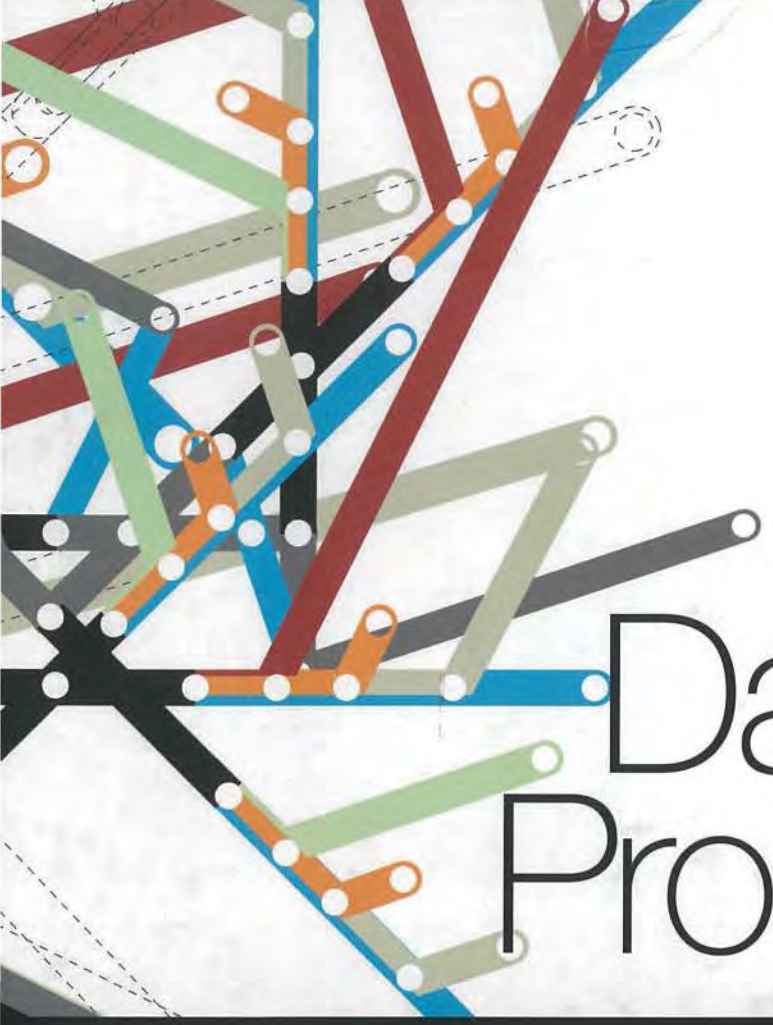


Database Processing

Fundamentals, Design, and Implementation

David M. Kroenke and David J. Auer

Edition 11

Amazon Ex. 1008.0001

Edition 11

Database Processing

Fundamentals, Design, and Implementation

David M. Kroenke

David J. Auer
Western Washington University

Prentice Hall
Boston Columbus Indianapolis New York San Francisco Upper Saddle River
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto
Delhi Mexico City Sao Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

Amazon Ex. 1008.0002

Executive Editor: Bob Horan
Editorial Director: Sally Yagan
AVP/Editor-in-Chief: Eric Svendsen
Assistant Editor: Kelly Loftus
Director of Marketing: Patrice Lumumba Jones
Senior Marketing Manager: Anne Fahlgren
Marketing Assistant: Susan Osterlitz
Senior Managing Editor: Judy Leale
Production Project Manager: Kelly Warsak
Senior Operations Supervisor: Arnold Vila
Operations Specialist: Ilene Kahn
Senior Art Director: Janet Slowik
Design Director: Christy Mahon
Cover and Interior Design: Karen Quigley
Manager, Rights and Permissions: Charles Morris
Manager, Cover Visual Research & Permissions: Karen Sanatar
Cover Illustration: Stockbyte/Veer/Corbis
Media Director: Lisa Rinaldi
Lead Media Project Manager: Denise Vaughn
Full-Service Project Management: Jennifer Welsch/BookMasters, Inc.
Composition: Integra Software Services
Printer/Binder: Quebecor World-Versailles
Cover Printer: Lehigh-Phoenix Color/Hagerstown
Text Font: 10/12 Kepler MM

Credits and acknowledgments borrowed from other sources and reproduced, with permission, in this textbook appear on appropriate page within text.

Microsoft® and Windows® are registered trademarks of the Microsoft Corporation in the U.S.A. and other countries. Screen shots and icons reprinted with permission from the Microsoft Corporation. This book is not sponsored or endorsed by or affiliated with the Microsoft Corporation.

MySQL®, the MySQL GUI Tools® (MySQL Query Brower® and MySQL Administrator®), the MySQL Command Line Client®, and MySQL Workbench® are registered trademarks of Sun Microsystems, Inc. in the U.S.A and other countries. Screenshots and icons reprinted with permission of Sun Microsystems, Inc. This book is not sponsored or endorsed by or affiliated with Sun Microsystems.

Eclipse® and The Eclipse PHP Development Tools (PDT) Project® are trademarks of the Eclipse Foundation, Inc. The Eclipse platform is copyright Eclipse copyright holders and others, 2000, 2007. Screenshots reprinted under the terms of the Eclipse Public License v1.0 available at www.eclipse.org/legal/epl-v10.html. This book is not sponsored or endorsed by or affiliated with the Eclipse Foundation, Inc.

PHP is copyright The PHP Group 1999–2008, and is used under the terms of the PHP Public License v3.01 available at http://www.php.net/license/3_01.txt. This book is not sponsored or endorsed by or affiliated with The PHP Group.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

Copyright © 2010, 2006, 2004, 2000, 1998 Pearson Education, Inc., publishing as Prentice Hall, One Lake Street, Upper Saddle River, New Jersey 07458. All rights reserved. Manufactured in the United States of America. This publication is protected by Copyright, and permission should be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission(s) to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458.

Many of the designations by manufacturers and seller to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed in initial caps or all caps.

Library of Congress Cataloging-in-Publication Data

Kroenke, David.
Database processing: fundamentals, design, and implementation/David M. Kroenke, David J. Auer.—11th ed.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-13-230267-8 (casebound : alk. paper)
1. Database management. I. Auer, David J. II. Title.

QA76.9.D3K7365 2009
005.74—dc22

2009011144

10 9 8 7 6 5 4 3 2

Prentice Hall
is an imprint of

PEARSON

www.pearsonhighered.com

ISBN 10: 0-13-230267-5
ISBN 13: 978-0-13-230267-8

Amazon Ex. 1008.0003

Brief Contents

Preface xvii

Part 1 • Getting Started 1

Chapter 1 Introduction 2
Chapter 2 Introduction to Structured Query Language 31

Part 2 • Database Design 93

Chapter 3 The Relational Model and Normalization 94
Chapter 4 Database Design Using Normalization 127
Chapter 5 Data Modeling and the Entity-Relationship Model 144
Chapter 6 Transforming Data Models in Database Designs 192

Part 3 • Database Implementation 235

Chapter 7 SQL for Database Construction and Application Processing 236
Chapter 8 Database Redesign 295

Part 4 • Multiuser Database Processing 319

Chapter 9 Managing Multiuser Databases 320
Chapter 10 Managing Databases with SQL Server 2008 351

ONLINE CHAPTER: SEE PAGE 423 FOR INSTRUCTIONS
Chapter 10A Managing Databases with Oracle 11g 10A-1

ONLINE CHAPTER: SEE PAGE 425 FOR INSTRUCTIONS
Chapter 10B Managing Databases with MySQL 5.1 10B-1

Part 5 • Database Access Standards 427

Chapter 11 The Web Server Environment 428
Chapter 12 Database Processing with XML 485
Chapter 13 Database Processing for Business Intelligence Systems 525

ONLINE APPENDICES: SEE PAGE 565 FOR INSTRUCTIONS
Appendix A Getting Started with Microsoft Access 2007 A-1
Appendix B The IDEF1X Standard B-1
Appendix C UML-Style Entity-Relationship Diagrams C-1
Appendix D Data Structures for Database Processing D-1
Appendix E The Semantic Object Model E-1

Amazon Ex. 1008.0004

Contents

Preface xvii

PART 1 • GETTING STARTED 1

Chapter 1: Introduction 2

Chapter Objectives 2
The Characteristics of Databases 3
A Note on Naming Conventions 3
A Database Has Data and Relationships 4
Databases Create Information 5
Database Examples 6
Single-User Database Applications 6
Multiuser Database Applications 6
E-Commerce Database Applications 7
Reporting and Data Mining Database Applications 7
The Components of a Database System 8
Database Applications and SQL 9
The DBMS 11
The Database 12
Personal Versus Enterprise-Class Database Systems 13
What Is Microsoft Access? 13
What Is an Enterprise-Class Database System? 15
Database Design 16
Database Design from Existing Data 16
Database Design for New Systems Development 17
Database Redesign 18
What You Need to Learn 19
A Brief History of Database Processing 20
The Early Years 20
The Emergence and Dominance of the Relational Model 22
Post-Relational Developments 23
Summary 24
Key Terms 25
Review Questions 26
Project Questions 28

Chapter 2: Introduction to Structured Query Language 31

Chapter Objectives 31
Cape Codd Outdoor Sports 32
The Retail Sales Data Extraction 33
RETAIL_ORDER Data 33
ORDER_ITEM Data 34
SKU_DATA Table 35
Data Extracts Are Common 35

Amazon Ex. 1008.0005

SQL Background 35
The SQL SELECT/FROM/WHERE Framework 36
Reading Specified Columns from a Single Table 36
Reading Specified Rows from a Single Table 38
Reading Specified Columns and Rows from a Single Table 39
Submitting SQL Statements to the DBMS 40
Using SQL in Microsoft Access 2007 40
Using SQL in Microsoft SQL Server 2008 46
Using SQL in Oracle Database 11g 48
Using SQL in Sun Microsystems MySQL 5.1 50
SQL Enhancements for Querying a Single Table 52
Sorting the Results 53
SQL WHERE Clause Options 55
Combing the SQL WHERE Clause and the SQL ORDER BY Clause 59
Performing Calculations in SQL Queries 59
Using SQL Built-in Functions 59
SQL Expressions in SQL SELECT Statements 62
Grouping in SQL SELECT Statements 64
Looking for Patterns in NASDAQ Trading 67
Investigating the Characteristics of the Data 68
Searching for Patterns in Trading by Day of Week 69
Querying Two or More Tables with SQL 70
Querying Multiple Tables with Subqueries 70
Querying Multiple Tables with Joins 73
Comparing Subqueries and Joins 76
Summary 77
Key Terms 77
Review Questions 78
Project Questions 82
Marcia's Dry Cleaning 86
Morgan Importing 89

PART 2 • DATABASE DESIGN 93

Chapter 3: The Relational Model and Normalization 94

Chapter Objectives 94
Relational Model Terminology 96
Relations 96
Characteristics of Relations 97
Alternative Terminology 98
Functional Dependencies 100
Finding Functional Dependencies 101
Keys 104
Normal Forms 106
Modification Anomalies 106
A Short History of Normal Forms 107
Normalization Categories 108
From First Normal Form to Boyce-Codd Normal Form 108
Eliminating Anomalies from Functional Dependencies 110
Eliminating Anomalies from Multivalued Dependencies 117
Fifth Normal Form 120
Domain/Key Normal Form 120
Summary 121
Key Terms 121
Review Questions 122

Amazon Ex. 1008.0006

Project Questions 124
Marcia's Dry Cleaning 125
Morgan Importing 126

Chapter 4: Database Design Using Normalization 127

Chapter Objectives 127
Assess Table Structure 128
Designing Updatable Databases 129
Advantages and Disadvantages of Normalization 129
Functional Dependencies 129
Normalizing with SQL 130
Choosing Not to Use BCNF 131
Multivalued Dependencies 132
Designing Read-Only Databases 132
Denormalization 132
Customized Duplicated Tables 134
Common Design Problems 135
The Multivalue, Multicolumn Problem 135
Inconsistent Values 136
Missing Values 137
The General-Purpose Remarks Column 138
Summary 139
Key Terms 139
Review Questions 139
Project Questions 141
Marcia's Dry Cleaning 142
Morgan Importing 143

Chapter 5: Data Modeling with the Entity-Relationship Model 144

Chapter Objectives 144
The Purpose of a Data Model 145
The Entity-Relationship Model 145
Entities 145
Attributes 146
Identifiers 146
Relationships 147
Maximum Cardinality 149
Minimum Cardinality 150
Entity-Relationship Diagrams and Their Versions 151
Variations of the E-R Model 151
E-R Diagrams Using the IE Crow's Foot Model 152
Strong Entities and Weak Entities 153
ID-Dependent Entities 153
Non-ID-Dependent Weak Entities 154
Subtype Entities 156
Patterns in Forms, Reports, and E-R Models 157
Strong Entity Patterns 158
ID-Dependent Relationships 162
Mixed Identifying and Nonidentifying Patterns 168
The For-Use-By Pattern 171
Recursive Patterns 172
The Data Modeling Process 174
The College Report 175
The Department Report 175
The Department/Major Report 176
The Student Acceptance Letter 178

