME, PROFESSOR.SALARY
FROM PROFESSOR
WHERE PROFESSOR.SALARY <=ANY
  (SELECT PROFESSOR.SALARY
   FROM PROFESSOR
   WHERE PROFESSOR.DCODE = 3) ;
```

#### Result:

NAME	SALARY
Laura Jackson	43000
Prester John	44000
Ram Raj	44000
Jeanine Troy	45000
John Smith	45000
Cathy Cobal	45000
Sunil Shetty	64000
Katie Shef	65000
Kobe Bryant	66000
Jessica Simpson	67000
Jack Nicklaus	67000
Mike Crick	69000
Alan Brodie	76000
Tony Hopkins	77000
Chelsea Bush	77000

Since `> ANY` returns all rows with a salary greater than the lowest salary associated with professors who work in department 3, the query in Example 3.1.10 displays all rows except for the professor with the lowest salary (i.e., Laura Jackson) and the two professors who have a null salary.

### Example 3.1.10

```
SELECT PROFESSOR.NAME, PROFESSOR.SALARY
FROM PROFESSOR
WHERE PROFESSOR.SALARY > ANY
  (SELECT PROFESSOR.SALARY
   FROM PROFESSOR
   WHERE PROFESSOR.DCODE = 3) ;
```

**Result:**

NAME	SALARY
John Nicholson	99000
Marie Curie	99000
Mike Faraday	92000
Chelsea Bush	77000
Tony Hopkins	77000
Alan Brodie	76000
Mike Crick	69000
Jessica Simpson	67000
Jack Nicklaus	67000
Kobe Bryant	66000
Katie Shef	65000
Sunil Shetty	64000
Cathy Cobal	45000
Jeanine Troy	45000
John Smith	45000
Prester John	44000
Ram Raj	44000

As expected, since  $\geq$  ANY returns all rows with a salary greater than or equal to the lowest salary associated with department 3, all rows are returned in Example 3.1.11 except for those associated with the professors who have null salaries.

**Example 3.1.11**

```
SELECT PROFESSOR.NAME, PROFESSOR.SALARY
FROM PROFESSOR
WHERE PROFESSOR.SALARY  $\geq$  ANY
  (SELECT PROFESSOR.SALARY
   FROM PROFESSOR
   WHERE PROFESSOR.DCODE = 3) ;
```

**Result:**

NAME	SALARY
John Nicholson	99000
Marie Curie	99000
Mike Faraday	92000
Chelsea Bush	77000
Tony Hopkins	77000
Alan Brodie	76000
Mike Crick	69000
Jessica Simpson	67000
Jack Nicklaus	67000
Kobe Bryant	66000

Katie Shef	65000
Sunil Shetty	64000
Cathy Cobal	45000
Jeanine Troy	45000
John Smith	45000
Prester John	44000
Ram Raj	44000
Laura Jackson	43000

As illustrated in Example 3.1.12, = ANY produces the same result as the IN operator. Note how the query in Example 3.1.12a restricts the rows displayed to those not associated with department 3.

### Example 3.1.12

```
SELECT PROFESSOR.NAME, PROFESSOR.SALARY
FROM PROFESSOR
WHERE PROFESSOR.SALARY = ANY
  (SELECT PROFESSOR.SALARY
   FROM PROFESSOR
   WHERE PROFESSOR.DCODE = 3) ;
```

621

#### Result:

NAME	SALARY
Tony Hopkins	77000
Chelsea Bush	77000
Alan Brodie	76000
Jack Nicklaus	67000
Jessica Simpson	67000
Laura Jackson	43000

### Example 3.1.12a

```
SELECT PROFESSOR.NAME, PROFESSOR.SALARY
FROM PROFESSOR
WHERE PROFESSOR.SALARY = ANY
  (SELECT PROFESSOR.SALARY
   FROM PROFESSOR
   WHERE PROFESSOR.DCODE = 3)
AND PROFESSOR.DCODE <> 3 ;
```

#### Result:

NAME	SALARY
Jack Nicklaus	67000

Although ANY and ALL are most commonly used with subqueries that return a set of numeric values, it is also possible to use them in conjunction with subqueries that return a set of character values.

### 12.3.1.3 The MAX and MIN Functions

The MAX and MIN functions can be used in place of > ANY, < ANY, > ALL, and < ALL when the WHERE clause of the outer query involves a numeric value. Examples 3.1.13 through 3.1.15 are equivalent to Examples 3.1.4 through 3.1.6 but use the MAX and MIN function instead of ANY or ALL.

#### Example 3.1.13 (Compare with Example 3.1.4)

```
SELECT PROFESSOR.NAME, PROFESSOR.SALARY
FROM PROFESSOR
WHERE PROFESSOR.SALARY >
    (SELECT MAX (PROFESSOR.SALARY)
     FROM PROFESSOR
     WHERE PROFESSOR.DCODE = 3) ;
```

#### Result:

NAME	SALARY
Mike Faraday	92000
Marie Curie	99000
John Nicholson	99000

622

#### Example 3.1.14 (Compare with Example 3.1.5)

```
SELECT PROFESSOR.NAME, PROFESSOR.SALARY
FROM PROFESSOR
WHERE PROFESSOR.SALARY <
    (SELECT MIN (PROFESSOR.SALARY)
     FROM PROFESSOR
     WHERE PROFESSOR.DCODE = 7) ;
```

#### Result:

NAME	SALARY
Ram Raj	44000
Prester John	44000
Laura Jackson	43000

#### Example 3.1.15 (Compare with Example 3.1.6)

```
SELECT PROFESSOR.NAME, PROFESSOR.SALARY
FROM PROFESSOR
WHERE PROFESSOR.SALARY <=
    (SELECT MIN (PROFESSOR.SALARY)
     FROM PROFESSOR
     WHERE PROFESSOR.DCODE = 7) ;
```

**Result:**

NAME	SALARY
John Smith	45000
Ram Raj	44000
Prester John	44000
Laura Jackson	43000
Cathy Cobal	45000
Jeanine Troy	45000

**12.3.1.4 Subqueries in the FROM Clause**

It is also possible to have nested subqueries in the FROM clause and in effect treat the subquery itself as if it were the name of a table.<sup>20</sup> This approach is often used when the subquery is a multiple-column subquery. As an example, suppose we are interested in listing all professors with a salary that is equal to or exceeds the average salary of all professors in their department.

**Example 3.1.17.** Display all professors with a salary that is equal to or exceeds the average salary of all professors in their department.

**SQL SELECT Statement:**

```
SELECT A.NAME, A.DCODE, A.SALARY, B."Department Average"
FROM PROFESSOR A
    JOIN (SELECT PROFESSOR.DCODE, AVG(PROFESSOR.SALARY) AS "Department Average"
          FROM PROFESSOR
          GROUP BY PROFESSOR.DCODE) B
ON A.DCODE = B.DCODE
AND A.SALARY >= B."Department Average";
```

Observe how the shaded subquery, in essence, creates a temporary table that records the average salary of the professors in each department. The syntax calls for the table alias B to be located outside the parenthetical expression of the subquery since the execution of the subquery yields a temporary (i.e., virtual) table. The Join operation uses the PROFESSOR table and concatenates a row from PROFESSOR (table alias A) with a row from the temporary table created by the subquery (table alias B) when (a) the department number of the row from A matches the department number of a row from B, and (b) the salary of the professor in the row from A exceeds the average salary of the professors in his or her department.

<sup>20</sup>Such a “temporary table” is more formally called an inline view.

**Result:**

NAME	DCODE	SALARY	Department Average
Mike Faraday	1	92000	67666.6667
Ram Raj	6	44000	44000
Prester John	6	44000	44000
Chelsea Bush	3	77000	68000
Tony Hopkins	3	77000	68000
Alan Brodie	3	76000	68000
Marie Curie	4	99000	88333.3333
John Nicholson	4	99000	88333.3333
Sunil Shetty	7	64000	58000
Katie Shef	7	65000	58000
Mike Crick	9	69000	57000

**12.3.1.5 Subqueries in the Column List**

624

The column list of a query can also include a subquery expression. A subquery in the column list must return a single value.

**Example 3.1.18.** Display all professors with a salary that exceeds the average salary of all professors in their department along with the amount by which the average is exceeded.

**SQL SELECT Statement:**

```

1 SELECT A.NAME, A.SALARY, A.DCODE,
2   ROUND ( (SELECT AVG(B.SALARY)
3   FROM PROFESSOR B
4   WHERE A.DCODE = B.DCODE) , 0 ) AS "Avg Dept Salary",
5   ROUND ( (A.SALARY - (SELECT AVG(B.SALARY) FROM PROFESSOR B
6   WHERE A.DCODE = B.DCODE) ) , 0 ) AS "Deviation"
7   FROM PROFESSOR A
8   WHERE A.SALARY IS NOT NULL
9   AND ROUND ((A.SALARY - (SELECT AVG(B.SALARY) FROM PROFESSOR B
10 WHERE A.DCODE = B.DCODE) ) , 0 ) > 0
11 ORDER BY "Deviation" DESC;
```

In order to explain this query, each line has been numbered. Two SELECT statements, each of which is the same, appear in the column list. The first (shown in *italics* on lines 2–4) determines the average salary for the professors in the department for a given professor, while the second (shown highlighted on lines 5 and 6) recalculates this average salary and uses it to determine the amount of the deviation between the salary of the professor and the average salary for the professors in the department. The WHERE clause in the main query (a) excludes from consideration those professors with a null salary (see line 8), and (b) includes only those professors whose salary exceeds that of their average salary in their department (note that the average salary of all professors in the professors' department is calculated a third time in lines 9 and 10). The ORDER BY clause on line 11 allows the result to be displayed in descending order by the amount of the deviation

between the salary of the professor and the average salary of the professors in their department.

**Result:**

NAME	SALARY	DCODE	Avg Dept Salary	Deviation
Mike Faraday	92000	1	67667	24333
Mike Crick	69000	9	57000	12000
Marie Curie	99000	4	88333	10667
John Nicholson	99000	4	88333	10667
Chelsea Bush	77000	3	68000	9000
Tony Hopkins	77000	3	68000	9000
Alan Brodie	76000	3	68000	8000
Katie Shef	65000	7	58000	7000
Sunil Shetty	64000	7	58000	6000

### 12.3.1.6 Subqueries in the HAVING Clause

In addition to appearing in the WHERE clause, the FROM clause, and in the column list, a subquery can also be used in the HAVING clause. As an example, consider the following example.

**Example 3.1.19.** Display the name and average professor salary (for all departments) whose average salary exceeds the average salary paid to all professors at Madeira College.

**SQL SELECT Statement:**

```
SELECT DEPARTMENT.NAME, AVG (PROFESSOR.SALARY)
FROM DEPARTMENT JOIN PROFESSOR
ON DEPARTMENT.DCODE = PROFESSOR.DCODE
GROUP BY DEPARTMENT.NAME
HAVING AVG (PROFESSOR.SALARY) >
(SELECT AVG (PROFESSOR.SALARY) FROM PROFESSOR);
```

**Result:**

NAME	AVG (PROFESSOR.SALARY)
Economics	78000
QA/QM	68000

The SELECT statement in the HAVING clause acts as a filter that insures the selection of only those departments (i.e., groups) with an average salary greater than the average salary of the entire college.

### 12.3.2 Multiple-Row Correlated Subqueries

A correlated subquery can be used if it is necessary to check if a nested subquery returns no rows. Correlated subqueries make use of the EXISTS operator, which returns the value of true if a set is non-empty.

**Example 3.2.1.** Display the names of professors who have offered at least one section.

```
SELECT PROFESSOR.NAME
FROM PROFESSOR
WHERE EXISTS
  (SELECT *
   FROM SECTION
   WHERE PROFESSOR.EMPID = SECTION.PROFID);
```

**Result:**

NAME
Cathy Cobal
Tony Hopkins
Ram Raj
Katie Shef

A correlated nested subquery is processed differently from an uncorrelated nested subquery. Instead of the execution of the subquery serving as input to its parent query (i.e., the outer query), the subquery in a correlated subquery is executed once for each row in the outer query. In addition, execution of the subquery stops and the EXISTS condition of the main query is declared true for a given row should the condition in the subquery be true. For example, using the data in the PROFESSOR and SECTION tables, the execution of the subquery for Ram Raj stops when the fourth row of the PROFESSOR table is evaluated against the seventh row of the SECTION table because **PROFESSOR.EMPID = SECTION.PROFID** at this point. Thus, in essence, each value of **PROFESSOR.NAME** is treated as a constant during the evaluation. If the NOT EXISTS operator were used instead of the EXISTS operator, the names of professors who have not offered a section would be displayed.

The NOT EXISTS operator can be used as a way to express the DIVIDE operator in SQL. Recall the example in Section 11.2.4 of Chapter 11, which poses the following question: What are the course numbers and course names of those courses offered in all quarters during which course sections are offered? An SQL statement that produces an answer to this query is given next. The individual lines have been numbered to facilitate discussion of the execution of the query.

**SQL SELECT Statement:**

```
1  SELECT COURSE.COURSE#, COURSE.NAME
2  FROM COURSE
3  WHERE NOT EXISTS
4      (SELECT DISTINCT (SUBSTR(A.SECTION#, 4, 1)
5          FROM SECTION A
6          WHERE NOT EXISTS
7              (SELECT *
8                  FROM SECTION B
9                  WHERE COURSE.COURSE# = B.COURSE#
10                 AND SUBSTR(A.SECTION#, 4, 1) = SUBSTR(B.SECTION#, 4, 1))) );
```

## Result

COURSE#	NAME
22QA375	Operations Research

Both the subquery that begins on line 4 and the subquery that begins on line 7 refer to the SECTION table. To prevent ambiguity, these two uses of the SECTION table have been assigned the aliases of A and B, respectively. Since the two uses of the NOT EXISTS operator may make this query difficult to understand, let's begin by assuming the first row retrieved from the COURSE table as part of the execution of lines 1–3 defines COURSE.COURSE# as 15ECON112. Replacing COURSE.COURSE# in line 9 with '15ECON112' causes the execution of lines 4–10 to generate the result shown here:

```

4      (SELECT DISTINCT (SUBSTR(A.SECTION#, 4, 1)
5          FROM SECTION A
6          WHERE NOT EXISTS
7              (SELECT *
8                  FROM SECTION B
9                  WHERE COURSE.COURSE# = B.COURSE#
10                 AND SUBSTR(A.SECTION#, 4, 1) = SUBSTR(B.SECTION#, 4, 1))) ;

```

627

## Result

SUBSTR(A.SECTION#, 4, 1)
W
U
A
S

Since Course# 15ECON112 does not appear at all in the SECTION table, the NOT EXISTS condition is true for each of the four quarters. On the other hand, when Course# 22QA375 replaces Course# 15ECON112 line 9, the NOT EXISTS condition is false for all four quarters (note that a section of Course# 22QA375 is offered during each quarter in the SECTION table) and thus "no rows selected" is the result when lines 4–10 are executed.

```

4      (SELECT DISTINCT (SUBSTR(A.SECTION#, 4, 1)
5          FROM SECTION A
6          WHERE NOT EXISTS
7              (SELECT *
8                  FROM SECTION B
9                  WHERE COURSE.COURSE# = B.COURSE#
10                 AND SUBSTR(A.SECTION#, 4, 1) = SUBSTR(B.SECTION#, 4, 1))) ;

```

## Result:

no rows selected.

In other words, the NOT EXISTS condition in line 3 is true for Course# 22QA375. This SQL formulation corresponds to the following informal statement: "Display the

course numbers of those courses such that there does not exist a quarter during which the course is not offered.” It is left as an exercise for the reader to determine the result when lines 4–10 are executed for other courses (e.g., Course# 22IS330).

### 12.3.3 Aggregate Functions and Grouping

In SQL, an aggregate function takes as input a set of values, one from each row in a group of rows, and returns one value as the result. As illustrated in Section 12.1.4, the COUNT function, one of the most commonly used aggregate functions, counts the non-NULL values in a column. Other aggregate functions include retrieving the sum, average, maximum, and minimum of a series of numeric values. Another type of request involves the grouping of rows in a table or tables by the value of some of their attributes and then applying an aggregate function independently to each group.

**Example 3.3.1.** Using the data in the STUDENT and TAKES tables, count the number of sections taken by each student. Be sure to include those students who have never taken a class. The individual lines have been numbered to facilitate the discussion of the execution of the query.

#### SQL SELECT Statement:

```

1  SELECT TAKES.SID, STUDENT.NAME, COUNT(*) AS "Sections Taken"
2  FROM STUDENT JOIN TAKES
3  ON STUDENT.SID = TAKES.SID
4  GROUP BY TAKES.SID, STUDENT.NAME
5  UNION
6  SELECT STUDENT.SID, STUDENT.NAME, 0
7  FROM STUDENT
8  WHERE STUDENT.SID NOT IN
9  (SELECT TAKES.SID FROM TAKES)
10 ORDER BY "Sections Taken" DESC;
```

In addition to illustrating the use of a grouping operation and an aggregate function, this query also contains a join, a nested subquery, and a union. Execution of the query begins with lines 1–4, which count the number of sections taken by those students who have taken at least one class. Note that a value of COUNT(\*) is obtained for each combination of a **TAKES.SID** and **STUDENT.NAME**. On the other hand, lines 6–10, when executed, begin with the execution of line 9 and identify those students who have not taken a section. For each qualifying student, the numeric literal zero (0)<sup>21</sup> is displayed along with the value of **STUDENT.SID**, **STUDENT.NAME**. Since the content of columns 1 and 2 on lines 1 and 6 share the same domain and COUNT(\*) and 0 also share the same domain (i.e., both are numeric values), the UNION operation on line 5 can take place. When a UNION operation takes place in a query, the sorting operation (i.e., represented by the ORDER BY clause on line 10) applies to the collective results from all SELECT statements involved in the UNION.

---

<sup>21</sup>In addition to listing column names, expressions (e.g., see Section 12.1.2), functions (e.g., COUNT(\*)), and SELECT statements (e.g., see Section 12.3), the SELECT list of a SELECT statement may contain either a numeric constant (e.g., the numeric literal 0) or a string constant.

**Result:**

SID	NAME	Sections Taken
KS39874	Sweety Kramer	3
BE76598	Elijah Baley	2
BG66765	Gladis Bale	2
KP78924	Poppy Kramer	2
GS76775	Shweta Gupta	1
KJ56656	Joumana Kidd	1
AJ76998	Jenny Aniston	0
DT87656	Tim Duncan	0
FR45545	Rick Fox	0
FV67733	Vanessa Fox	0
HJ45633	Jenna Hopp	0
HT67657	Troy Hudson	0
JD35477	Diana Jackson	0
OD76578	Daniel Olive	0
SD23556	David Sane	0
SW56547	Wanda Seldon	0

629

The same result could be obtained using the Left Outer Join shown next, since the value of COUNT(**TAKES.SID**) is zero when a row for a student not enrolled in a section is concatenated with the row of null values in TAKES.

**SQL SELECT Statement:**

```
SELECT STUDENT.SID, STUDENT.NAME, COUNT(TAKES.SID) AS "Sections Taken"
FROM STUDENT LEFT OUTER JOIN TAKES
ON STUDENT.SID = TAKES.SID
GROUP BY STUDENT.SID, STUDENT.NAME
ORDER BY "Sections Taken" DESC
```

**Example 3.3.2.** Display the maximum, minimum, total, and average salary for the professors affiliated with each department. In addition, count the number of professors in each department as well as the number of professors in each department with a not null salary.

**SQL SELECT Statement:**

```
SELECT DEPARTMENT.NAME AS "Dept Name", DEPARTMENT.DCODE AS "Dept Code",
MAX(PROFESSOR.SALARY) AS "Max Salary", MIN(PROFESSOR.SALARY) AS "Min Salary",
SUM(PROFESSOR.SALARY) AS "Total Salary",
ROUND(AVG(PROFESSOR.SALARY),0) AS "Avg Salary", COUNT(*) AS "Size",
COUNT(PROFESSOR.SALARY) AS "# Sals"
FROM DEPARTMENT JOIN PROFESSOR
ON DEPARTMENT.DCODE = PROFESSOR.DCODE
GROUP BY DEPARTMENT.NAME, DEPARTMENT.DCODE
ORDER BY "# Sals" DESC;
```

**Result:**

Dept Name	Dept Code	Max Salary	Min Salary	Total Salary	Avg Salary	Size	# Sals
QA/QM	3	77000	43000	340000	68000	5	5
Economics	1	92000	45000	203000	67667	3	3
IS	7	65000	45000	174000	58000	3	3
Economics	4	99000	67000	265000	88333	3	3
Mathematics	6	44000	44000	88000	44000	3	2
Philosophy	9	69000	45000	114000	57000	3	2

Since department names are not unique but required as part of the output, grouping must be done on both **DEPARTMENT.NAME** and **DEPARTMENT.DCODE**. In addition, instead of grouping by **DEPARTMENT.DCODE**, grouping could have been by **DEPARTMENT.COLLEGE** had the name of the college housing the department been required as opposed to the department code.

## Chapter Summary

---

A query expressed in relational algebra involves a sequence of operations that, when executed in the order specified, produces the desired results. SQL, the most common way that relational algebra is implemented for data retrieval operations in a relational database, is the subject of Chapter 12. The SQL SELECT statement is used to express a query and is the most important statement in the language. Every SELECT statement, when executed, produces as its result a table that consists of one or more columns and zero or more rows. Six clauses make up a SELECT statement. Two of these clauses, the SELECT clause and the FROM clause, are required. The SELECT clause identifies the columns, calculated values, and literals to appear in the result table. All column names that appear in the SELECT clause must have their corresponding tables or views listed in the FROM clause.

The other four clauses—WHERE, GROUP BY, HAVING, and ORDER BY—are optional. The WHERE clause of the SELECT statement includes a search condition that consists of an expression involving constant values, column names, and comparison operators. The ORDER BY clause allows the result table to be sorted on the values that appear in the SELECT clause. If specified, the ORDER BY clause must be the final clause in the SELECT statement. The GROUP BY clause is used to form groups of rows of the result table based on column values. When grouping of rows occurs, all aggregate functions (e.g., COUNT, SUM, AVG) are computed on the individual groups and not the entire table. If used, the HAVING clause follows the GROUP BY clause. The HAVING clause functions as a WHERE clause for groups, keeping some groups and eliminating other groups from further consideration.

631

A data field without a value in it is said to be a null value. A null value can occur in a data field where a value is unknown or where a value is not meaningful. In an SQL SELECT statement, the only comparison operators that can be used with null values are IS NULL and IS NOT NULL. Any other operator (e.g., =, >, <) used with a null value will always produce an unknown (i.e., false) result.

SQL queries are based on one or more tables or views and often take the form of subqueries and joins. A subquery is an SQL SELECT statement embedded within another query or even another subquery. Subqueries may appear in the FROM clause, the column list, the WHERE clause, and the HAVING clause. The SQL SELECT statement is used to implement each of the relational algebra operations, including both inner and outer joins. Use of the SQL SELECT statement in joining tables is required for all queries where the result comes from more than one table.

A complete list of the SQL SELECT statement features appears in Appendix B. After studying this chapter, it is hoped that readers will be able to use the features discussed and, where necessary, adapt them to their specific database platform with a minimum of difficulty.

## Exercises

---

1. Describe the six clauses that can be used in the syntax of the SQL SELECT statement. Which two clauses must be part of each SELECT statement?
2. What is the difference between a SELECT statement used in conjunction with the relational algebra Selection operation and a SELECT statement used in conjunction with the relational algebra Projection operation?

- 632
3. Of what value is the use of parentheses when making use of the rules of operator precedence?
  4. What is the difference between a character field that contains a null value and a character field that contains a single blank space?
  5. Which comparison operators can be used when searching for null values? Which comparison operators cannot be used when searching for null values? What is the result when these unacceptable comparison operators are used when searching for null values?
  6. What is the result of an attempt to add, subtract, multiply, or divide two number fields, one of which contains a null value?
  7. How are null values treated when one or more appears during the execution of a group function?
  8. When must a GROUP BY clause be used in a query?
  9. What SQL operator (i.e., keyword) is used in conjunction with pattern matching?
  10. What is the difference between a SELECT statement that uses COUNT (\*) and a SELECT statement that uses COUNT (column name)? How does COUNT (column name) differ from COUNT (DISTINCT column name)?
  11. Why is it important to be aware of the distinction between the CHAR and VARCHAR data types?
  12. What is the difference between a Cross Join, an Inner Join, and an Outer Join?
  13. What is the difference between the JOIN ... USING and the JOIN ... ON approaches for joining tables? Which approach must be used if the requirement that attributes have unique names over the entire relational schema is enforced?
  14. What is a subquery, and where can subqueries appear within an SQL SELECT statement?
  15. What do the ALL and ANY operators do when used in a subquery?
  16. You must have completed Exercise 10 in Chapter 10 before beginning this exercise, and thus have used the SQL Data Definition Language to populate the tables for the three relations DRIVER, TICKET\_TYPE, and TICKET. Once the three tables have been populated, write SQL Select statements to satisfy the following information requests:
    - a. Display the names of all drivers.
    - b. Display the license numbers of all drivers who have been issued a ticket.
    - c. Display the names of all drivers who have been issued a ticket.
    - d. Display the license numbers of all drivers who have never been issued a ticket.
    - e. Display the names of all drivers who have never been issued a ticket.
    - f. Count the number of tickets issued for each offense. Include as part of what you display any offense for which a ticket has not been issued.
    - g. For each ticket issued, display the name of the driver, the ticket number, and the nature of the offense. Order the results in ascending order by the name of the driver; and, within each driver, order the results by ticket number.
  17. You must have completed Exercise 11 in Chapter 10 before beginning this exercise, and thus have used the SQL Data Definition Language to populate the tables for the three

relations COMPANY, STUDENT, and INTERNSHIP. Once the three tables have been populated, write SQL Select statements to satisfy the following information requests:

- a. Display the total monthly stipend received by each student.
- b. For each internship, display the year and quarter offered, headquarters of the company offering the internship, and location of the internship. The output should be displayed in ascending order by the headquarters of the company offering the internship.
- c. Display the names of all students who have not participated in an internship.
- d. Display the number of internships offered for each year, quarter, and internship location.
- e. Use pattern matching to display the names of those students whose name begins with an uppercase A and ends with some letter other than a lowercase a.
- f. Use a Natural Join to display the name, major, and status of those students with an internship in the same city where the company is headquartered.
- g. Use a subquery to display the name, major, and status of those students with an internship in the same city where the company is headquartered.
- h. Display the difference between the average stipend offered by Company A and the average stipend offered by all other companies, excluding Company A.
- i. Use a Left Outer Join to display the total monthly stipend received by each student, including those students who have not participated in an internship. What, if anything, makes you uncomfortable about the result obtained?
- j. Use a Union to display the total monthly stipend received by each student, including those students who have not participated in an internship.



# CHAPTER 13

# ADVANCED DATA MANIPULATION USING SQL

SQL for data manipulation is covered extensively in Chapter 12. In this chapter, we offer the reader a glimpse of some advanced features of SQL via an assortment of simple and easy-to-grasp examples. The discussion begins in Section 13.1 with an examination of a number of character-based built-in functions that can be used in an SQL statement anywhere a constant of the same data type can be used, whereas Section 13.2 focuses on functions that facilitate the manipulation of dates and times. The next four sections introduce SQL's features for writing hierarchical queries, using Extended GROUP BY clauses, working with analytical functions, and incorporating elements of spreadsheet modeling into the SQL SELECT statement. Hierarchical relationships exist in an organization chart, a bill of materials, or a family tree. Section 13.3 discusses the use of the CONNECT BY clause and the PRIOR operator in processing data of this type. Section 13.4 discusses the GROUP BY clause, which makes aggregating data from different perspectives simpler and more efficient. These enhancements take the form of the ROLLUP and CUBE operators supplemented by the GROUPING SETS extension to the GROUP BY clause, the GROUPING function, the GROUPING\_ID function, and the GROUP\_ID function. SQL's analytical functions and MODEL clause are business intelligence tools that allow data to be retrieved, analyzed, and reported. Two of SQL's analytical functions, ranking functions and window functions, are addressed in Section 13.5. The SQL MODEL clause, introduced in Section 13.6, makes it possible to define a multidimensional array on query results and then apply rules on the array to calculate new values. Section 13.7 concludes the chapter with a series of examples that apply a number of the SQL features introduced in this chapter and in Chapter 12.

As was the case in parts of Chapter 12, the execution of the queries in this chapter includes feedback as to the number of rows retrieved. In an effort to conserve space, information of this type is omitted for those queries where the number of rows retrieved is obvious.

## **13.1 SELECTED SQL:2003 BUILT-IN FUNCTIONS**

A built-in function can be used in an SQL expression anywhere that a constant of the same data type can be used. A large number of built-in functions is supported by popular SQL implementations. Column 1 of Table 13.1 contains the names of some of the most widely-used of the character-based built-in functions that comprise the SQL:2003 standard.

SQL:2003 Function	Oracle 10g Implementation	Chapter 13 Sections Containing Useful Examples
CASE	CASE and DECODE	CASE (13.1.7.2) DECODE (13.1.7.1)
CHAR_LENGTH	LENGTH	13.1.2
Concatenation	CONCAT	13.1.1
CURRENT_DATE	CURRENT_DATE	13.2
CURRENT_TIME	CURRENT_DATE (with mask)	13.2
Date/Time Conversions	TO_CHAR, TO_DATE	13.2
POSITION	INSTR	13.1.5
SUBSTRING	SUBSTR	13.1.1
TRANSLATE	TRANSLATE	13.1.4
TRIM	LTRIM and RTRIM	13.1.3

**TABLE 13.1** Selected SQL:2003 built-in functions and their Oracle equivalents

This section covers the use of the functions listed in Table 13.1 via a variety of short examples in the context of the SQL SELECT statement. Note that Oracle's SQL requires use of the FROM keyword in every SQL SELECT statement. Thus, many of the Oracle SQL examples in this section make use of the DUAL table. The DUAL table has one column, DUMMY CHAR(1), and one row with a value of "X." As we will see, the DUAL table is useful when a SELECT statement is issued to display data that does not exist in a table. It is particularly useful when you want to display a numeric or character literal in a SELECT statement.

Column 2 of Table 13.1 contains names of selected SQL:2003 standard built-in functions used by Oracle, while column 3 contains the section numbers in Chapter 13 where examples of the use of these functions can be found.

It is important to note that the syntax and some of the functionality of some of the SQL:2003 functions in this section varies across database platforms. In short, the material in this chapter, along with the material in Chapters 10, and 12, is intended to be a highly useful but not necessarily stand-alone reference to SQL. As such, the reader may need to supplement the material in this textbook with product-specific documentation.

### 13.1.1 The SUBSTRING Function

The purpose of the **SUBSTRING** function is to extract a substring from a given string. The format of the SUBSTRING function in the SQL:2003 standard is:

```
SUBSTRING (source FROM n FOR len)
```

where:

- *n* indicates the character position where the search begins
- *len* represents the length of the search.

Oracle implements the SUBSTRING function with the **SUBSTR (char, m [,n])** function, which returns a portion of *char*, beginning at character *m*, that is *n* characters long (if *n* is omitted, to the end of *char*). The first position of *char* is 1. Floating point numbers passed as arguments to SUBSTR are automatically converted to integers.

The SELECT statement in SUBSTRING Example 1 goes into the character string 'ABCDEFG' beginning at character position 3 and returns the next four characters, thus displaying "CDEF."

### SUBSTRING Example 1

```
SELECT SUBSTR ('ABCDEFG' , 3 , 4) "Substring" FROM DUAL;
```

**Result:** CDEF<sup>1</sup>

As shown in Examples 2 and 3, if the position where the search begins is not an integer, the value of the position argument **m** is truncated.

### SUBSTRING Example 2

```
SELECT SUBSTR ('ABCDEFG' , 3 . 1 , 4) "Substring" FROM DUAL;
```

**Result:** CDEF

### SUBSTRING Example 3

```
SELECT SUBSTR ('ABCDEFG' , 3 . 7 , 4) "Substring" FROM DUAL;
```

**Result:** CDEF

The position where the search begins can also be a negative number. In this case, characters beginning with the rightmost characters in the string are stripped off. As expected, the SELECT statement in Example 4 illustrates that a value of -5 for the starting position of the search produces the same result as when the starting position of the search has a value of 3.

### SUBSTRING Example 4

```
SELECT SUBSTR ('ABCDEFG' , -5 , 4) "Substring" FROM DUAL;
```

**Result:** CDEF

As shown in Example 5, if the number of characters to be searched is omitted, the number of characters returned extends to the end of the string.

### SUBSTRING Example 5

```
SELECT SUBSTR ('ABCDEFG' , -1) FROM DUAL;
```

**Result:** G

---

<sup>1</sup>The output of each function is accompanied by a column heading. Since the format used to display column headings varies by function and by product, only the value returned by the function is displayed in this section.

Observe that if the position where the search begins is a negative value and the number of characters to be stripped off is greater than the absolute value of *value where the search begins*, the number of characters returned extends only to the end of the string (see Example 6).

### SUBSTRING Example 6

```
SELECT SUBSTR ('ABCDEFG', -1, 3) FROM DUAL;
```

**Result:** G

However, as indicated in Examples 7 and 8, if the number of characters to be stripped off is zero or negative, a null value is displayed for the result.

### SUBSTRING Example 7

```
SELECT SUBSTR ('ABCDEFG', -1, 0) FROM DUAL;
```

**Result:** null value

### SUBSTRING Example 8

```
SELECT SUBSTR ('ABCDEFG', -1, -5) FROM DUAL;
```

**Result:** null value

The SUBSTR function is often used in conjunction with the concatenation operator (||). Example 9 illustrates their use together in displaying the name and phone number of all professors with phone numbers that end with two digits ranging between 45 and 65.

### SUBSTRING Example 9

```
SELECT PROFESSOR.NAME, ' (' || SUBSTR (PROFESSOR.PHONE, 1, 3) || ')'
    || SUBSTR (PROFESSOR.PHONE, 4, 3) || '-' ||
    SUBSTR (PROFESSOR.PHONE, 7, 4) "Phone" FROM PROFESSOR
 WHERE SUBSTR (PROFESSOR.PHONE, 9, 2) BETWEEN 45 AND 65;
```

**Result:**

NAME	Phone
John Smith	(523) 556-7645
John B Smith	(523) 556-7556
Sunil Shetty	(523) 556-6764
Katie Shef	(523) 556-8765
Cathy Cobal	(523) 556-5345
Jeanine Troy	(523) 556-5545
Tiger Woods	(523) 556-5563

7 rows selected.

### 13.1.2 The CHAR\_LENGTH (char) Function<sup>2</sup>

The SQL:2003 CHAR\_LENGTH (*char*)<sup>3</sup> function returns the length of the character string *char*. Oracle uses the LENGTH (*char*) syntax to implement the CHAR\_LENGTH function in the SQL:2003 standard. The LENGTH function returns a numeric value.

#### LENGTH Example 1