Amazon Ex. 1008.0007

Summary 180
Key Terms 181
Review Questions 182
Project Questions 184
Marcia's Dry Cleaning 191
Morgan Importing 191

Chapter 6: Transforming Data Models into Database Designs 192

Chapter Objectives 192
Create a Table for Each Entity 193
Selecting the Primary Key 194
Specifying Candidate (Alternate) Keys 195
Specify Column Properties 195
Verify Normalization 197
Create Relationships 198
Relationships Between Strong Entities 198
Relationships Using ID-Dependent Entities 201
Relationships with a Weak Non-ID-Dependent Entity 206
Relationships in Mixed Entity Designs 206
Relationships Between Supertype and Subtype Entities 208
Recursive Relationships 208
Representing Ternary and Higher-Order Relationships 210
Relational Representation of the Highline University Data Model 213
Design for Minimum Cardinality 214
Actions When the Parent Is Required 216
Actions When the Child Is Required 217
Implementing Actions for M-O Relationships 218
Implementing Actions for O-M Relationships 219
Implementing Actions for M-M Relationships 219
Designing Special Case M-M Relationships 219
Documenting the Minimum Cardinality Design 220
An Additional Complication 222
Summary of Minimum Cardinality Design 222
The View Ridge Gallery Database 222
Summary of Requirements 222
The View Ridge Data Model 223
Database Design with Data Keys 224
Minimum Cardinality Enforcement for Required Parents 225
Minimum Cardinality Enforcement for the Required Child 227
Column Properties for the View Ridge Database Design Tables 228
Summary 230
Key Terms 231
Review Questions 231
Project Questions 233
Marcia's Dry Cleaning 234
Morgan Importing 234

PART 3 • DATABASE IMPLEMENTATION 235

Chapter 7: SQL for Database Construction and Application Processing 236

Chapter Objectives 236
The View Ridge Gallery Database 237
SQL DDL, DML, and a New Type of Join 237

Amazon Ex. 1008.0008

Managing Table Structure with SQL DDL 238
Creating the View Ridge Database 238
Using the SQL CREATE TABLE Statement 239
Variations in SQL Data Types 239
Creating the ARTIST Table 240
Creating the WORK Table and the 1:N ARTIST-to-WORK Relationship 244
Implementing Required Parent Rows 245
Implementing 1:1 Relationships 245
Casual Relationships 245
Creating Default Values and Data Constraints with SQL 246
Creating the View Ridge Database Tables 248
The SQL ALTER Statement 251
The SQL DROP TABLE Statement 252
SQL DML Statements 252
The SQL INSERT Statement 252
Populating the View Ridge Database Tables 253
The SQL UPDATE Statement 259
The SQL DELETE Statement 260
New Forms of Join 260
The SQL JOIN ON Syntax 260
Outer Joins 262
Using SQL Views 266
Using SQL Views to Hide Columns and Rows 268
Using SQL Views to Display Results of Computed Columns 269
Using SQL Views to Hide Complicated SQL Syntax 270
Layering Built-in Functions 270
Using SQL Views for Isolation, Multiple Permissions, and Multiple Triggers 273
Updating SQL Views 273
Embedding SQL in Program Code 274
Using SQL Triggers 275
Using Triggers to Provide Default Values 276
Using Triggers to Enforce Data Constraints 278
Using Triggers to Update Views 279
Using Triggers to Implement Referential Integrity Actions 279
Using Stored Procedures 281
Advantages of Stored Procedures 281
The WORK_AddWorkTransaction Stored Procedure 283
Summary 283
Key Terms 285
Review Questions 286
Project Questions 290
Marcia's Dry Cleaning 293
Morgan Importing 294

Chapter 8: Database Redesign 295

Chapter Objectives 295
The Need for Database Redesign 296
SQL Statements for Checking Functional Dependencies 296
Correlated Subqueries 297
Analyzing the Existing Database 302
Reverse Engineering 302
Dependency Graphs 303
Database Backup and Test Databases 304
Changing Table Names and Table Columns 305
Changing Table Names 305
Adding and Dropping Columns 306

Amazon Ex. 1008.0009

Changing a Column Data Type or Column Constraints 307
Adding and Dropping Constraints 308
Changing Relationship Cardinalities and Properties 308
Changing Minimum Cardinalities 308
Changing Maximum Cardinalities 309
Adding and Deleting Tables and Relationships 312
Forward Engineering(?) 312
Summary 313
Key Terms 314
Review Questions 314
Project Questions 316
Marcia's Dry Cleaning 317
Morgan Importing 318

PART 4 • MULTIUSER DATABASE PROCESSING 319

Chapter 9: Managing Multiuser Databases 320

Chapter Objectives 320
Database Administration 321
Managing the Database Structure 322
Concurrency Control 323
The Need for Atomic Transactions 324
Resource Locking 328
Optimistic Versus Pessimistic Locking 330
Declaring Lock Characteristics 331
Consistent Transactions 332
Transaction Isolation Level 333
Cursor Type 334
Database Security 335
Processing Rights and Responsibilities 335
DBMS Security 337
DBMS Security Guidelines 337
Application Security 339
SQL Injection Attack 340
Database Backup and Recovery 340
Recovery via Reprocessing 341
Recovery via Rollback/Rollforward 341
Managing the DMBS 343
Maintaining the Data Repository 344
Summary 345
Key Terms 346
Review Questions 347
Project Questions 348
Marcia's Dry Cleaning 349
Morgan Importing 350

Chapter 10: Managing Databases with SQL Server 2008 351

Chapter Objectives 351
Installing SQL Server 2008 352
The Microsoft SQL Server 2008 Management Studio 353
Creating an SQL Server 2008 Database 354
SQL Server 2008 Utilities 356
SQL CMD and Microsoft PowerShell 356
Microsoft SQL CLR 356

Amazon Ex. 1008.0010

SQL Server 2008 GUI Displays 357
SQL Server 2008 SQL Statements and SQL Scripts 357
Creating and Populating the View Ridge Database Tables 360
Creating the View Ridge Database Table Structure 360
Reviewing Database Structures in the SQL Server GUI Display 364
Indexes 368
Populating the VRG Tables with Data 369
Creating Views 379
SQL Server 2008 Application Logic 381
Transact-SQL 382
Stored Procedures 385
Triggers 393
Concurrency Control 407
Transaction Isolation Level 408
Cursor Concurrency 408
Locking Hints 409
SQL Server 2008 Security 410
SQL Server Database Security Settings 412
SQL Server 2008 Backup and Recovery 414
Backing Up a Database 414
SQL Server Recovery Models 415
Restoring a Database 416
Database Maintenance Plans 416
Topics Not Discussed in This Chapter 417
Summary 417
Key Terms 417
Review Questions 418
Project Questions 420
Marcia's Dry Cleaning 421
Morgan Importing 421

ONLINE CHAPTER: SEE PAGE 423 FOR INSTRUCTIONS

Chapter 10A: Managing Databases with Oracle Database 11g 10A-1

Chapter Objectives 10A-1
Installing Oracle Database 11g 10A-2
Oracle Database 11g Documentation 10A-4
Oracle and Java 10A-5
Oracle Database 11g Administration and Development Tools 10A-5
The Oracle Database Configuration Assistant 10A-6
The Oracle Enterprise Manager 11g Database Control 10A-6
Oracle Tablespaces 10A-8
Oracle Security 10A-11
User Privileges 10A-11
Creating a User Account 10A-11
Creating a Role 10A-13
Oracle Application Development Tools 10A-15
Oracle SQL*Plus 10A-15
Oracle SQL Developer 10A-16
Oracle Schemas 10A-17
Oracle Database 11g 2008 SQL Statements and SQL Scripts 10A-18
Creating and Populating the View Ridge Database Tables 10A-20
Creating the View Ridge Database Table Structure 10A-20
Reviewing Database Structures in the SQL Developer GUI Display 10A-23

Amazon Ex. 1008.0011

Indexes 10A-26
Populating the VRG Tables 10A-28
Creating Views 10A-36
Application Logic 10A-39
Oracle PL/SQL 10A-40
Stored Procedures 10A-42
Triggers 10A-50
Concurrency Control 10A-61
Read Committed Transaction Isolation Level 10A-62
Serializable Transaction Isolation Level 10A-63
Read-Only Transaction Isolation 10A-63
Additional Locking Comments 10A-63
Oracle Backup and Recovery 10A-64
Oracle Recovery Facilities 10A-64
Types of Failure 10A-64
Topics Not Discussed in This Chapter 10A-65
Summary 10A-65
Key Terms 10A-66
Review Questions 10A-67
Project Questions 10A-69
Marcia's Dry Cleaning 10A-70
Morgan Importing 10A-70

ONLINE CHAPTER: SEE PAGE 425 FOR INSTRUCTIONS

Chapter 10B: Managing Databases with MySQL 5.1 10B-1

Chapter Objectives 10B-1
The MySQL 5.1 DBMS 10B-2
Installing and Updating MySQL 10B-2
Configuring MySQL 10B-2
MySQL Storage Engines 10B-3
The MySQL GUI Utilities 10B-4
Creating and Using a MySQL Database 10B-6
MySQL Utilities 10B-7
MySQL Command Line Client 10B-8
MySQL GUI Displays 10B-8
MySQL SQL Statements and SQL Scripts 10B-10
Creating and Populating the View Ridge Database Tables 10B-13
Creating the View Ridge Database Table Structure 10B-13
Reviewing Database Structures in the MySQL GUI Display 10B-16
Indexes 10B-17
Populating the VRG Tables with Data 10B-22
Creating Views 10B-30
MySQL Application Logic 10B-33
MySQL Procedural Statements 10B-34
Stored Procedures 10B-36
Triggers 10B-43
A Last Word on MySQL Stored Procedures and Triggers 10B-53
Concurrency Control 10B-54
MySQL 5.1 Security 10B-55
MySQL Database Security Settings 10B-56
MySQL 5.1 Server Backup and Recovery 10B-58
Backing Up a MySQL Database 10B-58
Restoring a MySQL Database 10B-59
Topics Not Discussed in This Chapter 10B-61

Amazon Ex. 1008.0012

Summary 10B-61
Key Terms 10B-62
Review Questions 10B-62
Project Questions 10B-63
Marcia's Dry Cleaning 10B-65
Morgan Importing 10B-65

PART 5 • DATABASE ACCESS STANDARDS 427

Chapter 11: The Web Server Environment 428

Chapter Objectives 428
The Web Database Processing Environment 429
The ODBC Standard 431
ODBC Architecture 431
Conformance Levels 432
Creating an ODBC Data Source Name 434
The Mircosoft .NET Framework and ADO.NET 437
OLE DB 438
ADO and ADO.NET 442
The ADO.NET Object Model 442
The JAVA Platform 446
JDBC 446
Java Server Pages (JSP) and Servlets 448
Apache Tomcat 449
Web Database Processing with PHP 449
Web Database Processing with PHP and Eclipse 451
Getting Started with HTML Web Pages 453
The index.html Web Page 453
Creating the index.html Web Page 453
Using PHP 456
Challenges for Web Database Processing 463
Web Page Examples with PHP 463
Example 1: Updating a Table 464
Example 2: Using PHP Data Objects (PDO) 470
Example 3: Invoking a Stored Procedure 470
Summmary 478
Key Terms 480
Review Questions 481
Project Questions 483
Marcia's Dry Cleaning 485
Morgan Importing 485

Chapter 12: Database Processing with XML 486

Chapter Objectives 486
The Importance of XML 487
XML as a Markup Language 488
XML Document Type Declarations 489
Materializing XML Documents with XSLT 489
XML Schema 494
XML Schema Validation 494
Elements and Attributes 495
Flat Versus Structured Schemas 496
Global Elements 500

Amazon Ex. 1008.0013

Creating XML Documents from Database Data 502
Using the SQL SELECT . . . FOR XML Statement 502
Multitable SELECT with FOR XML 506
An XML Schema for All CUSTOMER Purchases 510
A Schema with Two Multivalue Paths 514
Why Is XML Important? 514
Additional XML Standards 520
Summary 522
Key Terms 523
Review Questions 523
Project Questions 524
Marcia's Dry Cleaning 525
Morgan Importing 525

Chapter 13: Database Processing for Business Intelligence Systems 526

Chapter Objectives 526
Business Intelligence Systems 526
The Relationship Between Operational and BI Systems 526
Reporting Systems and Data Mining Applications 527
Reporting Systems 528
Data Mining Applications 528
Data Warehouses and Data Marts 528
Components of a Data Warehouse 528
Data Warehouses Versus Data Marts 531
Dimensional Databases 532
Reporting Systems 539
RFM Analysis 540
Producing the RFM Report 541
Reporting System Components 544
Report Types 545
Report Media 545
Report Modes 546
Report System Functions 546
OLAP 549
Data Mining 552
Unsupervised Data Mining 554
Supervised Data Mining 555
Three Popular Data Mining Techniques 555
Market Basket Analysis 555
Using SQL for Market Basket Analysis 555
Summary 557
Key Terms 558
Review Questions 559
Project Questions 561
Marcia's Dry Cleaning 563
Morgan Importing 564

APPENDICES

ONLINE APPENDICES: SEE PAGE 565 FOR INSTRUCTIONS

Appendix A: Getting Started with Microsoft Access 2007 A-1
Chapter Objectives A-1
Table Keys A-2
Relationships Among Tables A-2

Amazon Ex. 1008.0014

Creating an Access Database A-3
The Microsoft Office Fluent User Interface A-5
The Ribbon and Command Tabs A-5
Contextual Command Tabs A-5
Database Objects and the Navigation Pane A-6
Closing a Database and Exiting Microsoft Access A-7
Opening an Existing Access Database A-8
Creating Access Database Tables A-9
Inserting Data into Tables—The Datasheet View A-17
Modifying and Deleting Data in Tables in the Datasheet View A-19
Creating Relationships Between Tables A-20
Working with Microsoft Access Queries A-23
Microsoft Access Forms and Reports A-26
Creating Access 2007 Forms A-26
Creating Access 2007 Reports A-27
Closing a Database and Exiting Access 2007 A-27
Key Terms A-28
Review Questions A-28

Appendix B: The IDEF1X Standard B-1
Chapter Objectives B-1
IDEF1X Entities B-1
IDEF1X Relationships B-2
Nonidentifying Connection Relationships B-2
Identifying Connection Relationships B-3
Nonspecific Relationships B-4
Categorization Relationships B-4
Domains B-6
Domains Reduce Ambiguity B-6
Domains Are Useful B-7
Base Domains and Typed Domains B-7
Key Terms B-8
Review Questions B-8

Appendix C: UML-Style Entity-Relationship Diagrams C-1
Chapter Objectives C-1
UML Entities and Relationships C-1
Representation of Weak Entities C-2
Representation of Subtypes C-2
OOP Constructs Introduced by UML C-3
The Role of UML in Database Processing Today C-4
Key Terms C-5
Review Questions C-5

Appendix D: Data Structures for Database Processing D-1
Chapter Objectives D-1
Flat Files D-1
Processing Flat Files in Multiple Orders D-2
A Note on Record Addressing D-2
Maintaining Order with Linked Lists D-3
Maintaining Order with Indexes D-5
B-Trees D-5
Summary of Data Structures D-7
Representing Binary Relationships D-8
Review of Record Relationships D-8
Representing Trees D-10

Amazon Ex. 1008.0015

Representing Simple Networks D-12
Representing Complex Networks D-13
Summary of Relationship Representations D-14
Secondary-Key Representations D-16
Linked-List Representation of Secondary Keys D-17
Index Representation of Secondary Keys D-18
Key Terms D-18
Review Questions D-19

Appendix E: The Semantic Object Model E-1
Chapter Objectives E-1
Semantic Objects E-2
Defining Semantic Objects E-2
Attributes E-3
Object Identifiers E-5
Attribute Domains E-6
Semantic Object Views E-6
Types of Objects E-7
Simple Objects E-7
Composite Objects E-8
Compound Objects E-11
Representing One-to-One Compound Objects E-12
Representing One-to-Many and Many-to-One Relationships E-14
Representing Many-to-Many Relationships E-14
Hybrid Objects E-17
Representing Hybrid Relationships E-20
Association Objects E-22
Parent/Subtype Objects E-24
Archetype/Version Objects E-26
Comparing the Semantic Object and the E-R Models E-29
Key Terms E-31
Review Questions E-32

Bibliography 567
Glossary 569
Index 591

Amazon Ex. 1008.0016

Preface

Fundamentals, Design, and Implementation

The 11th edition of *Database Processing* refines the organization and content of this classic textbook to reflect a new teaching and professional workplace environment. The organization and topic selection of the 11th edition is designed to:

- Present an early introduction to SQL queries.
- Use a "spiral approach" to database design.
- Use a consistent, generic Information Engineering (IE) Crow's Foot E-R diagram notation for data modeling and database design.
- Provide a detailed discussion of specific normal forms within a discussion of normalization that focuses on pragmatic normalization techniques.
- Use current DBMS technology: Microsoft Access 2007, Microsoft SQL Server 2008, Oracle Database 11g, and MySQL 5.1.
- Create Web database applications based on widely used Web development technology.
- Provide an introduction to business intelligence (BI) systems.
- Discuss the dimensional database concepts used in database designs for data warehouses and OnLine Analytical Processing (OLAP).

These changes have been made because it has become obvious that the basic structure of the earlier editions (up to and including the 9th edition—the 10th edition introduced many of the changes we use in the 11th edition) was designed for a teaching environment that no longer existed. The structural changes to the book were made for several reasons:

- Unlike the early years of database processing, today's students have ready access to data modeling and DBMS products.
- Today's students are too impatient to start a class with lengthy conceptual discussions on data modeling and database design. They want to do something, see a result, and obtain feedback.
- In the current economy, students need to reassure themselves that they are learning marketable skills.

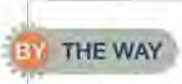
Early Introduction of SQL DML

Given these changes in the classroom environment, this book provides an early introduction to SQL data manipulation language (DML) SELECT statements. The discussion of SQL data definition language (DDL) and additional DML statements occurs in Chapters 7 and 8. By presenting SQL SELECT statements in Chapter 2, students learn early in the class how to query data and obtain results, seeing firsthand some of the ways that database technology will be useful to them.

The text assumes that students will work through the SQL statements and examples with a DBMS product. This is practical today, because nearly every student has access to Microsoft Access. Therefore, Chapters 1 and 2 and Appendix A are written to support an early introduction of Microsoft Access 2007 and the use of Access 2007 for SQL queries (Access 2007 QBE query techniques are also covered).

Amazon Ex. 1008.0017

If a non–Access-based approach is desired, versions of SQL Server 2008, Oracle Database, and MySQL 5.1 are readily available for use. Free versions of the three major DBMS products covered in this book (SQL Server 2008 Express, Oracle Express 10g, and MySQL 5.1 Community Edition) are available for download. Further, the text can be purchased with a licensed educational version of Oracle Database 11g personal edition (this is a developer license) as well. Alternatively, a trial copy of MySQL 5.1 Enterprise Edition is also available as a download. Thus, students can actively use a DBMS product by the end of the first week of class.



The presentation and discussion of SQL is spread over three chapters so that students can learn about this important topic in small bites. SQL SELECT statements are taught in Chapter 2. SQL DDL and SQL DML statements are presented in Chapter 7. Correlated subqueries and EXISTS/NOT EXISTS statements are described in Chapter 8. Each topic appears in the context of accomplishing practical tasks. Correlated subqueries, for example, are used to verify functional dependency assumptions, a necessary task for database redesign.

This box illustrates another feature used in this book: BTW boxes are used to separate comments from the text discussion. Sometimes they present ancillary material; other times they reinforce important concepts.

A Spiral Approach to Database Design

Today, databases arise from three sources: (1) from the integration of existing data from spreadsheets, data files, and database extracts, (2) from the development of new information systems projects, and (3) from the need to redesign an existing database to adapt to changing requirements. We believe that the fact that these three sources exist present instructors with a significant pedagogical opportunity. Rather than teach database design just once from data models, why not teach database design three times, once for each of these sources? In practice, this idea has turned out to be even more successful than expected.

Design Iteration 1: Databases from Existing Data

Considering the design of databases from existing data, if someone were to e-mail us a set of tables and say, "Create a database from them," how would we proceed? We would examine the tables in light of normalization criteria and then determine whether the new database was for query only or whether it was for query and update. Depending on the answer, we would denormalize the data, joining them together, or we would normalize the data, pulling them apart. All of which is important for students to know and understand.

Therefore, the first iteration of database design gives instructors a rich opportunity to teach normalization, not as a set of theoretical concepts, but rather as a useful toolkit for making design decisions for databases created from existing data. Additionally, the construction of databases from existing data is an increasingly common task that is often assigned to junior staff members. Learning how to apply normalization to the design of databases from existing data not only provides an interesting way of teaching normalization, it is also common and useful!

We prefer to teach and use a pragmatic approach to normalization, and present this approach in Chapter 3. However, we are aware that many instructors like to teach normalization in the context of a step-by-step normal form presentation (1NF, 2NF, 3NF, then BCNF), and Chapter 3 includes material to support this approach as well.

In today's workplace, large organizations are increasingly licensing standardized software from vendors such as SAP, Oracle, and Siebel. Such software already has a database design. But with every organization running the same software, many are learning that they can only gain a competitive advantage if they make better use of the data in those predesigned databases. Hence, students who know how to extract data and create read-only databases for reporting and data mining have obtained marketable skills in the world of ERP and other packaged software solutions.

Amazon Ex. 1008.0018

Design Iteration 2: Data Modeling and Database Design

The second source of databases is from new systems development. Although not as common as in the past, many databases are still created from scratch. Thus, students still need to learn data modeling, and they still need to learn how to transform data models into database designs.

The IE Crow's Foot Model as a Design Standard

This edition uses a generic, standard IE Crow's Foot notation. Your students should have no trouble understanding the symbols and using the data modeling or database design tool of your choice.

IDEF1X (which was used as the preferred E-R diagram notation in the 9th edition of this text) is explained in Appendix B in case your students will graduate into an environment where it is used, or if you prefer to use it in your classes. UML is explained in Appendix C in case you prefer to use UML in your classes.

BY THE WAY The choice of a data modeling tool is somewhat problematic. The two most readily available tools, Microsoft Visio and Sun Microsystems MySQL Workbench, are database design tools, not data modeling tools. Neither can produce an N:M relationship as such (as a data model requires), but have to immediately break it into two 1:N relationships (as database design does). Therefore, the intersection table must be constructed and modeled. This confounds data modeling with database design in just the way that we are attempting to teach students to avoid.

To be fair to Visio, it is true that data models with N:M relationships can be drawn using either the standard Visio drawing tools or the Entity Relationship shapes dynamic connector. These are shown in Figure FM-01, where the N:M relationships ENTITY-ONE-to-ENTITY-THREE and ENTITY-FOUR-to-ENTITY-FIVE were drawn using the dynamic connector and show the correct line end symbols.

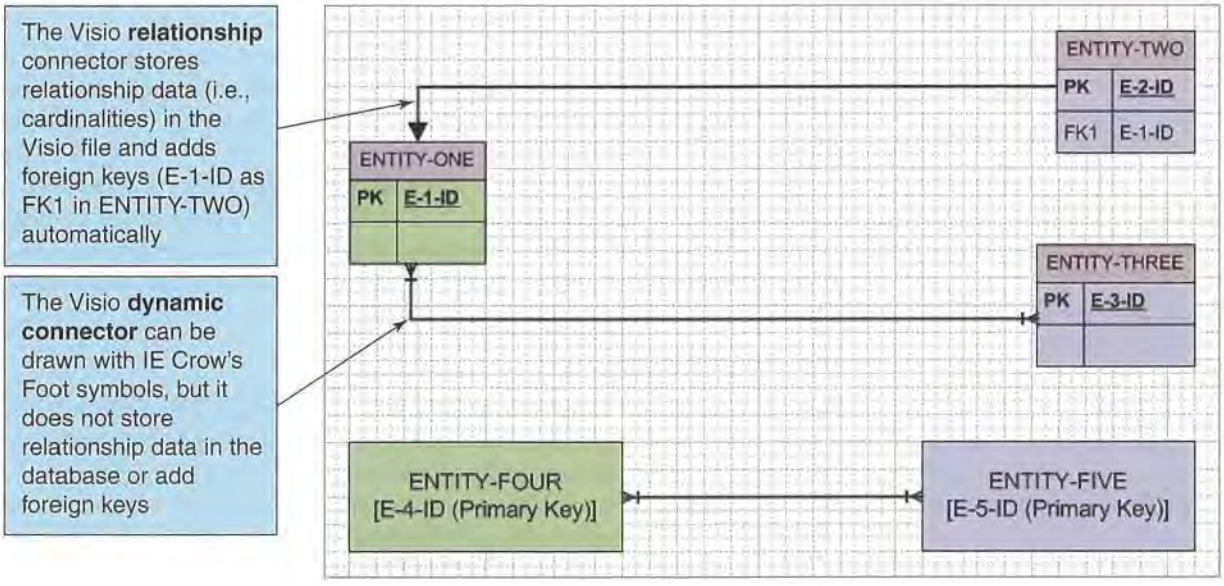
ENTITY-ONE and ENTITY-THREE are Visio database entity objects, and Visio stores object data such as column type, primary key, and so on, about the entities, but *not* about the relationship between them. ENTITY-FOUR and ENTITY-FIVE were drawn using the Visio rectangle drawing tool, and no database object data is stored by these shapes or the relationship between them.