```
SELECT LENGTH ('ABCDEFG') FROM DUAL;
```

**Result:** 7

Using the TEXTBOOK table introduced in Section 12.1.5 of Chapter 12, the SELECT statements in Examples 2 and 3 illustrate the difference when the LENGTH function is applied to a column defined as a VARCHAR data type (**TEXTBOOK.TITLE**) versus one defined as a CHAR data type (**TEXTBOOK.PUBLISHER**).

#### LENGTH Example 2

```
SELECT TEXTBOOK.TITLE, TEXTBOOK.PUBLISHER, LENGTH(TEXTBOOK.TITLE)
FROM TEXTBOOK;
```

**Result:**

TITLE	PUBLISHER	LENGTH (TEXTBOOK.TITLE)
Database Management	Thomson	19
Linear Programming	Prentice-Hall	18
Simulation Modeling	Springer	19
Systems Analysis	Thomson	16
Principles of IS	Prentice-Hall	16
Economics For Managers		22
Programming in C++	Thomson	18
Fundamentals of SQL		19
Data Modeling		13

9 rows selected.

639

#### LENGTH Example 3

```
SELECT TEXTBOOK.TITLE, TEXTBOOK.PUBLISHER,
LENGTH(TEXTBOOK.PUBLISHER)
FROM TEXTBOOK;
```

---

<sup>2</sup>The LENGTH (*char*) function is called the CHAR\_LENGTH (*string*) function in the SQL:2003 standard. It represents the length of a character string.

<sup>3</sup>The LENGTH (*char*) function is the LEN (*char*) function in SQL Server.

**Result:**

TITLE	PUBLISHER	LENGTH (TEXTBOOK.PUBLISHER)
Database Management	Thomson	13
Linear Programming	Prentice-Hall	13
Simulation Modeling	Springer	13
Systems Analysis	Thomson	13
Principles of IS	Prentice-Hall	13
Economics For Managers		
Programming in C++	Thomson	13
Fundamentals of SQL		
Data Modeling		13

9 rows selected.

640

### 13.1.3 The TRIM Function

The SQL:2003 standard includes three basic **TRIM functions** for trimming characters from a string. **TRIM (LEADING unwanted FROM string)** trims off any leading occurrences of *unwanted*; **TRIM (TRAILING unwanted FROM string)** trims off any trailing occurrences of *unwanted*; and **TRIM (BOTH unwanted FROM string)** trims both leading and trailing occurrences of *unwanted*. TRIM is particularly useful with a CHAR data type in cases where it is desirable to remove unwanted blank spaces from the end of the string.

Oracle supports LTRIM and RTRIM functions. **LTRIM (string [,unwanted])** removes unwanted characters from the beginning of a *string*, while **RTRIM (string [,unwanted])** removes *unwanted* characters from the end of a *string*. As illustrated in Trim Example 1, the *unwanted* argument is a string that contains the characters to be trimmed, and defaults to a single space.

#### TRIM Example 1

```
SELECT LTRIM('      LAST WORD') FROM DUAL;
```

**Result:** LAST WORD

The SELECT statement in TRIM Example 2 trims the leftmost 3 lowercase x's from the character string 'xxxXxxLASTWORD'.

#### TRIM Example 2

```
SELECT LTRIM ('xxxXxxLAST WORD', 'x') FROM DUAL;
```

**Result:** XxxLAST WORD

All forms of the TRIM function are case sensitive. Thus, the SELECT statement in TRIM Example 3 does not result in trimming any characters from the string 'xxxXxxLAST WORD'.

#### TRIM Example 3

```
SELECT LTRIM ('xxxXxxLAST WORD', 'X') FROM DUAL;
```

**Result:** xxxXxxLAST WORD

TRIM Example 4 illustrates how the Oracle LTRIM function not only trims the word “Systems” from the beginning of the textbook title *Systems Analysis*, it also trims the leading “S” from all titles that begin with the letter “S” (i.e., *Simulation Modeling*). This is because in Oracle’s LTRIM function it is important to note that any character string that begins with any of the characters included in *unwanted* will be trimmed.

#### TRIM Example 4

```
SELECT TEXTBOOK.TITLE, LTRIM(TEXTBOOK.TITLE, 'Systems') "Trimmed Title"
FROM TEXTBOOK;
```

##### **Result:**

TITLE	Trimmed Title
Database Management	Database Management
Linear Programming	Linear Programming
Simulation Modeling	imulation Modeling
Systems Analysis	Analysis
Principles of IS	Principles of IS
Economics For Managers	Economics For Managers
Programming in C++	Programming in C++
Fundamentals of SQL	Fundamentals of SQL
Data Modeling	Data Modeling

9 rows selected.

641

The SELECT statements in TRIM Examples 5 through 7 provide additional details on how the Oracle LTRIM function works. Since there are no characters prior to the character string “LAST WORD” other than those in the set ‘xX’ all of the leading x’s and X’s are trimmed in TRIM Example 5.

#### TRIM Example 5

```
SELECT LTRIM ('xxxxXxxLAST WORD', 'xX') FROM DUAL;
```

##### **Result:** LAST WORD

Since the leading x’s in *char* are in the set ‘yx’, they are trimmed and thus not displayed in the result shown in TRIM Example 6.

#### TRIM Example 6

```
SELECT LTRIM ('xxxxXxxLAST WORD', 'yx') FROM DUAL;
```

##### **Result:** XxxLAST WORD

On the other hand, as illustrated by the SELECT statement in TRIM Example 7, since the leftmost character(s) in *unwanted* do not include a ‘y’ or ‘X’, no characters are trimmed in the result.

**TRIM Example 7**

```
SELECT LTRIM ('xxxxxxLAST WORD', 'yX') FROM DUAL;
```

**Result:** xxxxLAST WORD

The RTRIM function operates on the rightmost characters in a string in the same way that LTRIM operates on the leftmost characters in a string. It is important to be careful when using the RTRIM function in conjunction with a CHAR data type. Trim Examples 8 and 9 illustrate the difference between the application of this function to a VARCHAR versus CHAR data type column.

Since a VARCHAR data type stores no trailing blanks, the rightmost ‘g’ in the titles “Linear Programming” and “Simulation Modeling” are trimmed by the Oracle RTRIM function in Trim Example 8.

**TRIM Example 8**

```
SELECT RTRIM(TEXTBOOK.TITLE, 'g') "Trimmed Result" FROM TEXTBOOK;
```

**Result:**

Trimmed Result
-----
Database Management
Linear Programmin
Simulation Modelin
Systems Analysis
Principles of IS
Economics For Managers
Programming in C++
Fundamentals of SQL
Data Modelin

9 rows selected.

Trim Example 9 illustrates how the RTRIM function can be used by Oracle to trim (i.e., remove) unwanted blank spaces at the end of a CHAR data type. Column 3 of the result in Trim Example 9 confirms that the **TEXTBOOK.PUBLISHER** column in the **TEXTBOOK** table is defined as a CHAR(13) data type, while column 5 verifies that the RTRIM function used in column 4 removed all trailing blank spaces from the end of all not-null publishers.

**TRIM Example 9**

```
SELECT TEXTBOOK.TITLE, TEXTBOOK.PUBLISHER,
LENGTH(TEXTBOOK.PUBLISHER) "Length Pub",
RTRIM(TEXTBOOK.PUBLISHER) "Trimmed Pub",
LENGTH(RTRIM(TEXTBOOK.PUBLISHER)) "Trimmed Length"
FROM TEXTBOOK;
```

**Result:**

TITLE	PUBLISHER	Length	Pub	Trimmed Pub	Trimmed Length
Database Management	Thomson	13	Thomson	Thomson	7
Linear Programming	Prentice-Hall	13	Prentice-Hall	Prentice-Hall	13
Simulation Modeling	Springer	13	Springer	Springer	8
Systems Analysis	Thomson	13	Thomson	Thomson	7
Principles of IS	Prentice-Hall	13	Prentice-Hall	Prentice-Hall	13
Economics For Managers					
Programming in C++	Thomson	13	Thomson	Thomson	7
Fundamentals of SQL					
Data Modeling		13			

9 rows selected.

**13.1.4 The TRANSLATE Function**

The SQL:2003 **TRANSLATE function** is used to translate characters into a string. Oracle's implementation of the TRANSLATE function has the format:

643

```
TRANSLATE (char, from_string, to_string)
```

The function searches *char*, replacing each occurrence of a character found in *from\_string* with the corresponding character from *to\_string*. Characters that are in *char* but not in *from\_string* are left untouched, whereas characters in *from\_string* but not in *to\_string* are deleted. For example, the following TRANSLATE function could be used to extract the identifier of the department from each course number.

```
SELECT COURSE#, TRANSLATE (COURSE#, 'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789',
'ABCDEFGHIJKLMNOPQRSTUVWXYZ') "Department"
FROM COURSE;
```

**Result:**

COURSE#	Department
05ARCH101	ARCH
15ECON112	ECON
18ACCT801	ACCT
18ECON123	ECON
20ECES212	ECES
22IS270	IS
22IS330	IS
22IS430	IS
22IS832	IS
22QA375	QA
22QA411	QA
22QA888	QA

12 rows selected.

None of the characters in **COURSE.COURSE#** are left untouched because each character is part of *from\_string*. However, since the characters ‘0123456789’ in *from\_string* do not appear in *to\_string* ‘ABCDEFGHIJKLMNPQRSTUVWXYZ’, the digits in **COURSE.COURSE#** are not returned by the TRANSLATE function.

### 13.1.5 The POSITION Function

The SQL:2003 POSITION (*target* IN *source*) function returns the position where the *target* string appears within the *source* string. Both the target string and the source string are character strings that have the same character set. The POSITION (*target* IN *source*) function returns a numeric value as follows:

- If the *target* string is of length zero (i.e., it is a null value), the result returned is one.
- Otherwise, if the *target* string occurs as a substring within the *source* string, the result returned is one greater than the number of characters in the *source* string that precede the first such occurrence.
- Otherwise, the result is zero.

Oracle implements the POSITION function with an INSTR function. The Oracle INSTR function has the following format:

```
INSTR (source, target [, position [, occurrence]])
```

The *position* argument is used to specify the starting position for the search in *source*, and *occurrence* makes it possible for a specific occurrence to be found. If *position* is negative, the search begins from the end of the string.

The SELECT statement in INSTR Example 1 locates the character position of the second occurrence of the character ‘S’ in the character string ‘MISSISSIPPI’ beginning at character position 5. When the value of *occurrence* is changed to a 1 (see INSTR Example 2), observe that a different ‘S’ is located.

#### INSTR Example 1

```
SELECT INSTR ('MISSISSIPPI', 'S', 5, 2) FROM DUAL;
```

**Result:** 7

#### INSTR Example 2

```
SELECT INSTR ('MISSISSIPPI', 'S', 5, 1) FROM DUAL;
```

**Result:** 6

The SELECT statement in INSTR Example 3 locates all textbooks that contain the character string ‘ing’ somewhere in the title. Since both *position* and *occurrence* are omitted, their values are assumed to be equal to 1.

#### INSTR Example 3

```
SELECT TEXTBOOK.TITLE, INSTR(TEXTBOOK.TITLE, 'ing') "Position of ing"
FROM TEXTBOOK
WHERE INSTR(TEXTBOOK.TITLE, 'ing') > 0;
```

**Result:**

TITLE	Position of ing
Linear Programming	16
Simulation Modeling	17
Programming in C++	9
Data Modeling	11

4 rows selected.

Observe that the value of the INSTR function returned for all other titles is zero.

### 13.1.6 Combining the INSTR and SUBSTR Functions

The INSTR and SUBSTR functions are often combined. The purpose of the query in the following example is to begin with the second word of each textbook title and display all titles with the character string ‘ing’ in the title. The individual lines have been numbered to facilitate discussion of the execution of the query.

```

1  SELECT TEXTBOOK.TITLE,
2  INSTR(SUBSTR(TEXTBOOK.TITLE, INSTR(TEXTBOOK.TITLE, ' ')+1), 'ing')
3          "ing string in word 2",
4  INSTR(SUBSTR(TEXTBOOK.TITLE,1), 'ing')
5          "ing string in overall title"
6  FROM TEXTBOOK
7  WHERE INSTR(SUBSTR(TEXTBOOK.TITLE,
8  INSTR(TEXTBOOK.TITLE, ' ')+1), 'ing') > 0;

```

645

**Result:**

TITLE	ing string in word 2	ing string in overall title
Linear Programming	9	16
Simulation Modeling	6	17
Data Modeling	6	11

3 rows selected.

The WHERE clause shown on lines 7 and 8 governs the titles displayed when the query is executed. The **INSTR(TEXTBOOK.TITLE, ' ')** function on line 8 is evaluated first and returns the character position of the first blank space character in the title of each textbook (i.e., it is responsible for skipping over the first word of the title). By adding 1 to the value returned by the INSTR function, the character position of the first character in the second word of the title is obtained. The **SUBSTR(TEXTBOOK.TITLE, INSTR(TEXTBOOK.TITLE, ' ')+1)** function on lines 7 and 8 is evaluated next. Since a value of  $n$  is not provided, all remaining characters beginning with the second word of the title are selected. Finally, the outer INSTR function on line 7 looks for the first occurrence of the character string ‘ing’ in the second or remaining words in the title. The values displayed in columns 2 and 3 indicate the values returned by the INSTR function when asked to find the character position of the character string ‘ing’ starting with the second word of the title (column 2) and when asked to find the character position of the character string ‘ing’ starting with the first word of the title (column 3).

### 13.1.7 The DECODE Function and the CASE Expression

The DECODE (*value, search\_value, result, default\_value*) function is used to compare *value* with *search\_value*. If the values are equal, the DECODE function returns *result*; otherwise, *default\_value* is returned. The DECODE function allows you to perform if-then-else logic in SQL within a row. Similar to the DECODE function, you can also use the CASE expression to perform if-then-else logic in SQL within a row.

As an example of both, using the STUDENT table shown in Figure 11.1 of Chapter 11, suppose we want to display the student id, name, and grade level of students in descending order by grade level:

```
SELECT SID, NAME, GRADELEVEL
FROM STUDENT
ORDER BY GRADELEVEL DESC;
```

SID	NAME	GRADELEVEL
DT87656	Tim Duncan	SR
SD23556	David Sane	SR
AJ76998	Jenny Aniston	SR
FV67733	Vanessa Fox	SO
HJ45633	Jenna Hopp	SO
GS76775	Shweta Gupta	SO
HT67657	Troy Hudson	JR
BE76598	Elijah Baley	JR
OD76578	Daniel Olive	JR
KS39874	Sweety Kramer	JR
KP78924	Poppy Kramer	JR
JD35477	Diana Jackson	GR
FR45545	Rick Fox	GR
KJ56656	Joumana Kidd	FR
SW56547	Wanda Seldon	FR
BG66765	Gladis Bale	FR

16 rows selected.

#### 13.1.7.1 Using the DECODE Function

While the grade levels shown in the previous SQL statement are listed in descending order alphabetically, they are not listed in accordance with their normal ordinal scale. Use of the DECODE function corrects this problem by allowing the rows to be in descending order by the decoded values of the GRADELEVEL column.

##### SQL Select Statement:

```
SELECT SID, NAME, GRADELEVEL
FROM STUDENT
ORDER BY DECODE (GRADELEVEL, 'FR', '1', 'SO', '2', 'JR', '3', 'SR', 4, 'GR', 5) DESC;
```

**Result:**

SID	NAME	GRADELEVEL
FR45545	Rick Fox	GR
JD35477	Diana Jackson	GR
AJ76998	Jenny Aniston	SR
DT87656	Tim Duncan	SR
SD23556	David Sane	SR
HT67657	Troy Hudson	JR
BE76598	Elijah Baley	JR
OD76578	Daniel Olive	JR
KS39874	Sweety Kramer	JR
KP78924	Poppy Kramer	JR
GS76775	Shweta Gupta	SO
HJ45633	Jenna Hopp	SO
FV67733	Vanessa Fox	SO
BG66765	Gladis Bale	FR
KJ56656	Joumana Kidd	FR
SW56547	Wanda Seldon	FR

16 rows selected

647

### 13.1.7.2 Using the CASE Expression

There are two types of CASE expressions: the Simple CASE Expression and the Searched CASE Expression.

**Simple CASE Expressions.** Simple CASE expressions use expressions to determine the returned value and have the following syntax:

```
CASE search_expression
    WHEN expression1 THEN result1
    WHEN expression2 THEN result2
    ...
    WHEN expressionN THEN resultN
    ELSE default_result
END
```

where: *search\_expression* is the expression being evaluated  
*expression1, expression2, ..., expressionN* are the expressions to be evaluated against *search\_expression*  
*result1, result2, ..., resultN* are the returned results (one for each possible expression)  
*default\_result* is the default result returned when no matching expression is found

**SQL Select Statement:**

```
SELECT SID, NAME, GRADELEVEL
FROM STUDENT
ORDER BY CASE GRADELEVEL
WHEN 'FR' THEN '1'
WHEN 'SO' THEN '2'
WHEN 'JR' THEN '3'
WHEN 'SR' THEN '4'
WHEN 'GR' THEN '5'
END DESC;
```

**Result:**

SID	NAME	GRADELEVEL
FR45545	Rick Fox	GR
JD35477	Diana Jackson	GR
AJ76998	Jenny Aniston	SR
DT87656	Tim Duncan	SR
SD23556	David Sane	SR
HT67657	Troy Hudson	JR
BE76598	Elijah Baley	JR
OD76578	Daniel Olive	JR
KS39874	Sweety Kramer	JR
KP78924	Poppy Kramer	JR
GS76775	Shweta Gupta	SO
HJ45633	Jenna Hopp	SO
FV67733	Vanessa Fox	SO
BG66765	Gladis Bale	FR
KJ56656	Joumana Kidd	FR
SW56547	Wanda Seldon	FR

16 rows selected.

**Searched CASE Expressions.** Searched CASE expressions use conditions to determine the returned value and have following syntax:

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    ...
    WHEN conditionN THEN resultN
    ELSE default_result
END
```

where      *condition1, condition2, ..., conditionN* are the conditions to be evaluated  
*result1, result2, ..., resultN* are the returned results (one for each possible condition)  
*default\_result* is the default result returned when no true condition is found

**SQL Select Statement:**

```

SELECT SID, NAME, GRADELEVEL
FROM STUDENT
ORDER BY CASE
WHEN GRADELEVEL = 'FR' THEN '1'
WHEN GRADELEVEL = 'SO' THEN '2'
WHEN GRADELEVEL = 'JR' THEN '3'
WHEN GRADELEVEL = 'SR' THEN '4'
WHEN GRADELEVEL = 'GR' THEN '5'
END DESC;

```

**Result:**

SID	NAME	GRADELEVEL
FR45545	Rick Fox	GR
JD35477	Diana Jackson	GR
AJ76998	Jenny Aniston	SR
DT87656	Tim Duncan	SR
SD23556	David Sane	SR
HT67657	Troy Hudson	JR
BE76598	Elijah Baley	JR
OD76578	Daniel Olive	JR
KS39874	Sweety Kramer	JR
KP78924	Poppy Kramer	JR
GS76775	Shweta Gupta	SO
HJ45633	Jenna Hopp	SO
FV67733	Vanessa Fox	SO
BG66765	Gladis Bale	FR
KJ56656	Joumana Kidd	FR
SW56547	Wanda Seldon	FR

16 rows selected.

You can also use comparison operators other than an equals sign in a searched CASE expression.

### 13.1.8 A Query to Simulate the Division Operation

Recall that the Division operation is useful when there is a need to identify tuples in one relation that match all tuples in another relation. Recall that the Division operation is described in Chapter 11 as capable of being expressed as a sequence of Projection, Cartesian Product, and Difference operations. Let's consider this description in the context of the following question that originally appeared in Chapter 11 referring to the Madeira College tables: *What are the course numbers and course names of those courses offered*

*in all quarters during which sections are offered?* For the convenience of the reader, the content of the COURSE and SECTION tables follows:

### COURSE Table

NAME	COURSE#	CREDIT	COLLEGE	HRS	DCODE
Intro to Economics	15ECON112	U	Arts and Sciences	3	1
Operations Research	22QA375	U	Business	2	3
Intro to Economics	18ECON123	U	Education	4	4
Supply Chain Analysis	22QA411	U	Business	3	3
Principles of IS	22IS270	G	Business	3	7
Programming in C++	20ECES212	G	Engineering	3	6
Optimization	22QA888	G	Business	3	3
Financial Accounting	18ACCT801	G	Education	3	4
Database Concepts	22IS330	U	Business	4	7
Database Principles	22IS832	G	Business	3	7
Systems Analysis	22IS430	G	Business	3	7
Architectural History	05ARCH101	U		3	

650

### SECTION Table

SECTION#	TIME	MAXST	ROOM	COURSE#	PROFID
101A2007	T1015	25		22QA375	HT54347
901A2006	W1800	35	Rhodes 611	22IS270	SK85977
902A2006	H1700	25	Lindner 108	22IS270	SK85977
201S2006	T1045	29	Lindner 110	22IS330	SK85977
301S2006	H1045	29	Lindner 110	22IS330	CC49234
401W2007	M1000	33	Braunstien 211	22IS832	CC49234
102A2007	W1800		Baldwin 437	20ECES212	RR79345
103U2007	T1015	33		22QA375	HT54347
104A2007	H1700	29	Lindner 108	22IS330	SK85977
105S2007	T1015	30		22QA375	HT54347
106W2007	T1015	20		22QA375	HT54347

The following SQL SELECT statement can also be used to display the course(s) offered during all quarters. Three steps are required to execute this statement. First, the number of distinct quarters in the SECTION table is determined by the SELECT statement:

```
(SELECT DISTINCT(COUNT(DISTINCT(SUBSTR(SECTION.SECTION#, 4, 1)))) FROM SECTION).
```

Second, the COURSE and SECTION tables are joined on their course number attributes that share the same domain. This join yields a total of 11 rows. Next, the rows associated with the result of the join are logically grouped by the combination of **COURSE.COURSE#** and **COURSE.NAME**, with the HAVING clause used to identify the subset of groups we want to consider. Finally, since one course (course number 22QA375) is

associated with all four quarters, only one course number is displayed: that for course number 22QA375.

#### **SQL SELECT Statement:**

```
SELECT DISTINCT COURSE.COURSE#, COURSE.NAME
FROM COURSE JOIN SECTION
ON COURSE.COURSE# = SECTION.COURSE#
GROUP BY COURSE.COURSE#, COURSE.NAME;
HAVING COUNT(*) = (SELECT DISTINCT(COUNT(DISTINCT(SUBSTR
(SECTION.SECTION#, 4, 1)))) FROM SECTION)
```

#### **Result:**

COURSE#	NAME
22QA375	Operations Research

## **13.2 SOME BRIEF COMMENTS ON HANDLING DATES AND TIMES**

Examples 1.2.7–1.2.9 in Section 12.1.2 of Chapter 12 illustrate how a simple mathematical operation can be included in the column-list of a query. In addition to operations involving columns and constants defined as numeric, mathematical operations can be performed on dates and times.

All DBMS vendors offer SQL functions to handle dates and times. Unfortunately, the implementation of date/time data types is far from standardized across vendors. This problem occurs because the ANSI SQL:2003 standard defines the support of date data types but does not say how those data types should be stored (Rob and Coronel, 2004). Given these differences, the material in this section is based on Oracle and contains examples that work with Oracle dates and times. Please refer to the SQL reference material for your database platform for the syntax associated with features comparable to those discussed here.

As indicated in Table 10.1 of Chapter 10, a date data type in SQL:2003 is 10 characters long in the format yyyy-mm-dd. While several database products make use of this format, Oracle uses as its default date format dd-mon-yy, where “dd” represents a two-digit day, “mon” a three-letter month abbreviation, and “yy” a two-digit year (e.g., the date July 29, 2013 would be represented as 29-jul-13). The Oracle default format can be changed to that of the SQL:2003 default by the following SQL statement:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'yyyy-mm-dd';
```

The remainder of the examples in this section makes use of the Oracle default format for representing a date.<sup>4</sup>

Although referenced as a non-numeric field, a date is actually stored internally in a numeric format that includes the century, year, month, day, hour, minute, and second. While dates appear as non-numeric fields when displayed, calculations can be performed

---

<sup>4</sup>NLS is an acronym for “National Language Support.”

with dates because they are stored internally as numeric data in accordance with the Julian calendar. A Julian date represents the number of days that have elapsed between a specified date and January 1, 4712 B.C.

The CURRENT\_DATE function is part of the SQL:2003 standard and is used to record the current date and time. For example, the following SELECT Statement calculates the tenure (in days) and the tenure (in years) for each professor in department 3.

#### **SQL SELECT Statement:**

```
SELECT PROFESSOR.NAME, CURRENT_DATE - PROFESSOR.DATEHIRED "Tenure in Days",
TRUNC((CURRENT_DATE - PROFESSOR.DATEHIRED)/365.25,0) "Tenure in Years"
FROM PROFESSOR
WHERE PROFESSOR.DCODE = 3;
```

#### **Result:**

NAME	Tenure in Days	Tenure in Years
Chelsea Bush	7387.63231	20
Tony Hopkins	6027.63231	16
Alan Brodie	4815.63231	13
Jessica Simpson	6541.63231	17
Laura Jackson	4685.63231	12

5 rows selected

652

The tenure in days contains a fractional component because, when referenced in a query, the CURRENT\_DATE function retrieves not only the current date but also the time of day. Thus, the previous query was executed when 63.231 percent of a 24-hour day had elapsed (i.e., the query was executed at approximately 3:10 PM).

Oracle uses the TO\_CHAR and TO\_DATE functions with dates and times. The TO\_CHAR function is used to extract the different parts of a date/time and convert them to a character string<sup>5</sup>, while the TO\_DATE function is used to convert character strings to a valid date format. Both functions make use of a format element (also known as a format mask). Table 13.2 contains a sample of format elements commonly used in conjunction with the TO\_CHAR and TO\_DATE functions. Table 13.3 shows the number format elements used with the TO\_CHAR function.

---

<sup>5</sup>The TO\_CHAR function can also be used to convert a number to a formatted character string. Format masks used in conjunction with numbers appear in Table 13.3.

Element	Description	Example
MONTH, Month, or month	Name of the month spelled out (padded with blank spaces to a total width of nine spaces); case follows format.	JULY, July, or july (5 spaces follows each representation of July)
MON, Mon, or mon	Three-letter abbreviation of the name of the month; case follows format.	JUL, Jul, or jul
MM	Two-digit numeric value of the month	7
D	Numeric value of the day of the week	Monday = 2
DD	Numeric value of the day of the month	23
DAY, Day, or day	Name of the day of the week spelled out (padded with blank spaces to a length of nine characters)	MONDAY, Monday, or monday (3 spaces follows each representation of Monday)
fm	"Fill mode." When this element appears, subsequent elements (such as MONTH) suppress blank padding, leaving a variable-length result.	fmMonth, yyyy produces a date such as March, 2014
DY	Three-letter abbreviation of the day of the week	MON, Mon, or mon
YYYY	Four-digit year	2014
YY	Last two digits of the year	14
YEAR, Year, or year	Spells out the year; case follows year.	TWO THOUSAND FOURTEEN
BC or AD	Indicates B.C. or A.D.	2014 A.D.
AM or PM	Meridian indicator	10:00 AM
J	Julian date; January 1, 4712 B.C. is day 1.	July 27, 2014 is Julian date 2456865
SS	Seconds (value between 0 and 59)	21
MI	Minutes (value between 0 and 59)	32
HH	Hours (value between 1 and 12)	9
HH24	Hours (value between 0 and 23)	13

**TABLE 13.2** Selected date and time format elements used with the TO\_CHAR and TO\_DATE functions

Element	Description	Example
9	Series of 9s indicates width of display (with insignificant leading zeros not displayed).	99999
0	Displays insignificant leading zeros	0009999
\$	Displays a floating dollar sign to prefix value	\$99999
.	Indicates number of decimals to display	999.99
,	Displays a comma in the position indicated	9,999

**TABLE 13.3** Selected number format elements used with the TO\_CHAR function

The following query illustrates the use of the TO\_CHAR function to display the date hired and salary of each professor in department 3, using the default format and an alternative format:

#### SQL Select Statement:

654

```
SELECT PROFESSOR.NAME, PROFESSOR.DATEHIRED "SQL:2003 Date Format",
TO_CHAR(PROFESSOR.DATEHIRED, 'DD-MON-YYYY') "Alternate Format",
PROFESSOR.SALARY "Default Format",
TO_CHAR(PROFESSOR.SALARY, '$99,999.00') "Alternate Format"
FROM PROFESSOR
WHERE PROFESSOR.DCODE = 3;
```

#### Result:

NAME	SQL:2003 Date Format	Alternate Format	Default Format	Alternate Format
Chelsea Bush	01-MAY-93	01-MAY-1993	77000	\$77,000.00
Tony Hopkins	20-JAN-97	20-JAN-1997	77000	\$77,000.00
Alan Brodie	16-MAY-00	16-MAY-2000	76000	\$76,000.00
Jessica Simpson	25-AUG-95	25-AUG-1995	67000	\$67,000.00
Laura Jackson	23-SEP-00	23-SEP-2000	43000	\$43,000.00

When inserting a date in a table, Oracle assumes a default time of 12:00 AM (midnight). Should it be necessary to associate a time other than 12:00 AM with a date, the TO\_DATE function can be used. For example, suppose we wish to insert the date and time of admission for each new patient into the PATIENT table. The INSERT statement that appears below illustrates how this could be done. Prior to and following the INSERT statement are two SELECT statements. The first two SELECT statements display the name and date of admission of each patient prior to the insertion of the new patient. Note that the first SELECT statement displays the date of admission using the default date format, while the second displays the date of admission using a format mask that includes the time portion of the date of admission. The use of the TO\_DATE function in the INSERT statement for patient Zhaoping Zhang allows both the date and time of her admission to be recorded.

**Name and Date of Admission of Patients in PATIENT Table Prior to Insertion.**

```
SELECT PATIENT.PAT_NAME, PATIENT.PAT_ADMIT_DT "Date of Admission"
FROM PATIENT;
```

PAT_NAME	Date of Admission
Davis, Bill	2013-07-07
Grimes, David	2013-07-12
Lisauckis, Hal	2014-06-06

3 rows selected.

```
SELECT PATIENT.PAT_NAME, TO_CHAR(PAT_ADMIT_DT, 'fmMonth dd, yyyy HH:MI AM')
"Date of Admission"
FROM PATIENT;
```

PAT_NAME	Date of Admission
Davis, Bill	July 7, 2013 12:0 AM
Grimes, David	July 12, 2013 12:0 AM
Lisauckis, Hal	June 6, 2014 12:0 AM

3 rows selected.

**Insertion of New Patient.**

```
INSERT INTO PATIENT VALUES ('ZZ', '06912', 'Zhang, Zhaoping', 'F', 35,
TO_DATE('2014-08-11 10:15 AM', 'YYYY-MM-DD HH:MI AM'), NULL, NULL, NULL);
```

1 row created.

**Name and Date of Admission of Patients in PATIENT Table After Insertion.**

```
SELECT PATIENT.PAT_NAME, PATIENT.PAT_ADMIT_DT "Date of Admission"
FROM PATIENT;
```

PAT_NAME	Date of Admission
Davis, Bill	2013-07-07
Grimes, David	2013-07-12
Lisauckis, Hal	2014-06-06
Zhang, Zhaoping	2014-08-11

4 rows selected.

```
SELECT PATIENT.PAT_NAME, TO_CHAR(PAT_ADMIT_DT, 'fmMonth dd, yyyy HH:MI AM')
"Date of Admission"
FROM PATIENT;
```

PAT_NAME	Date of Admission
Davis, Bill	July 7, 2013 12:0 AM
Grimes, David	July 12, 2013 12:0 AM
Lisauckis, Hal	June 6, 2014 12:0 AM
Zhang, Zhaoping	August 11, 2014 10:15 AM

4 rows selected.

Suppose patient Zhaoping Zhang is discharged on August 12 at 3:35 PM (i.e. the value of the CURRENT\_DATE function is 3:35 PM). The following SELECT statement records her length of stay:

```
SELECT PATIENT.PAT_NAME, CURRENT_DATE - PATIENT.PAT_ADMIT_DT "Length of Stay"
FROM PATIENT
WHERE PATIENT.P#A = 'ZZ' AND PATIENT.P#N = '06912';
```

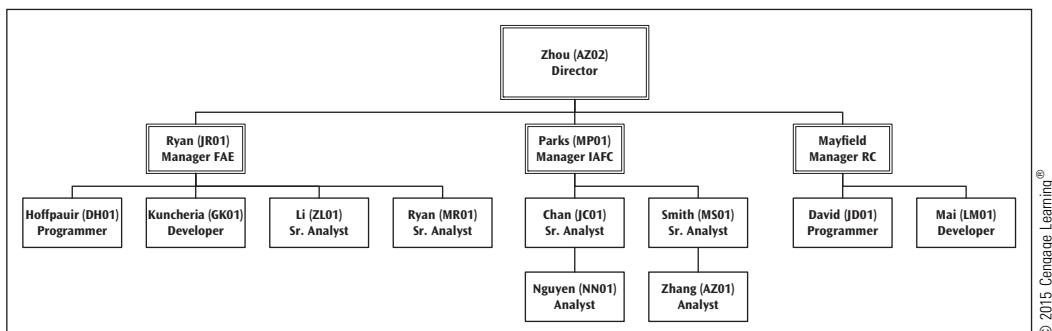
Pat_name	Length of Stay
Zhang, Zhaoping	1.2222338

656

Note that the 5 hours and 20 minutes (the difference between the time of day when she was discharged on August 12 and the time of day when she was admitted on August 11) is 22.22 percent of one day.

### 13.3 HIERARCHICAL QUERIES

We often encounter data that is organized in the form of a hierarchy, such as the people who work in an organization. Consider AZ Consultants, a small consulting company headquartered in Calgary, Alberta that specializes in creating custom software for clients in the financial services area. For how AZ Consultants is structured, see Figure 13.1



© 2015 Cengage Learning®

**FIGURE 13.1** Organization chart for AZ Consultants

In other words, Alicia Zhou, the company founder, serves as its director, with James Ryan, Michael Parks, and Ron Mayfield serving as managers of these three departments: Finance and Accounting Excellence (FAE), Risk and Compliance (RC), and Internal Audit and Financial Controls (IAFC). The CONSULTANT table contains data on each of the company's 14 consultants:

```
SELECT * FROM CONSULTANT;
```

ID	NAME	GENDER	TITLE	DID	SALARY	HIREDATE	REPTS_TO
GK01	Kuncheria, Ginnu	M	Developer	FAE	80000	28-FEB-09	JR01
JR01	Ryan, James R.	M	Manager	FAE	135000	24-NOV-08	AZ02
DH01	Hoffpauir, Deb	F	Programmer	FAE	80000	09-DEC-09	JR01
JC01	Chan, Jackie	M	Sr. Analyst	IAFC	80000		MP01
AZ01	Zhang, Anthony	M	Analyst	IAFC	65000	25-NOV-09	MS01
AZ02	Zhou, Alicia	F	Director		225000	03-SEP-08	
MS01	Smith, Maranda	F	Sr. Analyst	IAFC	72500	15-MAY-12	MP01
RM01	Mayfield, Ron	M	Manager	RC	120000	11-NOV-08	AZ02
MR01	Ryan, Michael	M	Sr. Analyst	FAE		14-FEB-13	JR01
MP01	Parks, Michael	M	Manager	IAFC	140000	09-DEC-08	AZ02
JD01	David, Jason	M	Programmer	RC	72500	24-NOV-11	RM01
NN01	Nguyen, Nicole	F	Analyst	IAFC	42000	22-MAR-12	JC01
LM01	Mai, Ly H.	F	Developer	RC	72000	17-MAR-09	RM01
ZL01	Li, ZP	F	Sr. Analyst	FAE	100000	31-MAY-11	JR01

14 rows selected.

The REPTS\_TO column refers back to the ID column and thus reflects the supervisor of the employee (if any).

Data organized in a hierarchy are said to form a tree the elements of which are called nodes. Four types of nodes make up a tree:

- *Root node*—The node at the top of the tree. For example, in Figure 13.1, the root node is Zhou, the Director.
- *Parent node*—A node that has one or more nodes below it. For example, in Figure 13.1, Ryan is the parent of Hoffpauir, Kuncheria, Li, and Ryan. Note that two different employees of AZ Consultants have the name Ryan.
- *Child node*—A node with one parent node above it. For example, in Figure 13.1, Zhang's parent is Smith.
- *Leaf node*—A node with no children. Hoffpauir and David are two of the eight leaf nodes.

### 13.3.1 Using the CONNECT BY and START WITH Clauses with the PRIOR Operator

To achieve a hierarchical display, a query must be constructed with the CONNECT BY clause and the PRIOR operator. The PRIOR operator along with the CONNECT BY and START WITH clauses makes use of the following syntax:

```
SELECT [LEVEL], column, expression, ...
FROM table
[WHERE where_clause]
[ [START WITH start_condition] [CONNECT BY PRIOR prior_condition] ]
```

where:

- LEVEL is a pseudo-column that tells you how far into the tree you are.
- start\_condition specifies where to start the hierarchical query from.
- prior\_condition specifies the relationship between the parent and child rows; the relationship between the parent and the child is established by placing the PRIOR operator before the parent column.

To find the children of a parent, SQL evaluates the expression qualified by the PRIOR operator for the parent row. Rows for which the condition is true are the children of the parent. Using the CONSULTANT table, the following CONNECT BY clause makes it possible to see the children of a parent:

CONNECT BY PRIOR ID = REPTS\_TO

The ID column is the parent, and the REPTS\_TO column is the child. The PRIOR operator is placed in front of the parent column ID. As illustrated in the queries that follow in Examples 1 and 2, depending on which column you prefix with the PRIOR operator, the direction of the hierarchy changes.

The START WITH clause determines the root rows of the hierarchy. The records for which the START WITH clause is true are selected first. All children are retrieved from these records going forward.

To find the children of a specific parent—for example, Michael Parks (or ID = 'MP01') in Example 1—SQL evaluates the expression qualified by the PRIOR operator for the parent row. Rows for which the condition is true are the children of the parent.

#### Example 1

```
SELECT * FROM CONSULTANT
START WITH ID = 'MP01'
CONNECT BY PRIOR ID = REPTS_TO;
```

ID	NAME	GENDER	TITLE	DID	SALARY	HIREDATE	REPTS_TO
MP01	Parks, Michael	M	Manager	IAFC	140000	09-DEC-08	AZ02
JC01	Chan, Jackie	M	Sr. Analyst	IAFC	80000		MP01
NN01	Nguyen, Nicole	F	Analyst	IAFC	42000	22-MAR-12	JC01
MS01	Smith, Maranda	F	Sr. Analyst	IAFC	72500	15-MAY-12	MP01
AZ01	Zhang, Anthony	M	Analyst	IAFC	65000	25-NOV-09	MS01

5 rows selected.

The START WITH clause determines the root rows<sup>6</sup> of the hierarchy. The records for which the START WITH clause is true are first selected. All children are retrieved from these records going forward.

- In this case, after displaying Michael Parks in the first row, the first consultant (i.e., child) reporting to Parks is displayed (Jackie Chan, employee id JC01) in row 2. Note that the content of the REPTS\_TO column in row 2 (the child) is equal to the content of the ID column in row 1 (the parent).
- After displaying Jackie Chan on row 2, the first consultant reporting to Chan is displayed (Nicole Nguyen, employee id NN01) in row 3.
- Since no consultant reports to Nicole Nguyen, the second consultant reporting to Michael Parks is displayed next (Maranda Smith, employee id MS01) in row 4.
- After displaying Maranda Smith, the first consultant reporting to Smith is displayed in row 5 (Anthony Zhang, employee id AZ01).
- Since no consultant reports to Anthony Zhang and no more consultants report to Maranda Smith and no more consultants report to Michael Parks, the execution of the query terminates with the fifth row.

In summary, note that Michael Parks is displayed first, then one of his senior analysts is displayed, followed by all (in this case, just the one) of the senior analyst's subordinates, followed by the other senior analyst, followed by all (in this case, just one) of her subordinates. Without the START WITH clause, SQL uses all rows in the table as root rows. It is left as an exercise to the reader to verify that such a query will generate 40 rows of output. Queries without a CONNECT BY clause generate a syntax error.

As mentioned previously, depending on which column you prefix with the PRIOR operator, the direction of the hierarchy changes. For example, consider the following query:

### Example 2

```
SELECT * FROM CONSULTANT
START WITH ID = 'NN01'
CONNECT BY ID = PRIOR REPTS_TO;
```

ID	NAME	GENDER	TITLE	DID	SALARY	HIREDATE	REPTS
NN01	Nguyen, Nicole	F	Analyst	IAFC	42000	22-MAR-12	JC01
JC01	Chan, Jackie	M	Sr. Analyst	IAFC	80000		MP01
MP01	Parks, Michael	M	Manager	IAFC	140000	09-DEC-08	AZ02
AZ02	Zhou, Alicia	F	Director		225000	03-SEP-08	

4 rows selected.

Rather than displaying all employees under Nicole Nguyen in the hierarchy, all employees above Nicole Nguyen in the hierarchy (i.e., all employees that Nicole Nguyen ultimately reports to) are displayed. In other words, this query starts at the child and traverses upward.

---

<sup>6</sup>In this example, there is only one root row.

### 13.3.2 Using the LEVEL Pseudo-Column

The LEVEL pseudo-column returns the number 1 for the root of the hierarchy, 2 for the child, 3 for the grandchild, and so on.

#### Example 3

```
SELECT LEVEL, ID, NAME, TITLE, REPTS_TO
FROM CONSULTANT
START WITH ID = 'AZ02'
CONNECT BY PRIOR ID = REPTS_TO;
```

LEVEL	ID	NAME	TITLE	REPTS_TO
1	AZ02	Zhou, Alicia	Director	
2	JR01	Ryan, James R	Manager	AZ02
3	DH01	Hoffpauir, Deb	Programmer	JR01
3	GK01	Kuncheria, Ginu	Developer	JR01
3	MR01	Ryan, Michael	Sr. Analyst	JR01
3	ZL01	Li, ZP	Sr. Analyst	JR01
2	MP01	Parks, Michael	Manager	AZ02
3	JC01	Chan, Jackie	Sr. Analyst	MP01
4	NN01	Nguyen, Nicole	Analyst	JC01
3	MS01	Smith, Maranda	Sr. Analyst	MP01
4	AZ01	Zhang, Anthony	Analyst	MS01
2	RM01	Mayfield, Ron	Manager	AZ02
3	JD01	David, Jason	Programmer	RM01
3	LM01	Mai, Ly H	Developer	RM01

14 rows selected.

Observe how the use of the LEVEL pseudo-column traverses the hierarchy from top to bottom, left to right.

The COUNT() function can be used with the LEVEL pseudo-column to obtain the number of levels in the “tree.”

#### Example 4

```
SELECT COUNT (DISTINCT LEVEL)
FROM CONSULTANT
START WITH ID = 'AZ02'
CONNECT BY PRIOR ID = REPTS_TO;
```

COUNT (DISTINCT LEVEL)

4

1 row selected.

The LEVEL pseudo-column makes it easier to understand the results of a query without a START WITH clause.

### 13.3.3 Formatting the Results from a Hierarchical Query

The results from a hierarchical query can be formatted using LEVEL and the LPAD() function<sup>7</sup>, which left-pads values with characters. The query in Example 5 indents a consultant's name with spaces based on the value of the LEVEL pseudo-column (LEVEL 1 is not padded, LEVEL 2 is padded by two spaces, LEVEL 3 by four spaces, and so on).

#### Example 5

```
SELECT LEVEL, LPAD(' ', 2*(LEVEL-1)) || NAME "Consultant Name"
FROM CONSULTANT
START WITH ID = 'AZ02'
CONNECT BY PRIOR ID = REPTS_TO;
```

LEVEL	Consultant Name
1	Zhou, Alicia
2	Ryan, James R.
3	Hoffpauir, Deb
3	Kuncheria, Ginu
3	Ryan, Michael
3	Li, ZP
2	Parks, Michael
3	Chan, Jackie
4	Nguyen, Nicole
3	Smith, Maranda
4	Zhang, Anthony
2	Mayfield, Ron
3	David, Jason
3	Mai, Ly H.

14 rows selected.

Note how use of the LPAD function allows you to visualize the hierarchy by indenting it with spaces. The length of the padded characters is calculated with the LEVEL pseudo-column. Of course, it is not necessary to display the value of the LEVEL pseudo-column.

### 13.3.4 Using a Subquery in a START WITH Clause

As shown in Example 6, a subquery can be used in a START WITH clause. Note that the START WITH clause uses the preposition IN. Had the START WITH clause used an equals sign, the error message stating that a single-row subquery returns more than one row would have been obtained (see Example 3.1.3 in Section 12.3 of Chapter 12).

---

<sup>7</sup>The LPAD(x, width, [ pad\_string]) is used to pad x with spaces to the left to bring the total length of the string up to width character. If a string is supplied in pad\_string, this string is repeated to the left to fill up the padded space. The resulting padded string is then returned.

**Example 6**

```
SELECT TITLE, LPAD(' ', 2*(LEVEL-1)) || NAME "Consultant Name"
FROM CONSULTANT
START WITH ID IN
  (SELECT ID FROM CONSULTANT
   WHERE TITLE = 'Manager')
CONNECT BY PRIOR ID = REPTS_TO;
```

TITLE	Consultant Name
Manager	Ryan, James R.
Developer	Kuncheria, Ginu
Programmer	Hoffpauir, Deb
Sr. Analyst	Ryan, Michael
Sr. Analyst	Li, ZP
Manager	Mayfield, Ron
Programmer	David, Jason
Developer	Mai, Ly H.
Manager	Parks, Michael
Sr. Analyst	Chan, Jackie
Analyst	Nguyen, Nicole
Sr. Analyst	Smith, Maranda
Analyst	Zhang, Anthony

13 rows selected.

A WHERE clause can also be used to eliminate a particular node from a query. For example, the query in Example 7 eliminates Michael Parks from the previous query.

**Example 7**

```
SELECT TITLE, LPAD(' ', 2*(LEVEL-1)) || NAME "Consultant Name"
FROM CONSULTANT
START WITH ID IN
  (SELECT ID FROM CONSULTANT
   WHERE TITLE = 'Manager'
   AND NAME NOT LIKE '%Parks%')
CONNECT BY PRIOR ID = REPTS_TO;
```

TITLE	Consultant Name
-----	-----
Manager	Ryan, James R.
Developer	Kuncheria, Ginu
Programmer	Hoffpauir, Deb
Sr. Analyst	Ryan, Michael
Sr. Analyst	Li, ZP
Manager	Mayfield, Ron
Programmer	David, Jason
Developer	Mai, Ly H.

8 rows selected.

### 13.3.5 The SYS\_CONNECT\_BY\_PATH Function

The SYS\_CONNECT\_BY\_PATH function is used in hierarchical queries to obtain a concatenated list of column values in the path from the root node to the current node. It accepts two input parameters of a character data type. The first parameter is the column that indicates the scalar value of the root node. The second parameter is the delimiter to be used as the node value separator. The SELECT statement in Example 8 displays a consultant id, the consultant name, and the node path starting from the root node element. In essence, what is displayed is a horizontal orientation of the vertical structure for each node.

#### Example 8

```
SELECT ID, NAME, SYS_CONNECT_BY_PATH (NAME, '/') "Path"
FROM CONSULTANT
START WITH ID = 'AZ02'
CONNECT BY PRIOR ID = REPTS_TO;
```

ID	NAME	Path
-----	-----	-----
AZ02	Zhou, Alicia	/Zhou, Alicia
JR01	Ryan, James R.	/Zhou, Alicia/Ryan, James R.
DH01	Hoffpauir, Deb	/Zhou, Alicia/Ryan, James R./Hoffpauir, Deb
GK01	Kuncheria, Ginu	/Zhou, Alicia/Ryan, James R./Kuncheria, Ginu
MR01	Ryan, Michael	/Zhou, Alicia/Ryan, James R./Ryan, Michael
ZL01	Li, ZP	/Zhou, Alicia/Ryan, James R./Li, ZP
MP01	Parks, Michael	/Zhou, Alicia/Parks, Michael
JC01	Chan, Jackie	/Zhou, Alicia/Parks, Michael/Chan, Jackie
NN01	Nguyen, Nicole	/Zhou, Alicia/Parks, Michael/Chan, Jackie/Nguyen, Nicole
MS01	Smith, Maranda	/Zhou, Alicia/Parks, Michael/Smith, Maranda
AZ01	Zhang, Anthony	/Zhou, Alicia/Parks, Michael/Smith, Maranda/Zhang, Anthony
RM01	Mayfield, Ron	/Zhou, Alicia/Mayfield, Ron
JD01	David, Jason	/Zhou, Alicia/Mayfield, Ron/David, Jason
LM01	Mai, Ly H.	/Zhou, Alicia/Mayfield, Ron/Mai, Ly H.

14 rows selected.

It is left as an exercise for the reader to explain the output displayed when the PRIOR operator is placed in front of the REPTS\_TO column.

### 13.3.6 Joins in Hierarchical Queries

Joins are permitted in hierarchical queries. The query in Example 9 makes use of the ASSIGNMENT table that records the assignment of consultants reporting to Michael Parks on projects. The structure and content of the ASSIGNMENT table follow:

DESCRIBE ASSIGNMENT

Name	Null?	Type
AS_P_ID	-----	NUMBER
AS_C_ID	-----	VARCHAR2 (5)
AS_S_DATE	-----	DATE
AS_C_DATE	-----	DATE
AS_HOURS	-----	NUMBER

SELECT \* FROM ASSIGNMENT;

664

AS_P_ID	AS_C_ID	AS_S_DATE	AS_C_DATE	AS_HOURS
100	MP01	24-FEB-13	25-MAR-13	75
100	JD01	01-MAR-13	25-MAR-13	40
100	ZL01	01-APR-13		60
150	JC01	15-MAR-13		100
200	LM01	09-MAR-13	02-MAY-13	40
200	DH01	25-MAR-13		40
250	NN01	16-APR-13	07-MAY-13	100
250	MS01	01-APR-13		50
300	GK01	01-MAY-13		30
350	AZ01	20-APR-13	07-MAY-13	50

10 rows selected.

Note that the START WITH and CONNECT BY clauses in Example 9 are simply part of the join condition.

#### Example 9

```
SELECT NAME, TITLE, DID, AS_C_ID, AS_P_ID, AS_HOURS
FROM CONSULTANT JOIN ASSIGNMENT
ON CONSULTANT.ID = AS_C_ID
START WITH ID = 'MP01'
CONNECT BY PRIOR ID = REPTS_TO;
```

NAME	TITLE	DID	AS_C_ID	AS_P_ID	AS_HOURS
Parks, Michael	Manager	IAFC	MP01	100	75
Chan, Jackie	Sr. Analyst	IAFC	JC01	150	100
Nguyen, Nicole	Analyst	IAFC	NN01	250	100
Smith, Maranda	Sr. Analyst	IAFC	MS01	250	50
Zhang, Anthony	Analyst	IAFC	AZ01	350	50

5 rows selected.

### 13.3.7 Incorporating a Hierarchical Structure into a Table

A foreign key constraint of the form:

```
CONSTRAINT REPTS_TO_FK FOREIGN KEY (REPTS_TO) REFERENCES CONSULTANT (ID);
```

is necessary to insure that the hierarchical structure of the CONSULTANT table is maintained. Assume that the CONSULTANT table was created using the following CREATE TABLE statement:

```
CREATE TABLE CONSULTANT
  (ID VARCHAR2(5) PRIMARY KEY,
  NAME VARCHAR2(15) NOT NULL,
  GENDER CHAR(1),
  TITLE VARCHAR2(12),
  DID VARCHAR2(5) REFERENCES DEPARTMENT (DID),
  SALARY NUMBER(6),
  HIREDATE DATE,
  REPTS_TO TO VARCHAR2(5));
```

665

and the rows were inserted in the order given here:

```
SELECT * FROM CONSULTANT;
```

ID	NAME	GENDER	TITLE	DID	SALARY	HIREDATE	REPTS_TO
GK01	Kuncheria, Ginu	M	Developer	FAE	80000	28-FEB-09	JR01
JR01	Ryan, James R	M	Manager	FAE	135000	24-NOV-08	AZ02
DH01	Hoffpauir, Deb	F	Programmer	FAE	80000	09-DEC-09	JR01
JC01	Chan, Jackie	M	Sr. Analyst	IAFC	80000		MP01
AZ01	Zhang, Anthony	M	Analyst	IAFC	65000	25-NOV-09	MS01
AZ02	Zhou, Alicia	F	Director		225000	03-SEP-08	
MS01	Smith, Maranda	F	Sr. Analyst	IAFC	72500	15-MAY-12	MP01
RM01	Mayfield, Ron	M	Manager	RC	120000	11-NOV-08	AZ02
MR01	Ryan, Michael	M	Sr. Analyst	FAE		14-FEB-13	JR01
MP01	Parks, Michael	M	Manager	IAFC	140000	09-DEC-08	AZ02
JD01	David, Jason	M	Programmer	RC	72500	24-NOV-11	RM01
NN01	Nguyen, Nicole	F	Analyst	IAFC	42000	22-MAR-12	JC01
LM01	Mai, Ly H	F	Developer	RC	72000	17-MAR-09	RM01
ZL01	Li, ZP	F	Sr. Analyst	FAE	100000	31-MAY-11	JR01

14 rows selected.

Note how the foreign key constraint has not been included in the CREATE TABLE statement. Indeed, had it been, the first row could not have been inserted into the CONSULTANT table because the consultant to whom Ginu Kuncheria reports did not exist in the table at this time.

Observe how the hierarchical structure is reflected in Example 10 despite the fact that there is no constraint in CONSULTANT that requires a consultant to report to an existing consultant.

### Example 10

```
SELECT * FROM CONSULTANT
START WITH ID = 'AZ02'
CONNECT BY PRIOR ID = REPTS_TO;
```

ID	NAME	GENDER	TITLE	DID	SALARY	HIREDATE	REPTS_TO
AZ02	Zhou, Alicia	F	Director		225000	03-SEP-08	
JR01	Ryan, James R	M	Manager	FAE	135000	24-NOV-08	AZ02
DH01	Hoffpauir, Deb	F	Programmer	FAE	80000	09-DEC-09	JR01
GK01	Kuncheria, Ginu	M	Developer	FAE	80000	28-FEB-09	JR01
MR01	Ryan, Michael	M	Sr. Analyst	FAE		14-FEB-13	JR01
ZL01	Li, ZP	F	Sr. Analyst	FAE	100000	31-MAY-11	JR01
MP01	Parks, Michael	M	Manager	IAFC	140000	09-DEC-08	AZ02
JC01	Chan, Jackie	M	Sr. Analyst	IAFC	80000		MP01
NN01	Nguyen, Nicole	F	Analyst	IAFC	42000	22-MAR-12	JC01
MS01	Smith, Maranda	F	Sr. Analyst	IAFC	72500	15-MAY-12	MP01
AZ01	Zhang, Anthony	M	Analyst	IAFC	65000	25-NOV-09	MS01
RM01	Mayfield, Ron	M	Manager	RC	120000	11-NOV-08	AZ02
JD01	David, Jason	M	Programmer	RC	72500	24-NOV-11	RM01
LM01	Mai, Ly H	F	Developer	RC	72000	17-MAR-09	RM01

14 rows selected.

Note how it is possible to insert a new consultant (King Nelson) who does not report to an existing consultant (i.e., there is no employee id ZZ02).

```
INSERT INTO CONSULTANT VALUES ('KN01', 'Nelson, King', 'M', 'Developer', 'FAE',
100000, '23-JUL-13', 'ZZ02');
```

1 row created.

However, since the CONNECT BY clause is not satisfied in the following hierarchical query (i.e., employee KN01 does not report to anyone in the hierarchy), consultant KN01 does not appear in the output generated by SELECT statement shown in Example 11.

### Example 11

```
SELECT * FROM CONSULTANT
START WITH ID = 'AZ02'
CONNECT BY PRIOR ID = REPTS_TO;
```

ID	NAME	GENDER	TITLE	DID	SALARY	HIREDATE	REPTS_TO
AZ02	Zhou, Alicia	F	Director		225000	03-SEP-08	
JR01	Ryan, James R	M	Manager	FAE	135000	24-NOV-08	AZ02
DH01	Hoffpauir, Deb	F	Programmer	FAE	80000	09-DEC-09	JR01
GK01	Kuncheria, Ginu	M	Developer	FAE	80000	28-FEB-09	JR01
MR01	Ryan, Michael	M	Sr. Analyst	FAE		14-FEB-13	JR01
ZL01	Li, ZP	F	Sr. Analyst	FAE	100000	31-MAY-11	JR01
MP01	Parks, Michael	M	Manager	IAFC	140000	09-DEC-08	AZ02
JC01	Chan, Jackie	M	Sr. Analyst	IAFC	80000		MP01
NN01	Nguyen, Nicole	F	Analyst	IAFC	42000	22-MAR-12	JC01
MS01	Smith, Maranda	F	Sr. Analyst	IAFC	72500	15-MAY-12	MP01
AZ01	Zhang, Anthony	M	Analyst	IAFC	65000	25-NOV-09	MS01
RM01	Mayfield, Ron	M	Manager	RC	120000	11-NOV-08	AZ02
JD01	David, Jason	M	Programmer	RC	72500	24-NOV-11	RM01
LM01	Mai, Ly H	F	Developer	RC	72000	17-MAR-09	RM01

14 rows selected.

Let's assign consultant KN01 to one of the existing consultants (consultant JR01):

667

```
UPDATE CONSULTANT SET REPTS_TO = 'JR01' WHERE ID = 'KN01';
```

1 row updated.

Since the CONNECT BY clause is now satisfied, consultant KN01 now appears in the highlighted row:

### Example 12

```
SELECT * FROM CONSULTANT
START WITH ID = 'AZ02'
CONNECT BY PRIOR ID = REPTS_TO;
```

ID	NAME	GENDER	TITLE	DID	SALARY	HIREDATE	REPTS_TO
AZ02	Zhou, Alicia	F	Director		225000	03-SEP-08	
JR01	Ryan, James R	M	Manager	FAE	135000	24-NOV-08	AZ02
DH01	Hoffpauir, Deb	F	Programmer	FAE	80000	09-DEC-09	JR01
GK01	Kuncheria, Ginu	M	Developer	FAE	80000	28-FEB-09	JR01
KN01	Nelson, King	M	Developer	FAE	100000	23-JUL-13	JR01
MR01	Ryan, Michael	M	Sr. Analyst	FAE		14-FEB-13	JR01
ZL01	Li, ZP	F	Sr. Analyst	FAE	100000	31-MAY-11	JR01
MP01	Parks, Michael	M	Manager	IAFC	140000	09-DEC-08	AZ02
JC01	Chan, Jackie	M	Sr. Analyst	IAFC	80000		MP01
NN01	Nguyen, Nicole	F	Analyst	IAFC	42000	22-MAR-12	JC01
MS01	Smith, Maranda	F	Sr. Analyst	IAFC	72500	15-MAY-12	MP01
AZ01	Zhang, Anthony	M	Analyst	IAFC	65000	25-NOV-09	MS01
RM01	Mayfield, Ron	M	Manager	RC	120000	11-NOV-08	AZ02
JD01	David, Jason	M	Programmer	RC	72500	24-NOV-11	RM01
LM01	Mai, Ly H	F	Developer	RC	72000	17-MAR-09	RM01

15 rows selected.

Now, let's add the constraint that requires each consultant to report to an existing consultant. Once this constraint has been added, new consultants can be safely added to the hierarchical structure as long as they report to an existing consultant.

```
ALTER TABLE CONSULTANT ADD CONSTRAINT REPTS_TO_FK
FOREIGN KEY (REPTS_TO) REFERENCES CONSULTANT (ID);
```

Table altered.

```
INSERT INTO CONSULTANT VALUES ('JA01', 'Abbott, John', 'M', 'Programmer', 'FAE',
100000, '24-JUL-13', 'ZZ02');
INSERT INTO CONSULTANT VALUES ('JA01', 'Abbott, John', 'M', 'Programmer', 'FAE',
*
ERROR at line 1:
ORA-02291: integrity constraint (BUILDHIERARCHY.REPTS_TO_FK) violated - parent
key not found
```

## 13.4 EXTENDED GROUP BY CLAUSES

668

SQL contains several extensions to the GROUP BY clause that allows data to be summarized from different perspectives. These extensions come in the form of the ROLLUP and CUBE operators, the GROUPING() function, the GROUPING SETS clause, the GROUPING\_ID function, and the GROUP\_ID function.

### 13.4.1 The ROLLUP Operator

The ROLLUP operator extends the GROUP BY clause to return a row containing a subtotal for each group along with a total for all groups. Assume Alicia Zhou, the Director of AZ Consultants, wants to calculate the total salaries paid to the consultants in each department. The SELECT statement in Example 1 uses GROUP BY to group the rows produced by joining the CONSULTANT and DEPARTMENT<sup>8</sup> tables by department id and uses SUM() to obtain the sum of the salaries for each department.

#### Example 1

```
SELECT DNAME "DEPARTMENT NAME", SUM(SALARY)
FROM CONSULTANT JOIN DEPARTMENT
ON CONSULTANT.DID = DEPARTMENT.DID
GROUP BY DNAME;

DEPARTMENT NAME                      SUM (SALARY)
-----                                 -----
Finance and Accounting Excellence    395000
Risk and Compliance                 264500
Internal Audit and Financial Controls 399500

3 rows selected.
```

---

<sup>8</sup>The following CREATE TABLE statement created the DEPARTMENT table:  
CREATE TABLE DEPARTMENT (DID VARCHAR2(5) PRIMARY KEY, DNAME VARCHAR2(40) NOT NULL, DMGR VARCHAR2(5));

The query in Example 2 rewrites the query in Example 1 using the ROLLUP operator. Notice how the additional row at the end contains the total salaries for the three departments.

### Example 2

```
SELECT DNAME "DEPARTMENT NAME", SUM(SALARY)
FROM CONSULTANT JOIN DEPARTMENT
ON CONSULTANT.DID = DEPARTMENT.DID
GROUP BY ROLLUP (DNAME) ;
```

DEPARTMENT NAME	SUM (SALARY)
Finance and Accounting Excellence	395000
Internal Audit and Financial Controls	399500
Risk and Compliance	264500
	1059000

4 rows selected.

669

### 13.4.2 Passing Multiple Columns to ROLLUP

When multiple columns are used with the ROLLUP operator, rows are then grouped into blocks that have identical values in the grouping columns. The SELECT statement in Example 3 passes the department name and gender columns to ROLLUP, which groups the rows by identical values in these columns. For example, note that the salaries are summed for all departments with the same name and gender combination values (this yields six different totals/rows). The ROLLUP operator also returns a row with the sum of the salaries for each department (this yields three different totals), along with another row at the very end with the total salaries for all consultants.<sup>9</sup> In other words, the ROLLUP operator first calculates the standard aggregate values specified in the GROUP BY clause. Then it creates progressively higher-level subtotals, moving from right to left through the list of grouping columns. At the end, it creates a grand total.

### Example 3

```
SELECT DNAME "DEPARTMENT NAME", GENDER, SUM(SALARY)
FROM CONSULTANT JOIN DEPARTMENT
ON CONSULTANT.DID = DEPARTMENT.DID
GROUP BY ROLLUP (DNAME, GENDER) ;
```

---

<sup>9</sup>When using the ROLLUP and CUBE operators, the order in which the rows are displayed can vary among different database products.

DEPARTMENT NAME	GENDER	SUM (SALARY)
Risk and Compliance	F	72000
Risk and Compliance	M	192500
Risk and Compliance		264500 an example of (a) below
Finance and Accounting Excellence	F	180000
Finance and Accounting Excellence	M	215000
Finance and Accounting Excellence		395000
Internal Audit and Financial Controls	F	114500
Internal Audit and Financial Controls	M	285000 an example of (b) below
Internal Audit and Financial Controls		399500
		1059000 the example of (c) below

10 rows selected.

Note how the result includes (a) the sum of all salaries in each department, (b) the sum of the salaries of the consultants in each department for each gender, and (c) the sum of the salaries of all consultants. As shown in Example 4, if a GROUP BY (DNAME, GENDER) clause had simply been used, only the six sums associated with (b) would be displayed.

670

#### Example 4

```
SELECT DNAME "DEPARTMENT NAME", GENDER, SUM(SALARY)
FROM CONSULTANT JOIN DEPARTMENT
ON CONSULTANT.DID = DEPARTMENT.DID
GROUP BY (DNAME, GENDER);
```

DEPARTMENT NAME	GENDER	SUM (SALARY)
Finance and Accounting Excellence	F	180000
Internal Audit and Financial Controls	M	285000
Risk and Compliance	M	192500
Finance and Accounting Excellence	M	215000
Internal Audit and Financial Controls	F	114500
Risk and Compliance	F	72000

6 rows selected.

The SELECT statement in Example 5 illustrates the totals generated when three columns are passed to the ROLLUP operator. Since there are five different jobs across the three departments and two genders, a total of 30 possible sums could be generated. Another six possible sums could be generated for each of the six department name and gender combinations. There would also be four additional sums (a sum for each department name along with a sum of the salaries for all consultants. However, given the fact that no department has more than one consultant of each gender for each of its job titles,

only 13 of the 30 possible job, department name, and gender combinations exist. When added to the sums for the six department name and gender combinations, the three departments, and the single grouping of all consultants, the query shown next produces a total of 23 rows.

### Example 5

```
SELECT DNAME "DEPARTMENT NAME", GENDER, TITLE, SUM(SALARY)
FROM CONSULTANT JOIN DEPARTMENT
ON CONSULTANT.DID = DEPARTMENT.DID
GROUP BY ROLLUP (DNAME, GENDER, TITLE);
```

DEPARTMENT NAME	GENDER	TITLE	SUM (SALARY)
Risk and Compliance	F	Developer	72000
Risk and Compliance	F		72000
Risk and Compliance	M	Manager	120000
Risk and Compliance	M	Programmer	72500
Risk and Compliance	M		192500
Risk and Compliance			264500
Finance and Accounting Excellence	F	Programmer	80000
Finance and Accounting Excellence	F	Sr. Analyst	100000
Finance and Accounting Excellence	F		180000
Finance and Accounting Excellence	M	Manager	135000
Finance and Accounting Excellence	M	Developer	80000
Finance and Accounting Excellence	M	Sr. Analyst	
Finance and Accounting Excellence	M		215000
Finance and Accounting Excellence			395000
Internal Audit and Financial Controls	F	Analyst	42000
Internal Audit and Financial Controls	F	Sr. Analyst	72500
Internal Audit and Financial Controls	F		114500
Internal Audit and Financial Controls	M	Analyst	65000
Internal Audit and Financial Controls	M	Manager	140000
Internal Audit and Financial Controls	M	Sr. Analyst	80000
Internal Audit and Financial Controls	M		285000
Internal Audit and Financial Controls			399500
Internal Audit and Financial Controls			1059000

671

23 rows selected.

### 13.4.3 Changing the Position of Columns Passed to ROLLUP

The query in Example 6 switches department name and gender in the ROLLUP operator so that gender is listed before department name. This causes ROLLUP to return a row with the sum of the salaries for each gender (highlighted rows 4 and 8) plus, of course, the sum of the salaries for all consultants (highlighted row 9).

**Example 6**

```
SELECT DNAME "DEPARTMENT NAME", GENDER, SUM(SALARY)
FROM CONSULTANT JOIN DEPARTMENT
ON CONSULTANT.DID = DEPARTMENT.DID
GROUP BY ROLLUP (GENDER, DNAME) ;
```

DEPARTMENT NAME	GENDER	SUM (SALARY)
Risk and Compliance	F	72000
Finance and Accounting Excellence	F	180000
Internal Audit and Financial Controls	F	114500
	F	366500
Risk and Compliance	M	192500
Finance and Accounting Excellence	M	215000
Internal Audit and Financial Controls	M	285000
	M	692500
		1059000

672

9 rows selected.

Any of the other aggregate functions can be used with ROLLUP (e.g., AVG(), COUNT(), MAX(), MEDIAN(), MIN(), STDDEV(), VARIANCE()).

#### 13.4.4 Using the CUBE Operator

The CUBE operator extends GROUP BY to return rows containing a subtotal of all combinations of columns included in the CUBE, along with a total at the end (i.e., it makes it possible to generate all possible combinations of groups). The SELECT statement in Example 7 passes department name and gender to CUBE, which groups the rows by identical values in those columns. Note that the salaries are summed for each department name and gender combination (this yields six different totals/rows) and the CUBE operator returns a row with the sum of the salaries in each department (this sum is also generated by using ROLLUP), along with the sum of all salaries in each gender and, finally, a row with the total salary of all consultants.

**Example 7**

```
SELECT DNAME "DEPARTMENT NAME", GENDER, SUM(SALARY)
FROM CONSULTANT JOIN DEPARTMENT
ON CONSULTANT.DID = DEPARTMENT.DID
GROUP BY CUBE(DNAME, GENDER) ;
```

DEPARTMENT NAME	GENDER	SUM (SALARY)
		1059000
	F	366500
	M	692500
Risk and Compliance		264500
Risk and Compliance	F	72000
Risk and Compliance	M	192500
Finance and Accounting Excellence		395000
Finance and Accounting Excellence	F	180000
Finance and Accounting Excellence	M	215000
Internal Audit and Financial Controls		399500
Internal Audit and Financial Controls	F	114500
Internal Audit and Financial Controls	M	285000

12 rows selected.

The query in Example 8 switches department name and gender so that gender is listed before department name. This still results in CUBE returning separate sums of the salaries for each department and for each gender.

673

### Example 8