Visio is at its best using the Visio relationship connector to connect entity objects. In this case, as shown in the ENTITY-ONE-to-ENTITY-TWO relationship, relationship data such as minimum and maximum cardinalities is stored in database objects and foreign keys (such as E-1-ID as FK1 in ENTITY-TWO) are automatically inserted in child

(*continued*)

Figure FM-01

Data Models in Microsoft Visio 2007



Amazon Ex. 1008.0019

entities. However, there is no *N:M* relationship connector in Visio, so we cannot model N:M relationships using a relationship connector. Without the ability to store relationship data for N:M relationships, it is problematic to use Visio as data modeling tool while it functions well as database design tool.

There are good data modeling tools available, but they tend to be more complex and expensive. Two examples are Visible Systems' Visible Analyst and Computer Associates' ERwin Data Modeler. Visible Analyst is available in a student edition (at a modest price), and a one-year time-limitedCA ERwin Data Modeler Community Edition suitable for class use can be downloaded from www.ca.com/us/software-trials.aspx—CA has limited the number of objects that can be created by this edition to 25 entities per model, and disabled some other features (see www.ca.com/files/TechnicalDocuments/erwin-community-edition-matrix_197107.pdf), but there is still enough functionality to make this product a possible choice for class use.

Database Design from E-R Data Models

As we discuss in Chapter 6, designing a database from data models consists of three tasks: representing entities and attributes with tables and columns; representing maximum cardinality by creating and placing foreign keys; and representing minimum cardinality via constraints, triggers, and application logic.

The first two tasks are straightforward. However, designs for minimum cardinality are more difficult. Required parents are easily enforced using NOT NULL foreign keys and referential integrity constraints. Required children are more problematic. In this book, however, we simplify the discussion of this topic by limiting the use of referential integrity actions and by supplementing those actions with design documentation. See the discussion around Figure 6-28.

Although the design for required children is complicated, it is important for students to learn. It also provides a reason for students to learn about triggers as well. In any case, the discussion of these topics is much simpler than it was in prior editions because of the use of the IE Crow's Foot model and the use of ancillary design documentation.



David Kroenke is the creator of the semantic object model (SOM). The SOM is presented in Appendix E; the E-R data model is used everywhere else in the text.

Design Iteration 3: Database Redesign

Database redesign, the third iteration of database design, is both common and difficult. As stated in Chapter 8, information systems cause organizational change. New information systems give users new behaviors, and as users behave in new ways, they require changes in their information systems.

Database redesign is by nature complex. Depending on your students, you may wish to skip it, and you can do so without loss of continuity. Database redesign is presented after the discussion of SQL DDL and DML in Chapter 7, because it requires the use of advanced SQL. It also provides a practical reason to teach correlated subqueries and EXISTS/NOT EXISTS statements.

Active Use of a DBMS Product

We assume that the students will actively use a DBMS product. The only real question becomes "which one?" Realistically, most of us have four alternatives to consider: Microsoft Access, Microsoft SQL Server, Oracle Database, or MySQL. You can use any of those products with this text, and tutorials for each of them are presented for Access 2007 (Appendix A), SQL Server 2008 (Chapter 10), Oracle Database 11g (Chapter 10A), and MySQL 5.1 (Chapter 10B). Given the limitations of class time, it is probably necessary to pick and use just one of these

Amazon Ex. 1008.0020

products. You can often devote a portion of a lecture to discussing the characteristics of each, but it is usually best to limit student work to one of them. The possible exception to this is starting the course with Microsoft Access, and then switching to a more robust DBMS product later in the course.

Using Microsoft Access 2007

The primary advantage of Microsoft Access is accessibility. Most students already have a copy, and, if not, copies are easily obtained. Many students will have used Access in their introductory or other classes. Appendix A is a tutorial on Access 2007 for students who have not used it but who wish to use it with this book.

However, Access has several disadvantages. First, as explained in Chapter 1, Access is a combination application generator and DBMS. Access confuses students because it confounds database processing with application development. Also, Access hides SQL behind its query processor and makes SQL appear as an afterthought rather than a foundation. Furthermore, as discussed in Chapter 2, Access does not correctly process some of the basic, SQL-92 standard statements in its default setup. Finally, Access does not support triggers. You can simulate triggers by trapping Windows events, but that technique is nonstandard and does not effectively communicate the nature of trigger processing.

Using Oracle, SQL Server, or MySQL

Choosing which of these products to use depends on your local situation. Oracle Database 11g, a superb enterprise-class DBMS product, is difficult to install and administer. However, if you have local staff to support your students, it can be an excellent choice. As shown in Chapter 10A, Oracle's SQL Developer GUI tool (or SQL*Plus if you are dedicated to this beloved command-line tool) is a handy tool for learning SQL, triggers, and stored procedures. In our experience, students require considerable support to install Oracle on their own computers, and you may be better off to use Oracle from a central server.

SQL Server 2008, although probably not as robust as Oracle, is easy to install on Windows machines, and it provides the capabilities of an enterprise-class DBMS product. The standard database administrator tool is the Microsoft SQL Server Management Studio GUI tool. As shown in Chapter 10, SQL Server 2008 can be used to learn SQL, triggers, and stored procedures.

MySQL 5.1, discussed in Chapter 10B is an open-source DBMS product that is receiving increased attention and market share. The capabilities of MySQL are continually being upgraded, and MySQL 5.1 supports stored procedures and triggers. MySQL also has excellent GUI tools (MySQL Query Browser and MySQL Administrator) and an excellent command-line tool (the MySQL Command Line Client). It is the easiest of the three products for students to install on their own computers. It also works with the Linux operating system, and is popular as part of the AMP (Apache-MySQL-PHP) package (known as WAMP on Windows and LAMP on Linux).



If the DBMS you use is not driven by local circumstances and you do have a choice, we recommend using SQL Server 2008. It has all of the features of an enterprise-class DBMS product, and it is easy to install and use. Another option is to start with Access 2007 if it is available, and switch to SQL Server 2008 at Chapter 7. Chapters 1 and 2 and Appendix A are written specifically to support this approach. A variant is to use Access 2007 as the development tool for forms and reports running against an SQL Server 2008 database.

If you prefer a different DBMS product, you can still start with Access 2007 and switch later in the course. You can order the text with a shrink-wrapped version of Oracle, while a trial version of SQL Server can be downloaded from Microsoft, and MySQL is so easy to download (and updated often enough) that it makes no sense to package a copy with the book.

Amazon Ex. 1008.0021

Focus on Database Application Processing

In this edition, we clearly draw the line between *application development* per se and *database application processing*. Specifically, we have:

- Focused on specific database dependent applications:
  - Web-based, database-driven applications
  - XML-based data processing
  - Business intelligence (BI) systems applications
- Emphasized the use of commonly available, multiple OS compatible application development languages.
- Limited the use of specialized vendor-specific tools and programming languages as much as possible.

There is simply not enough room in this book to provide even a basic introduction to, for example, Visual Basic .NET and Java. Therefore, rather than attempting to introduce these languages, we leave them for other classes where they can be covered at an appropriate depth. Instead, we focus on basic tools that are relatively straightforward to learn and immediately applicable to database-driven applications. We use PHP as our Web development language, and we use the readily available Eclipse integrate development environment (IDE) as our development tool. The result is a very focused final section of the book, where we deal specifically with the interface between databases and the applications that use them.

BY THE WAY Although we try to use widely available software as much as possible, there are, of course, exceptions where we must use vendor specific tools. For BI applications, for example, we draw on Microsoft Excel's PivotTable capabilities and the Microsoft SQL Server 2008 Data Mining Add-ins for Microsoft Office 2007. However, there are either alternatives to these tools (OpenOffice.org DataPilot capabilities, the Palo OLAP Server) or the tools are generally available for download.

Business Intelligence Systems and Dimensional Databases

This edition increases coverage of business intelligence (BI) systems (Chapter 13). The chapter now includes a discussion of dimensional databases, which are the underlying structure for data warehouses, data marts, and OLAP servers. It still covers data management for data warehouses and data marts and also describes reporting and data mining applications, including OLAP.

Chapter 13 includes coverage of two applications that should be particularly interesting to students. The first is RFM analysis, a reporting application frequently used by mail order and e-commerce companies. The complete RFM analysis is accomplished in Chapter 13 through the use of standard SQL statements. Additionally, this chapter includes a market basket analysis that is also performed using SQL correlated subqueries. This chapter can be assigned at any point after Chapter 8 and could be used as a motivator to illustrate the practical applications of SQL midcourse.

Overview of the Chapters in the 11th Edition

Chapter 1 sets the stage by introducing database processing, describing basic components of database systems, and summarizing the history of database processing. If the students are using Access 2007 for the first time (or need a good review), they will also need to study Appendix A at this point. Chapter 2 presents SQL SELECT statements. It also includes sections on how to submit SQL statements to Access, SQL Server, Oracle, and MySQL.

Amazon Ex. 1008.0022

The next four chapters, Chapters 3 through 6, present the first two iterations of database design. Chapter 3 presents the principles of normalization using Boyce-Codd normal form. It describes the problems of multivalued dependencies and explains how to eliminate them. This foundation in normalization is applied in Chapter 4 to the design of databases from existing data.

Chapters 5 and 6 describe the design of new databases. Chapter 5 presents the E-R data model. Traditional E-R symbols are explained, but the majority of the chapter uses IE Crow's Foot notation. Chapter 5 provides a taxonomy of entity types, including strong, ID-dependent, weak but not ID-dependent, supertype/subtype and recursive. The chapter concludes with a simple modeling example for a university database.

Chapter 6 describes the transformation of data models into database designs by converting entities and attributes to tables and columns, by representing maximum cardinality by creating and placing foreign keys, and by representing minimum cardinality via carefully designed DBMS constraints, triggers, and application code. The primary section of this chapter parallels the entity taxonomy in Chapter 5.

Chapter 7 presents SQL DDL and DML. SQL DDL is used to implement the design of an example introduced in Chapter 6. INSERT, UPDATE, and DELETE statements are discussed, as are SQL views. Additionally, the principles of embedding SQL in program code are presented, and triggers and stored procedures are explained.

Database redesign, the third iteration of database design, is described in Chapter 8. This chapter presents SQL correlated subqueries and EXISTS/NOT EXISTS statements and uses those statements in the redesign process. Reverse engineering is described, and basic redesign patterns are illustrated and discussed.

Chapters 9, 10, 10A, and 10B consider the management of multiuser organizational databases. Chapter 9 describes database administration tasks, including concurrency, security, and backup and recovery. Chapters 10, 10A, and 10B then describe SQL Server 2008, Oracle Database 11g, and MySQL 5.1, respectively. These chapters show how to use these products to create database structures and process SQL statements. They also explain concurrency, security, and backup and recovery with each product. The discussion in Chapters 10, 10A, and 10B parallels the order of discussion in Chapter 9 as much as possible, although rearrangements of some topics are made as needed to support the discussion of a specific DBMS product.



We have significantly extended our coverage of Access, SQL Server, Oracle Database, and MySQL in this book. In order to keep the bound book to a reasonable length, and to keep the cost of the book down, we have chosen to provide some materials by download from our Web site at www.pearsonhighered.com/kroenke. There you will find:

- Chapter 10A—Managing Databases with Oracle Database 11g
- Chapter 10B—Managing Databases with MySQL 5.1
- Appendix A—Getting Started with Microsoft Access 2007
- Appendix B—The IDEF1X Standard
- Appendix C—UML-Style Entity-Relationship Diagrams
- Appendix D—Data Structures for Database Processing
- Appendix E—The Semantic Object Model

Chapters 11, 12, and 13 address standards for accessing databases. Chapter 11 presents ODBC, OLE DB, ADO.NET, ASP.NET, JDBC, and Java Server Pages (JSP). It then introduces PHP (and the Eclipse IDE) and illustrates the use of PHP for the publication of databases via Web pages. Chapter 12 describes the integration of XML and database technology. The chapter begins with a primer on XML and then shows how to use the FOR XML SQL statement in SQL Server.

Chapter 13 concludes the text with a discussion of BI systems, dimensional data models, data warehouses, and data marts. It illustrates the use of SQL for RFM reporting analysis and for market basket analysis.

Amazon Ex. 1008.0023

Supplements

This text is accompanied by a wide variety of supplements. Please visit the text's Web site at www.pearsonhighered.com/kroenke to access the instructor and student supplements described below. Please contact your Pearson sales representative for more details. All supplements were written by David Auer.