```
SELECT DNAME "DEPARTMENT NAME", GENDER, SUM(SALARY)
FROM CONSULTANT JOIN DEPARTMENT
ON CONSULTANT.DID = DEPARTMENT.DID
GROUP BY CUBE(GENDER, DNAME) ;
```

DEPARTMENT NAME	GENDER	SUM (SALARY)
		1059000
Risk and Compliance		264500
Finance and Accounting Excellence		395000
Internal Audit and Financial Controls		399500
	F	366500
Risk and Compliance	F	72000
Finance and Accounting Excellence	F	180000
Internal Audit and Financial Controls	F	114500
	M	692500
Risk and Compliance	M	192500
Finance and Accounting Excellence	M	215000
Internal Audit and Financial Controls	M	285000

12 rows selected.

### 13.4.5 The GROUPING () Function

The GROUPING () function accepts a column and returns a 0 or 1. The function returns 1 when it encounters a null value created by a ROLLUP or CUBE operation. It is used to help distinguish summary rows from any rows that are a result of null values. The GROUPING () function is only used in queries that use ROLLUP or CUBE. As illustrated in Examples 9 and 10, the GROUPING () function is useful when you want to display a value when a null value would otherwise be returned.

#### 13.4.5.1 Using GROUPING () with a Single Column in a ROLLUP

Recall Example 2 (repeated here) for the basic ROLLUP, where the last row that contains the total salaries for the three departments has a null value for the department name:

```
SELECT DNAME "DEPARTMENT NAME", SUM(SALARY)
FROM CONSULTANT JOIN DEPARTMENT
ON CONSULTANT.DID = DEPARTMENT.DID
GROUP BY ROLLUP(DNAME);
```

674

DEPARTMENT NAME	SUM (SALARY)
Finance and Accounting Excellence	395000
Internal Audit and Financial Controls	399500
Risk and Compliance	264500
	1059000

4 rows selected.

As shown in the SELECT statement in Example 9, the GROUPING () function can be used to determine whether a value in a given column is null. Note that GROUPING () returns 0 for the rows that have non-null department name values and 1 for the last row that has a null department name.

#### Example 9

```
SELECT GROUPING(DNAME), DNAME "DEPARTMENT NAME", SUM(SALARY)
FROM CONSULTANT JOIN DEPARTMENT
ON CONSULTANT.DID = DEPARTMENT.DID
GROUP BY ROLLUP(DNAME);
```

GROUPING(DNAME)	DEPARTMENT NAME	SUM (SALARY)
0	Finance and Accounting Excellence	395000
0	Internal Audit and Financial Controls	399500
0	Risk and Compliance	264500
1		1059000

4 rows selected.

The GROUPING () function eliminates any ambiguities. Whenever you see a value of 1 in a column where the GROUPING () function is applied, it indicates the presence of what is referred to as a “super aggregate row,” such as a subtotal or grand total row created with the ROLLUP or CUBE operator.<sup>10</sup>

### 13.4.5.2 Using the DECODE () Function to Convert the Returned Value from the GROUPING () Function

As illustrated in Example 10, the DECODE () function can be used with the GROUPING () function to add a label to what would otherwise be displayed as a null value for a column (the department name column in this case).

#### Example 10

```
SELECT DECODE (GROUPING (DNAME) , 1, 'All Departments', DNAME) "DEPARTMENT NAME",
SUM(SALARY)
FROM CONSULTANT JOIN DEPARTMENT
ON CONSULTANT.DID = DEPARTMENT.DID
GROUP BY ROLLUP (DNAME) ;
```

675

DEPARTMENT NAME	SUM (SALARY)
Finance and Accounting Excellence	395000
Internal Audit and Financial Controls	399500
Risk and Compliance	264500
All Departments	1059000

4 rows selected.

The DECODE () and GROUPING () functions can also be used to display a meaningful label for multiple column values. The SELECT statement in Example 11 replaces the null values in a ROLLUP based on the department name and gender columns.

#### Example 11

```
SELECT DECODE(GROUPING(DNAME) , 1, 'All Departments', DNAME) "DEPARTMENT NAME",
DECODE(GROUPING(GENDER) , 1, 'Both Genders', GENDER) "GENDER", SUM(SALARY)
FROM CONSULTANT JOIN DEPARTMENT
ON CONSULTANT.DID = DEPARTMENT.DID
GROUP BY ROLLUP (DNAME, GENDER) ;
```

---

<sup>10</sup>The rows where GROUPING (DNAME) is equal to 0 exist because of the GROUP BY clause.

DEPARTMENT NAME	GENDER	SUM (SALARY)
Risk and Compliance	F	72000
Risk and Compliance	M	192500
Risk and Compliance	Both Genders	264500
Finance and Accounting Excellence	F	180000
Finance and Accounting Excellence	M	215000
Finance and Accounting Excellence	Both Genders	395000
Internal Audit and Financial Controls	F	114500
Internal Audit and Financial Controls	M	285000
Internal Audit and Financial Controls	Both Genders	399500
All Departments	Both Genders	1059000

10 rows selected.

### 13.4.6 The GROUPING SETS Extension to the GROUP BY Clause

The GROUPING SETS extension to the GROUP BY clause is used to compute and display selective results for the set of groups you want to create. The SELECT statement in Example 12 uses the GROUPING SETS extension to get just the subtotals for salaries by department name and gender.

676

#### Example 12

```
SELECT DNAME "DEPARTMENT NAME", GENDER, SUM(SALARY)
FROM CONSULTANT JOIN DEPARTMENT
ON CONSULTANT.DID = DEPARTMENT.DID
GROUP BY GROUPING SETS (DNAME, GENDER) ;
```

DEPARTMENT NAME	GENDER	SUM (SALARY)
Finance and Accounting Excellence		395000
Risk and Compliance		264500
Internal Audit and Financial Controls		399500
	M	692500
	F	366500

5 rows selected.

Note how the GROUPING SETS extension eliminates the highlighted rows in Example 13 that would appear had the GROUP BY CUBE (DNAME, GENDER) been used.

#### Example 13

```
SELECT DNAME "DEPARTMENT NAME", GENDER, SUM(SALARY)
FROM CONSULTANT JOIN DEPARTMENT
ON CONSULTANT.DID = DEPARTMENT.DID
GROUP BY CUBE (DNAME, GENDER) ;
```

DEPARTMENT NAME	GENDER	SUM (SALARY)
		1059000
	F	366500
	M	692500
Risk and Compliance		264500
Risk and Compliance	F	72000
Risk and Compliance	M	192500
Finance and Accounting Excellence		395000
Finance and Accounting Excellence	F	180000
Finance and Accounting Excellence	M	215000
Internal Audit and Financial Controls		399500
Internal Audit and Financial Controls	F	114500
Internal Audit and Financial Controls	M	285000

12 rows selected.

While the GROUPING SETS extension allows the subtotals for the department name and gender columns to be returned; the total for all consultants is not returned. The GROUPING\_ID () described below makes it possible to obtain this total.

677

### 13.4.7 The GROUPING\_ID ()

The GROUPING\_ID () filters rows using a HAVING clause so as to exclude rows that do not contain a subtotal or total. The GROUPING\_ID () accepts one or more columns and returns the decimal equivalent (i.e., base 10 value) of the GROUPING bit vector. The GROUPING bit vector is computed by combining the results of a call to the GROUPING () function for each column in order.

Previously, we saw that the GROUPING () function returns 1 when the column value is null and returns 0 when the column value is non-null. For example:

- If both department name and gender are non-null, GROUPING () would return 0 for both columns. The result for department name is combined with the result for gender, giving a bit vector of 00, whose decimal equivalent is 0 (i.e.,  $0 * 2^1 + 0 * 2^0 = 0$ ). The GROUPING\_ID () function therefore returns 0 when department name and gender are non-null.
- If department name is non-null (the GROUPING bit is 0) but gender is null (the GROUPING bit is 1), the resulting bit vector is 01, and thus the GROUPING\_ID () function returns 1.
- If department name is null (the GROUPING bit is 1) but gender is non-null (the GROUPING bit is 0), the resulting bit vector is 10, and thus GROUPING\_ID () returns 2.
- If department name is null (the GROUPING bit is 1) and gender is null (the GROUPING bit is 1), the resulting bit vector is 11, and thus GROUPING\_ID () returns 3.

The SELECT statement in Example 14 passes department name and gender to the GROUPING\_ID () function. Note that the output from the GROUPING\_ID () function shown in the GROUPING ID VALUE column agrees with the expected returned values described earlier.

#### Example 14

```
SELECT DNAME "DEPARTMENT NAME", GENDER,
GROUPING(DNAME) "DNAME GROUP",
GROUPING(GENDER) "GENDER GROUP",
GROUPING_ID(DNAME,GENDER) "GROUPING ID VALUE",
SUM(SALARY)
FROM CONSULTANT JOIN DEPARTMENT
ON CONSULTANT.DID = DEPARTMENT.DID
GROUP BY CUBE(DNAME, GENDER);
```

DEPARTMENT NAME	GENDER	DNAME GROUP	GENDER GROUP	GROUPING ID VALUE	SUM(SALARY)
		1	1	3	1059000
	F	1	0	2	366500
	M	1	0	2	692500
Risk and Compliance		0	1	1	264500
Risk and Compliance	F	0	0	0	72000
Risk and Compliance	M	0	0	0	192500
Finance and Accounting Excellence		0	1	1	395000
Finance and Accounting Excellence	F	0	0	0	180000
Finance and Accounting Excellence	M	0	0	0	215000
Internal Audit and Financial Controls		0	1	1	399500
Internal Audit and Financial Controls	F	0	0	0	114500
Internal Audit and Financial Controls	M	0	0	0	285000

678

12 rows selected.

One useful application of the GROUPING\_ID () is to filter rows by using it as part of a HAVING clause. As illustrated in Example 15, a HAVING clause can be used to exclude rows that do not contain a subtotal or total by checking to see if the GROUPING\_ID () returns a value greater than 0.

#### Example 15

```
SELECT DNAME "DEPARTMENT NAME", GENDER, SUM(SALARY)
FROM CONSULTANT JOIN DEPARTMENT
ON CONSULTANT.DID = DEPARTMENT.DID
GROUP BY CUBE(DNAME, GENDER)
HAVING GROUPING_ID(DNAME,GENDER) > 0;
```

DEPARTMENT NAME	GENDER	SUM (SALARY)
		1059000
	F	366500
	M	692500
Risk and Compliance		264500
Finance and Accounting Excellence		395000
Internal Audit and Financial Controls		399500

6 rows selected.

The CUBE operator with the HAVING clause shown earlier considers only the sum of the salaries for each department as a separate subtotal, the sum of the salaries for each gender as a separate subtotal, and the sum of the salaries for all consultants (i.e., departments/genders) as a separate subtotal.

### 13.4.8 Using a Column Multiple Times in a GROUP BY Clause

You can use a column multiple times in a GROUP BY clause. This makes it possible for you to reorganize your data or report on different groupings of data. For example, the SELECT statement in Example 16 contains a GROUP BY clause that uses department name twice, once to group by department name and the second in a ROLLUP.

#### Example 16

```
SELECT DNAME "DEPARTMENT NAME", GENDER, SUM(SALARY)
FROM CONSULTANT JOIN DEPARTMENT
ON CONSULTANT.DID = DEPARTMENT.DID
GROUP BY DNAME, ROLLUP(DNAME, GENDER);
```

DEPARTMENT NAME	GENDER	SUM (SALARY)
Risk and Compliance	F	72000
Risk and Compliance	M	192500
Finance and Accounting Excellence	F	180000
Finance and Accounting Excellence	M	215000
Internal Audit and Financial Controls	F	114500
Internal Audit and Financial Controls	M	285000
Risk and Compliance		264500
Finance and Accounting Excellence		395000
Internal Audit and Financial Controls		399500
Risk and Compliance		264500
Finance and Accounting Excellence		395000
Internal Audit and Financial Controls		399500

12 rows selected.

Note that the last three rows are duplicates of the previous three rows. These duplicates can be eliminated using the GROUP\_ID () function illustrated in Examples 17 and 18.

If the department name column is referenced only as part of the ROLLUP operator, as we have seen previously, the output includes the sum of the salaries of all consultants in addition to the sum of the salaries for each department. However, when you group by department name before using the ROLLUP operator, the sum of the salaries of all consultants is not calculated.

The GROUPING\_ID () distinguishes among duplicate groupings by removing duplicate rows returned by a GROUP BY clause. GROUP\_ID () does not accept any parameters. If n duplicates exist for a particular grouping, GROUPING\_ID () returns numbers in the range of 0 to n-1. The query in Example 18 rewrites the query in Example 17 that references the department name column two times so as to include the output of the GROUP\_ID () function. Note that GROUP\_ID () returns 0 for all rows except for the last three, which are duplicates of the previous three rows. For these rows, GROUP\_ID () returns a value of 1.

### Example 17

```
SELECT DNAME "DEPARTMENT NAME", GENDER, GROUP_ID(), SUM(SALARY)
FROM CONSULTANT JOIN DEPARTMENT
ON CONSULTANT.DID = DEPARTMENT.DID
GROUP BY DNAME, ROLLUP(DNAME, GENDER);
```

680

DEPARTMENT NAME	GENDER	GROUP_ID()	SUM(SALARY)
Risk and Compliance	F	0	72000
Risk and Compliance	M	0	192500
Finance and Accounting Excellence	F	0	180000
Finance and Accounting Excellence	M	0	215000
Internal Audit and Financial Controls	F	0	114500
Internal Audit and Financial Controls	M	0	285000
Risk and Compliance		0	264500
Finance and Accounting Excellence		0	395000
Internal Audit and Financial Controls		0	399500
Risk and Compliance		1	264500
Finance and Accounting Excellence		1	395000
Internal Audit and Financial Controls		1	399500

12 rows selected.

### Example 18

```
SELECT DNAME "DEPARTMENT NAME", GENDER, SUM(SALARY)
FROM CONSULTANT JOIN DEPARTMENT
ON CONSULTANT.DID = DEPARTMENT.DID
GROUP BY DNAME, ROLLUP(DNAME, GENDER)
HAVING GROUP_ID() = 0;
```

DEPARTMENT NAME	GENDER	SUM (SALARY)
Risk and Compliance	F	72000
Risk and Compliance	M	192500
Finance and Accounting Excellence	F	180000
Finance and Accounting Excellence	M	215000
Internal Audit and Financial Controls	F	114500
Internal Audit and Financial Controls	M	285000
Risk and Compliance		264500
Finance and Accounting Excellence		395000
Internal Audit and Financial Controls		399500

9 rows selected.

## 13.5 USING THE ANALYTICAL FUNCTIONS

SQL includes a number of useful functions that allow you to analyze, aggregate, and rank stored data. Analytical functions execute queries fairly quickly because they allow you to make one pass through the data rather than write multiple queries or complicated SQL to achieve the same result.

The general syntax of analytical functions is as follows:

```
analytic_function (<argument>, <argument>, ...) OVER (<query_partitioning_clause>
<order_by_clause> <>windowing_clause>)
```

where the following apply:

- **analytic\_function** specifies the name of an analytic function (e.g., RANK, DENSE\_RANK, AVG, CORR).
- **argument** - analytic functions take anywhere from 0 to 3 arguments.
- **OVER** indicates that the analytic function operates after the results of the FROM, WHERE, GROUP BY, and HAVING clauses have been formed.
- **query\_partitioning\_clause** refers to a clause that logically breaks a single result set into N groups (or partitions), according to the criteria set by the partition expressions. The analytic functions are applied to each group independently.
- **order\_by\_clause** refers to a clause that specifies how the data is sorted within each group (or partition). It is applied to the result of an analytic function.
- **windowing\_clause** defines a sliding or anchored window of data on which the analytic function will operate within a group. It lets you compute moving and accumulative aggregates—such as moving averages, moving sums, or cumulative sums—by choosing only certain data within a specified window.

There are slight variations in the general syntax for certain analytical functions, with some requiring specific clauses and others not. The QUERY\_PARTITIONING\_CLAUSE allows you to split a result into smaller subsets on which you can apply the analytical

functions. The ORDER\_BY\_CLAUSE is much like the familiar ordering clause; however, it is applied to the result of an analytical function. The WINDOWING\_CLAUSE lets you compute moving and accumulative aggregates—such as moving averages, moving sums, or cumulative sums—by choosing only certain data within a specified window.

Rischert (2010) describes query processing with analytical functions as being performed in three steps:

1. The joins, WHERE, GROUP BY, and HAVING clauses are carried out.
2. The following analysis of the results from Step 1 takes place:
  - If any partitioning clause is listed, the rows are split into appropriate partitions. These partitions are formed after the GROUP BY clause, so you may be able to analyze data by partition, not just the expressions of the GROUP BY clause.
  - If a windowing clause is involved, the ranges of the sliding windows of rows are determined. The analytical functions are based against the specified window and allow moving averages and moving sums.
  - Analytic functions may have an ORDER BY clause as part of the function specification that allows you to order the result before the analytical function is applied.
3. The third step occurs if an ORDER BY clause is present at the end of the statement and the results are sorted accordingly.

682

This section illustrates several analytical functions in the context of the MONTHLY\_SALES and MONTH tables that appear in Figure 13.2. The MONTHLY\_SALES table contains the number of units sold per month for each of five products. The MONTH table contains the names of the months of the year. It is included so that the examples that follow can display month names instead of month numbers.

### 13.5.1 Analytical Function Types

Price (2004) organizes the analytical functions into the following types:

- *ranking functions*—Used to calculate ranks, percentiles, and n-tiles (quartiles, etc.)
- *inverse percentage functions*—Used to calculate the value that corresponds to a percentile
- *window functions*—Used to calculate cumulative and moving aggregates
- *reporting functions*—Used to calculate things like market shares
- *lag and lead functions*—Make it possible for you to get a value in a row where that row is a certain number of rows away from the current row
- *first and last functions*—Used to get the first and last values in an ordered group
- *linear regression functions*—Used to fit an ordinary-least-squares regression line to a set of number pairs
- *hypothetical rank and distribution functions*—Used to calculate the rank and percentile that a new row would have if you inserted it into a table

MONTHLY_SALES TABLE			MONTHLY_SALES TABLE (CONTINUED)		
PRODUCT	MNUM	UNITS SOLD	PRODUCT	MNUM	UNITS SOLD
1	1	91	4	1	65
1	2	61	4	2	76
1	3	53	4	3	90
1	4	98	4	4	62
1	5	91	4	5	53
1	6	63	4	6	83
1	7	88	4	7	53
1	8	74	4	8	93
1	9	59	4	9	65
1	10	58	4	10	59
1	11	86	4	11	96
1	12	81	4	12	83
2	1	52	5	1	86
2	2	98	5	2	83
2	3	85	5	3	68
2	4	93	5	4	84
2	5	51	5	5	58
2	6	55	5	6	62
2	7	57	5	7	69
2	8	61	5	8	71
2	9	69	5	9	52
2	10	66	5	10	78
2	11	56	5	11	93
2	12	67	5	12	74
3	1	77	MONTH TABLE		
3	2	96	MNUM	MNAME	
3	3	54			
3	4	59			
3	5	81	1	January	
3	6	81	2	February	
3	7	73	3	March	
3	8	97	4	April	
3	9	79	5	May	
3	10	95	6	June	
3	11	96	7	July	
3	12	58	8	August	
			9	September	
			10	October	
			11	November	
			12	December	

**FIGURE 13.2** MONTHLY\_SALES and MONTH tables

Sections 13.5.2–13.5.4 illustrate the use of the ranking functions, window functions, and the first and last functions. The reader is encouraged to refer to Price (2004) and Rischert (2010) for a more in-depth discussion of analytical function types.

### 13.5.2 The RANK () and DENSE\_RANK () Functions

The RANK () and DENSE\_RANK () functions are used to rank items in a group. The difference between RANK () and DENSE\_RANK () is in the way they handle items that tie. As illustrated next, RANK () leaves a gap in the sequence when there is a tie, but DENSE\_RANK () leaves no gap. The SELECT statement in Example 1 illustrates the use of RANK () and DENSE\_RANK () to get the rankings of total monthly units sold by product for five products. Note the use of the keyword OVER in the syntax when calling the RANK () and DENSE\_RANK () functions. Because the total units sold for products 4 and 5 are the same, RANK () and DENSE\_RANK () return slightly different ranks, with the RANK () function leaving a gap in the sequence.

#### Example 1

```
SELECT PRODUCT, SUM(UNITS_SOLD),
RANK() OVER (ORDER BY SUM(UNITS_SOLD) DESC) RANK,
DENSE_RANK() OVER (ORDER BY SUM(UNITS_SOLD) DESC) DENSE_RANK
FROM MONTHLY_SALES
GROUP BY PRODUCT
ORDER BY RANK;
```

684

PRODUCT	SUM(UNITS_SOLD)	RANK	DENSE_RANK
3	946	1	1
1	903	2	2
4	878	3	3
5	878	3	3
2	810	5	4

5 rows selected.

Since there are five products in the MONTHLY\_SALES table, there are five groups to rank. Twelve “monthly units sold” amounts are added together to produce the five sums.

The PARTITION BY clause is used when there is a need to divide the groups into subgroups. If you need to subdivide the “monthly units sold” by product, so that you can obtain which months generated the best sales for each product, you can use the PARTITION BY clause, as shown in Example 2.

### Example 2

```
SELECT PRODUCT, MNAME, UNITS_SOLD,
RANK() OVER (PARTITION BY PRODUCT ORDER BY UNITS_SOLD DESC) RANK
FROM MONTHLY_SALES JOIN MONTH
ON MONTH.MNUM = MONTHLY_SALES.MNUM
ORDER BY PRODUCT, RANK;
```

PRODUCT	MNAME	UNITS_SOLD	RANK
1	April	98	1
1	January	91	2
1	May	91	2
1	July	88	4
1	November	86	5
1	December	81	6
1	August	74	7
1	June	63	8
1	February	61	9
1	September	59	10
1	October	58	11
1	March	53	12
2	February	98	1
2	April	93	2
2	March	85	3
2	September	69	4
2	December	67	5
2	October	66	6
2	August	61	7
2	July	57	8
2	November	56	9
2	June	55	10
2	January	52	11
2	May	51	12
3	August	97	1
3	November	96	2
3	February	96	2
3	October	95	4
3	May	81	5
3	June	81	5
3	September	79	7

PRODUCT	MNAME	UNITS SOLD	RANK
3	January	77	8
3	July	73	9
3	April	59	10
3	December	58	11
3	March	54	12
4	November	96	1
4	August	93	2
4	March	90	3
4	June	83	4
4	December	83	4
4	February	76	6
4	January	65	7
4	September	65	7
4	April	62	9
4	October	59	10
4	July	53	11
4	May	53	11
5	November	93	1
5	January	86	2
5	April	84	3
5	February	83	4
5	October	78	5
5	December	74	6
5	August	71	7
5	July	69	8
5	March	68	9
5	June	62	10
5	May	58	11
5	September	52	12

60 rows selected.

The ranks shown in Example 2 reveal that the number of units sold for certain products were the same for multiple months and that the month of November was the best selling month for products 4 and 5. In fact, as shown in Example 3, the month of November ranked first in terms of total units sold for the five products.

### Example 3

```
SELECT MNAME, SUM(UNITS SOLD),
RANK() OVER (ORDER BY SUM(UNITS SOLD) DESC) RANK
FROM MONTHLY_SALES JOIN MONTH
ON MONTH.MNUM = MONTHLY_SALES.MNUM
GROUP BY MNAME
ORDER BY RANK;
```

MNAME	SUM(UNITS_SOLD)	RANK
November	427	1
February	414	2
August	396	3
April	396	3
January	371	5
December	363	6
October	356	7
March	350	8
June	344	9
July	340	10
May	334	11
September	324	12

12 rows selected.

### 13.5.3 Using ROLLUP, CUBE, and GROUPING SETS Operators with Analytical Functions

687

You can use the ROLLUP, CUBE, and GROUPING SETS operators with the analytical functions. The SELECT statement in Example 4 uses ROLLUP and RANK () to get the rankings of total units sold by product. Note that the ROLLUP clause is responsible for the first row.

#### Example 4

```
SELECT PRODUCT, SUM(UNITS_SOLD),
RANK() OVER (ORDER BY SUM(UNITS_SOLD) DESC) RANK
FROM MONTHLY_SALES
GROUP BY ROLLUP (PRODUCT)
ORDER BY RANK;
```

PRODUCT	SUM(UNITS_SOLD)	RANK
	4415	1
3	946	2
1	903	3
4	878	4
5	878	4
2	810	6

6 rows selected.

The SELECT statement in Example 5 uses GROUPING SETS and RANK () to get just the sales amount subtotal rankings. Since there are five products and 12 months, 17 rows of output are generated.

**Example 5**

```
COLUMN RANK NOPRINT11
SELECT PRODUCT, MNAME, SUM(UNITS_SOLD) ,
RANK() OVER (ORDER BY SUM(UNITS_SOLD) DESC) RANK
FROM MONTHLY_SALES JOIN MONTH
ON MONTH.MNUM = MONTHLY_SALES.MNUM
GROUP BY GROUPING SETS (PRODUCT, MNAME)
ORDER BY RANK;
```

PRODUCT	MNAME	SUM(UNITS_SOLD)
3		946
1		903
4		878
5		878
2		810
	November	427
	February	414
	August	396
	April	396
	January	371
	December	363
	October	356
	March	350
	June	344
	July	340
	May	334
	September	324

688

17 rows selected.

### 13.5.4 Using the Window Functions

Window functions are used to calculate things such as cumulative sums and moving averages<sup>12</sup> within (a) a specified range of rows, (b) a range of values, or (c) an interval of time. The term “window” is used because processing of results involves a sliding range of rows returned by a query. A window has a defined starting point and ending point. For example, a window that defines a cumulative sum starts with the first row and then slides

---

<sup>11</sup>Different database products have a number of supporting commands that control the display attributes for a single column or all columns. Oracle 10g's SQL\*Plus environment allows the command COLUMN RANK NOPRINT prior to the SELECT statement to suppress the display of the column headed by the alias RANK, leaving only the PRODUCT, MNAME, and SUM(UNITS\_SOLD) columns displayed.

<sup>12</sup>A moving average is used to analyze a series of data points by creating a series of averages of different subsets of the full data set. Moving averages are typically used with time-series data.

forward with each subsequent row. A moving average has sliding starting and ending rows for a constant logical or physical range.

You can use windows with the following functions: SUM(), AVG(), MAX(), MIN(), COUNT(), VARIANCE(), and STDEV(). You can also use windows with FIRST\_VALUE() and LAST\_VALUE(), which return the first and last values in a window.

The SELECT statement in Example 6 performs a cumulative sum to compute the cumulative sales amount, starting with January and ending in December. Note how each monthly sales amount is added to the cumulative amount that grows after each month.

### Example 6

```
COLUMN MONTH.MNUM NOPRINT13
SELECT MONTH.MNUM, MNAME, SUM(UNITS_SOLD), SUM(SUM(UNITS_SOLD))
OVER (ORDER BY MONTH.MNUM ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
"CUMULATIVE SALES"
FROM MONTHLY_SALES JOIN MONTH
ON MONTHLY_SALES.MNUM = MONTH.MNUM
GROUP BY MONTH.MNUM, MNAME
ORDER BY MONTH.MNUM;
```

689

MNAME	SUM(UNITS_SOLD)	CUMULATIVE SALES
January	371	371
February	414	785
March	350	1135
April	396	1531
May	334	1865
June	344	2209
July	340	2549
August	396	2945
September	324	3269
October	356	3625
November	427	4052
December	363	4415

12 rows selected.

Notice the highlighted expression utilized in the query:

```
SUM(SUM(UNITS_SOLD)) OVER (ORDER BY MONTH.MNUM ROWS BETWEEN UNBOUNDED PRECEDING AND
CURRENT ROW)
```

In this expression:

- **SUM(UNITS\_SOLD)** computes the sum of the units sold for a given month. The **outer SUM()** computes the cumulative amount.

---

<sup>13</sup>While the MNUM column is needed to join the MONTH and MONTHLY\_SALES tables, it is not necessary to display both the month number and month name.

- **ORDER BY MONTH.MNUM** orders the rows read by the query by month.
- **ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW** defines the starting and ending points of the window. The starting point includes all rows read by the query, as indicated by unbounded preceding; the ending point of the window is the current row. Current row is actually the default, and thus the window size could have been implicitly specified using ROWS UNBOUNDED PRECEDING and the results of the query would be the same.

Thus the entire expression computes the cumulative sum of the amount for each month starting at the first row read by the query (i.e., the units sold for the month of January). Each row in the window is processed one at a time, starting with the first row in the window (i.e., the row for January). As each row is processed, the current row's amount is added to the cumulative amount and the end of the window moves down to the next row. Processing continues until the last row read by the query is processed.

The SELECT statement in Example 7 uses a cumulative sum to compute the cumulative sales amount, starting with April and ending in September. Note the use of just rows unbounded preceding to define the starting point of the window implicitly indicates the end of the window to be the current row.

690

### Example 7

```
SELECT MONTH.MNUM, MNAME, SUM(UNITS_SOLD), SUM(SUM(UNITS_SOLD))
OVER (ORDER BY MONTH.MNUM ROWS UNBOUNDED PRECEDING)
"CUMULATIVE SALES"
FROM MONTHLY_SALES JOIN MONTH
ON MONTHLY_SALES.MNUM = MONTH.MNUM
AND MONTH.MNUM BETWEEN 4 AND 9
GROUP BY MONTH.MNUM, MNAME
ORDER BY MONTH.MNUM;
```

MNUM	MNAME	SUM(UNITS_SOLD)	CUMULATIVE SALES
4	April	396	396
5	May	334	730
6	June	344	1074
7	July	340	1414
8	August	396	1810
9	September	324	2134

6 rows selected.

The query in Example 8 computes the moving average of the sales amount between (i.e., involving) the current month and the previous three months.

**Example 8**

```
SELECT MONTH.MNUM, MNAME, SUM(UNITS_SOLD), AVG(SUM(UNITS_SOLD))
OVER (ORDER BY MONTH.MNUM ROWS BETWEEN 3 PRECEDING AND CURRENT ROW)
"MOVING AVERAGE"
FROM MONTHLY_SALES JOIN MONTH
ON MONTHLY_SALES.MNUM = MONTH.MNUM
GROUP BY MONTH.MNUM, MNAME
ORDER BY MONTH.MNUM;
```

MNAME	SUM(UNITS_SOLD)	MOVING AVERAGE
January	371	371
February	414	392.5
March	350	378.333333
April	396	382.75
May	334	373.5
June	344	356
July	340	353.5
August	396	353.5
September	324	351
October	356	354
November	427	375.75
December	363	367.5

12 rows selected.

Notice the highlighted expression used to compute the moving average:

```
AVG(SUM(UNITS_SOLD)) OVER (ORDER BY MONTH.MNUM ROWS BETWEEN 3 PRECEDING AND
CURRENT ROW)
```

In this expression:

- **SUM(UNITS\_SOLD)** computes the sum of the units sold for a given group of months. The **outer AVG()** computes the average.
- **ORDER BY MONTH.MNUM** orders the rows read by the query by month.
- **ROWS BETWEEN 3 PRECEDING AND CURRENT ROW** defines the starting point of the window as including the three rows preceding the current row; the ending point of the window is the current row. Since current row is the default, **ROWS 3 PRECEDING** would have accomplished the same thing.

The entire expression means compute the moving average of the sales amount between the current month and the previous three months. Because, for the first two months, less than the full three months of data are available, the moving average is based on only the months available. Once the month of April is reached, each moving average is based on four months of data.

Both the starting point and the ending point of the window begin at row 1 read by the query. The ending point of the window moves down after each row is processed. The starting point of the window only moves down after row 4 has been processed, after which time the starting point of the window moves down after each row is processed. Processing continues until the last row read by the query is processed.

**FIRST\_VALUE()** and **LAST\_VALUE()** are functions used to get the first and last rows in a window. The **SELECT** statement in Example 9 uses **FIRST\_VALUE()** and **LAST\_VALUE()** to get the previous and next month's sales amount.

### Example 9

```
SELECT MONTH.MNUM, MNAME, SUM(UNITS_SOLD) ,
FIRST_VALUE(SUM(UNITS_SOLD)) OVER (ORDER BY MONTH.MNUM
ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) "PREVIOUS AMOUNT",
LAST_VALUE(SUM(UNITS_SOLD)) OVER (ORDER BY MONTH.MNUM
ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) "NEXT AMOUNT"
FROM MONTHLY_SALES JOIN MONTH
ON MONTHLY_SALES.MNUM = MONTH.MNUM
GROUP BY MONTH.MNUM, MNAME
ORDER BY MONTH.MNUM;
```

692

MNAME	SUM(UNITS_SOLD)	PREVIOUS AMOUNT	NEXT AMOUNT
January	371	371	414
February	414	371	350
March	350	414	396
April	396	350	334
May	334	396	344
June	344	334	340
July	340	344	396
August	396	340	324
September	324	396	356
October	356	324	427
November	427	356	363
December	363	427	363

12 rows selected.

## 13.6 A QUICK LOOK AT THE MODEL CLAUSE

The MODEL clause is an extension to the SQL Select Statement and makes it possible to treat relational data as multidimensional arrays and to define formulas on the arrays. As such, the MODEL clause performs inter-row calculations by allowing you to access a column in a row like a cell in an array. The MODEL clause is a powerful feature that allows you to change any cell in the query's result set using data from any other cell (similar to the way a spreadsheet works). This permits you to perform calculations in a

similar manner to spreadsheet calculations. For example, the MONTHLY\_SALES table contains sales information for the months in 2013. The MODEL clause can be used to calculate sales in future months based on sales in 2013.

### 13.6.1 MODEL Clause Concepts

The MODEL clause defines a multidimensional array by mapping the columns of a query into three groups: partition, dimension, and measure columns. These elements perform the following tasks:

- Partitions define logical blocks of the result set in a way similar to the partitions of the analytical functions. Each partition is viewed by the formulas as an independent array.
- Dimensions identify each measure cell within a partition. These columns are identifying characteristics such as month, year, and product.
- Measures are numeric values, such as units sold. Each cell is accessed within its partition by specifying its full combination of dimensions.

Figure 13.3 gives a conceptual overview of a Model based on the data in Figure 13.2 for product 1. Three parts are shown in the figure. The top segment shows the concept of dividing the table into partitioning, dimension, and measure columns. The middle segment shows rules that calculate the units sold for product 1 for months 1 and 2 of year 2014. The third part shows the output of a query applying the rules to the data. The unshaded output is data retrieved from the database, while the shaded output shows rows calculated from the rules.

It is important to note that the MODEL clause does not update existing data in tables, nor does it insert new data into tables. In order to change values in a table, the MODEL results must be supplied to an INSERT or UPDATE or MERGE<sup>14</sup> statement.

### 13.6.2 Basic Syntax of the MODEL Clause

The basic elements of the MODEL clause syntax follow. While there are some other elements of the syntax, those shown here represent the core functionality.