For Students

- Many of the *Sample Databases* used in this text are available in Access, Oracle, SQL Server, and MySQL format.
- Glossary

For Instructors

- The *Instructor's Resource Manual* provides sample course syllabi, teaching suggestions, and answers to end-of-chapter review, project, and case questions.
- The *Test Item File* and *TestGen* include an extensive set of test questions in multiple-choice, true/false, fill-in-the-blank, short-answer, and essay format. The difficulty level and where the topic is covered in the text are noted for each question. The Test Item File is available in Microsoft Word and in TestGen. TestGen is a comprehensive suite of tools for testing and assessment. It enables instructors to easily create and distribute tests for their courses, either by printing and distributing them through traditional methods or by online delivery via a LAN. TestGen features Screen Wizards to assist you as you move through the program, and the software is backed with full technical support. TestGen conversions for WebCT and BlackBoard course management systems are available.
- *PowerPoint Presentation Slides* feature lecture notes that highlight key terms and concepts. Instructors can customize the presentation by adding their own slides or editing the existing ones.
- The *Image Library* is a collection of the text art organized by chapter. This includes all figures, tables, and screenshots (as permission allows) to enhance class lectures and PowerPoint presentations.

Materials for Your Online Course

Prentice Hall supports our adopters using online courses by providing files ready for upload into both WebCT and BlackBoard course management systems for our testing, quizzing, and other supplements. Please contact your local Pearson sales representative for further information on your particular course.

Acknowledgments

We are grateful for the support of many people in the development of this 11th edition and previous editions. Thanks to Rick Mathieu at James Madison University for interesting and insightful discussions on the database course. Professor Doug MacLachlan from the Marketing Department at the University of Washington was most helpful in understanding the goals, objectives, and technology of data mining, particularly as it pertains to marketing. Don Nilson of the Microsoft Corporation helped us understand the importance of XML to database processing.

In addition, we wish to thank the reviewers of this edition:

Ann Aksut, *Central Piedmont Community College*

Allen Badgett, *Oklahoma City University*

Rich Beck, *Washington University*

Amazon Ex. 1008.0024

Jeffrey J. Blessing, *Milwaukee School of Engineering*
Jason Deane, *Virginia Polytechnic Institute and State University*
Barry Flaschbart, *Missouri University of Science and Technology*
Dianne Hall, *Auburn University*
Jeff Hassett, *University of Utah*
Barbara Hewitt, *Texas A&M, Kingsville*
William Hochstettler, *Franklin University*
Nitin Kale, *University of Southern California, Los Angeles*
Johnny Li, *South University*
Mike Morris, *Southeastern Oklahoma State University*
Catherine Ricardo, *Iona College*
Kevin Roberts, *DeVry University*
Ioulia Rytikova, *George Mason University*
Christelle Scharff, *Pace University*
K. David Smith, *Cameron University*
Marcia Williams, *Bellevue Community College*

David Kroenke would like to thank our editor, Bob Horan, for his advice and assistance through four editions, so far(!), of this text. I also thank Kelly Loftus for keeping this project on track. Most importantly, I wish to thank my wife, Lynda, for her love and support through so many of these writing projects!

David Auer would like to thank David Kroenke for his generosity, insight, and support during this project. Our editor, Bob Horan; our assistant editor, Kelly Loftus; and our production project manager, Kelly Warsak, for their professionalism, insight, support, and assistance in the development of this project. My wife, Donna, for her love, encouragement, and patience while I was completing this project.

David Kroenke
Seattle, Washington

David Auer
Bellingham, Washington

Amazon Ex. 1008.0025

About the Authors

David M. Kroenke



Work Experience

David M. Kroenke has more than 35 years experience in the computer industry. He began as a computer programmer for the U.S. Air Force, working both in Los Angeles and at the Pentagon, where he developed one of the world's first DBMS products while part of a team that created a computer simulation of World War III. That simulation served a key role for strategic weapons studies during a 10-year period of the Cold War.

From 1973 to 1978, Kroenke taught in the College of Business at Colorado State University. In 1977, he published the first edition of *Database Processing*, a significant and successful textbook that, over 30 years later, you now are reading in its 11th edition. In 1978, he left Colorado State and joined Boeing Computer Services, where he managed the team that designed database management components of the IPAD project. After that, he joined with Steve Mitchell to form Mitchell Publishing and worked as an editor and author, developing texts, videos, and other educational products and seminars. Mitchell Publishing was acquired by Random House in 1986. During these years he also worked as an independent consultant, primarily as a database disaster repairman helping companies recover from failed database projects.

In 1982, Kroenke was one of the founding directors of the Microrim Corporation. From 1984 to 1987, he served as the Vice President of Product Marketing and Development and managed the team that created and marketed the DBMS product R:base 5000 as well as other related products.

For the next five years, Kroenke worked independently while he developed a new data modeling language called the *semantic object model*. He licensed this technology to the Wall Data Corporation in 1992 and then served as the Chief Technologist for Wall Data's SALSA line of products. He was awarded three software patents on this technology.

Since 1998, Kroenke has continued consulting and writing. His current interests concern the practical applications of data mining techniques on large organizational databases. An avid sailor, he wrote *Know Your Boat: The Guide to Everything That Makes Your Boat Work*, which was published by McGraw-Hill in 2002.

Consulting

Kroenke has consulted with numerous organizations during his career. In 1978, he worked for Fred Brooks, consulting with IBM on a project that became the DBMS product DB2. In 1989, he consulted for the Microsoft Corporation on a project that became Microsoft Access. In the 1990s, he worked with Computer Sciences Corporation and with General Research Corporation for the development of technology and products that were used to model all of the U.S. Army's logistical data as part of the CALS project. Additionally, he has consulted for Boeing Computer Services, the U.S. Air Force Academy, Logicon Corporation, and other smaller organizations.

Publications

- *Database Processing*, Pearson Prentice Hall, 11 editions, 1977–present (coauthor with David Auer, 11th edition)
- *Database Concepts*, Pearson Prentice Hall, four editions, 2004–present (coauthor with David Auer, third and fourth editions)

Amazon Ex. 1008.0026

- *Using MIS*, Pearson Prentice Hall, two editions, 2006–present
- *Experiencing MIS*, Pearson Prentice Hall, two editions, 2007–present
- *MIS Essentials*, Pearson Prentice Hall, one edition, 2009
- *Know Your Boat: The Guide to Everything That Makes Your Boat Work*, McGraw-Hill, 2002
- *Management Information Systems*, Mitchell Publishing/Random House, three editions, 1987–1992
- *Business Computer Systems*, Mitchell Publishing/Random House, five editions, 1981–1990
- *Managing Information for Microcomputers*, Microrim Corporation, 1984 (coauthor with Donald Nilson)
- *Database Processing for Microcomputers*, Science Research Associates, 1985 (coauthor with Donald Nilson)
- *Database: A Professional's Primer*, Science Research Associates, 1978

Teaching

Kroenke taught in the College of Business at Colorado State University from 1973 to 1978. He also has taught part-time in the Software Engineering program at Seattle University. From 1990 to 1991, he served as the Hanson Professor of Management Science at the University of Washington. Most recently, he taught at the University of Washington from 2002 to 2008. During his career, he has been a frequent speaker at conferences and seminars for computer educators. In 1991, the International Association of Information Systems named him Computer Educator of the Year.

Education

B.S., Economics, U.S. Air Force Academy, 1968
M.S., Quantitative Business Analysis, University of Southern California, 1971
Ph.D., Engineering, Colorado State University, 1977

Personal

Kroenke is married, lives in Seattle, and has two grown children and three grandchildren. He enjoys skiing, sailing, and building small boats. His wife tells him he enjoys gardening as well.

David J. Auer



Work Experience

David J. Auer has more than 25 years experience teaching college-level business and information systems courses and for the last 15 years has worked professionally in the field of information technology. He served as a commissioned officer in the U.S. Air Force, with assignments to NORAD and the Alaskan Air Command in air defense operations. He later taught both business administration and music classes at Whatcom Community College, and business courses for the Chapman College Residence Education Center at Whidbey Island Naval Air Station. He was a founder of the Puget Sound Guitar Workshop (now in its 35th year of operations). He worked as a psychotherapist and organizational development consultant for the Whatcom Counseling and Psychiatric Clinic's Employee Assistance Program, and provided training for the Washington State Department of Social and Health Services. He has taught for Western Washington University's College of Business and Economics since 1981 and has been the college's Director of Information Systems and Technology Services since 1994.

Publications

- *Database Processing*, Pearson Prentice Hall, one edition, 2009 (coauthor with David Kroenke)
- *Database Concepts*, Pearson Prentice Hall, two editions, 2007–present (coauthor with David Kroenke)

Amazon Ex. 1008.0027

- *Network Administrator: NetWare 4.1*, Course Technology, 1997 (coauthors Ted Simpson and Mark Ciampa)
- *New Perspectives on Corel Quattro Pro 7.0 for Windows 95*, Course Technology, 1997 (coauthors June Jamrich Parsons, Dan Oja, and John Leschke)
- *New Perspectives on Microsoft Excel 7 for Windows 95—Comprehensive*, Course Technology, 1996 (coauthors June Jamrich Parsons and Dan Oja)
- *New Perspectives on Microsoft Office Professional for Windows 95—Intermediate*, Course Technology, 1996 (coauthors June Jamrich Parsons, Dan Oja, Beverly Zimmerman, Scott Zimmerman, and Joseph Adamski)
- *The Student's Companion for Use with Practical Business Statistics* (3rd ed.), Irwin, 1993
- *Microsoft Excel 5 for Windows—New Perspectives Comprehensive*, Course Technology, 1995 (coauthors June Jamrich Parsons and Dan Oja)
- *Introductory Quattro Pro 6.0 for Windows*, Course Technology, 1995 (coauthors June Jamrich Parsons and Dan Oja)
- *Introductory Quattro Pro 5.0 for Windows*, Course Technology, 1994 (coauthors June Jamrich Parsons and Dan Oja)
- *The Student's Companion for Use with Practical Business Statistics* (2nd ed.), Irwin, 1993

Teaching

Auer has taught in the College of Business and Economics at Western Washington University from 1981 to the present. From 1975 to 1981, he taught part time for community colleges, and from 1981 to 1984 he taught part time for the Chapman College Residence Education Center System. During his career, he has taught a wide range of courses in Quantitative Methods, Production and Operations Management, Statistics, Finance and Management Information Systems. In MIS, he has taught Principles of Management Information Systems, Business Database Development, Computer Hardware and Operating Systems, and Telecommunications and Network Administration.

Education

B.A., English Literature, University of Washington, 1969
B.S., Mathematics and Economics, Western Washington University, 1978
M.A., Economics, Western Washington University, 1980
M.S., Counseling Psychology, Western Washington University, 1991

Personal

Auer is married, lives in Bellingham, Washington, and has two grown children and five grandchildren. He is active in his community, where he has been President of his neighborhood association and served on the City of Bellingham Planning and Development Commission. He enjoys music, playing acoustic and electric guitar, five-string banjo, and a bit of mandolin.

Amazon Ex. 1008.0028

Part 1

Getting Started

The two chapters in Part 1 provide an introduction to database processing. In Chapter 1, we consider the characteristics of databases and describe important database applications. Chapter 1 also describes the various database components and provides a survey of the knowledge you need to learn from this text. The chapter also summarizes the history of database processing.

You will start working with a database in Chapter 2 and use that database to learn how to use Structured Query Language (SQL), a database-processing language, to query database data. You will learn how to query both single and multiple tables, and you will use SQL to investigate a practical example—looking for patterns in stock market data. Together, these two chapters will give you a sense of what databases are and how they are processed.



Amazon Ex. 1008.0029



1 Introduction

Chapter Objectives

- To understand the nature and characteristics of databases.
- To survey some important and interesting database applications.
- To gain a general understanding of tables and relationships.
- To describe the components of a Microsoft Access database system and explain the functions they perform.
- To describe the components of an enterprise-class database system and explain the functions they perform.
- To define the term database management system (DBMS) and describe the functions of a DBMS.
- To define the term database and describe what is contained within the database.
- To define the term metadata and provide examples of metadata.
- To define and understand database design from existing data.
- To define and understand database design as new systems development.
- To define and understand database design in database redesign.
- To understand the history and development of database processing.

This chapter introduces database processing. We will first consider the nature and characteristics of databases and then survey a number of important and interesting database applications. Next, we will describe the components of a database system and then, in general terms, describe how databases are designed. After that, we will survey the knowledge that you need to work with databases as an application developer or as a database

Amazon Ex. 1008.0030

administrator. Finally, we conclude this introduction with a brief history of database processing.