```
<prior clauses of SELECT statement>
MODEL [main]
[reference models]
[PARTITION BY (<cols>)]
[DIMENSION BY (<cols>)]
[MEASURES (<cols>)]
[RULES
(  <cell_assignment> = <expression> ...)
```

---

<sup>14</sup>The MERGE statement allows you to merge rows from one table into another. The MERGE statement is not covered in this book.

Mapping of columns to model entities:			
PRODUCT	YEAR	MONTH	UNITS_SOLD
Partition	Dimension	Dimension	Measure
Rules:			
$\text{UNITS\_SOLD [1, 2014]} = \text{UNITS\_SOLD [1, 2013]}$ $\text{UNITS\_SOLD [2, 2014]} = \text{UNITS\_SOLD [2, 2013]} + \text{UNITS\_SOLD [3, 2013]}$			
Output of MODEL Clause:			
PRODUCT	YEAR	MONTH	UNITS_SOLD
Partition	Dimension	Dimension	Measure
1	2013	1	91
1	2013	2	61
1	2013	3	53
1	2013	4	98
1	2013	5	91
1	2013	6	63
1	2013	7	88
1	2013	8	74
1	2013	9	59
1	2013	10	58
1	2013	11	86
1	2013	12	81
1	2014	1	91
1	2014	2	114

**FIGURE 13.3** Segments of a MODEL

### 13.6.3 An Example of the MODEL Clause

The MODEL clause must be preceded by an SQL SELECT statement. All examples in this section are based on the following query. This query serves as the basis on which the elements in the MODEL clause are executed.

```
SELECT PRODUCT, YEAR, MNUM, UNITS_SOLD
FROM MONTHLY_SALES
WHERE PRODUCT = 1;
```

PRODUCT	YEAR	MNUM	UNITS_SOLD
1	2013	1	91
1	2013	2	61
1	2013	3	53
1	2013	4	98
1	2013	5	91
1	2013	6	63
1	2013	7	88
1	2013	8	74
1	2013	9	59
1	2013	10	58
1	2013	11	86
1	2013	12	81

12 rows selected.

The SELECT statement in Example 1 retrieves the sales amount for each month in 2013 for product 1 and computes the predicted sales for January, February, and March of 2014, based on sales in 2013. It is used as a vehicle by which to illustrate the basic elements of the MODEL clause syntax.

### Example 1

```
SELECT PRODUCT, YEAR, MNUM, UNITS_SOLD
FROM MONTHLY_SALES
WHERE PRODUCT = 1
MODEL
PARTITION BY (PRODUCT)
DIMENSION BY (MNUM, YEAR)
MEASURES (UNITS_SOLD)
(UNITS_SOLD [1, 2014] = UNITS_SOLD [1, 2013],
UNITS_SOLD [2, 2014] = UNITS_SOLD [2, 2013] + UNITS_SOLD [3, 2013],
UNITS_SOLD [3, 2014] = ROUND (UNITS_SOLD [3, 2013] * 1.25, 2))
ORDER BY YEAR, MNUM;
```

Breaking the query down:

- **PARTITION BY (PRODUCT)** specifies that the results are partitioned by **PRODUCT**.
- **DIMENSION BY (MNUM, YEAR)** specifies that the dimensions of the array are month number and year. This means that you access a column in a row by supplying the month number and year.
- **MEASURES (UNITS\_SOLD)** specifies that each cell in the array contains the number of units sold and the array name is **UNITS\_SOLD**. To access the cell in the **UNITS\_SOLD** array for January 2013, you use **UNITS\_SOLD [1, 2013]**, which returns a number of units sold amount.

- After **MEASURES** come three lines that compute the future sales for January, February, and March of 2014. The following three lines constitute the rules of the model:
  - **UNITS\_SOLD [1, 2014] = UNITS\_SOLD [1, 2013]** sets the sales amount for January 2014 to the amount for January 2013.
  - **UNITS\_SOLD [2, 2014] = UNITS\_SOLD [2, 2013] + UNITS\_SOLD [3, 2013]** sets the sales amount for February 2014 to the amount for February 2013 plus March 2013.
  - **UNITS\_SOLD [3, 2014] = ROUND (UNITS\_SOLD [3, 2013] \* 1.25, 2)** sets the sales amount for March 2014 to the rounded value of the sales amount for March 2013 multiplied by 1.25.
- **ORDER BY prd\_type\_id, year, month** orders the results returned by the entire query.

The output of the query in Example 1 is shown next. Note that the results contain the units sold for all months in 2013 for product 1 plus the predicted number of units sold for the first three months of 2014.

696

PRODUCT	YEAR	MNUM	UNITS_SOLD
1	2013	1	91
1	2013	2	61
1	2013	3	53
1	2013	4	98
1	2013	5	91
1	2013	6	63
1	2013	7	88
1	2013	8	74
1	2013	9	59
1	2013	10	58
1	2013	11	86
1	2013	12	81
1	2014	1	91
1	2014	2	114
1	2014	3	66.25

### 13.6.3.1 Accessing a Range of Cells Using BETWEEN and AND

You can access a range of cells on the right side of a formula using the BETWEEN and AND keywords. The SELECT statement in Example 2 sets the number of units sold for January 2014 to the rounded average of the number of units sold between January and March 2013 (see shaded lines).

**Example 2**

```

SELECT YEAR, MNUM, UNITS_SOLD
FROM MONTHLY_SALES
WHERE PRODUCT = 1
MODEL
DIMENSION BY (MNUM, YEAR)
MEASURES (UNITS_SOLD)
(UNITS_SOLD [1, 2014] = ROUND (AVG(UNITS_SOLD) [MNUM BETWEEN 1 AND 3, 2013], 2))
ORDER BY YEAR, MNUM;

```

YEAR	MNUM	UNITS_SOLD
2013	1	91
2013	2	61
2013	3	53
2013	4	98
2013	5	91
2013	6	63
2013	7	88
2013	8	74
2013	9	59
2013	10	58
2013	11	86
2013	12	81
2014	1	68.33

13 rows selected.

697

**13.6.3.2 Accessing All Cells Using ANY and IS ANY**

You can access all cells using the ANY and IS ANY predicates. You use ANY with positional notation and IS ANY with symbolic notation (see shaded line). The SELECT statement in Example 3 sets the sales amount for January 2014 to the rounded sum of the sales for all months and years.

**Example 3**

```

SELECT YEAR, MNUM, UNITS_SOLD
FROM MONTHLY_SALES
WHERE PRODUCT = 1
MODEL
DIMENSION BY (MNUM, YEAR)
MEASURES (UNITS_SOLD)
(UNITS_SOLD [1, 2014] = ROUND (AVG(UNITS_SOLD) [ANY, YEAR IS ANY], 2))
ORDER BY YEAR, MNUM;
/

```

YEAR	MNUM	UNITS_SOLD
2013	1	91
2013	2	61
2013	3	53
2013	4	98
2013	5	91
2013	6	63
2013	7	88
2013	8	74
2013	9	59
2013	10	58
2013	11	86
2013	12	81
2014	1	75.25

13 rows selected.

698

### 13.6.3.3 Getting the Current Value of a Dimension Using CURRENTV()

The CURRENTV() function provides the current value of a dimension. This function is used in the SELECT statement given in Example 4 to set the sales amount for the first month of 2014 to 1.25 times the sales of the same month in 2013. Observe the use of CURRENTV() on the shaded line to get the current month.

#### Example 4

```
SELECT PRODUCT, YEAR, MNUM, UNITS_SOLD
FROM MONTHLY_SALES
WHERE PRODUCT = 1
MODEL
PARTITION BY (PRODUCT)
DIMENSION BY (MNUM, YEAR)
MEASURES (UNITS_SOLD)
(UNITS_SOLD [1, 2014] = ROUND(UNITS_SOLD [CURRENTV(), 2013] * 1.25,2),
UNITS_SOLD [2, 2014] = UNITS_SOLD [2, 2013] + UNITS_SOLD [3, 2013],
UNITS_SOLD [3, 2014] = ROUND (UNITS_SOLD [3, 2013] * 1.25, 2))
ORDER BY YEAR, MNUM;
```

PRODUCT	YEAR	MNUM	UNITS_SOLD
1	2013	1	91
1	2013	2	61
1	2013	3	53
1	2013	4	98
1	2013	5	91
1	2013	6	63
1	2013	7	88
1	2013	8	74

```

1 2013      9      59
1 2013     10      58
1 2013     11      86
1 2013     12      81
1 2014      1 113.75
1 2014      2      114
1 2014      3      66.25

```

15 rows selected.

### 13.6.3.4 Accessing Cells Using a FOR Loop

You can access cells using a FOR loop. The SELECT statement in Example 5 sets the sales amount for the first three months of 2014 to 1.25 times the sales of the same months in 2013. Note the use of the FOR loop and the INCREMENT keyword that specifies the amount to increment month by during each iteration of the loop (see shaded line).

#### Example 5

```

SELECT PRODUCT, YEAR, MNUM, UNITS_SOLD
FROM MONTHLY_SALES
WHERE PRODUCT = 1
MODEL
PARTITION BY (PRODUCT)
DIMENSION BY (MNUM, YEAR)
MEASURES (UNITS_SOLD)
(UNITS_SOLD [FOR MNUM FROM 1 TO 3 INCREMENT 1, 2014] = ROUND(UNITS_SOLD [CURRENTV(),
2013] * 1.25,2))
ORDER BY YEAR, MNUM;

```

699

PRODUCT	YEAR	MNUM	UNITS_SOLD
1	2013	1	91
1	2013	2	61
1	2013	3	53
1	2013	4	98
1	2013	5	91
1	2013	6	63
1	2013	7	88
1	2013	8	74
1	2013	9	59
1	2013	10	58
1	2013	11	86
1	2013	12	81
1	2014	1	113.75
1	2014	2	76.25
1	2014	3	66.25

15 rows selected.

## 13.7 A POTPOURRI OF OTHER SQL QUERIES

Date (1995) has described SQL as an *extremely redundant* language in the sense that there is almost always a variety of ways to formulate the same query in SQL. The advantage of this flexibility is that users have the option of choosing the approach with which they are most comfortable. For example, a join can be used as an alternative way of expressing many subqueries. In addition, as we have seen in Section 12.3 of Chapter 12, subqueries and join conditions may appear in the FROM clause or even in the column list of the SELECT clause.

The declarative, nonprocedural nature of SQL, which allows the user to specify what the intended results of the query are, rather than specifying the details of how the result should be obtained, can also be a disadvantage. Ideally, the user should be concerned only with specifying the query correctly. The DBMS, on the other hand, should then take the query and execute it efficiently. While DBMS products all make use of a variety of techniques to process, optimize, and execute queries written in SQL, in practice it helps if the user is aware of which types of constructs in a query are more expensive to process in terms of performance than others. See Chapter 15 of Elmasri and Navathe (2007) for an interesting discussion of query processing and optimization.

700

The remainder of this section contains six examples of SQL queries that combine a number of the features introduced in Chapter 12 as well as in this chapter. Their purpose is not so much to specify the most efficient way of expressing the query as to illustrate the application of several SQL features and functions. Since each example makes use of SQL running on an Oracle 10g platform, readers working with some other database platform may wish to refer to their SQL reference material to resolve any syntactical differences created by syntax unique to Oracle 10g, such as differences in function names.

### 13.7.1 Concluding Example 1

*Suppose Madeira College is interested in finding out how many professors have been hired each month. A review of the data in the PROFESSOR table identifies two interesting problems. First, there have been months during which no professors have been hired (i.e., February, March, April, and July). Second, there are two professors (i.e., John B. Smith and Jeanine Troy) whose hire date is unavailable (i.e., unknown). The following SQL SELECT statement represents one way to count the number of professors hired during each month and also includes (a) one row for each month during which no professor has been hired, and (b) a row that records the number of professors for which a date hired is unavailable. The individual lines have been numbered to facilitate the discussion.*

#### SQL Select Statement:

```

1 SELECT MONTHS .MNUM,
2 NVL (TO _CHAR (PROFESSOR .DATEHIRED , 'Month' ) ,MONTHS .MNAME) AS "Month",
3 COUNT (PROFESSOR .DATEHIRED) AS "Number Hired"
4 FROM MONTHS LEFT OUTER JOIN PROFESSOR

```

```

5 ON MONTHS.MNUM = TO_CHAR(PROFESSOR.DATEHIRED, 'mm')
6 GROUP BY MONTHS.MNUM,
7 NVL(TO_CHAR(PROFESSOR.DATEHIRED, 'Month'), MONTHS.MNAME)
8 UNION
9 SELECT '13', 'Unknown', COUNT(*)
10 FROM PROFESSOR
11 WHERE PROFESSOR.DATEHIRED IS NULL
12 GROUP BY '13', 'Unknown';

```

1. Since the months February, March, April, and July do not appear in the PROFESSOR table, the MONTHS Table was created using the following CREATE TABLE statement:  
**CREATE TABLE MONTHS (MNUM VARCHAR(2), MNAME VARCHAR(10));**  
and a row containing a two-character month number (e.g., 01 for January, ..., 12 for December) and the full name of the month inserted for each of the 12 months of the year.
2. Lines 6 and 7 indicate that grouping is first done by month number and then by month name. Grouping by month name alone would cause the months to be displayed in alphabetical (i.e., April, August, December, etc.) as opposed to chronological (i.e., January, February, March, etc.) order.
3. Lines 2 and 7 make use of the null value function NVL (expression1, expression2) as a way to replace a null value with a string in the results of a query. If expression1 is a null value, the NVL function returns expression2. If expression1 is not null, the NVL function returns expression1.
4. Line 4 indicates a left outer join involving the MONTHS and PROFESSOR tables. This allows an unmatched month number in the MONTHS table to be combined with the row of null values in the PROFESSOR table as a result of the left outer join. Use of the NVL function in lines 2 and 7 is required so that the name of the month in the MONTHS table can replace the null value associated with the **PROFESSOR.DATEHIRED** on the row of null values in the **PROFESSOR** table.
5. COUNT(PROFESSOR.DATEHIRED) is used on line 3 so that the null value in the **PROFESSOR.DATEHIRED** column on rows associated with months during which no professor was hired will allow the initial value of the accumulator to remain at zero. If COUNT(PROFESSOR.DATEHIRED) is replaced by COUNT(\*), the number of professors hired during each of the months of February, March, April, and July will be 1.
6. Execution of the query on lines 1–7 will not include a row that records the number of professors for which a date hired is unavailable. However a query that contains the union of the SELECT statement on lines 1–7 with the SELECT statement on lines 9–13 will include a final row that shows the number of professors for which a date hired is unavailable (i.e., unknown).

7. Execution of the query that appears above generates the following result:

**Result:**

MNUM	Month	Number Hired
01	January	2
02	February	0
03	March	0
04	April	0
05	May	5
06	June	5
07	July	0
08	August	2
09	September	1
10	October	1
11	November	1
12	December	1
13	Unknown	2

702

Different database products have a number of supporting commands that control the display attributes for a single column or all columns. For example, in Oracle 10g's SQL\*Plus environment, adding the command:

**COLUMN MNUM NOPRINT**

before line 1 would suppress the display of the column headed Mnum and leave only the Month and Number Hired columns displayed.

### 13.7.2 Concluding Example 2

*Based on just the data in the TAKES table, display the number of grade points, classes taken, and grade point average for the classes taken for each junior. Include not only those juniors who have taken classes but also those juniors who have not taken any classes. In addition, display the student id and name of each junior. The following SQL SELECT statement represents one way to express this information requirement. The individual lines have been numbered to facilitate the discussion.*

**SQL Select Statement:**

```

1  SELECT STUDENT.SID, STUDENT.NAME,
2  SUM (DECODE (RTRIM(TAKES.GRADE) , 'A' , 4 , 'B' , 3 , 'C' , 2 , 'D' , 1 , 'F' , 0))
     "Grade Points",
3  COUNT (TAKES.SID) "Classes Taken",
4  SUM (DECODE (RTRIM(TAKES.GRADE) , 'A' , 4 , 'B' , 3 , 'C' , 2 , 'D' , 1 , 'F' ,
0 )) /COUNT (TAKES.SID) "GPA"
5  FROM TAKES JOIN STUDENT ON STUDENT.SID = TAKES.SID
6  AND STUDENT.GRADELEVEL = 'JR'
7  GROUP BY STUDENT.SID, STUDENT.NAME
8  UNION
9  SELECT STUDENT.SID, STUDENT.NAME, 0 , 0 , 0
10 FROM STUDENT

```

```

11 WHERE STUDENT.GRADELEVEL = 'JR'
12 AND STUDENT.SID NOT IN
13     (SELECT TAKES.SID FROM TAKES)
14 ORDER BY 5 DESC, 3 DESC, 2 ASC;

```

- Line 2 makes use of the DECODE function introduced in Section 13.1.7 as the Oracle implementation of the SQL:2003 CASE expression (see Table 13.1). Recall that the DECODE function facilitates character-by-character substitution. For every value it sees in a field, DECODE checks for a match in a series of *if/then* tests. The format of the DECODE function is:

```
DECODE (expr, search1, result1, [search2, result2,] ... [default])
```

The purpose is to compare *expr* to each search value and to return the result if *expr* equals the *search* value. If no match is found, the DECODE function returns the *default* value. If a default is not supplied, the default will return null. In line 2, if **TAKES.GRADE** is equal to "A," the value returned to the DECODE function is 4; if **TAKES.GRADE** is equal to "B," the value returned to the DECODE function is 3, etc. Note that before the DECODE function is applied to **TAKES.GRADE**, the RTRIM function is applied. This is necessary because **TAKES.GRADE** is defined as a CHAR(5) data type, and thus removal of the trailing blanks is necessary before **TAKES.GRADE** is equal to any of the five possible letter grades. Had **TAKES.GRADE** been defined as a VARCHAR data type, use of the RTRIM function would have been unnecessary.

- Grouping by **STUDENT.SID** and **STUDENT.NAME** on line 7 is necessary to allow (a) both columns to be displayed as part of the result, and (b) number of grade points, number of classes taken, and grade point average of each student to be calculated.
- Execution of the SELECT statement on lines 1–7 only permits the results for those graduate students who have taken a course to be displayed. The SELECT statement on lines 9–13 allows the results for those graduate students who have never taken a course to be displayed.
- Note that the ORDER BY clause applies to the collective results from all SELECT statements involved in the query. The ORDER BY clause on line 14 causes the results to be displayed in descending order by the student's grade point average and classes taken, and in ascending order by undergraduate major.
- Execution of the query shown above produces the following result:

**Result:**

SID	NAME	Grade Points	Classes Taken	GPA
KS39874	Sweety Kramer	12	3	4
KP78924	Poppy Kramer	8	2	4
BE76598	Elijah Baley	6	2	3
OD76578	Daniel Olive	0	0	0
HT67657	Troy Hudson	0	0	0

### 13.7.3 Concluding Example 3

*Display all professors whose salary exceeds that of their department head. For each qualifying professor, display his or her name and salary along with the name and salary of the department head. The following query represents one way to satisfy this information request. The individual lines have been numbered to facilitate the discussion.*

#### SQL Select Statement:

```

1  SELECT A.NAME AS "Prof Name", A.SALARY AS "Prof Salary",
2  DEPARTMENT.NAME "Dept Name",
3  B.NAME AS "Dept Head", B.SALARY AS "Dept Head Salary",
4  DEPARTMENT.NAME "Dept Name"
5  FROM (PROFESSOR A JOIN DEPARTMENT
6  ON A.DCODE = DEPARTMENT.DCODE)
7  JOIN PROFESSOR B ON DEPARTMENT.HODID = B.EMPID
8  AND A.SALARY > B.SALARY;

```

1. Since both the salary of a professor and the salary of the professor's department head are recorded in the PROFESSOR table, this query references the PROFESSOR table twice (once using table alias A and once using table alias B) in order to compare the salary of a professor with that of his or her department head.
2. The ON clauses that appear on lines 4 and 5 work together to identify the professor in copy B of PROFESSOR serving as department head of the professor under investigation in copy A of PROFESSOR. The condition on line 6 ensures that the salary of the professor exceeds that of his or her department head.
3. Execution of the query yields the result shown here:

#### Result:

Prof Name	Prof Salary	Dept Name	Dept Head	Dept Head Salary	Dept Name
Chelsea Bush	77000	QA/QM	Alan Brodie	76000	QA/QM
Tony Hopkins	77000	QA/QM	Alan Brodie	76000	QA/QM
Sunil Shetty	64000	IS	Cathy Cobal	45000	IS
Katie Shef	65000	IS	Cathy Cobal	45000	IS

### 13.7.4 Concluding Example 4

*Display the names of all professors who have the same first name as at least one other professor.*

#### SQL Select Statement:

```

SELECT PROFESSOR.NAME FROM PROFESSOR
WHERE SUBSTR(PROFESSOR.NAME, 1, INSTR(PROFESSOR.NAME, ' ') - 1) IN
  (SELECT SUBSTR(PROFESSOR.NAME, 1, INSTR(PROFESSOR.NAME, ' ') - 1)
   FROM PROFESSOR
   GROUP BY SUBSTR(PROFESSOR.NAME, 1, INSTR(PROFESSOR.NAME, ' ') - 1)
   HAVING COUNT(*) > 1);

```

Since the PROFESSOR table does not have separate columns for first name, last name, and middle initial, the subquery makes use of the SUBSTR and INSTR functions along with the GROUP BY and HAVING clauses to retrieve the first names that appear more than one time in the PROFESSOR table. The main query, when executed, follows by displaying the names of all professors whose first name appears in the set of first names retrieved by the subquery. Execution of this query produces the following result. It is left as an exercise for the reader to order the result in ascending order by last name.

**Result:**

```
NAME
-----
John Smith
Mike Faraday
John B Smith
John Nicholson
Mike Crick
```

### 13.7.5 Concluding Example 5

*Display the names of those department chairs who do not teach a section of a course.*

705

**SQL Select Statement:**

```
SELECT PROFESSOR.NAME FROM PROFESSOR
WHERE PROFESSOR.EMPID IN
(SELECT DEPARTMENT.HODID FROM DEPARTMENT
WHERE NOT EXISTS
(SELECT * FROM SECTION
WHERE DEPARTMENT.HODID = SECTION.PROFID));
```

Although this query involves three tables (PROFESSOR, DEPARTMENT, and SECTION), it is rather straightforward. The two subqueries of the main query identify the department heads who do not teach a section and provide input to the main query that displays their names.

**Result:**

```
NAME
-----
Marie Curie
Mike Faraday
Mike Crick
Alan Brodie
```

## Chapter Summary

---

In addition to the aggregate functions introduced in Chapter 12, the SQL:2003 standard contains a number of built-in functions for working with strings, dates, and times. Although the functionality associated with these functions is available in practically all database products, different products contain slightly different syntax from that specified in the standard (e.g., Oracle's SUBSTR function versus the SUBSTRING function in the SQL:2003 standard). While such variations in syntax limit the portability of SQL SELECT statements, it is hoped that, after studying Chapters 10 through 13, readers will be able to use the features discussed here and, where necessary, adapt them to their specific database platform with a minimum of difficulty.

Hierarchical queries can be used to retrieve records from a table by their natural relationship, be it a family tree, an employee/supervisor tree, or a bill of materials. Hierarchical queries incorporate key words such as START WITH, CONNECT BY, and PRIOR into the Select Statement. The START WITH clause defines the root rows of the hierarchy; the CONNECT BY clause explains the relationship between the parent and child; and the key word PRIOR, when prefixing a column name, is used to indicate whether the hierarchy goes from parent to child (i.e., top to bottom) or from child to parent (bottom to top).

SQL:2003 contains the following enhancements to the GROUP BY clause: (a) ROLLUP and CUBE extensions to the GROUP BY clause, (b) three GROUPING functions, and (c) the GROUPING SETS expression ROLLUP, which enables a SELECT statement to create subtotals that roll up from the most detailed level to a grand total, following a grouping list of columns specified in the ROLLUP clause. CUBE is an extension of ROLLUP, taking a specified set of grouping columns and creating subtotals for all of their possible combinations. GROUPING SETS is an expression that allows only a selectively specified set of groups to be displayed. The three grouping functions allow you to determine (a) which rows are subtotals and (b) the exact level of aggregation for a given subtotal.

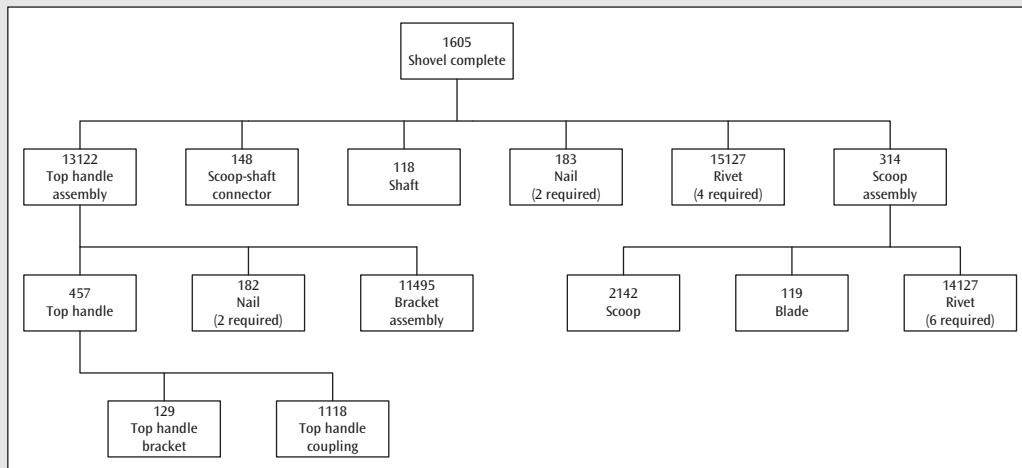
SQL:2003 introduces a new family of analytical functions. These functions make it possible to calculate (a) rankings and percentiles, (b) moving window aggregates, (c) lag/lead analysis, (d) first/last analysis, and (e) least squares regression. Query processing using analytical functions takes place in three steps. First, all joins, WHERE, GROUP BY, and HAVING clauses are carried out. Second, the query results are divided into groups of rows called partitions. A query result set may be partitioned into just one partition holding all rows, a few large partitions, or many small partitions holding just a few rows each. Third, if the query has an ORDER BY clause at its end, the ORDER BY is processed to allow for precise ordering of the results.

The SQL MODEL clause allows you to view query results in the form of multidimensional arrays and then apply formulas to calculate new array values. The formulas can be sophisticated interdependent calculations with inter-row and inter-array references. The MODEL clause defines a multidimensional array by mapping the columns of a query into three groups: partitioning, dimension, and measure columns. Partitions define logical blocks of the result set in a way similar to the partitions of the analytical functions. Each partition is viewed by the formulas as an independent array. Dimensions identify each measure cell within a partition. These columns are

identifying characteristics such as date, region, and product name. Measures contain numeric values such as sales. Each cell is accessed within its partition by specifying its full combination of dimensions.

## Exercises

1. Display the subassemblies of a snow shovel in the same top to bottom order as shown here:



707

2. Display all of the assemblies for which a snow shovel is a subassembly.
3. Display all of the assemblies for which a Bracket assembly is a subassembly.
4. Display all subassemblies of a snow shovel along with the number of required units of each. Indent each component in such a way as to appropriately depict the hierarchical structure.
5. Display the entire composition of all subassemblies of the Shovel complete that are either manufactured by Sears or use a component manufactured by Sears.
6. Display all subassemblies manufactured by Sears that require more than one unit.
7. Display all subassemblies and their components where the subassembly contains the words "Top Handle."

Exercises 8 thru 23 are based on monthly milk production (in millions of pounds of milk) for the northern, eastern, southern, and western regions of the United States for the years 2006 through 2010 shown in the following MILK\_PROD and REGIONS tables:

## Chapter 13

MILK\_PROD Table

YEAR	MONTH	POUNDS	REGION	YEAR	MONTH	POUNDS	REGION	YEAR	MONTH	POUNDS	REGION
----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----
2006	1	118	N	2009	10	124	N	2008	7	208	E
2006	2	118	N	2009	11	111	N	2008	8	224	E
2006	3	149	N	2009	12	121	N	2008	9	232	E
2006	4	177	N	2010	1	117	N	2008	10	261	E
2006	5	226	N	2010	2	118	N	2008	11	261	E
2006	6	215	N	2010	3	147	N	2008	12	243	E
2006	7	189	N	2010	4	166	N	2009	1	201	E
2006	8	172	N	2010	5	193	N	2009	2	199	E
2006	9	140	N	2010	6	195	N	2009	3	237	E
2006	10	143	N	2010	7	158	N	2009	4	246	E
2006	11	134	N	2010	8	159	N	2009	5	257	E
2006	12	143	N	2010	9	134	N	2009	6	228	E
2007	1	137	N	2010	10	126	N	2009	7	207	E
2007	2	124	N	2010	11	107	N	2009	8	230	E
2007	3	150	N	2010	12	110	N	2009	9	239	E
2007	4	181	N	2006	1	193	E	2009	10	258	E
2007	5	212	N	2006	2	196	E	2009	11	261	E
2007	6	208	N	2006	3	236	E	2009	12	235	E
2007	7	184	N	2006	4	255	E	2010	1	198	E
2007	8	175	N	2006	5	264	E	2010	2	190	E
2007	9	140	N	2006	6	237	E	2010	3	239	E
2007	10	130	N	2006	7	220	E	2010	4	241	E
2007	11	122	N	2006	8	234	E	2010	5	269	E
2007	12	134	N	2006	9	247	E	2010	6	243	E
2008	1	142	N	2006	10	271	E	2010	7	213	E
2008	2	142	N	2006	11	260	E	2010	8	247	E
2008	3	152	N	2006	12	251	E	2010	9	231	E
2008	4	163	N	2007	1	212	E	2010	10	268	E
2008	5	211	N	2007	2	210	E	2010	11	258	E
2008	6	204	N	2007	3	244	E	2010	12	261	E
2008	7	183	N	2007	4	256	E	2006	1	211	S
2008	8	172	N	2007	5	274	E	2006	2	210	S
2008	9	145	N	2007	6	251	E	2006	3	248	S
2008	10	132	N	2007	7	217	E	2006	4	265	S
2008	11	114	N	2007	8	224	E	2006	5	276	S
2008	12	127	N	2007	9	240	E	2006	6	247	S
2009	1	121	N	2007	10	266	E	2006	7	227	S
2009	2	121	N	2007	11	257	E	2006	8	238	S
2009	3	132	N	2007	12	239	E	2006	9	248	S
2009	4	150	N	2008	1	196	E	2006	10	265	S
2009	5	185	N	2008	2	189	E	2006	11	276	S
2009	6	183	N	2008	3	236	E	2006	12	262	S
2009	7	160	N	2008	4	242	E	2007	1	224	S
2009	8	150	N	2008	5	249	E	2007	2	215	S
2009	9	136	N	2008	6	237	E	2007	3	256	S

## Advanced Data Manipulation Using SQL

YEAR	MONTH	POUNDS	REGION	YEAR	MONTH	POUNDS	REGION	YEAR	MONTH	POUNDS	REGION
2007	4	272	S	2006	2	103	W	2009	10	175	W
2007	5	286	S	2006	3	120	W	2009	11	172	W
2007	6	266	S	2006	4	147	W	2009	12	150	W
2007	7	239	S	2006	5	167	W	2010	1	159	W
2007	8	261	S	2006	6	173	W	2010	2	158	W
2007	9	272	S	2006	7	152	W	2010	3	160	W
2007	10	296	S	2006	8	142	W	2010	4	183	W
2007	11	292	S	2006	9	116	W	2010	5	184	W
2007	12	285	S	2006	10	97	W	2010	6	189	W
2008	1	240	S	2006	11	85	W	2010	7	215	W
2008	2	231	S	2006	12	93	W	2010	8	208	W
2008	3	282	S	2007	1	86	W	2010	9	204	W
2008	4	282	S	2007	2	86	W	2010	10	183	W
2008	5	309	S	2007	3	96	W	2010	11	195	W
2008	6	289	S	2007	4	125	W	2010	12	193	W
2008	7	255	S	2007	5	147	W				
2008	8	273	S	2007	6	138	W				
2008	9	289	S	2007	7	138	W				REGIONS Table;
2008	10	309	S	2007	8	135	W				
2008	11	305	S	2007	9	108	W				REG_CODE REG_NAME
2008	12	290	S	2007	10	102	W				
2009	1	241	S	2007	11	95	W	N			Northern Region
2009	2	234	S	2007	12	110	W	E			Eastern Region
2009	3	278	S	2008	1	143	W	S			Southern Region
2009	4	294	S	2008	2	134	W	W			Western Region
2009	5	300	S	2008	3	127	W				
2009	6	279	S	2008	4	121	W				
2009	7	246	S	2008	5	110	W				
2009	8	270	S	2008	6	107	W				
2009	9	270	S	2008	7	111	W				
2009	10	296	S	2008	8	114	W				
2009	11	306	S	2008	9	122	W				
2009	12	283	S	2008	10	134	W				
2010	1	231	S	2008	11	143	W				
2010	2	236	S	2008	12	130	W				
2010	3	282	S	2009	1	132	W				
2010	4	278	S	2009	2	124	W				
2010	5	307	S	2009	3	126	W				
2010	6	279	S	2009	4	134	W				
2010	7	250	S	2009	5	136	W				
2010	8	278	S	2009	6	145	W				
2010	9	293	S	2009	7	140	W				
2010	10	316	S	2009	8	140	W				
2010	11	316	S	2009	9	172	W				
2010	12	316	S								
2006	1	105	W								

- 710
8. Display the number of pounds of milk produced in each region during the 2006–2010 time period.
  9. Display the number of pounds of milk produced in each region during the 2006–2010 time period. In addition, include in the output you display the total number of pounds of milk produced across all regions. Use the GROUPING function and the DECODE function to display an appropriate label for the number of pounds produced across all regions.
  10. Display the number of pounds of milk produced in each region each year, along with the same totals you displayed in your answer to the previous question. Use the GROUPING function and the DECODE function to display an appropriate label for the number of pounds produced in all years for each region and for all years for all regions.
  11. Modify the previous query so that instead of displaying as one of the totals the total number pounds of milk produced in each region, the total number of pounds of milk displayed in each year across all regions is displayed.
  12. How many rows of output would be produced if the ROLLUP used in your query were to take the following form: GROUP BY ROLLUP (YEAR, MONTH, REG\_NAME)? Write a query to verify your result.
  13. How many rows of output would be produced if the ROLLUP used in your query were to take the following form: GROUP BY ROLLUP (MONTH, YEAR, REG\_NAME)? Write a query to verify your result.
  14. Display the number of pounds of milk produced in each region each year along with the total number of pounds produced in each region, the total number of pounds of milk produced during each year, and the total number of pounds of milk produced overall.
  15. Display only the total number of pounds of milk produced in each region and in each year. Order the results by the totals generated.
  16. Display the number of pounds of milk produced in each year, along with the ranking of each year in terms of the number of pounds of milk produced. The rankings should be displayed in descending order (i.e., year with the top production should receive the highest rank).
  17. Display the number of pounds of milk produced in each region during the 2006–2010 period, along with the ranking of each region in terms of the number of pounds of milk produced. The ranks should be displayed in descending order (i.e., the region with the top production should receive the highest rank).
  18. Display the number of pounds of milk produced in each month during the 2006–2010 period, along with the ranking of each month in terms of the number of pounds of milk produced. The ranks should be displayed in descending order (i.e., the month with the top production should receive the highest rank).
  19. Display a separate set of rankings for the number of pounds of milk produced in each month for each region.
  20. Display the sum along with the cumulative sum of the production of milk across the regions for the 2006–2010 period, starting with January and ending in December.
  21. Display the sum along with a three-month moving average of the production of milk across the regions for the 2006–2010 period, starting with January and ending in December.
  22. Display the sum along with a three-month moving average of the production of milk across the regions for the 2006–2010 period, starting with January, 2006 and ending in December, 2010.

23. Write an SQL query that makes use of the MODEL clause to estimate the monthly production of milk in 2011 for each region to be equal to the average of the production of milk for that month during the 2006 through 2010 time period.

## SQL Project

---

This project<sup>15</sup> is based on eight tables (AIRPORT, FLIGHT, DEPARTURES, PASSENGER, RESERVATION, EQUIP\_TYPE, PILOTS, and TICKET) that contain data about Belle Airlines. The script required to create and populate these eight tables can be downloaded from [www.course.com](http://www.course.com) (search on the ISBN of this book) or obtained from your instructor.

### Some Background on Belle Airlines

Belle Airlines is a regional carrier that operates primarily in the southwestern United States. At the present time, Belle Airlines operates its own reservation information system. To simplify our analysis, we will assume that all reservations on Belle Airlines flights are placed through Belle Airlines employees. Flights are not booked through travel agents, and Belle Airlines does not participate in industry-wide reservation services. Each flight is assigned a unique flight number and has its own set of flight characteristics (*i.e., flight number, origin, destination, departure time, arrival time, meal code, base fare, mileage between origin and destination, and number of changes in time zone between the origin and destination of the flight*). Departures of each flight are stored in the Departures table. Each departure contains four attributes: *flight number, departure date, pilot id, and equipment number*.

Belle Airlines flies out of airports located all over the country. Data on these airports is stored in the Airport table. Data on these airports includes: *a three-character airport code, location of the airport, elevation, phone number, hub airline that operates out of the airport*. Since Belle Airlines flies out of airports located all over the country, Belle Airlines pilots live all over the country. Data on these pilots is stored in the Pilots table, which contains the following attributes: *pilot id, pilot name, social security number, street address, city, state, zip code, flight pay, date of birth, and date hired*. The company also owns its own fleet of airplanes. Data on these airplanes is stored in the Equip\_Type table, which contains the following attributes: *equipment number, equipment type, seating capacity, fuel capacity, and miles per gallon*.

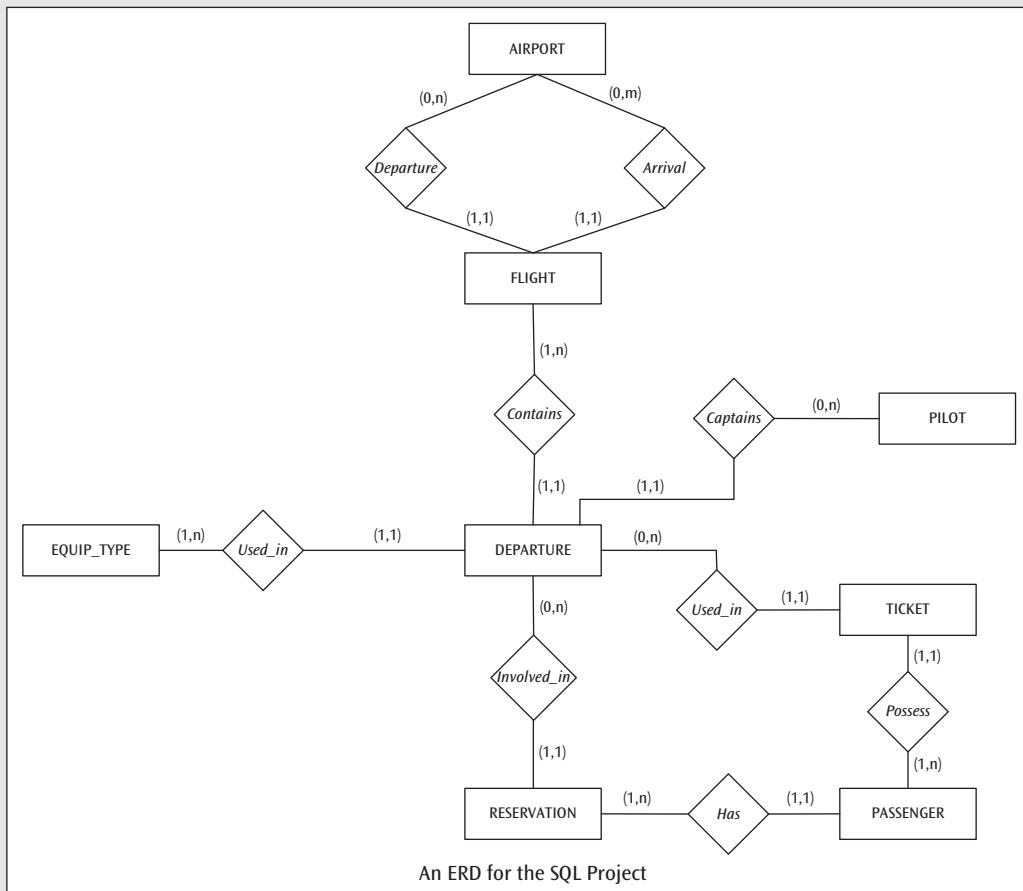
Three additional tables populate the Belle Airlines database: the Passenger table (with these attributes: passenger name, itinerary number, and confirmation number), the Reservation table (with these attributes: confirmation number, reservation date, reservation name, reservation phone, reservation flight number, and reservation flight date), and the Ticket table (with these attributes: itinerary number, flight number, flight date, and seat assignment). See the ER diagram that follows for a description of the relationships among these eight tables. In addition, once the eight tables are created and their contents printed, the following two examples can be used to illustrate the relationship between the Passenger, Reservation, and Ticket tables. The key notion here is that one passenger can make a reservation for any number of other passengers on a flight (*i.e.*, departure).

---

<sup>15</sup>This project is an adaptation of an example that appears in Lorents and Morgan (1998).

## ERD for SQL Project

712



**Example 1.** On April 1, Ole Olson (assigned confirmation number 1 in the Reservation table) reserved two tickets on Flight Number 15, scheduled to leave on April 1, and return via Flight Number 329 on April 1 and April 10. As a result, the Passenger table shows Ole Olson as Itinerary Number 1, and his wife, Lena Olson, as Itinerary Number 2. Observe that both have Confirmation Number 1 since both reservations were booked at the same time by Ole. Observe that the Ticket Table shows that Ole (Itinerary Number 1) is sitting in Seat 10D on Flight Number 15 and in Seat 12D in Flight Number 329. On the other hand, his wife, Lena (Itinerary Number 2) is sitting in Seat 10E on Flight Number 15 and in Seat 12E in Flight Number 329.

**Example 2.** On April 17, Andy Anderson (assigned confirmation number 6 in the Reservation table) reserved two tickets on Flight Number 102, scheduled to leave on April 18. As a result, the Passenger table shows Andy Anderson as Itinerary Number 12, and his wife, Gloria Anderson, as Itinerary Number 13. Once again, observe that both have Confirmation Number 6 since both reservations were booked at the same time by Andy. Observe that the Ticket table shows that Andy (Itinerary Number 12) has a ticket on Flight Number 102, with seat

assignment 10B. The same is true for Gloria (Itinerary Number 13). She has a ticket on Flight Number 102 and seat assignment 7C.

The 101 queries that follow should collectively provide a good introduction to SQL. Please note that in some cases it is quite clear which columns (fields) need to be displayed. In other cases, the columns to be displayed are left to individual judgment. Use DISTINCT to minimize the repetition of duplicate rows, and don't be afraid to use column aliases (especially when a column heading would otherwise be the content of a numeric or character function); they can make the output of a query more readable. In addition, please try to avoid wraparound as much as possible. In all cases, please refrain from the temptation to believe that the results are correct the first time output from a query is obtained. In other words, please try to display enough columns (fields), and study the output in order to verify the accuracy of the result.

In order to challenge the student, the 101 queries are for the most part randomly ordered. In other words, queries 1–10 are not necessarily the easiest to write, nor is query 101 necessarily the most difficult to write. One may want to begin working with queries that seem to involve either one or two tables, such as FLIGHT and PASSENGER. These queries involve the use of the LIKE operator, numeric functions (e.g., COUNT, MIN, MAX, AVG, etc.), grouping, nested subqueries, and joins.

1. Display the origin, destination, departure time from origin, and arrival time at destination for all flights that occur in the same time zone. Your results should be displayed in order by flight number.
2. Display the code, location, and elevation of all airports without a hub airline. Your results should be in descending order by elevation.
3. Display the departures originating from Los Angeles, CA. Include in your results flights from Los Angeles for which no departures currently exist. Los Angles, CA and not LAX should be used in the WHERE clause of your query.
4. Display the flight numbers and the codes for the origins and destinations of all flight reservations made by Andy Anderson.
5. Display the seating capacity, fuel capacity, and miles per gallon for all aircraft manufactured by Boeing. Information about each equipment type should be displayed only once.
6. Display the names of all pilots who live outside of the state of Texas. Order the results in alphabetical order by last name.
7. Display the flight number, flight date, fare, origin, and destination for all tickets with a flight date of July 2006. Use the fare in the FLIGHT table as the fare for the ticket. Order your results in ascending order by flight date and within flight date by flight number.
8. Display all flights that originate at an airport without a hub airline.
9. Display all flights that arrive at an airport without a hub airline.
10. Display all flights that both originate and arrive at an airport without a hub airline.
11. Display all departures that are flown by an aircraft *not* manufactured by Boeing. Your results should be in ascending order by departure date and within departure date by flight number.
12. Display the distance divided by the fare for each flight. For each flight, display the flight number, the origin, the destination, the distance, the fare, and the quotient. Your results

should be in descending order by the quotient and rounded to two places to the right of the decimal point. Create a descriptive column alias for the quotient.

13. Display the total number of flights that originate from each point of origin.
14. Revise the previous query so that instead of displaying the code for each point of origin, the location of each point of origin from the AIRPORT table is displayed.
15. Revise the previous query to also include the display of those locations where no flights originate.
16. Display the average flight pay for pilots that live in each state.
17. Display the name and flight pay for those pilots whose flight pay exceeds the average flight pay for all pilots.
18. Display the name and flight pay for those pilots whose flight pay exceeds the average flight pay for all pilots in the state in which they reside.
19. Display the date of the most recent departure flown by each pilot. Include in what you display the name of the pilot.
20. Display not only the date of the most recent departure by each pilot but also the number of days since the last departure date. Truncate the number of days (i.e., if number of days is 37.67655, display 37) to zero places to the right of the decimal point. Order the result in descending order by the number of days.
21. Display the number of departures that involve flights for each of the three time zone differences.
22. Display the number of airports located in each state.
23. Display the number of departures where the distance flown is greater than or equal to 1000 miles.
24. Display the difference in age between the oldest and youngest pilot.
25. For each type of aircraft, display the total distance that can be flown before refueling. Display your results in descending order by total distance that can be flown.
26. For each passenger listed in the PASSENGER table, display the name of the person responsible for his or her reservation.
27. For each passenger listed in the PASSENGER table, display the name of the person responsible for his or her reservation only if the passenger himself or herself was not responsible for making the reservation.
28. For each reservation in the RESERVATION table, display the name of the pilot who will be piloting the flight.
29. Display those tickets that include only one flight.
30. Display the name of the passengers whose tickets include only one flight.
31. What flights leave Phoenix for Los Angeles between 3:00 PM and midnight? Display each flight's flight number, city name of the flight's origin, city name of the flight's destination, departure time, and arrival time.
32. What are the fares from Phoenix to Los Angeles if Belle Airlines is running a 20 percent discount special off the current fares? Display each flight's flight number, city name of the flight's origin, city name of the flight's destination, and discounted fare.

33. Andy Anderson wants to know the passenger and ticket information on all passengers for which he has made reservations. For each reservation, display the passenger name, flight number, and flight date and seat assignment.
34. Display the maximum fare for flights originating at each airport if that fare is greater than \$100. For each qualifying airport, display the airport code, location, and maximum fare. Display the results in descending order by fare.
35. Display the flight number of the flights that have no ticketed passengers scheduled on April 18, 2006. Display the results in ascending order by flight number.
36. What is the passenger count for each flight that has more than one ticketed passenger scheduled? For each qualifying flight, display the flight number and number of passengers.
37. Display the city name of the flight's origin and city name of the flight's destination as well as all data about the flight for all tickets held by Pete Peterson.
38. Display the names and phone numbers of persons who have reservations on flights leaving Phoenix, Arizona on May 17, 2006 for each flight booked with fewer than three passengers.
39. Display all flights where the origination time is later than at least one of the Minneapolis to Phoenix flights.
40. Display all flights that leave later than all flights going from Phoenix to Los Angeles.
41. Display the flight information (origination, destination, and times) on all passengers who are flying under a reservation made by Pete Peterson. The origin and destination should include the entire name of the city.
42. Display the number of departures associated with each pilot in ascending order by pilot name. Include all pilots, even those without any departures, in your results.
43. Display the names of those pilots who were not assigned to a departure during April 2006.
44. A passenger wants to fly from Phoenix to Los Angeles and back in a single day. He needs at least five hours in Los Angeles to get to and from the airport and conduct his business. List the flight numbers, origin times, and destination times of the flights that will accommodate his schedule.
45. What flights from Flagstaff to Phoenix have connecting flights in Phoenix going on to Los Angeles? Allow 40 minutes for a connection.
46. Display the total of the fares for all tickets related to Pete Peterson. Use the fare in the FLIGHT table in your calculation.
47. Display the total number of tickets sold for each flight across all dates.
48. Display the total of the fares collected for each flight on each date. Assume all tickets were sold at full fare and that the fares come from the FLIGHT table.
49. Display the maximum fare for flights between each origin and destination airport (e.g., Phoenix to Los Angeles, Phoenix to Flagstaff, Phoenix to San Francisco). As part of your result, display the name of the city where the airport is located—not the code.
50. Display the total miles flown by each pilot. Display the name of each pilot along with the miles flown. Display the results in descending order by miles flown. Include all pilots in your result—even those who have not yet flown a flight.

- 716
51. Display the names of all pilots who have flown a Boeing 727. Please display the first name followed by the middle initial and the last name of each pilot.
  52. Display the names of the passengers with tickets on the July 23, 2006 departure of Flight Number 104. Your results should be displayed in ascending order by seat number.
  53. Display the flight number and date of all departures originating from Phoenix that serve either a snack or nothing to passengers.
  54. Display all flights with either a California origination or destination.
  55. Display all flights that depart on one day and arrive the next day.
  56. Display the total compensation to each pilot in April 2006. The total compensation for a pilot is the product of the pilot's flight pay times the number of flights flown. Include pilots who did not fly any flights in April 2006.
  57. Under the assumption that fuel costs \$2.31 per gallon, display the total cost of fuel for all departures flown by each aircraft. Include the equipment number and equipment type of each aircraft.
  58. Display the name and age of the youngest pilot.
  59. Display the name and hiredate of all pilots who were hired during the 1990s. The qualifying pilots should be displayed in descending order by length of service.
  60. Display the name and age of all pilots who were less than 40 years of age when hired and are now older than 47 years of age.
  61. Display the names of those passengers whose last name begins with the letter "A." Each unique passenger name should be displayed only one time.
  62. Display the confirmation number, reservation date, reservation name, phone number of the person making the reservation, and flight number for those reservations made during April 2006 for flights scheduled to depart sometime after April 2006. In addition, the area code of the phone number should be separated from the first three digits by a hyphen, and the first three digits of the phone number should be separated from the last four digits by another hyphen.
  63. In a Boeing 727, seats A and E are window seats. Display the name, flight number, and departure date for all passengers who have reserved a window seat.
  64. Display the names of passengers who have flown on a flight piloted by William B. Pasewark. Include in your results the flight number as well as the date of the flight. Display the result in order by passenger name.
  65. Produce a list showing the total number of pilots hired in each year. The list should contain the calculated current average flight pay for all pilots hired during that year. The resulting list should be in ascending order by year.
  66. Display the total number of pilots who live in each state in alphabetical order by state.
  67. Display the total number of departures associated with each meal type.
  68. Display the names of those pilots scheduled for a departure in an airplane manufactured by Boeing Corporation during April 2006.
  69. Display the equipment number and equipment type of airplanes flown by Stuart Long.

70. Display the first and last name of all pilots who have flown more than one type of aircraft. Display the results in ascending order by last name and include in the result the equipment types each qualifying pilot has flown.
71. Display just the first and last name (first name followed by last name with no middle initial) of pilots who have not been assigned to a flight during May 2006. *Note:* This is a challenging query.
72. Display the first name and last name of those passengers who have a ticket with at least two departures.
73. For the oldest pilot, display his or her name, and aircraft flown.
74. Display the names of those passengers who at one time or another have made at least one reservation.
75. Display the name and age of the pilot with the highest pay among those pilots younger than 50 years of age.
76. Display information about all departures flown by the aircraft with the largest fuel capacity. Include in your results the flight number, departure date, origin, and destination.
77. Display the first and last name of those passengers whose reservation was made at least one month in advance of their departure. In addition, display the flight date, origin, and destination.
78. How many flights are flown within the same state? Display the flight number, origin city, destination city, and type of aircraft used.
79. Display the total number of miles flown on ticketed departures by each pilot. If a pilot has never flown a ticketed flight, display a zero.
80. Display the names of those pilots who live in a city that does not have an airport.
81. Display the types of aircraft used to fly to or from the cities with the highest or lowest elevation.
82. Display the name and phone number of the passenger booked on the most reservations.
83. Display the name, age, and hiredate of the pilots who fly either to or from the city with the highest elevation. The result should include the name of the city.
84. Display the name(s) of all persons making a reservation for Lena Olson.
85. Display information on tickets associated with a person whose last name is Peterson.
86. Display the number of tickets associated with those people who make reservations.
87. Display the number of tickets sold on each row. The results should be displayed in ascending order by row number.
88. Display the number of tickets associated with each seat in ascending order by seat.
89. Display the names of passengers who have never made a reservation themselves.
90. Display the total fare associated with each reservation. Base your calculation on the fare associated with each flight.
91. Display the total number of tickets sold during each month.
92. For each ticket, display the name of the passenger, the name of the person making the reservation, the name of the pilot, the fare, the fuel capacity of the airplane assigned to the departure, and the flight's point of origination and point of destination.

93. For each flight, display the number of departures associated with flights within the same time zone.
94. Display the location of those airports that have been neither the point of origination nor the point of destination for any Belle Airlines flights.
95. Display flights without any departures in May 2006.
96. Calculate the number of scheduled departures of flights from each airport during May 2006 that include lunch or dinner. Display the number of flights for each qualifying airport.
97. Assume that airports with a hub airline offer flights for just that airline (e.g., the only airline flying out of Phoenix is Belle Airlines, the only airline flying out of Minneapolis is Northwest). Calculate the number of departures associated with each hub airline.
98. Display the flying time associated with each flight. Remember to take into account the difference in time zones from the point of origination versus the point of destination.
99. Display the number of tickets associated with each flight that has a California destination.
100. Calculate the total fare paid by each passenger. Use the fare in the FLIGHT table.
101. Display the number of departures associated with each date in ascending order by date.

# APPENDIX A

# DATA MODELING ARCHITECTURES BASED ON THE INVERTED TREE AND NETWORK DATA STRUCTURES

Appendix A begins with illustrations of the inverted tree and network data structures—the two basic data structures that underlie the data models used today for designing databases. Discussion of these data structures is followed by a brief overview of how the hierarchical and CODASYL data model architectures express the inverted tree and network data structures, respectively.

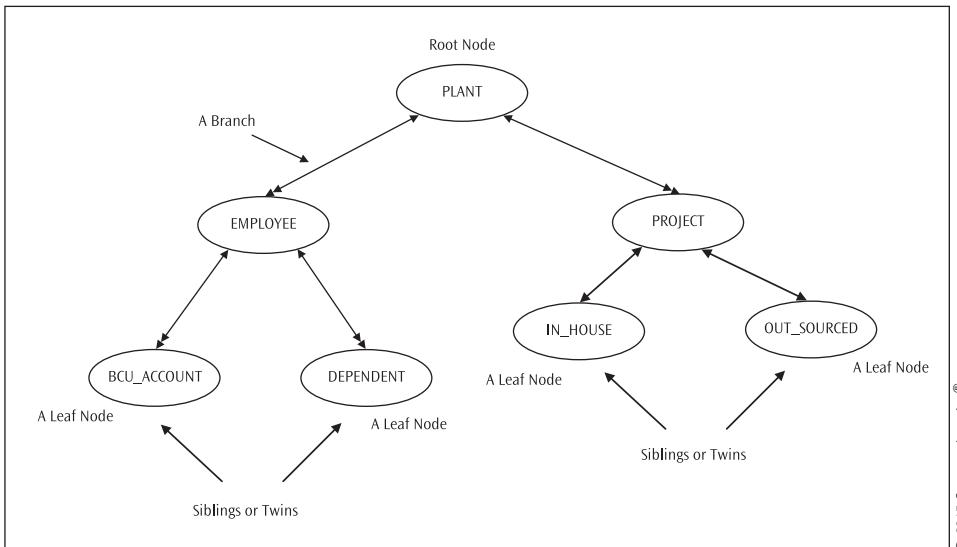
## A.1 LOGICAL DATA STRUCTURES

---

From a data modeling and database design perspective, there are just two basic logical data structures—the inverted tree structure and the network data structure—and the latter subsumes the former. Nonetheless, an exposure to both logical data structures is worthwhile because the three major logical data modeling architectures, viz., relational, hierarchical, and CODASYL, implement one or both of these structures.

### A.1.1 Inverted Tree Structure

The inverted tree is a data structure in which the elements of the structure have only 1:1 and 1:n relationships with one another. Figure A.1 contains a variation of the Presentation Layer ER diagram in Chapter 3, Figure 3.3 that takes the form of an inverted tree. An inverted tree consists of a hierarchy of nodes connected by branches. Each node in the tree, except the root node located at the top of the tree, has exactly one parent node and zero or more child nodes. Child nodes with the same parent are called twins or siblings. A root node has no parent, and a node with no child nodes is called a leaf node.

© 2015 Cengage Learning<sup>®</sup>**FIGURE A.1** An inverted tree structure

In an inverted tree, a **1:n** relationship is referred to as a **parent-child relationship type (PCR type)**.<sup>1</sup> Six PCRs appear in Figure A.1: a PLANT may employ many employees,<sup>2</sup> an EMPLOYEE may have many bank accounts, an EMPLOYEE may have many dependents, a PLANT may control many projects, a PROJECT may be done in-house, and a PROJECT may be outsourced.<sup>3</sup> Observe that while it is possible for nodes to have more than one child (e.g., both EMPLOYEE and PROJECT are child nodes of PLANT; BCU\_ACCOUNT and DEPENDENT are child nodes of EMPLOYEE), each node has exactly one parent. If this were not true (e.g., if BCU\_ACCOUNT is a child of both EMPLOYEE and DEPENDENT), an inverted tree structure would not exist. Thus, clearly, the Presentation Layer ER diagram in Figure 3.3 does not reflect an inverted tree structure. In short, in an inverted tree structure a node can participate as a parent in several PCRs; but as a child in only one PCR. That is, a parent node can have several child nodes, while a child node can have only one parent.

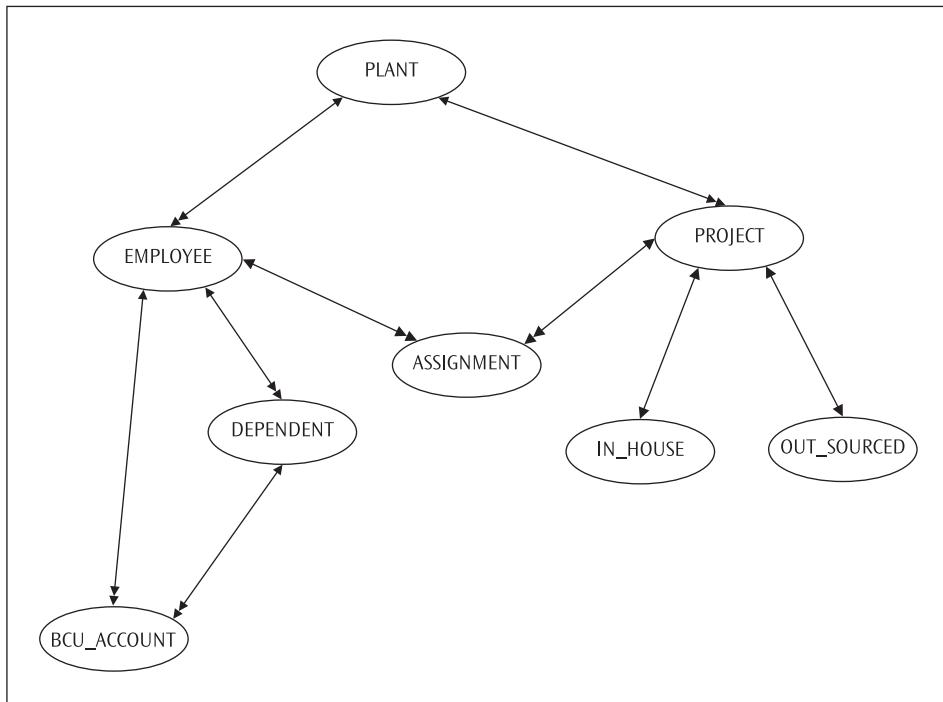
<sup>1</sup>When two nodes exhibit a **1:1** relationship, either can be designated as the parent or the child.

<sup>2</sup>Observe that the line connecting EMPLOYEE to PLANT has a single arrowhead on the PLANT end indicating that an employee works for (is associated with just one PLANT); since a PLANT can have many EMPLOYEES, the same line connecting PLANT to EMPLOYEE has two (i.e., multiple) arrowheads on the EMPLOYEE end.

<sup>3</sup>The semantics of this illustration allow for a project to be an in-house or an outsourced project or contain no more than one in-house and one outsourced component.

### A.1.2 Network Data Structure

A network or plex structure is also composed of nodes and branches but unlike an inverted tree structure, a child node in a network structure can have multiple parents. Figure A.2 shows how the network data structure allows BCU\_ACCOUNT to have two parents (EMPLOYEE and DEPENDENT). Likewise, ASSIGNMENT is a child node with two parents, viz., EMPLOYEE and PROJECT.<sup>4</sup> In essence, in a network data structure, a node can participate in more than one PCR as a parent as well as a child.



**FIGURE A.2** A network structure

Thus, both inverted tree and network data structures permit a node to participate as a parent in multiple PCRs. That a node can participate as a child in only one PCR is an additional constraint specific to the inverted tree structure. When this constraint is relaxed and a node is permitted to also participate as a child in multiple PCRs, a network structure eventuates. From this it should be clear that an inverted tree structure is a special case of the network structure. In other words, implementation of the latter implicitly implements the former.

<sup>4</sup>Observe that the network data structure is similar to the EER construct Specialization Lattice (see Section 4.1.3 in Chapter 4).

## A.2 LOGICAL DATA MODEL ARCHITECTURES

There are three major architectures for logical data models. The **relational data model**, discussed in Chapter 6, literally monopolizes contemporary DBMS architectures. However, the **hierarchical** and **CODASYL data models** precede the relational data model and have historical value in that many legacy systems still run on DBMS platforms that implement hierarchical or CODASYL architectures. Two more architectures are currently emerging and are often referred to as post-relational data models: the **object-oriented data model** and the **object-relational data model**. The former is considered an aggressive competitor to the relational data model, while the latter containing the features of both relational and object-oriented constructs is seen at least by some [e.g., Date (2004)] as the future of practical data modeling for database design. These models are the subject of Appendix B.

### A.2.1 Hierarchical Data Model

The hierarchical data model was developed in the early 1960s as a joint effort between IBM and two of its customers, Rockwell and Caterpillar. This architecture essentially implements the inverted tree structure. Both manufacturing organizations needed a bill-of-materials processor that would print the total number of each component required to meet a given production schedule for a particular product.<sup>5</sup> This narrow problem required a specific solution. Solving the problem led IBM to market a DBMS known as IMS that could handle the bill-of-materials problem well but was limited in flexibility since it was not only incapable of representing anything other than an inverted tree structure, but also required regeneration of the whole data model whenever addition of node(s) became necessary.

In IBM's implementation of the hierarchical model, files are called segments or nodes and record occurrences are called segment occurrences or segment instances. Fields are called fields or data items. Relationships between segments are called parent-child relationships. In a 1:n relationship, the segment occurring on the one side is the parent segment and the segment on the many side is called the child segment.

Mapping (or transforming) a conceptual schema to a logical schema using the hierarchical architecture requires four steps. First, each entity type must be mapped to a segment type. Second, since the hierarchical data model does not allow for the direct representation of network structures, any network structure must be reduced to a collection of inverted trees.<sup>6</sup> Third, after transforming the conceptual schema to one or more inverted tree structures, a decision must be made as to the hierarchical ordering of the segments.<sup>7</sup> The final step involves mapping each attribute to a field of a segment.

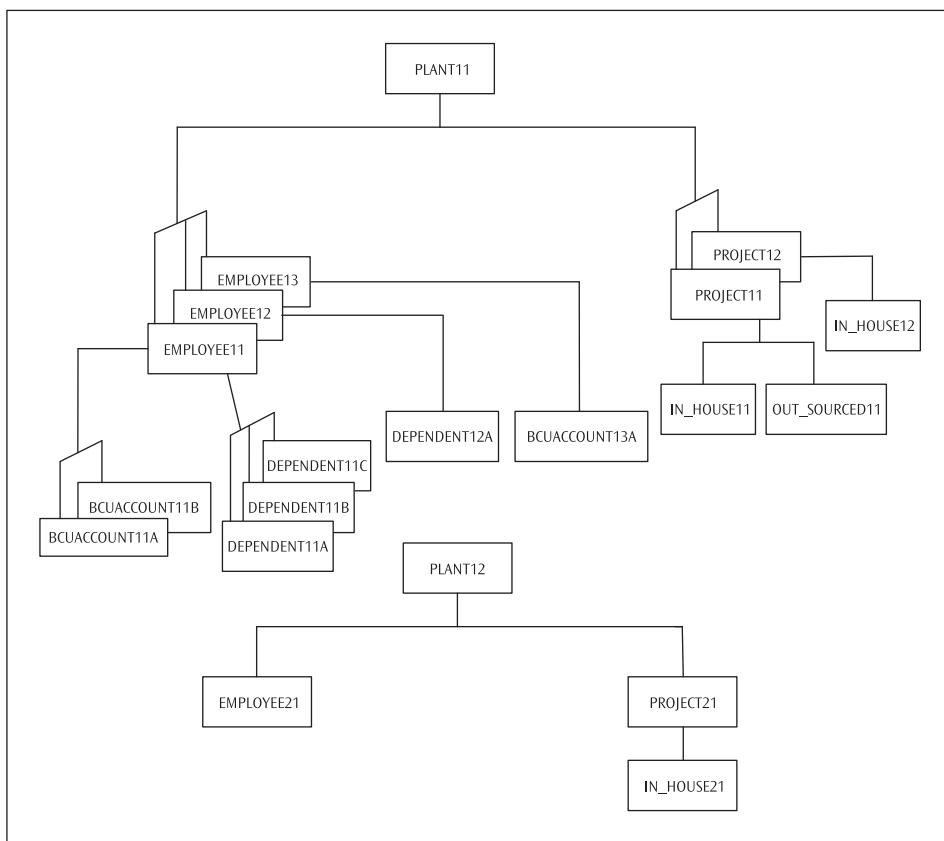
<sup>5</sup>A bill of materials is sequence of 1:n relationships between the subassemblies necessary for producing an item. (Shepherd, 1990, p. 67).

<sup>6</sup>For example, if BCU\_ACCOUNT can be a child of both EMPLOYEE and DEPENDENT, Figure A.1 would have to be revised to show a BCU\_ACCOUNT node under DEPENDENT as well as a second BCU\_ACCOUNT node under EMPLOYEE, thus introducing possible redundant bank account data. However, in practice this data redundancy is handled through the use of logical pointers. Figures A.3 and A.4 illustrate a situation of this nature.

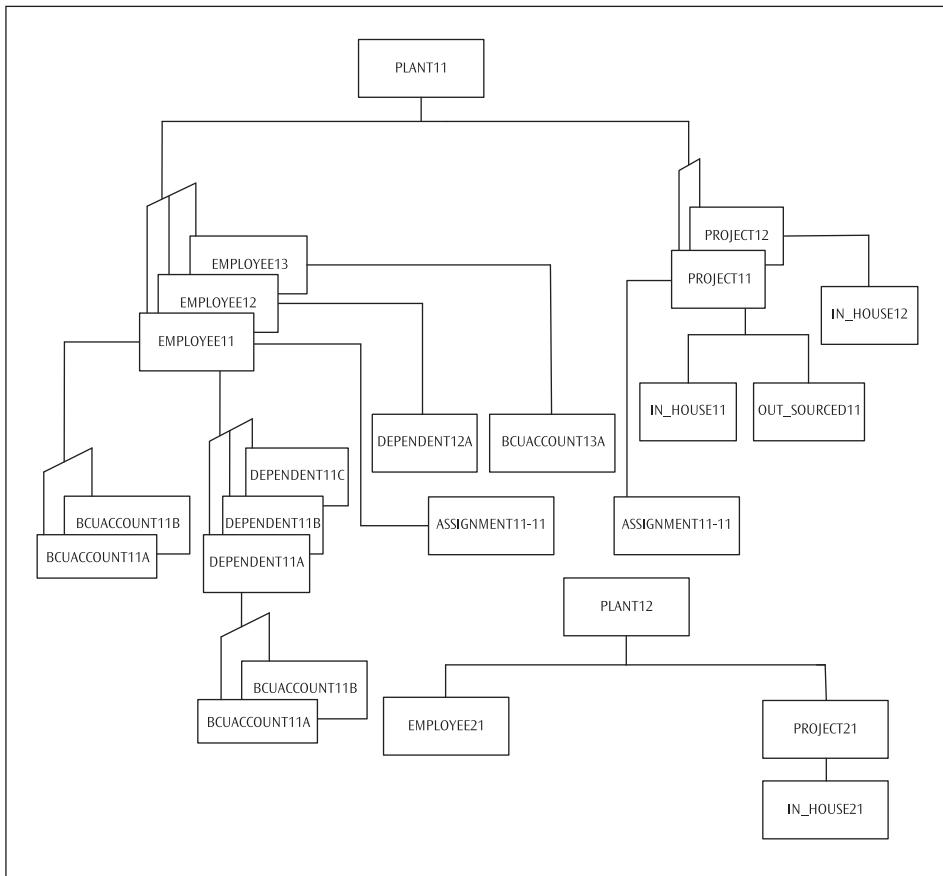
<sup>7</sup>This is an issue of navigational efficiency through the hierarchy. Several strategies are available for ordering the segments. However, this topic is beyond the scope of this book.

## Data Modeling Architectures Based on the Inverted Tree and Network Data Structures

Figure A.3 shows two occurrences (one for PLANT11 and a second for PLANT12) of the inverted tree structure that appears in Figure A.1. Figure A.4 shows two occurrences of the network structure that appears in Figure A.2 implemented in a hierarchical architecture. Since in the case shown in Figure A.2 it is possible for a bank account to be associated with both an employee and a dependent, the segment occurrences for bank accounts 11a and 11b must appear redundantly as children of both EMPLOYEE11 and DEPENDENT11A (one of the three dependents of EMPLOYEE11). Likewise, the segment occurrence for assignment 11-11 also appears redundantly under EMPLOYEE11 and PROJECT11. While this exemplifies how a network structure is accommodated in a hierarchical architecture, from a practical standpoint, the resulting redundancy often induces processing inefficiencies that are bound to reduce the database system performance.



**FIGURE A.3** Two occurrences of the hierarchical structure shown in Figure A.1



**FIGURE A.4** Two occurrences of the network structure shown in Figure A.2

The Data Definition Language for IMS is Data Language/1 (DL/1). DL/1 makes it possible to map a conceptual schema to a logical schema as well as to a physical schema. In DL/1, a field is the smallest unit of data, a segment is a group of related fields, a database record is a hierarchically structured group of segments, and a database is a collection of database record occurrences of one or more database record types. DL/1 makes use of a database description (DBD) to define the way data is stored for use by IMS. In addition, the DBD also defines the format, length, and location of each data item to be accessed by the DL/1 data manipulation language. The following lines illustrate a DBD that reflects the hierarchical structure in Figure A.1. It should be noted that many of the details associated with a complete DBD (e.g., the length and location of each field) are not shown.

DBD            NAME = BEARCAT  
SEGM          NAME = PLANT  
                  PLANT Field Descriptions

SEGM NAME = EMPLOYEE, PARENT = PLANT  
                  EMPLOYEE Field Descriptions

SEGM NAME = BCU\_ACCOUNT, PARENT = EMPLOYEE  
                  BCU\_ACCOUNT Field Descriptions

SEGM NAME = DEPENDENT, PARENT = EMPLOYEE  
                  DEPENDENT Field Descriptions

SEGM NAME = PROJECT, PARENT = PLANT  
                  PROJECT Field Descriptions

SEGM NAME = IN\_HOUSE, PARENT = PROJECT  
                  IN\_HOUSE Field Descriptions

SEGM NAME = OUT\_SOURCED, PARENT = PROJECT  
                  OUT\_SOURCED Field Descriptions

The DBD begins with a DBD macro<sup>8</sup> which among other things assigns a name to the hierarchical structure. Each segment description is headed by a SEGM macro that names the segment, indicates its total length in bytes (not shown here), and gives the name of its parent. The first segment, or root, has no parent. Each field within a segment is represented by a FIELD macro. Applications access an IMS database through DL/1 data

---

<sup>8</sup>A macro is a short program consisting of several operations saved in a file under a certain name, which can be invoked from within another program.

## Appendix A

manipulation language commands embedded in a host language such as COBOL, PL/1, and C. These commands navigate the hierarchical structure of the database to retrieve, insert, update, or delete segments. For this reason, IMS and other implementations of the hierarchical data model are sometimes referred to as navigational systems.

The hierarchical data model allows only one relationship between any two segments and requires that each relationship be explicitly defined when the database is created. While satisfactory for applications that are inherently hierarchical in nature and for which query transactions are stable, to a great extent, databases based on the hierarchical data model are used today only in ongoing legacy systems that must be maintained. It should be noted that the hierarchical data model does not permit the direct representation of a network data structure such as the one that appears in Figure A.2. Approaches for transforming data relationships that involve network structures into a hierarchical architecture involve the use of logical pointers and are discussed in Johnson (1997).

### A.2.2 CODASYL Data Model

During the 1960s efforts to develop a standard database theory were evolving along many paths. In 1965, the Conference on Data Systems Languages (CODASYL) established a Database Task Group (DBTG) to develop a database model for processing using COBOL. Such a model was developed and submitted first in 1971 to the American National Standards Institute (ANSI) for its consideration as a national standard. Although revised several times throughout the 1970s and during the first part of the 1980s, the model was never accepted as a national standard. Nevertheless, during this time several commercial database management systems based on what came to be known as the CODASYL data model (or DBTG network model) were implemented by some vendors (e.g., IDS by Honeywell, DMS-1100 by Univac, IDMS by Cullinet Corporation).

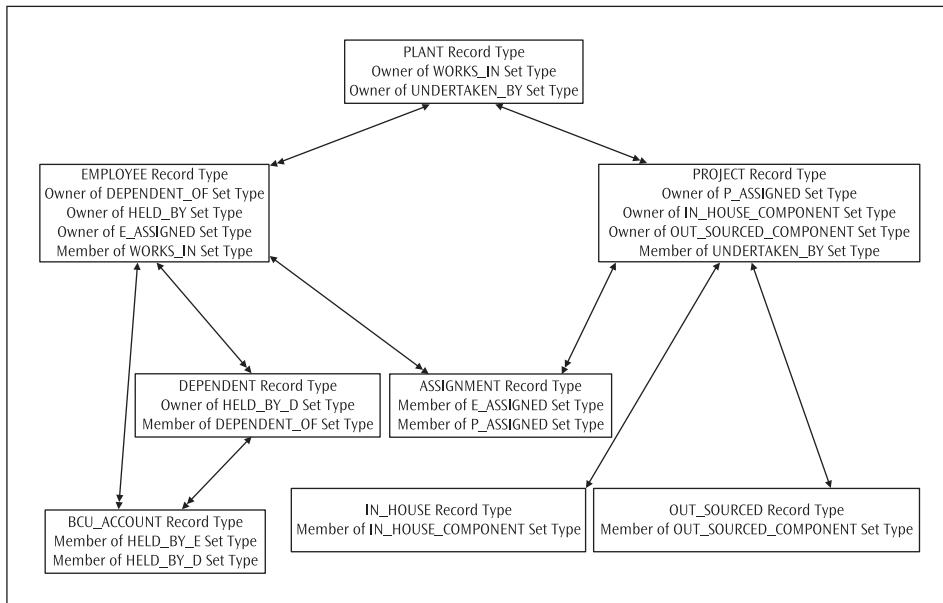
The DBTG network model adheres to the ANSI/SPARC three-schema architecture described in Section 1.5 of Chapter 1. The conceptual level (the logical view of all the data and relationships in the database) is called the **schema**. The external level (the users' views of the data needed for various applications) is called the **subschema**. The internal level (the physical details of storage) is implicit in the implementation.

In the CODASYL data model, the database is a collection of records and sets controlled by a single schema. A record type is a named collection of data items, whereas a set type is a named relationship between an owner record type and a member record type.<sup>9</sup> A set occurrence consists of one owner occurrence and all member occurrences of that set. Within a set occurrence, members are ordered on a member data item(s) or arranged in accordance with a time sequence.

Figure A.5 depicts the nine sets that relate plant, project, employee, dependent, BCU\_account, assignment, in\_house, and out\_sourced from Figure A.2 together. Observe that unlike the hierarchical data model, the CODASYL data model allows both an inverted tree structure and a network structure to be directly represented. This makes it possible for BCU\_ACCOUNT to be shown as a member of two sets (the EMPLOYEE/BCU\_ACCOUNT set and the DEPENDENT/BCU\_ACCOUNT set).

---

<sup>9</sup>A record type is equivalent to a segment type in a hierarchical architecture. Likewise, a set type is the same as a PCR, an owner represents the parent and the member is the child.



**FIGURE A.5** The set representation of Figure A.2

Like the hierarchical data model, implementations of the CODASYL data model include a data definition language. The following lines illustrate how the eight record types and nine set types for the structure in Figure A.5 can be declared:

```

SCHEMA NAME IS BEARCAT
RECORD NAME IS PLANT
RECORD NAME IS EMPLOYEE
RECORD NAME IS PROJECT
RECORD NAME IS DEPENDENT
RECORD NAME IS BCU_ACCOUNT
RECORD NAME IS ASSIGNMENT
RECORD NAME IS IN_HOUSE
RECORD NAME IS OUT_SOURCED
SET NAME IS WORKS_IN
  OWNER IS PLANT
  MEMBER IS EMPLOYEE
SET NAME IS UNDERTAKEN_BY
  OWNER IS PLANT
  MEMBER IS PROJECT
SET NAME IS DEPENDENT_OF
  OWNER IS EMPLOYEE
  MEMBER IS DEPENDENT
SET NAME IS HELD_BY_E
  OWNER IS EMPLOYEE
  MEMBER IS BCU_ACCOUNT
  
```

```
SET NAME IS HELD_BY_D
    OWNER IS DEPENDENT
    MEMBER IS BCU_ACCOUNT
SET NAME IS E_ASSIGNED
    OWNER IS EMPLOYEE
    MEMBER IS ASSIGNMENT
SET NAME IS P_ASSIGNED
    OWNER IS PROJECT
    MEMBER IS ASSIGNMENT
SET NAME IS IN_HOUSE_COMPONENT
    OWNER IS PROJECT
    MEMBER IS IN_HOUSE
SET NAME IS OUT_SOURCED_COMPONENT
    OWNER IS PROJECT
    MEMBER IS OUT_SOURCED
```

As was the case for the DBD for the hierarchical data model that appears in Section A.2.1, many of the details associated with the definition of the record types and sets in this schema are not provided. For example, the definition of a record type includes several clauses that specify the scheme used to locate the record and to define the individual data items that comprise the record type. Besides identifying the owner and member of each set type, the definition of each set type includes rules for the insertion of new member records and for moving existing records from one set occurrence to another. Interested readers may wish to refer to Shepherd (1990) for an excellent introduction to the CODASYL as well as the hierarchical data model.

## Summary

---

The hierarchical data model is the oldest of the data models and organizes data in the form of an inverted tree consisting of a hierarchy of parent and child segments, where a child is allowed to have only one parent. Coinciding with the development of the hierarchical data model was the CODASYL data model, which allowed more than one parent per child. Both of these models were used primarily during the mainframe era as vehicles for describing the structure of data as well as data manipulation operations but are no longer used as the basis for database systems today. While a number of legacy systems structured in accordance with these models remain in use today, many predict that they will be phased out over time as the number of qualified staff declines due to retirement and retraining.

## Selected Bibliography

---

- Date, C. J. (2004) *An Introduction to Database Systems*, Eighth Edition, Addison-Wesley.  
Kroenke, D. M. (1977) *Database Processing*, Science Research Associates, Inc.  
Shepherd, J. C. (1990) *Database Management: Theory and Application*, Richard D. Irwin, Inc.



# APPENDIX B

# OBJECT-ORIENTED DATA MODELING ARCHITECTURES

Object-oriented concepts have drawn considerable attention among researchers and practitioners since the late 1980s and have significantly influenced efforts to incorporate in the DBMS the ability to process complex data types beyond just storage and retrieval. Appendix B briefly introduces the reader to object-oriented concepts exclusively from a database or, to be more precise, from a data modeling perspective.

## B.1 THE OBJECT-ORIENTED DATA MODEL

---

The needs of most business database applications to date are reasonably satisfied using simple data types (e.g., numbers and character strings). To this extent, the traditional database system architectures overviewed thus far (i.e., hierarchical, CODASYL, and relational) have been capable of providing adequate support to the commercial needs of modern enterprises. As the use of database systems spreads to wider domains (e.g., engineering and medicine), the need for handling complex data types has become apparent (e.g., CAD/CAM, CIM, CASE,<sup>1</sup> image processing, document handling) and the limitations imposed by the currently prevalent relational models have emerged as obstacles in these newer application domains. In current database environments this problem is somewhat mitigated by storing complex data outside the purview of the DBMS for concomitant access by application programs. Recent improvements in relational database systems allow for storing complex data types as large objects (LOBs)<sup>2</sup> in the database. However, the DBMS support involves only storage and retrieval. All other processing of these complex data types is still the responsibility of the application programs—i.e., the host language in which SQL is embedded, for storage and retrieval support only. Another related issue of concern has been the inadequacy of the database query language (i.e., SQL) in handling complex data types.

---

<sup>1</sup>CAD/CAM stands for Computer-aided Design/Computer-aided Manufacturing, CIM is the acronym for Computer-integrated Manufacturing and CASE is the acronym for Computer-aided Software Engineering)

<sup>2</sup>SQL:1999 supports the Large Object (LOB) data type with two possible variants—Binary Large Object (BLOB) and Character Large Object (CLOB). A LOB has a unique id called a *locator* which allows LOBs to be manipulated without extensive copying. LOBs are typically stored separately from the tuples in whose attributes they appear.

## Appendix B

As a consequence, object-oriented (OO) concepts have drawn considerable attention among researchers and practitioners since the late 1980s and have significantly influenced efforts to incorporate the ability to process complex data types beyond just storage and retrieval by the DBMS itself. Object orientation in software engineering emerged originally in the programming languages arena (e.g., C++, SMALLTALK, Java). The initial motivation for OO database and Object DBMS (ODBMS) has essentially been due to a desire to seamlessly integrate DBMS functionality in the programming language environment so that the limitation of lack of persistence<sup>3</sup> of data in programming languages can be overcome. Several experimental ODBMS prototypes (e.g., ORION, IRIS, ODE)<sup>4</sup> and commercial products (e.g., GEMSTONE/OPEL of Gemstone Systems, ONTOS of Ontos Corporation, and Versant of Versant Object Technology) have been developed. Yet none have found widespread usage in the business application domain (Elmasri and Navathe, 2004). Our interest in this book is to provide a general understanding of OO concepts exclusively from a database or, to be more precise, from a data modeling perspective—certainly not from the programming perspective; in the OO world there is a difference in these perspectives.

### B.1.1 Overview of OO Concepts

The core concept of the OO approach is the idea of “bundling” data and the operations pertaining to the data as integrated units called ‘objects.’ The internal structure of an object is not made visible to any user of the object; the users only know the attributes and the methods (i.e., operations) that the object is capable of executing on the data (i.e., attribute values). The expectation is that the objects more closely resemble their counterparts in the world at large and the user should not have to be overly concerned about the technology-oriented constructs (e.g., bits, bytes, fields, records, and even entities and relationships). For instance, the general mental image of a Flight often includes an Aircraft, Passengers, and Crew as a unified object. Join operations that connect a specific aircraft, a crew, and a group of passengers to form a flight is viewed as an implementation detail from which the users can be spared. While the simplicity of this example may not quite reflect the complexity equivalent to CAD/CAM or medical imaging applications, the example clearly demonstrates the concept. In essence, the goal is to raise the level of abstraction in data modeling and database design an additional notch. The concept is referred to as **encapsulation** and represents a paradigm shift from traditional database principles where data and methods are treated independently as complementary units.

From a data modeling perspective, object orientation espouses four notions: object structure, object class, inheritance, and object identity.

---

<sup>3</sup>Persistence in the OO paradigm refers to continued existence of data even after the program that created it has terminated.

<sup>4</sup>ORION was developed at Microelectronic and Computer Technology Corporation (MCC), Austin, TX; IRIS was developed by Hewlett-Packard; and ODE was developed at AT&T Bell Labs, now a part of Lucent Technologies.

### B.1.1.1 Object Structure

So, what is an object? In the basic OO approach, “everything is an object.” Some objects never change and are known as **immutable objects** (e.g., integers like 7, 1, 43 and strings like “Cincinnati,” “Star Trek”); others are **mutable**—i.e., variable (e.g., Flight, Airport). In a loose sense, an object corresponds to an entity in the ER modeling grammar except that an entity does not encapsulate methods with its data. An object constitutes a *state* (value) and a *behavior* (operations). In contrast with the OO programming languages where objects are transient—i.e., exist only for the duration of program execution, the objects in a OODB are persistent—i.e., they exist beyond the program termination and can be retrieved and shared by other programs at a later time. A set of variables analogous to attributes in the ER modeling grammar carry the data for an object, a set of pre-defined messages (with or without parameters) invoke methods for the objects, and a set of pre-defined methods (a body of code to execute operations) respond to the messages. Together, the variables, messages, and methods constitute an object. All interactions between an object and the rest of the system (essentially, other objects) are via messages.

### B.1.1.2 Object Class

Every object belongs to a *type*, and this is usually referred to as an **object class**. Objects that have variables of the same name and type, respond to the same set of messages, and use the same set of methods belong to the same object class. Individual objects belonging to an object class are also called **object instances** or just **instances**. The notion of an object class is equivalent to the notion of an entity type/class in the ER modeling grammar except for the methods component of an object class. A consortium of ODBMS vendors and users called the Object Management Group (OMG) has proposed an *Object Model* which forms the basis for an *Object Definition Language* (ODL) and an *Object Query Language* (OQL). Detailed discussion of the object model is beyond the scope of this book. The reader is directed to the references in the Selected Bibliography for additional material.

### B.1.1.3 Class Inheritance

Class inheritance is analogous to type inheritance in the EER modeling grammar except that in class inheritance, objects of the subclass not only inherit the public instance variables<sup>5</sup> of the superclass, but also the methods associated with the superclass. The former is called *structural inheritance* and the latter *behavioral inheritance*. The ability to reuse methods of a superclass with the related subclasses because of the behavioral inheritance property (*principle of substitutability*) is referred to as the property of **polymorphism**. In general, an OO database schema tends to employ an unusually large number of classes.

The inheritance principle goes hand in hand with the concept of **class hierarchies**. Once again, a class hierarchy in an OO database schema is analogous to type hierarchy (Specialization/generalization) in the EER modeling grammar. Some object systems also

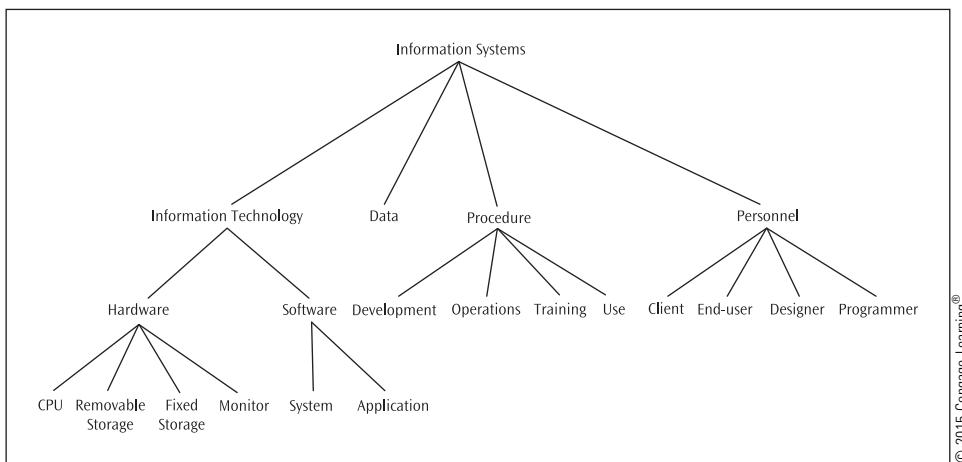
---

<sup>5</sup>In a pure sense, all instance variables are hidden from the user. While logically unnecessary, in practice, objects typically expose physical representation of some instance variables usually via some special syntax and these are called **public instance variables**. Therefore, the truly hidden instance variables are labeled **private instance variables**.

## Appendix B

support *multiple inheritance*—i.e., the ability of a subclass to inherit variables and methods from multiple superclasses. Modeled as a directed acyclic graph (DAG), this concept is similar to that of a specialization lattice in the EER modeling grammar (see Section 4.1.3 in Chapter 4).

Another related concept that is rather useful is known as *object containment*. The idea essentially conceptualizes objects as containing (i.e., being a **part of**) other objects in addition to public and private instance variables and methods. Sometimes these objects are referred to as complex objects or composite objects. Portrayal of multiple levels of containment leads to a **containment hierarchy**. Containment allows different users to view objects at different granularities. The concept essentially replicates the **aggregation** constructs in the EER modeling grammar in the OO context. Figure B.1 shows a containment hierarchy for a complex object called **Information Systems**. In this containment hierarchy, a business analyst can focus attention on the **Procedure** objects without any concern about **Information Technology**, **Data**, and **Personnel** objects. Likewise, a computer engineer can choose to limit the scope of his/her analysis to the hardware objects. The **Information Systems** manager, on the other hand, may use the containment hierarchy to monitor the whole **Information Systems**. In applications where an object is a part of several objects, the containment relationship can be portrayed as a DAG instead of a hierarchy.



© 2015 Cengage Learning®

**FIGURE B.1** Containment hierarchy in an object database

### B.1.1.4 Object Identity

In an object database every mutable object that is stored (i.e., persistent) is uniquely identified by an Object ID (OID) which is typically a system-generated (conceptual) address. As a consequence, OIDs can be used elsewhere in the database as a (conceptual) pointer (e.g., in containment hierarchies). The OID is intended for independent identification of an object in the database and for managing inter-object references. The value of

an OID is therefore meant for internal use by the database system and so is not visible to the user. The fundamental property of an OID is that it is immutable. Therefore, it is preferred that an OID value be retired when the associated object is removed from the database instead being reassigned to another object. For these reasons, the value of OIDs should not be a function of any variable in the database schema. Likewise, basing the value of an OID on a physical storage address is also discouraged. A commonly practiced strategy in object databases is the use of system-generated long integers as OID values and using an index (or hash table) to map the OID values to a physical storage address. Immutable objects like numbers and character strings usually do not have OIDs since they are typically stored within an object and cannot be referenced from outside the object. Note that OIDs do not eliminate the need for user-defined keys (e.g., candidate keys or primary key) because OIDs are not only prohibited for use in external interactions, but also are often not user-friendly means of external interaction. With respect to inter-object reference, the use of OID is somewhat similar to the use of a foreign key in a relational data model, except that an OID can point to an object anywhere in the OODBMS while a foreign key in an RDBMS is constrained to reference an attribute in a specific referenced relation. Lack of such a restriction in OODBMS imposes the responsibility for proper references on the application program. Use of OIDs for inter-object reference as in containment hierarchies is essentially equivalent to the low-level pointer mechanism originally defined in the CODASYL data model. Date (1998) asserts that OIDs have no place in the data model as far as the user is concerned.

### B.1.2 A Note on UML

With the advent of the OO approach, Unified Modeling Language (UML) is becoming increasingly popular in the software engineering discipline. The attraction of UML may be attributed to the fact that it seeks to combine data and process (method) specifications in a single unified grammar. From a non-object paradigm data modeling and database design perspective, only the class diagram of UML is of some relevance. A class diagram is similar in many ways to the EER diagram except that the class diagram has provisions to specify methods along with attribute specifications in line with the OO approach. The Object Management Group has also endorsed UML as the standard for object modeling. Since UML combines commonly accepted concepts from several OO approaches, it is applicable to almost any application domain and is also programming language and operating system platform independent. A UML class diagram for a slightly enhanced Bearcat, Inc. ER diagram (see Figure 3.3) is shown in Figure B.2. Detailed treatment of UML grammar is outside the scope of this book<sup>6</sup> and can be found in software engineering textbooks that employ the OO development approach.

---

<sup>6</sup>Rational Rose is one of the popular CASE tools for drawing UML diagrams.

## Appendix B

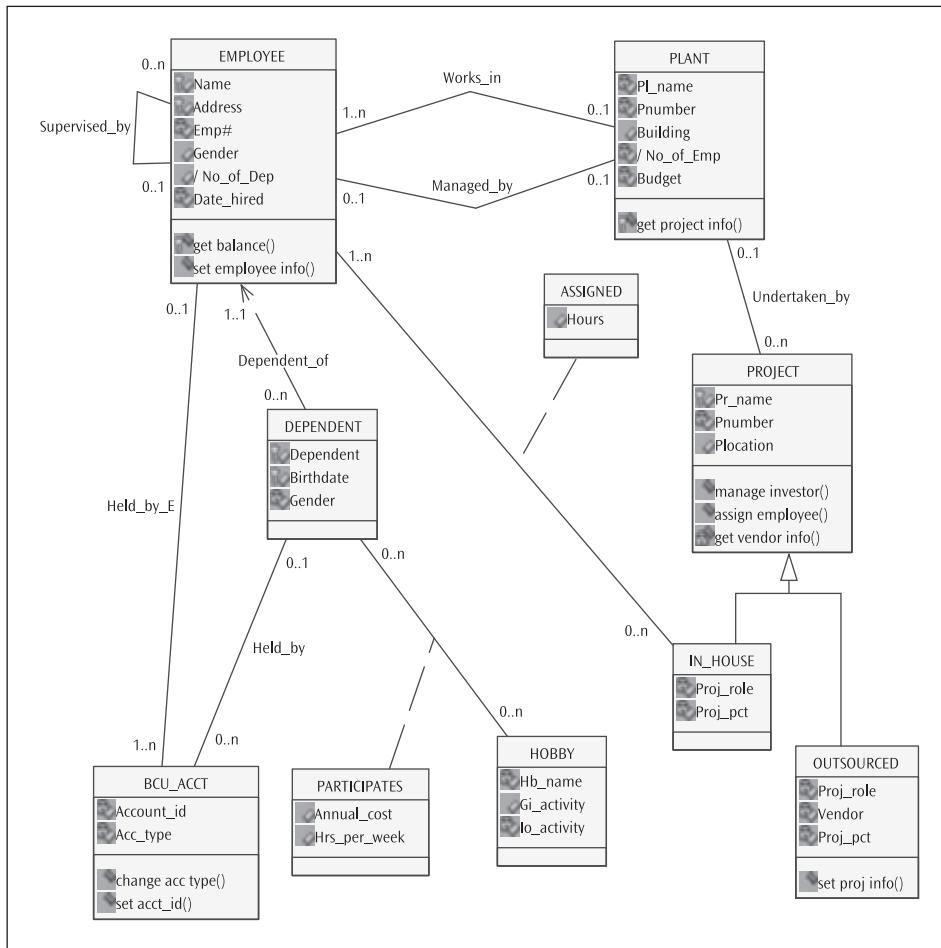


FIGURE B.2 UML class diagram for Bearcat, Inc.

In essence, OO databases are simply persistent extensions to OO programming languages. While persistent memory of programming languages may perhaps be viewed as databases, they are *application-specific*—i.e., they must first be tailored to an application, say, CAD/CAM; such a tailored DBMS is essentially useless for another application, say, a medical imaging system. OODBMS is not a shared, general purpose DBMS as a DBMS is expected to be. An OODBMS may be thought of as a sort of DBMS construction kit (Date, 2004, p. 846), while a second generation DBMS (e.g., RDBMS), once installed, is ready for use to build databases for various applications. There are other issues pertaining to *ad hoc* queries, treatment of derived objects, specification and enforcement of declarative integrity constraints that are application-independent, data independence and view definitions, application-independent database design, performance considerations, etc., where an object database may be inadequate. For a thoughtful treatise of these issues, see Date (2004).

## B.2 THE OBJECT-RELATIONAL DATA MODEL

In this section, we present a brief overview of the conceptual foundation for a class of DBMS called the object-relational system. OO databases discussed in the previous section are simply persistent extensions to OO programming languages like C++, SMALLTALK, and Java. In contrast, object-relational data models seek to extend the relational data model to support complex data types and other OO features. Proposing the object-relational data model, Stonebraker (1990) asserted that “second-generation systems made a major contribution in two areas, non-procedural data access and data independence, and these advances must not be compromised by third-generation systems.” The underlying idea is that third-generation DBMSs should subsume rather than displace their second-generation counterparts lest the proven features of the second-generation DBMS be lost forever.

We briefly mentioned the problems associated with OODBMS (e.g., *ad hoc* queries, treatment of derived objects, specification and enforcement of declarative integrity constraints that are application-independent, data independence and view definitions, application-independent database design, performance considerations) in the last section. The principal motivation behind object-relational database systems (ORDBMS) is one of extending the RDBMS to support OO features without sacrificing the sound theoretical foundation of the RDBMS. OO features like (1) user-defined **abstract data types** (ADTs) that store complex data types and encapsulate methods for processing these data types, (2) inheritance, and (3) **structured types** where an attribute can capture more than an atomic value (e.g., sets, tuples, arrays, and sequences) when incorporated as extensions to the RDBMS are expected to overcome the current limitations of the RDBMS in supporting certain types of applications. Structured data types enable a *nested relational data model* by allowing domains to have non-atomic values. That is, an attribute value in a tuple can be a relation and that relation may contain other relations essentially enabling a nesting of relation schemas.

Several DBMS vendors have released object-relational DBMS products, also known as *universal servers* (e.g., IBM’s universal database version of DB2, universal data option of Informix Dynamic Server, Oracle Universal Server). Sql:1999 has made significant strides towards incorporating advanced OO features while preserving the relational foundation of declarative access to data. For instance, SQL:1999, while including new data types for large objects and ARRAY type constructors, also enforces referential integrity on OIDs in the object-relational data model—i.e., all the OIDs that appear in an attribute of a relational schema in the object-relational data model must reference the same target relation schema. ORDBMS offers a convenient migration path for current users of RDBMS and affords the flexibility of using OO features selectively for appropriate applications.

## Summary

---

The need to handle complex data types and to represent an object as consisting of both the data structure and set of operations that can be used to manipulate it (i.e., the concept of encapsulation) led to the creation of the object-oriented data model, which is intricately woven with the OO programming languages. In fact, OODBMS in principle is simply equivalent to adding persistence to OO programming language. The object-relational data model incorporating selected OO constructs as an extension to the relational data model has been proposed as an alternative to combat this problem. This has triggered a debate between proponents of the object-oriented data model and the relational data model. Both sides agree that the relational model is capable of supporting standard business applications, but lacks the capability to support special applications using complex data types. They, however, disagree as to whether extensions to the relational model can overcome this limitation. Proponents of the relational data model claim that the relational data model is a necessary part of any database management system and believe that extensions to the relational model (i.e., the object-relational data model) can address its limitations by effectively incorporating OO constructs as an extension to the theoretically sound relational data model.

## Selected Bibliography

---

- Date, C. J. (2004) *An Introduction to Database Systems*, Eighth Edition, Addison-Wesley.
- Elmasri, R. and Navathe, S. B. (2003) *Fundamentals of Database Systems*, Fourth Edition, Addison-Wesley.
- Ramakrishnan, R. and Gehrke, J (2000) *Database Management Systems*, McGraw Hill.
- Stonebraker, M. et al. (1990) "Third-Generation Database System Manifesto," *ACM SIGMOD Record*, 19, 3.

# SELECTED BIBLIOGRAPHY

- Abrial, J. "Data Semantics." *Data Base Management*. Eds. J. W. Klimbie and K. L. Koffeman. Amsterdam: North-Holland, 1974.
- Armstrong, W. W. "Dependence Structures of Data Base Relationships." *Proceedings of IFIP Congress*, Stockholm, Sweden, 1974.
- Batini, C., M. Lenzerini, and S. B. Navathe. "A Comparative Analysis of Methodologies for Database Schema Integration." *ACM Computing Surveys*, 8, 4 (December) 323–364, 1986.
- Batini, C., S. Ceri, and S. B. Navathe. *Conceptual Database Design: An Entity-Relationship Approach*. Boston: Addison-Wesley, 1991.
- Booch, G., J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*, Second Edition. Boston: Addison-Wesley, 2005.
- Catriel, B., R. Fagin, and J. H. Howard. "A Complete Axiomatization for Functional and Multi-valued Dependencies." *Proceedings of the ACM/SIGMOD International Conference on Management of Data*, Toronto, Canada (August) 1977.
- Chen, P. "The Entity Relationship Model: Toward a Unified View of Data," *ACM Transactions on Database Systems*, 1 (March) 9–36, 1976.
- Chen, P. "The Entity-Relationship Approach to Logical Database Design." *Information Technology in Action: Trends and Perspectives*. Ed. R. Y. Wang. Upper Saddle River, NJ: Prentice Hall, 1993.
- Chiang, R. H. L., T. M. Barron, and V. C. Storey. "Reverse Engineering of Relational Databases: Extraction of an ER Model from a Relational Database" *Data & Knowledge Engineering*, 12, 107–142, 1994.
- Chiang, R. H. L., T. M. Barron, and V. C. Storey. "A Framework for the Design and Evaluation of Reverse Engineering Methods for Relational Databases." *Data & Knowledge Engineering*, 21, 55–77, 1997.
- Codd, E. F. "A Relational Model for Large Shared Data Banks." *Communications of the ACM*, 13, 6 (June) 377–387, 1970.
- Connolly, T. M., and C. Begg. *Database Systems: A Practical Approach to Design, Implementation, and Management*, 4th Edition. Boston: Addison-Wesley, 2005.
- Courtney, J. F., and D. B. Paradice. *Database Systems for Management*. St. Louis: Times Mirror/Mosby College Publishing, 1988.
- Darwen, H. "The Role of Functional Dependencies in Query Decomposition." In C. J. Date and H. Darwen, *Relational Database Writings 1989–1991*, 133–154. Boston: Addison-Wesley, 1992.
- Date, C. J. *An Introduction to Database Systems*, 8th Edition. Boston: Addison-Wesley, 2004.
- Date, C. J., and H. Darwen. *A Guide to the SQL Standard*, Fourth Edition. Boston: Addison-Wesley, 1997.
- Date, C. J., and H. Darwen. *Foundation for Object/Relational Databases*. Boston: Addison-Wesley, 1998.
- Dey, D., V. C. Storey, and T. M. Barron. "Improving Database Design through the Analysis of Relationships." *ACM Transactions on Database Systems*, 24, 4 (December) 453–486, 1999.
- Elmasri, R., and S. B. Navathe. *Fundamentals of Database Systems*, Sixth Edition. Boston: Addison-Wesley, 2010.
- Everest, G. C. *Database Management Objectives, System Functions and Administration*. New York: McGraw-Hill, 1986.
- Fagin, R. "Normal Forms and Relational Database Operators." *Proceedings of the ACM/SIGMOD International Conference on Management of Data*, (May/June) 1979.

## Selected Bibliography

- Fagin, R. "A Normal Form for Relational Databases that is Based on Domains and Keys." *ACM Transactions on Database Systems*, 6, 3 (September) 1981.
- Fahrner, C., and G. Vossen. "A Survey of Database Design Transformations Based on the Entity-Relationship Model." *Data & Knowledge Engineering*, 15, 213–250, 1995.
- Gennick, J. *SQL Pocket Guide*. Sebastopol, CA: O'Reilly Media, Inc., 2004.
- Groff, J. R., and P. N. Weinberg. *SQL: The Complete Reference*, Second Edition. New York: McGraw-Hill/Osborne, 2002.
- Gulutzan, P., and T. Pelzer. *SQL-99 Complete, Really*. Lawrence, KS: R&D Books, 1999.
- Hansen, G. W., and J. V. Hansen. *Database Management and Design*. Upper Saddle River, NJ: Prentice Hall, 1992.
- Hawryszkiewycz, I. T. *Database Analysis and Design*, Second Edition. London: Macmillan Publishing Company, 1991.
- Hoffer, J. A., M. B. Prescott, and F. R. McFadden. *Modern Database Management*, Sixth Edition. Upper Saddle River, NJ: Prentice-Hall, 2002.
- Howe, D. R. *Data Analysis for Data Base Design*, Second Edition. Edward Arnold, 1989.
- Johnson, J. L. *Database: Models, Languages, Design*. New York: Oxford University Press, 1997.
- Kent, W. "A Simple Guide to the Five Normal Forms in Relational Database Theory." *Communications of the ACM*, 26, 2 (February) 120–125, 1983.
- Kifer, M., A. Bernstein, and P. M. Lewis. *Database Systems: An Application-Oriented Approach*, Second Edition. Boston: Addison-Wesley, 2005a.
- Kifer, M., A. Bernstein, and P. M. Lewis. *Databases and Transactions Processing: An Application-Oriented Approach*, Second Edition. Boston: Addison-Wesley, 2005b.
- Kim, Y., and S. T. March. "Comparing Data Modeling Formalisms." *Communications of the ACM*, 38, 6 (June) 103–112, 1995.
- King, K. *SQL Tips & Techniques*. Portland, OR: Premier Press, 2002.
- Kroenke, D. M. *Database Processing: Fundamentals, Design, and Implementation*, Ninth Edition. Upper Saddle River, NJ: Prentice Hall, 2004.
- Loney, K. *ORACLE DATABASE 10g: The Complete Reference*. Boston: McGraw-Hill/Osborne, 2004.
- Lorentz, A. C., and J. N. Morgan. *Database Systems: Concepts, Management and Applications*. Fort Worth: Dryden Press, 1998.
- Lorie, R. A., and J. J. Daudenarde. *SQL & Its Applications*. Upper Saddle River, NJ: Prentice-Hall, 1991.
- Mannila, H., and K. Raiha. *The Design of Relational Databases*. Boston: Addison-Wesley, 1992.
- Markowitz, V. M., and A. Shoshani. "Representing Extended Entity-Relationship Structures in Relational Databases." *ACM Transactions on Database Systems*, 17, 423–464, 1992.
- Martin, J. *Principles of Data-Base Management*. Upper Saddle River, NJ: Prentice Hall, 1976.
- Morris-Murphy, L. L. *Oracle9i: SQL with an Introduction to PL/SQL*. Brooks/Cole: Course Technology, 2003.
- Navathe, S. B. "Evolution of Data Modeling for Databases." *Communications of the ACM*, 35, 9 (September) 112–123, 1992.
- Nolan, R. L. "Computer Data Bases: The Future is Now." *Harvard Business Review*, 51, 4 (September–October) 98–114, 1973.
- Pratt, P. J., and J. J. Adamski. *Database Systems Management and Design*, Third Edition. New York: Boyd & Fraser Publishing Company, 1994.
- Premelani, W. J., and M. R. Blaha. "An Approach to Reverse Engineering of Relational Databases." *Communications of the ACM*, 37, 5 (May) 42–49, 1994.
- Price, J. *ORACLE DATABASE 10g: SQL*. New York: McGraw-Hill/Osborne, 2004.
- Ram, S. "Deriving Functional Dependencies from the Entity Relationship Model." *Communications of the ACM*, 38, 9 (September) 95–107, 1995.

- Ramakrishnan, R., and J. Gehrke. *Database Management Systems*. New York: McGraw-Hill, 2000.
- Rischert, A. *Oracle SQL: By Example*. Upper Saddle River, NJ: Prentice-Hall, 2010.
- Rob, P., and C. Coronel. *Database Systems: Design, Implementation, and Management*, Seventh Edition. Brooks/Cole: Course Technology, 2006.
- Shepherd, J. C. *Database Management: Theory and Application*. Boston: Richard D. Irwin, Inc., 1990.