This chapter assumes a minimal knowledge of database use. It assumes that you have used a product such as Microsoft Access to enter data into a form, to produce a report, and possibly to execute a query. If you have not done these things, you should obtain a copy of Microsoft Access 2007 and work through the tutorial in Appendix A.

The Characteristics of Databases

The purpose of a database is to help people keep track of things, and the most commonly used type of database is the **relational database**. We will discuss the relational database model in depth in Chapter 3, so for now we just need to understand a few basic facts about how a relational database helps people track things of interest to them.

First, a relational database stores data in tables. **Data** are recorded facts and numbers. A **table** has rows and columns, like those in a spreadsheet. A database usually has multiple tables, and each table contains data about a different type of thing. For example, Figure 1-1 shows a database with two tables: the STUDENT table holds data about students, and the CLASS table holds data about classes.

Each **row** of a table has data about a particular occurrence or **instance** of the thing of interest. For example, each row of the STUDENT table has data about one of four students: Cooke, Lau, Harris, and Greene. Similarly, each row of the CLASS table has data about a particular class. Because each row *records* the data for a specific instance, rows are also known as **records**. Each **column** of a table stores a characteristic common to all rows. For example, the first column of STUDENT stores StudentNumber, the second column stores LastName, and so forth.

BY THE WAY

A table and a *spreadsheet* (also known as a *worksheet*) are very similar in that you can think of both as having rows, columns, and cells. The details that define a table as something different from a spreadsheet are discussed in Chapter 3. For now, the main differences you will see are that tables have column names instead of identifying letters (for example, *Name* instead of *A*) and that the rows are not necessarily numbered.

Although, in theory, you could switch the rows and columns by putting instances in the columns and characteristics in the rows, this is never done. Every database in this text, and 99.999999 percent of all databases throughout the world, store instances in rows and characteristics in columns.

A Note on Naming Conventions

In this text, table names appear in capital letters. This convention will help you to distinguish table names in explanations. However, you are not required to set table names in capital letters. Microsoft Access and similar programs will allow you to write a table name as STUDENT, student, Student, stuDent, or in some other way.

Additionally, in this text column names begin with a capital letter. Again, this is just a convention. You could write the column name Term as term, teRm, TERM, or in any other way. To ease readability, we will sometimes create compound column names in which the first letter of each element of the compound word is capitalized. Thus, in Figure 1-1, the STUDENT table has columns StudentNumber, LastName, FirstName, and EmailAddress. Again, this capitalization is just a convenient convention. However, following these or other consistent conventions will make interpretation of database structures easier. For example, you will always know that STUDENT is the name of a table and that Student is the name of a column of a table.

Amazon Ex. 1008.0031

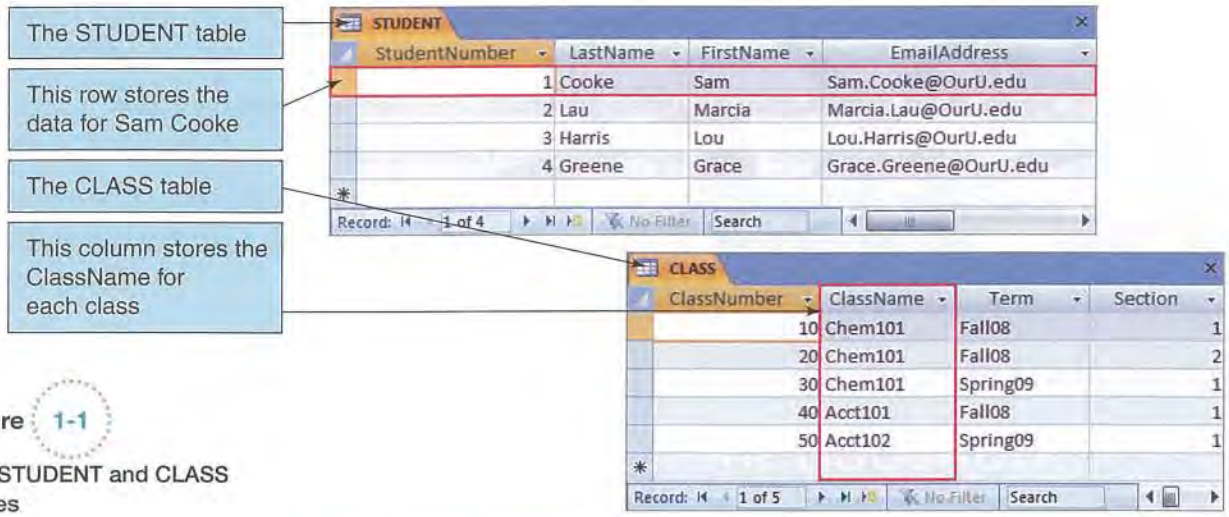


Figure 1-1

The STUDENT and CLASS Tables

A Database Has Data and Relationships

Figure 1-1 illustrates how database tables are structured to store data, but a database is not complete unless it also shows the relationships among the rows of data. To see why this is important, examine Figure 1-2. In this figure, the database contains all of the basic data shown in Figure 1-1 together with a GRADE table. Unfortunately, the relationships between the data are missing. In this format, the GRADE data are useless. It is like the joke about the sports commentator who announced: "Now for tonight's baseball scores: 2–3, 7–2, 1–0, and 4–5." The scores are useless without knowing the teams that earned them. Thus, a database contains both data and the relationships among the data.

Figure 1-3 shows the complete database that contains not only the data about students, classes, and grades, but also the relationships among the rows in those tables. For example, StudentNumber 100, who is Sam Cooke, earned a Grade of 3.7 in ClassNumber 10, which is Chem101. He also earned a Grade of 3.5 in ClassNumber 40, which is Acct101.

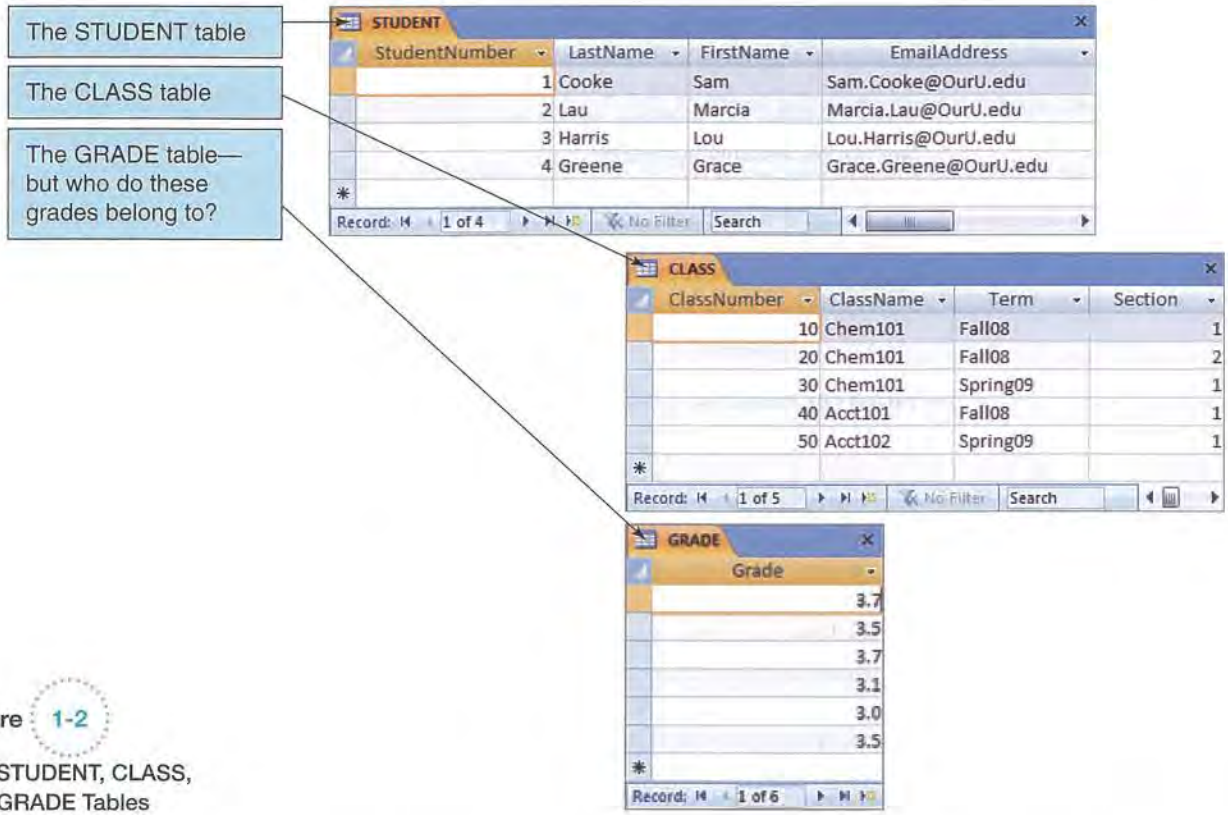


Figure 1-2

The STUDENT, CLASS, and GRADE Tables

Amazon Ex. 1008.0032

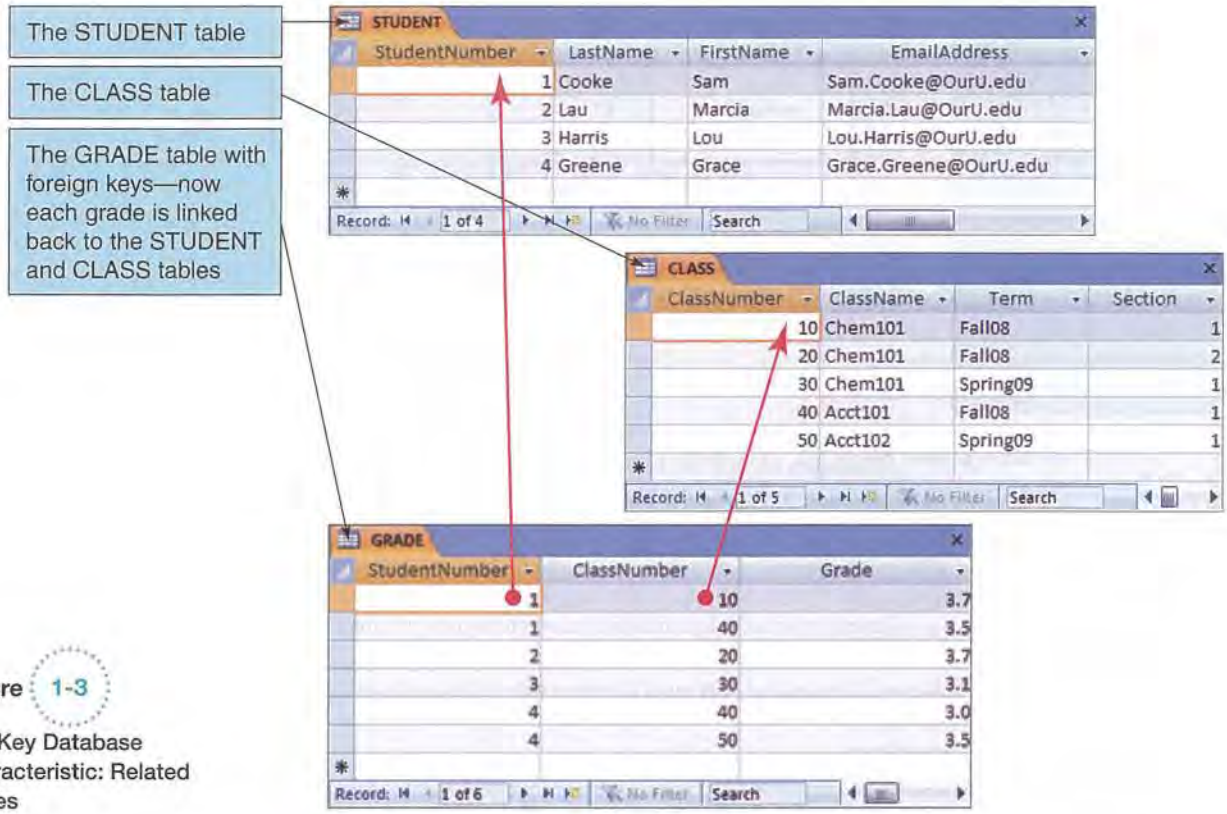


Figure 1-3

The Key Database Characteristic: Related Tables

Figure 1-3 illustrates an important characteristic of database processing. Each row in a table is uniquely indentified by a **primary key**, and the values of these keys are used to create the relationships between the tables. For example, in the STUDENT table, StudentNumber serves as the primary key. Each value of StudentNumber is unique, and uniquely indentifies on student. Thus, StudentNumber 100 identifies Sam Cooke. Similarly, ClassNumber in the CLASS table identifies each class.

By comparing Figures 1-2 and 1-3, we can see how the primary keys of STUDENT and CLASS were added to the GRADE table to provide GRADE with a primary key of (StudentNumber, ClassNumber) to uniquely identify each row. More important, in GRADE StudentNumber and ClassNumber each now serves as a **foreign key**. A foreign key provides the link between two tables. By adding a foreign key, we create a **relationship** between the two tables.

Figure 1-4 shows a Microsoft Access 2007 view of the tables and relationships shown in Figure 1-3. In Figure 1-4, primary keys in each table are marked with key symbols, and connecting lines representing the relationships are drawn from the foreign keys (in GRADE) to the corresponding primary keys (in STUDENT and CLASS). The symbols on the relationship line (the number 1 and the infinity symbol) mean that, for example, one student in STUDENT can be linked to many grades in GRADE.

Databases Create Information

In order to make decisions, we need information upon which to base those decisions. Since we have already defined *data* as recorded facts and numbers, we can now define[1] **information** as:

- Knowledge derived from data.
- Data presented in a meaningful context.
- Data processed by summing, ordering, averaging, grouping, comparing or other similar operations.

[1] These definitions are from David M. Kroenke's books *Using MIS* (2nd ed.) (Upper Saddle River, NJ: Prentice-Hall, 2008) and *Experiencing MIS* (2nd ed.) (Upper Saddle River, NJ: Prentice-Hall, 2009). See these books for a full discussion of these definitions, as well as a discussion of a fourth definition, "a difference that makes a difference."

Amazon Ex. 1008.0033

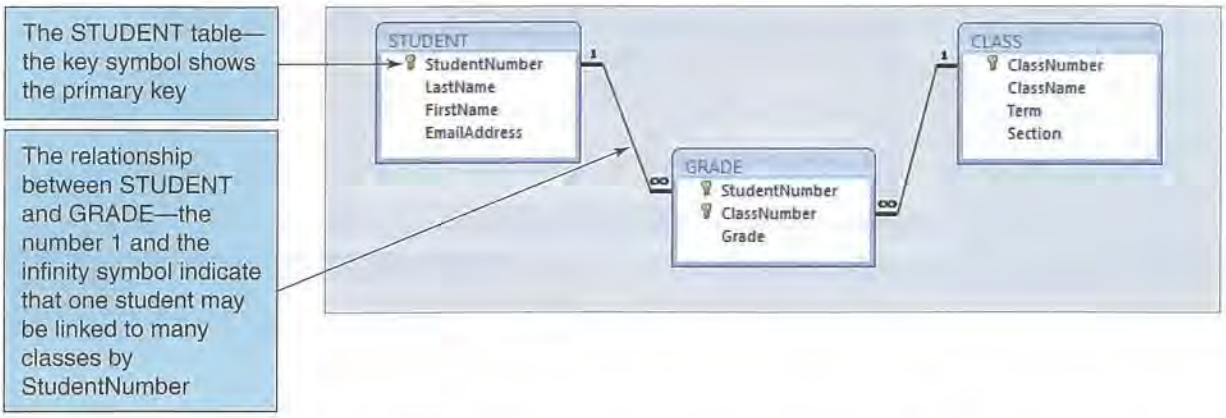


Figure 1-4
Microsoft Access 2007 View of Tables and Relationships

Databases record facts and figures, so they record data. They do so, however, in a way that enables them to produce information. The data in Figure 1-3 can be manipulated to produce a student's GPA, the average GPA for a class, the average number of students in a class, and so forth. In Chapter 2, you will be introduced to a language called Structured Query Language (SQL) that you can use to produce information from database data.

To summarize, relational databases store data in tables, and they represent the relationships among the rows of those tables. They do so in a way that facilitates the production of information. We will discuss the relational database model in depth in Part 2 of this book.

Database Examples

Today, database technology is part of almost every information system. This fact is not surprising when we consider that every information system needs to store data and the relationships among those data. Still, the vast array of applications that use this technology is staggering. Consider, for example, the applications listed in Figure 1-5.

Single-User Database Applications

In Figure 1-5, the first application is used by a single salesperson to keep track of the customers she has called and the contacts that she's had with them. Most salespeople do not build their own contact manager applications; instead, they license products such as GoldMine (see *www.frontrange.com/goldmine*) or ACT! (see *www.act.com*).

Multiuser Database Applications

The next applications in Figure 1-5 are those that involve more than one user. The patient-scheduling application, for example, may have 15 to 50 users. These users will be appointment clerks, office administrators, nurses, dentists, doctors, and so forth. A database like this one may have as many as 100,000 rows of data in perhaps 5 or 10 different tables.

When more than one user employs a database application, there is always the chance that one user's work may interfere with another's. Two appointment clerks, for example, might assign the same appointment to two different patients. Special concurrency-control mechanisms are used to coordinate activity against the database to prevent such conflict. You will learn about these mechanisms in Chapter 9.

The third row of Figure 1-5 shows an even larger database application. A customer relationship management (CRM) system is an information system that manages customer contacts from initial solicitation through acceptance, purchase, continuing purchase, support, and so forth. CRM systems are used by salespeople, sales managers, customer service and support staff, and other personnel. A CRM database in a larger company might have 500 users and 10 million or more rows in perhaps 50 or more tables. According to Microsoft, in 2004 Verizon had an SQL Server customer database that contained more than 15 terabytes of data. If that data were published in books, a bookshelf 450 miles long would be required to hold them.

Enterprise resource planning (ERP) is an information system that touches every department in a manufacturing company. It includes sales, inventory, production planning, purchasing,

Amazon Ex. 1008.0034

Application	Example Users	Number of Users	Typical Size	Remarks
Sales contact manager	Salesperson	1	2,000 rows	Products such as GoldMine and Act! are database centric.
Patient appointment (doctor, dentist)	Medical office	15 to 50	100,000 rows	Vertical market software vendors incorporate databases into their software products.
Customer Relationship Management (CRM)	Sales, marketing, or customer service departments	500	10 million rows	Major vendors such as Siebel and PeopleSoft build applications around the database.
Enterprise Resource Planning (ERP)	An entire organization	5,000	10 million+ rows	SAP uses a database as a central repository for ERP data.
E-commerce site	Internet users	Possibly millions	1 billion+ rows	Drugstore.com has a database that grows at the rate of 20 million rows per day!
Digital dashboard	Senior managers	500	100,000 rows	Extractions, summaries, and consolidations of operational databases.
Data mining	Business analysts	25	100,000 to millions+	Data are extracted, reformatted, cleaned, and filtered for use by statistical data mining tools.

Figure 1-5

Example Database Uses

and other business functions. SAP is the leading vendor of ERP applications, and a key element of its product is a database that integrates data from these various business functions. An ERP system may have 5,000 or more users and perhaps 100 million rows in several hundred tables.

E-Commerce Database Applications

E-commerce is another important database application. Databases are a key component of e-commerce order entry, billing, shipping, and customer support. Surprisingly, however, the largest databases at an e-commerce site are not order-processing databases. The largest databases are those that track customer browser behavior. Most of the prominent e-commerce companies, such as Amazon.com (*www.amazon.com*) and Drugstore.com (*www.drugstore.com*) keep track of the Web pages and the Web page components that they send to their customers. They also track customer clicks, additions to shopping carts, order purchases, abandoned shopping carts, and so forth.

E-commerce companies use Web activity databases to determine which Web page items are popular and successful and which are not. They also can conduct experiments to determine if a purple background generates more orders than a blue one, and so forth. Such Web usage databases are huge. For example, Drugstore.com adds 20 million rows to its Web log database each day!

Reporting and Data Mining Database Applications

Two other example applications in Figure 1-5 are digital dashboards and data mining applications. These applications use the data generated by order processing and other operational systems to produce information to help manage the enterprise. Such applications do not generate new data; instead they summarize existing data to provide insights to management. Digital dashboards and other reporting systems assess past and current performance. Data

Amazon Ex. 1008.0035

mining applications predict future performance. We will consider such applications in Chapter 15. The bottom line is that database technology is used in almost every information system and involves databases ranging in size from a few thousand rows to many millions of rows.

BY THE WAY Do not assume that just because a database is small that its structure is simple. For example, consider parts distribution for a company that sells $1 million in parts per year and parts distribution for a company that sells $100 million in parts per year. Despite the difference in sales, the companies have similar databases. Both have the same kinds of data, about the same number of tables of data, and the same level of complexity in data relationships. Only the amount of data varies from one to the other. Thus, although a database for a small business may be small, it is not necessarily simple.

The Components of a Database System

As shown in Figure 1-6, a database system is typically defined to consist of four components: users, the database application, the database management system (DBMS), and the database. However, given the importance of **Structured Query Language (SQL)**, an internationally recognized standard language that is understood by all commercials DBMS products, in database processing and the fact that database applications typically send SQL statements to the DBMS for processing, we can refine our illustration of a database system to appear as shown in Figure 1-7.

Starting from the right of Figure 1-7, the **database** is a collection of related tables and other structures. The **database management system (DBMS)** is a computer program used to create, process, and administer the database. The DBMS receives requests encoded in SQL and translates those requests into actions on the database. The DBMS is a large, complicated program that is licensed from a software vendor; companies almost never write their own DBMS programs.

A **database application** is a set of one or more computer programs that serves as an intermediary between the user and the DBMS. Application programs read or modify database data by sending SQL statements to the DBMS. Application programs also present data to users in the format of forms and reports. Application programs can be acquired from software vendors, and they are also frequently written in-house. The knowledge you gain from this text will help you write database applications.

Users, the fourth component of a database system, employ a database application to keep track of things. They use forms to read, enter, and query data, and they produce reports to convey information.

Figure 1-6
The Components of a Database System

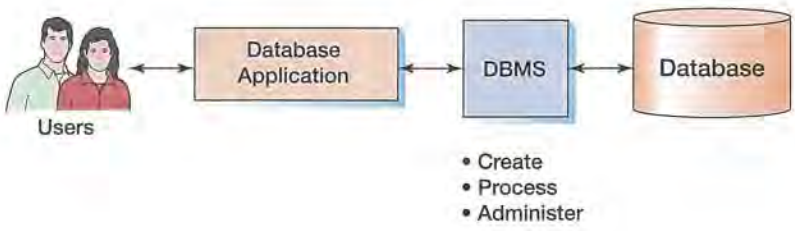
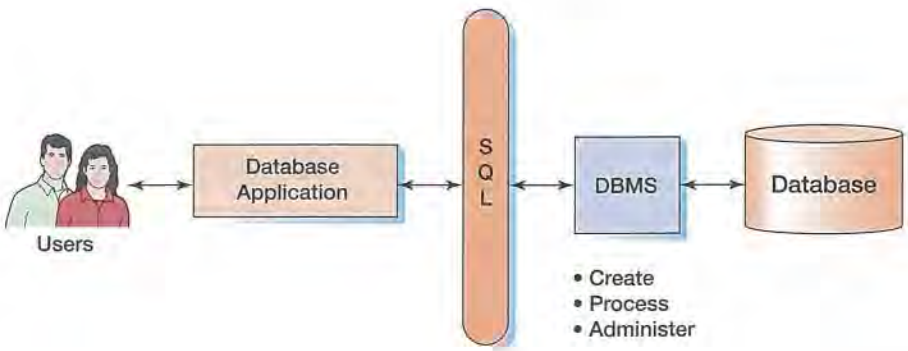


Figure 1-7
The Components of a Database System with SQL



Amazon Ex. 1008.0036

Figure 1-8

Basic Functions of Application Programs

- Create and process forms
- Process user queries
- Create and process reports
- Execute application logic
- Control application

Database Applications and SQL

Figure 1-7 shows the database applications that users interact with directly. Figure 1-8 lists the basic functions of database applications.

First, an application program creates and processes forms. Figure 1-9 shows a typical form for entering and processing student enrollment data for the Student-Class-Grade database shown in Figures 1-3 and 1-4. Notice that this form hides the structure of the underlying tables from the user. By comparing the tables and data in Figures 1-3 and 1-4 to the form in Figure 1-9, we can see that data from the CLASS table appears at the top of the form, while data from the STUDENT table is presented in a tabular section labeled Class Enrollment Data.