- Silberschatz, A., H. F. Korth, and S. Sudarshan. *Database Systems Concepts*, Sixth Edition. New York: McGraw-Hill, 2010.
- Smith, J. M., and D. C. P. Smith. "Data Abstractions: Aggregation and Generalization." *ACM Transactions on Database Systems*, 2 (June) 105–133, 1977.
- Song, I., M. Evans, and E. K. Park. "A Comparative Analysis of Entity-Relationship Diagrams." *Journal of Computer & Software Engineering*, 3, 4, 427–459, 1995.
- Sunderraman, R. *Oracle9i Programming: A Primer*. Boston: Addison-Wesley, 2003.
- Teory, T. J. *Database Modeling and Design*, 5th Edition. Burlington, MA: Morgan Kaufman, 2011.
- Teorey, T. J., D. Yang, and J. P. Fry. "A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model." *Computing Surveys*, 18, 2 (June) 197–222, 1986.
- Turban, E., E. McLean, and J. Wetherbe. *Information Technology for Management*. Hoboken, NJ: John Wiley & Sons, Inc., 1996.
- Wand, Y., and R. Weber. "Research Commentary: Information Systems and Conceptual Modeling—A Research Agenda." *Information Systems Research*, 13, 4, 363–376, 2002.
- Waters, R. C., and E. Chikofsky. "Reverse Engineering: A Brief Summary of the Papers Presented at the May 1993 ACM/IEEE Computer Society's Working Conference on Reverse Engineering." *Communications of the ACM*, 37, 5 (May) 22–25, 1994.



# INDEX

## A

abstract data types (ADTs), 737  
access  
    direct, 3, 8  
    RAID, 22  
    sequential, 3  
aggregate entity type, 210  
Aggregate Function and grouping, 561–562  
aggregate functions (SQL), 581, 628–630  
aggregation in EER modeling, 166–168  
aggregations, mapping, 326–327  
ALL operators in SQL subqueries, 615–621  
algorithm for non-loss 3NF, 436–442  
ALTER TABLE statement (SQL), 517–519, 532  
American National Standards Institute (ANSI), 6–10, 24  
analysis approach, 85–86  
analytical functions  
    DENSE\_RANK () functions, 684–687  
    RANK () functions, 684–687  
    ROLLUP, CUBE, and GROUPING SETS  
        operators and, 687–688  
    syntax of, 681  
    types, 682  
AND keyword (SQL), 696–697  
AND logical operator, 571, 572  
anomalies, deletion, insertion and update, 400, 402  
ANSI/SPARC three-schema architecture, 6–10, 24  
ANY function (SQL), 697–698  
ANY operators in SQL subqueries, 615–621  
application-based constraints, 284  
application-specific databases, 736  
architecture  
    ANSI/SPARC three-schema architecture, 6–10, 24

based on inverted tree, network data structures, 719–728  
of file-processing systems, 8–9  
arities in relational data model, 281  
Armstrong's axioms, 356, 358, 366–368, 390  
AS keyword (SQL), 576n  
associations, objects, and relationships, 31–32  
asterisks (\*) in SQL statements, 570  
atomic attributes, 33  
attribute-defined specialization, 149  
attribute preservation, 407  
attributes  
    closure of set of, and FDs, 372–373  
    defining, 149  
    entity types and, 32–34, 70  
    and functional dependencies (FDs), 365–366  
    key, 386  
    prime, non-prime, 386–389  
    of relationship types, 96  
    representing in ER diagrams (ERDs), 85  
    in SQL, 567n  
    and weak entity types, 53

## B

base entity types and weak entity types, 52–57  
BCNF relational scheme, normalization, 467–498  
Bearcat Incorporated (case study)  
    data requirements, EER model, 170–181  
    ER modeling for, 79–81  
    fine-granular design-specific ER model, 309–315  
    presentation layer ER model for, 86–104  
behavioral inheritance, 733  
BETWEEN keyword (SQL), 696–697  
BETWEEN operator (SQL), 577  
binary decomposition, 412

## Index

binary operations, 293–295  
 binary operators  
     Cartesian Product operator, 546–549  
     Divide operator, 557–560  
     join operators, 551–557  
     in relational algebra, 539  
     set theoretic operators, 549–551  
     SQL queries based on, 597–613  
 binary relationship type, 38–42, 198  
 Boyce-Codd normal form (BCNF), 396,  
     404–407, 414–418, 457  
 bridge entity type, 115  
 business rules  
     and data modeling errors, 119–133  
     described, 18, 35, 70  
     and functional dependencies (FDs), 374  
     user-specified, 19

## C

candidate keys  
     of relation schema, deriving, 374–386, 390  
     and unique identifiers, 286, 287  
 canonical cover, 368–369  
 cardinality  
     constraint for relationship types, 43–51  
     in relational data model, 282  
 cardinal ratio, notation, 83  
 Cartesian Product operator, 546–549  
 cascade rule, 59  
 CASE expressions, 647–649  
 case sensitivity, SQL, 570  
 CASE (computer-aided engineering) tools,  
     82–83  
 CASE (computer-aided software engineering)  
     tools, 15  
 categorization in superclass/subclass  
     relationships, 157–160, 162–165, 188  
 CHAR\_LENGTH (char) function (SQL),  
     639–640  
 chasm traps, 251–253  
 Chen, Peter, 30, 83  
 child nodes, 657  
 classes  
     entity, 31–32  
     object, 143–144

class hierarchies, 733  
 cluster entity type, 57–58, 70, 205–224,  
     364  
     decomposing, 240–241  
 clustering recursive relationship type,  
     221–224  
 CODASYL (Conference on Data Systems  
     Languages) data model, 726–728  
     schemas and, 726–728  
     subschemas and, 726  
 Codd, E. F., 563  
 collective type inheritance, 166  
 comparison operators, 575, 615  
 completeness constraints, 148  
 complex relationships modeling  
     beyond ternary relationship types,  
     205–224  
     composites of weak relationship types,  
     230–233  
 Cougar Medical Associated (case study),  
     257–272  
 decomposition of complex relationship  
     constructs, 234–245  
 inter-relationship integrity constraint,  
     224–230  
 introduction, 197  
 mapping, 336–344  
 ternary relationship type, 198–205  
 validating conceptual design, 246–257  
 weak relationship type, 224–230  
 composite  
     attributes, 33, 96  
     entity type, 115  
     relationship type, inclusion, exclusion  
     dependencies in, 230–233  
 computer-aided software engineering (CASE)  
     tools, 15, 82–83  
 conceptual data modeling, 19–21  
     applying ER modeling grammar to,  
     81–82  
     and data modeling errors, 119–133  
     ER modeling grammar, 32–69  
     framework, ER modeling primitives, 30–32  
     introduction, 27–29  
     validating conceptual design, EER,  
     246–257

conceptual design validation, EER, 246–257  
chasm traps, 251–253  
fan traps, 246–251  
introduction, 246  
miscellaneous semantic traps, 253–257  
conceptual schema of metadata in databases, 6–8  
condition-defined subclasses, 148  
condition precedence sequence in weak relationships, 226  
CONNECT BY clause (SQL), 658–659  
connection traps  
    chasm traps, 251–253  
    described, 246  
    fan traps, 246–251  
connectors, superclass, 148, 151  
constraints. *See also specific constraint*  
    cardinality, 43–45  
    completeness, 148  
    declarative, 284  
    deletion, 58–69  
    disjointness, 148  
    domains, 36–37  
    entity integrity, 288  
    foreign key, 290  
    integrity, 35–36  
    referential integrity, 290  
    state, 284  
    structural, of relationship types, 43–51  
containment hierarchy, 734  
context in conceptual data model, 28  
controlled redundancy, 362, 401, 407  
converting presentation layer ER model to EER diagram, 168–170  
Cougar Medical Associated (case study), 257–272  
    conceptual model for, 259–266  
    described, 257–259  
    design-specific ER model for, 266–272  
CREATE TABLE statement (SQL), 507–517  
CROSS keyword (SQL), 597  
Crow's Foot notation, 83  
CUBE operator, 672–673  
CURRENT\_DATE function (SQL), 652  
CURRENTV () function, 698–699

**D**

data  
    described, 1–2  
    dictionaries described, 2, 12–15  
    independence. *See data independence*  
    management. *See data management*  
    redundancy, and database normalization, 355–357, 360–362, 395–396, 422, 457  
warehouse, 12  
database creation  
    base table specification in SQL/DDL, 507–524  
    data definition using SQL, 507  
    data population using SQL, 524–531  
database description (DBD), 724  
database design  
    characteristics of database systems, 10–17  
    data models and, 17–19  
    normalization. *See normalization*  
    relational data model. *See relational data model*  
database management systems (DBMSs)  
    defined, 10  
    described, 12–15  
databases  
    application-specific, 736  
    creation. *See database creation*  
    described, 10  
    designing. *See database design*  
    normalization. *See normalization*  
data control languages (DCLs), 12  
data definition language (DDL), 12  
data independence  
    defined, 8–9  
    logical, 8  
    physical, 8  
data integrity  
    database system and, 16  
    in file-processing systems, 4  
    and relational data model, 283–291  
Data Language/1 (DL/1), 724  
data management, described, 3  
data manipulation languages (DMLs), 12–13

## Index

- data modeling
  - architectures based on inverted tree, network data structures, 719–728
  - conceptual. *See* conceptual data modeling
  - logical. *See* logical data modeling
  - physical, 503
  - reverse engineering, role of, 442–456
- data modeling errors, 119–133
- data models
  - CODASYL (Conference on Data Systems Languages), 726–728
  - and database design, 17–19
  - described, 17
  - hierarchical, 722–726
  - logical, 19–21, 21–22
  - nested relational, 737
  - object-oriented (OO), 731–736
  - object-relational, 737
  - physical, 22–24
- data sets described, 3
- data types
  - abstract, 737
  - described, 32–33
- dates and times, handling in SQL, 651–656
- DBMS. *See* database management systems
- DCLs (data control languages), 12
- DDBMS (distributed database management system), 12
- DDBs (distributed databases), 12
- DDL (data definition language), 12
- declarative constraints, 284
- DECODE function (SQL), 646–647
- decomposed design-specific ER model, 111–119
  - final model, 117–119
  - resolution of m:n relationship types, 115–116
  - resolution of multi-valued attributes, 112–115
- decomposition
  - binary, 412
  - of cluster entity type, 240–241
  - of complex relationship constructs, 234–245
  - deriving candidate keys by, 379–383
  - lossless-join, 408, 414–418
- of recursive relationship types, 241–243
- of relationship type with multi-valued attribute, 235–240
- of ternary and higher-order relationship types, 234–235
- of weak relationship type, 244–245
  - when complete, 422–423
- defining attributes, 149
- defining condition, 148
- defining predicate, 148
- degree of relationship types, 38
- degrees in relational data model, 281
- DELETE statement (SQL), 528–529, 532
- deletion anomaly, 361, 400, 402
- deletion constraint, 58–69, 97–104
  - for intra-entity class relationships, 182–187
- denormalization, 442–443
- DENSE\_RANK () functions, 684–687
- dependencies
  - exclusion, 228, 231–233
  - existence, 46, 54
  - functional. *See* functional dependencies
  - inclusion, 225
  - join, and 5NF, 480–497
  - multi-valued dependency (MVD), 467–472
- dependency preservation, 408–411, 414–418
- derived attributes, 33, 95
- design, database. *See* database design
- Design-Specific ERD model, 563
- design-specific ER model, 104–111
  - and Cougar Medical Associated (case study), 266–272
  - decomposed, 111–119
  - final, 117–119
- desktop database systems, 11
- diagrams
  - EER, 168–170
  - ER. *See* ER diagrams
  - instance, 40
- dictionary, data, 2, 12–15
- difference operations, 294–295
- direct access, 3, 8
- directed acyclic graph (DAG), 734
- disjointness constraints, 148
- DISTINCT qualifier (SQL), 579

distributed database management system (DDBMS), 12  
 distributed databases (DDBs), 12  
 Divide operator, 557–560  
 Division operations (SQL), 649–651  
 DMLs (data manipulation languages), 12–13  
 domain constraints, 497  
 domain-key normal form (DK/NF), 497  
 domains  
     constraints, 36–37  
     entities, and attributes, 31  
 DROP TABLE statement (SQL), 519–524, 532

## E

EER diagrams, converting presentation layer to, 168–170  
 EER (enhanced entity-relationship) modeling.  
*See also* ER modeling  
 aggregation, 166–168  
 choosing appropriate construct, 160–165  
 information-preserving grammar for constructs, 328–336  
 specialization, generalization, and categorization, 145–154  
 superclass/subclass relationships, 142–143  
 elements, data, 2–3  
 encapsulation, object-oriented programming, 732  
 enhanced entity-relationship modeling.  
*See* EER modeling  
 enterprise database systems, 11–12  
 entity classes, 31–32, 143–145  
 entity integrity constraints, 288  
 entity relationship. *See* ER  
 entity sets described, 31  
 entity types. *See also specific entity type*  
     and attributes, 32–34, 70  
 bridge, 115  
 cluster, 57–58, 70  
 composite, 115  
 described, 358, 374  
 gerund, 115  
 mapping, 298–299  
 Equijoin operations, 602–603  
 Equijoin operator, 552–553

ER diagrams (ERDs)  
     and complex relationships, 197  
     and connection traps, 246–257  
     described, 81–82  
     development of, 86–96  
     mapping deletion rules to, 97–104  
 ER (entity relationship) modeling. *See also* EER modeling  
     applying ER modeling grammar to conceptual modeling process, 81–82  
 Bearcat Incorporated (case study), 79–81  
 decomposed design-specific, 111–119  
 described, 19–21, 30, 70  
 design-specific, 104–111  
 modeling grammar, 32–69  
 presentation layer ER model, 82–85  
 primitives, 30–32  
 errors  
     conceptual data modeling, 119–133  
     connection traps, 246–257  
 event procedure sequence in weak relationships, 226  
 EXCEPT keyword (SQL), 601  
 exclusion dependencies, 228, 231–233  
 existence dependencies, 46, 54  
 external schema of metadata in databases, 6–8

## F

fan (connection) traps, 246–251  
 fast-track algorithm, 436–442  
 fifth normal form (5NF), 480–497  
 file-processing systems  
     vs. database systems, 11–12  
     limitations of, 3–6  
     as two-schema architecture, 9  
 files in data management, 3  
 final design-specific ER model, 117–119  
 fine-granular design-specific ER model, 309–315  
 first normal form (1NF), 396–398  
 foreign key constraints, 290  
 FOR loop (SQL), 699  
 forms, normal. *See* normalization  
 fourth normal form (4NF), 472–479

## Index

Full Outer Join operations, 556–557, 611–612  
 functional dependencies (FDs)  
     closure of set of attributes, 372–373  
     deriving candidate keys, 374–386  
     described, 365, 390  
     desirable and undesirable, 395–396, 457  
     interference rules, minimal cover for, 359–372  
     normalization. *See* normalization  
 functions. *See also specific function*  
 SQL-2003 built-in, 635–651

## G

generalization  
     and specialization, 157–160  
     and specialization in superclass/subclass relationships, 146–154  
 gerund entity type, 115  
 Get Well Pharmacists, Inc. (case study),  
     modeling complex relationships, 203–205  
 grammar in conceptual data model, 28, 81–82  
 GROUP BY clause (SQL), 581, 676–677, 679–681  
 grouping  
     aggregate functions and, 561–562, 628–630  
     rows in SQL, 580–583  
 GROUPING () function, 674–676  
 GROUPING\_ID (), 677–680  
 GROUPING SETS extension, 676–677

## H

‘Has-a’ relationship, 143  
 HAVING clause (SQL), 582  
 hierarchical data model, 722–726  
 hierarchical queries, 656–668, 706  
 hierarchies  
     aggregation, 166–168  
     specialization/generalization, 154–157  
 higher-order relationship types, decomposing, 234–235

## I

identifiers, unique  
     of entity types, 37–38  
     in relational data model, 284–289, 345  
 immutable objects, 733  
 inclusion dependency, 225, 230–231  
 inference rules for functional dependencies (FDs), 366–367  
 information  
     described, 1–2  
     preservation issue in relational data model, 297–298  
 Information Resource Directory Systems (IRDSs), 30  
 inheritance  
     class, 733–734  
     collective type, 166  
     multiple type, 156  
     selective type, of category, 159  
     type inheritance property, 146  
 IN keyword (SQL), 575  
 Inner Join operations, 555–556  
 IN operator (SQL), 614–615  
 insertion anomaly, 360, 400, 402  
 INSERT statement (SQL), 525–527, 532  
 instance  
     diagrams, 40  
     relation, 359, 366  
 instances, object, 733  
 INSTR function (SQL), 645  
 integration, data, in file-processing systems, 5, 11, 16  
 integrity  
     constraints, 35–36, 284  
     data, 4, 16  
 inter-entity class relationship, 143  
 interference rules for multi-valued dependency, 470–472  
 internal schema of metadata in databases, 8–9  
 inter-relationship integrity constraint, 224–230  
 intersection operations, 294–295  
 intra-entity class (Is-a), 157–158  
 intra-entity class relationships, 143–145  
     deletion constraint for, 182–187

inverted tree structures, data modeling  
architectures based on, 719–728  
IRDSs (Information Resource Directory Systems), 30  
IS ANY function (SQL), 697–698  
‘Is-a-part-of’ relationships, 166

**J**

join dependencies (JDs) and 5NF, 480–497  
join operations  
examples of, 602–613  
natural join operation, 295  
and operators, 551–557  
join operators, 551–557

**K**

key attributes, 37, 386  
key constraints, 497

**L**

lattice, specialization, 156  
leaf nodes, 657  
Left Outer Join operations, 555–556, 608–610  
LEVEL pseudo-column (SQL), 660  
life cycle  
database design, 19–24  
data modeling/database design (fig.), 18  
LIKE operator, 593, 596  
logical  
data independence, 8  
data modeling, 21–22, 722–728  
data structures, 719–721  
data types, 33  
logical data modeling  
introduction, 277–279  
relational data model. *See relational data model*  
logical data structures, 719–721  
inverted tree structure, 719–720  
network data structure, 721  
logical schema, mapping ER model to, 298–315

lossless-join  
decomposition, 408, 414–418  
property, 411–414  
LPAD() function, 661  
LTRIM function (SQL), 640–643

**M**

Madeira College (case study), modeling  
complex relationships, 198–203, 216–221  
maintenance of file-processing systems, 5  
mandatory attributes, 34, 95  
mapping  
aggregations, 326–327  
complex ER model, 336–344  
enhanced ER model constructs to logical schema, 321–328  
entity types, 298–299  
ER model to logical schema, 298–315  
information-preserving, relational data model, 315–320  
specialization, specialization hierarchy, lattice, 321–326  
materialized views in relational data model, 296–297  
MAX function (SQL), 622–623  
metadata described, 2  
methods in conceptual data model, 28  
min, max notation, for structural constraints, 104–111  
MIN function (SQL), 622–623  
minimal cover, 368–369  
MINUS keyword (SQL), 601  
m:n relationship types, resolution of, 115–116  
MODEL clause (SQL), 706  
basic syntax of, 693–694  
concepts, 693  
example of, 694–699  
modeling  
complex relationships, 197  
conceptual data. *See conceptual data modeling*  
data. *See data modeling*  
ER (entity relationship). *See ER modeling*

## Index

models  
 data, 17–24  
 ER (entity relationship), 19–21  
 relational data. *See* relational data model  
 modification anomalies, 361  
 multiple inheritance, 156, 734  
 multi-valued attributes, 33–34, 95  
     decomposition of, 235–240  
     resolution of, 112–115  
 multi-valued dependency (MVD), 467–472, 478–479, 498  
 mutable objects, 733  
 MySQL built-in functions, 635–651

## N

names, role, and entity types, 42  
 Natural Join operations, 603–608  
 Natural Join operator, 553–554  
 nested relational data model, 737  
 network data structure, 721  
 NF (normal forms). *See* normalization  
 nodes, 657  
 non-key attributes, 37  
 non-prime attributes, 386–389  
 normalization  
     Boyce-Codd normal form (BCNF), 404–407  
     comprehensive approach to, 424–442  
     and data redundancy, 355–357  
     and denormalization, 442–443  
     described, 363, 390, 395–396, 457  
     domain-key normal form (DK/NF), 497  
     fifth normal form (5NF), 480–497  
     first normal form (1NF), 396–398  
     fourth normal form (4NF), 472–478  
     higher normal forms, 467  
     multi-valued dependency (MVD), 467–472  
     normal forms summary, 418–419  
     second normal form (2NF), 398–401  
     side effects of, 407–418  
     third normal form (3NF), 401–404  
 notation for ER diagrams, 82–85  
 NOT IN keyword (SQL), 575, 577  
 NOT IN operator (SQL), 614–615

nulls in relational data model, 281  
 null values, handling in SQL, 583–592

## O

object classes, 733  
 entity classes as, 143  
 object containment, 734  
 Object Definition Language (ODL), 733  
 Object IDs (OIDs), 734–735  
 object instances, 733  
 Object Management Group (OMG), 733, 735  
 object-oriented (OO) data model, 722, 731–736  
 Object Query Language (OQL), 733  
 object-relational data model, 722, 737  
 object-relational DBMS products, 737  
 objects  
     complex, 734  
     composite, 734  
     defined, 732  
     immutable, 733  
     mutable, 733  
     structure, 733  
 object types and ER modeling primitives, 30–31  
 ODL (Object Definition Language), 733  
 operators. *See also specific operator*  
     binary. *See* binary operators  
     comparison, 575, 615  
     CUBE, 672–673  
     PRIOR, 658–659  
     in relational algebra generally, 563  
     ROLLUP, 668–672  
     SQL precedence, 573  
 optional attributes, 34  
 OQL (Object Query Language), 733  
 Oracle SQL, built-in functions, 635–651  
 ORDER\_BY\_CLAUSE (SQL), 681–682  
 OR logical operator, 571, 572  
 Outer Join operations, 555–557, 608–612

## P

parent-child relationship (PCR)  
 described, 44  
 and inverted tree structure, 720

parent nodes, 657  
 partial category, 159  
 partial dependencies, 398  
 partial keys, 96  
 partial specialization, 148  
 participation constraint  
     cardinality and, 46  
     for entity types, 45  
 pattern matching in SQL, 593–597  
 PCR (parent-child relationship), 44  
 percent sign (%)  
     in SELECT statement, 593–594  
     in SQL statements, 596  
 physical  
     data independence, 8  
     data modeling, 22–24  
 polymorphism, object-oriented programming, 733  
 POSITION function (SQL), 644–645  
 predicate-defined subclasses, 148  
 presentation layer ER model, 82–85,  
     86–104  
     Bearcat Incorporated (case study),  
         170–181  
     and design-specific ER model, 266–272  
 preservation, attribute and dependency, 407  
 primary keys  
     and prime attributes, 387–389  
     in relational data model, 288  
 prime attributes, 386–389  
 primitives, ER modeling, 30–32  
 principle of substitutability, 733  
 PRIOR operator, 658–659  
 procedure-defined specialization, 149  
 program-data independence  
     in database systems, 16–17  
     in file-processing systems, 5–6  
 Projection operation, 292–293  
 Project operator, 544–545  
 properties of superclass/subclass relationships,  
     145–146

**Q**

quaternary relationship type, 38–42, 212  
 queries, evaluating SQL, 568

QUERY\_PARTITIONING\_CLAUSE (SQL),  
     681  
 quotation marks (“”) in SQL statements, 572

**R**

RANK () functions, 684–687  
 RDBMSs (relational database management  
     solutions) and SQL, 504–505  
 records in data management, 3  
 record types described, 2–3  
 recursive relationships, 48, 221–224  
     decomposing, 241–243  
 redundancy, data, 355–357, 360–362,  
     395–396  
 referential integrity constraints, 290  
 relation, relation schema, 282–283  
 relational algebra  
     basic operators (table), 560  
     binary operations, 293–295  
     described, 280  
     natural join operation, 295  
     and relational data model, 563  
     unary operations, 292–293  
 relational data model  
     characteristics of a relation, 282–283  
     database implementation using, 503–505  
     data integrity, 283–291  
     described, 280, 345  
     information preservation issues, 297–298  
     information-preserving grammar for  
         enhanced ER modeling constructs,  
         328–336  
     information-preserving mapping, 315–320  
     logical data model architectures, 722–728  
     mapping enhanced ER model constructs to  
         logical schema, 321–328  
     mapping ER model to logical schema,  
         298–315  
     relational algebra, 291–295  
     views and materialized views in, 296–297  
 relation instance, 359, 366  
 relations, in SQL, 507  
 relation schemas  
     deriving candidate keys, 374–386  
     described, 397

## Index

relationships. *See also specific relationship*  
 complex. *See complex relationships*  
 in conceptual data modeling, 31  
 (min, max) notation for the structural  
 constraints of, 104–111  
 parent-child relationship (PCR), 44  
 relationship sets described, 39  
 relationship types  
   attributes of, 96  
   described, 38–42, 70  
   mapping, 300–309  
   resolution of m:n, 115–116  
   structural constraints of, 43–51  
 REPTS\_TO column (SQL), 657  
 requirements specification in database  
   design, 18–21, 35  
 resolution of m:n relationship types, 115–116  
 restrict rule, 59  
 reverse engineering role in data modeling,  
   442–456, 458  
 Right Outer Join operations, 555–556,  
   610–611  
 roles, names, and entity types, 42  
 ROLLUP operator, 668–672  
 root nodes, 657  
 RTRIM function (SQL), 640–643  
 rules, business. *See business rules*

## S

schema-based constraints, 284  
 schemas  
   and CODASYL data model, 726–728  
   in relational data model, 282  
 scripts  
   conceptual modeling, 28  
   SQL, 506, 506n  
 SC/se relationships, 144  
 second normal form (2NF), 398–401  
 Selection operation, 292–293  
 selective type inheritance, 159  
 Select operator, 542–544  
 SELECT statement (SQL), 569–631  
 semantic  
   data modeling errors, 119–133  
   errors, and connection traps, 253–257

integrity constraints, 36, 82, 103, 284  
 semantic modeling, 27  
 semicolons (;) in SQL statements, 569  
 Semi-Join operations, 560–561, 612–613  
 Semi-Minus operations, 561, 612–613  
 sequential access, 3  
 servers, universal, 737  
 set default rule, 60  
 set null rule, 60  
 set theoretic operators, 549–551  
 shared subclass, 156  
 simple attributes, 33  
 single-valued attributes, 33–34  
 SPARC (Standards Planning and Requirements  
   Committee), 6  
 specialization, 149  
   and categorization, 157–160  
   and generalization in superclass/subclass  
     relationships, 146–154, 188  
   hierarchy, and lattice, 154–157, 188  
 specialization lattice, 156  
 SQL (Structured Query Language). *See also*  
   *specific statements*  
   built-in functions, 2003 standard,  
     635–651  
   database data population using, 524–531  
   database implementation using, 504–505  
   and database systems, 10  
   data definition language (DDL) and, 12  
   as data manipulation language (DML), 13  
   dates and times, handling, 651–656  
   pattern matching in, 593–597  
   queries based in single table, 569–597, 631  
   queries based on binary operators,  
     597–613, 631  
   query examples, 700–705  
   subqueries, 613–630, 631  
     using generally, 568–569  
 SQL/DCL (SQL data control language), 504  
 SQL/DDL (SQL data definition language)  
   base database table specification in,  
     507–524  
   described, 504–505  
 SQL/DML (SQL data manipulation language),  
   504  
 SQL scripts, 506, 506n

SQL-2003 standard  
 built-in functions, 635–651  
 described, 507, 532, 593  
**Standards Planning and Requirements Committee (SPARC)**, 6  
**START WITH clause (SQL)**, 658–659, 661–663  
 state constraints in relational data model, 284  
 stored attributes, 33  
 structural constraints  
   min, max notation for, 104–111  
   of relationship types, 43–51  
 structural inheritance, 733  
 structured data types, 737  
**Structured Query Language. See SQL (Structured Query Language)**  
 subclass, shared, 156  
 subclass (sc) entity types, 144  
 subgroups (sc's), specialization and generalization, 146–147  
 subschemas, and CODASYL data model, 726  
 substitutability, principle of, 733  
**SUBSTR function (SQL)**, 645  
**SUBSTRING function (SQL)**, 636–638  
 summarizing data in SQL, 580–583  
 superclass connectors, 148, 151  
 superclass/subclass relationships  
   described, 142–143  
   general properties, 145–146  
   specialization and generalization, 146–154, 188  
 superkeys  
   and candidate keys, 379–383  
   and unique identifiers, 284  
 syntactic data modeling errors, 119–133  
 synthesis approach, 85  
 synthesizing candidate keys, 375–379

**T**

tables and rows, in SQL, 507, 567n  
 ternary relationship type, 38–42, 198–205  
   decomposing, 234–235  
 Get Well Pharmacists, Inc. (case study), 203–205

Madeira College (case study), 198–203, 216–221  
**Theta Join operations**, 608  
**Theta Join operator**, 554  
 third normal form (3NF), 401–404  
   non-loss, 436–442  
 times and dates, handling in SQL, 651–656  
**TO\_CHAR, TO\_DATE function (SQL)**, 652–656  
 total category, 159  
 total specialization, 148  
 transitive dependencies, 401  
**TRANSLATE function (SQL)**, 643–644  
**TRIM function (SQL)**, 640–643  
 trivial dependencies, 366–367  
 tuples  
   in relational data model, 281  
   in SQL, 631  
 type inheritance property, 146

**U**

UML (Unified Modeling Language), 735–736  
 unary operations, 292–293  
 unary operators  
   in relational algebra, 563  
   Select and Project operators, 542–545  
 underscore character (\_) in SQL statements, 596  
 Unified Modeling Language (UML), 735–736  
 union compatibility, 293  
 unions  
   described, 293–295  
   set theoretic operators, 549–551  
 unique identifiers, 96  
   of entity types, 37–38  
   in relational data model, 284–289, 345  
   and weak entity types, 52–53  
 uniqueness constraint, 36  
 Universal Relation Schema (URS) assumption, 316  
 universal servers, 737

## Index

universe of interest and data models, 17  
update anomaly, 361, 400  
UPDATE statement (SQL), 530–531  
URS (Universal Relation Schema) assumption, 316  
user-defined specialization, 149

## V

views  
materialized, 296–297  
and materialized views in relational data model, 296

## W

Wand and Weber conceptual data modeling framework, 28, 70  
warehouse, data, 12  
weak entity types, base entity types and, 52–57  
weak relationship type, 224–230  
composites of, 230–233  
decomposing, 244–245  
wildeards in SQL, 593  
window functions (SQL), 688–692  
WINDOWING\_CLAUSE (SQL), 681–682  
workgroup database systems, 11