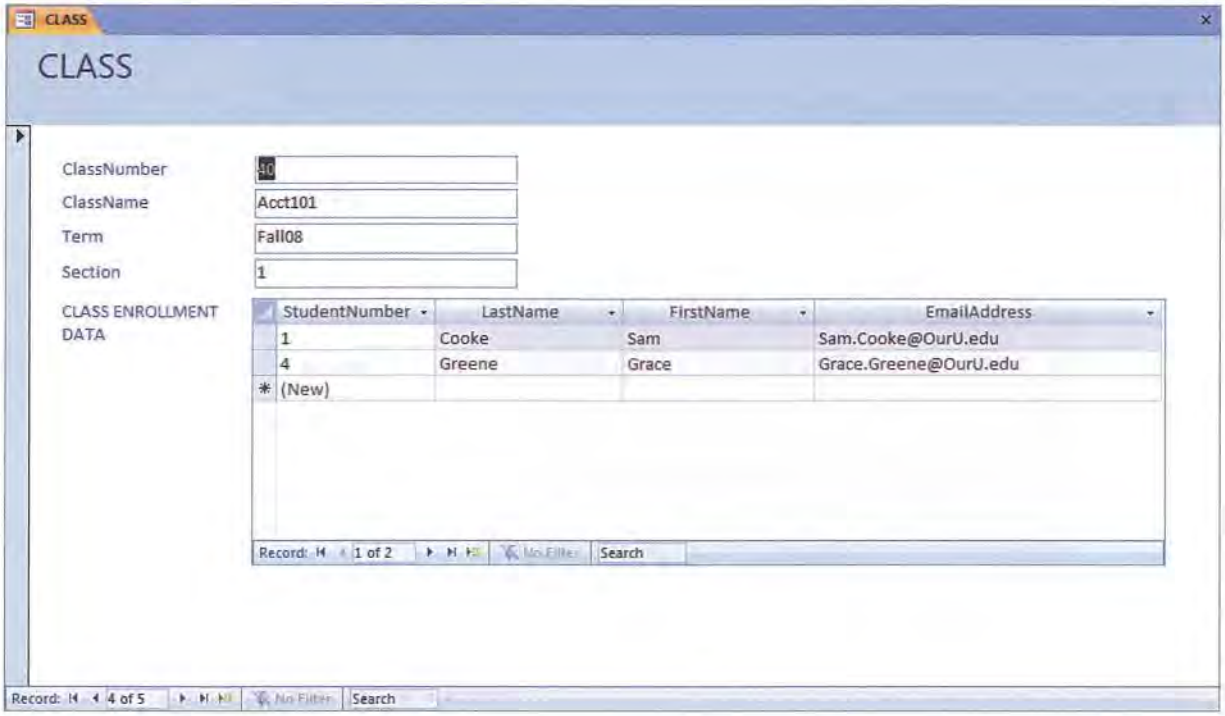
The goal of this form, like that for all data entry forms, is to present the data in a format that is useful for the users, regardless of the underlying table structure. Behind the form, the application processes the database in accordance with the users' actions. The application generates an SQL statement to insert, update, or delete data for any of the tables that underlie this form.

The second function of application programs is to process user queries. The application program first generates a query request and sends it to the DBMS. Results are then formatted and returned to the user. Applications use SQL statements and pass them to the DBMS for processing. To give you a taste of SQL, here is a sample SQL statement for processing the STUDENT table in Figure 1-1:

```
SELECT      LastName, FirstName, EmailAddress
FROM        STUDENT
WHERE       StudentNumber > 2;
```

Figure 1-9

An Example Data Entry Form



Amazon Ex. 1008.0037

Figure 1-10
Example SQL Query Results

Query1

LastName	FirstName	EmailAddress
Harris	Lou	Lou.Harris@OurU.edu
Greene	Grace	Grace.Greene@OurU.edu
*		

Record: 1 of 2 No Filter Search

This SQL statement is a query statement, which asks the DBMS to obtain specific data from a database. In this case, the query asks for the last name, first name, and e-mail address of all students having a StudentNumber greater than 2. The results of this SQL statement are shown (as displayed in Microsoft Access 2007) in Figure 1-10. As shown in Figure 1-10, running this SQL statement will produce the LastName, FirstName, and EmailAddress for students Harris and Greene.

The third function of an application is to create and process reports. This function is somewhat similar to the second because the application program first queries the DBMS for data (again using SQL). The application then formats the query results as a report. Figure 1-11 shows a report that displays all the Student-Class-Grade database shown in Figure 1-3 sorted by Class Number, Section and Last Name. Notice that the report, like the form in Figure 1-9, is structured according to the users' needs, not according to the underlying table structure.

In addition to generating forms, queries, and reports, the application program takes other actions to update the database in accordance with application-specific logic. For example, suppose a user using an order-entry application requests 10 units of a particular item. Suppose further that when the application program queries the database (via the DBMS), it finds that only 8 units are in stock. What should happen? It depends on the logic of that particular application. Perhaps no units should be removed from inventory, and the user should be notified, or perhaps the 8 units should be removed and 2 more placed on back order. Perhaps some other action should be taken. Whatever the case, it is the job of the application program to execute the appropriate logic.

Finally, the last function for application programs listed in Figure 1-8 is to control the application. This is done in two ways. First, the application needs to be written so that only logical options are presented to the user. For example, the application may generate a menu with user choices. In this case, the application needs to ensure that only appropriate choices are available. Second, the application needs to control data activities with the DBMS. The application might

Figure 1-11
Example Report

Class Grade Report

Class Number	Class Name	Term	Section	Last Name	First Name	Grade
10	Chem101	Fall08	1			
				Cooke	Sam	3.7
20	Chem101	Fall08	2			
				Lau	Marcia	3.7
30	Chem101	Spring09	1			
				Harris	Lou	3.1
40	Acct101	Fall08	1			
				Cooke	Sam	3.5
				Greene	Grace	3.0
50	Acct102	Spring09	1			
				Greene	Grace	3.5

Amazon Ex. 1008.0038

direct the DBMS, for example, to make a certain set of data changes as a unit. The application might tell the DBMS to either make all these changes or none of them. You will learn about such control topics in Chapter 9.

The DBMS

The DBMS, or database management system, creates, processes, and administers the database. A DBMS is a large, complicated product that is almost always licensed from a software vendor. One DBMS product is Microsoft Access. Other commercial DBMS products are Oracle Database, from Oracle Corporation; DB2, from IBM Corporation; SQL Server, from Microsoft Corporation; and MySQL, from Sun Microsystems.[2] Dozens of other DBMS products exist, but these five have the lion's share of the market. Figure 1-12 lists the functions of a DBMS.

A DBMS is used to create a database and to create the tables and other supporting structures inside that database. As an example of the latter, suppose that we have an EMPLOYEE table with 10,000 rows and that this table includes a column, DepartmentName, that records the name of the department in which an employee works. Furthermore, suppose that we frequently need to access employee data by DepartmentName. Because this is a large database, searching through the table to find, for example, all employees in the accounting department would take a long time. To improve performance, we can create an index (akin to the index at the back of a book) for DepartmentName to show which employees are in which departments. Such an index is an example of a supporting structure that is created and maintained by a DBMS.

The next two functions of a DBMS are to read and modify database data. To do this, a DBMS receives SQL and other requests and transforms those requests into actions on the database files. Another DBMS function is to maintain all the database structures. For example, from time to time it might be necessary to change the format of a table or another supporting structure. Developers use a DBMS to make such changes.

With most DBMS products, it is possible to declare rules about data values and have a DBMS enforce them. For example, in the Student-Class-Grade database tables in Figure 1-3, what would happen if a user mistakenly entered a value of 900 for StudentNumber in the GRADE table? No such student exists, so such a value would cause numerous errors. To prevent this situation, it is possible to tell the DBMS that any value of StudentNumber in the GRADE table must already be a value of StudentNumber in the STUDENT table. If no such value exists, the insert or update request should be disallowed. The DBMS then enforces these rules, which are called **referential integrity constraints**.

The last three functions of a DBMS listed in Figure 1-12 have to do with database administration. A DBMS controls **concurrency** by ensuring that one user's work does not inappropriately interfere with another user's work. This important (and complicated) function is discussed in Chapter 9. Also, a DBMS contains a security system that ensures that only authorized users perform authorized actions on the database. For example, users can be prevented from seeing certain data. Similarly, users' actions can be confined to making only certain types of data changes on specified data.

Finally, a DBMS provides facilities for backing up database data and recovering it from backups, when necessary. The database, as a centralized repository of data, is a valuable

Figure 1-12
Functions of a DBMS

- Create database
- Create tables
- Create supporting structures (e.g., indexes)
- Read database data
- Modify (insert, update, or delete) database data
- Maintain database structures
- Enforce rules
- Control concurrency
- Provide security
- Perform backup and recovery

[2] As this book goes to press, Oracle Corporation has concluded an agreement with Sun Microsystems to purchase Sun Microsystems. This will make Oracle the owner of both Oracle Database and MySQL. See http://www.sun.com/third-party/global/oracle/.

Amazon Ex. 1008.0039

organizational asset. Consider, for example, the value of a book database to a company such as Amazon.com. Because the database is so important, steps need to be taken to ensure that no data will be lost in the event of errors, hardware or software problems, or natural or human catastrophes.

The Database

The last component in Figure 1-7 is the database. A database is a self-describing collection of integrated tables. **Integrated tables** are tables that store both data and the relationships among the data. The tables in Figure 1-3 are *integrated* because they store not just student, class, and grade data, but also data about the relationships among the rows of data.

A database is **self-describing** because it contains a description of itself. Thus, databases contain not only tables of user data, but also tables of data that describe that user data. Such descriptive data is called **metadata**, because it is data about data. The form and format of metadata varies from DBMS to DBMS. Figure 1-13 shows generic metadata tables that describe the tables and columns for the database in Figure 1-3.

You can examine metadata to determine if particular tables, columns, indexes, or other structures exist in a database. For example, the following statement queries the Microsoft SQL

Figure 1-13
Typical Metadata Tables

USER_TABLES Table

TableName	NumberColumns	PrimaryKey
STUDENT	3	StudentNumber
CLASS	4	ClassNumber
GRADE	3	(StudentNumber, ClassNumber)

USER_COLUMNS Table

ColumnName	TableName	DataType	Length (bytes)
StudentNumber	STUDENT	Integer	4
LastName	STUDENT	Text	25
FirstName	STUDENT	Text	25
EmailAddress	STUDENT	Text	100
ClassNumber	CLASS	Integer	4
Name	CLASS	Text	25
Term	CLASS	Text	12
Section	CLASS	SmallInteger	2
StudentNumber	GRADE	Integer	4
ClassNumber	GRADE	Integer	4
Grade	GRADE	Decimal	(2, 1)

Amazon Ex. 1008.0040

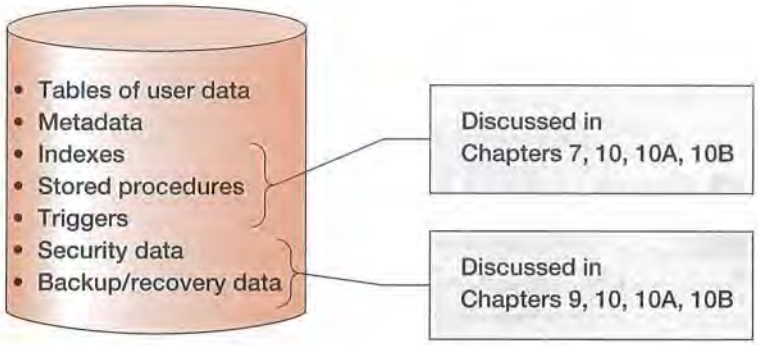


Figure 1-14
Database Contents

Server metadata table SYSOBJECTS to determine if a user table (Type = 'U') named CLASS exists in the database. If it does, the table is dropped (removed) from the database.

```
IF EXITS
      (SELECT     *
      FROM        SYSOBJECTS
      WHERE       [Name]='CLASS'
         AND      Type='U')
   DROP TABLE CLASS;
```

Do not be concerned with the syntax of this statement. You will learn what it means and how to write such statements yourself as we proceed. For now, just understand that this is one way that database administrators use metadata.

> **BY THE WAY** Because metadata is stored in tables, you can use SQL to query it, as just illustrated. Thus, by learning how to write SQL to query user tables, you will also learn how to write SQL to query metadata. To do that, you just apply the SQL statements to metadata tables rather than user tables.

In addition to user tables and metadata, databases contain other elements, as shown in Figure 1-14. These other components will be described in detail in subsequent chapters. For now, however, understand that indexes are structures that speed the sorting and searching of database data. Triggers and stored procedures are programs that are stored within the database. Triggers are used to maintain database accuracy and consistency and to enforce data constraints. Stored procedures are used for database administration tasks and are sometimes part of database applications. You will learn more about these different elements in Chapters 7, 10, 10A, and 10B.

Security data defines users, groups, and allowed permissions for users and groups. The particulars depend on the DBMS product in use. Finally, backup and recovery data are used to save database data to backup devices as well as to recover the database after failure. You will learn more about security and backup and recovery data in Chapters 9, 10, 10A, and 10B.

Personal Versus Enterprise-Class Database Systems

We can divide database systems and DBMS products into two classes: personal database systems and enterprise-class database systems.

What Is Microsoft Access?

We need to clear up a common misconception: Microsoft Access is *not* just a DBMS. Rather, it is a **personal database system**: a DBMS *plus* an application generator. Although Microsoft Access contains a DBMS engine that creates, processes, and administers the database, it also contains form, report, and query components that are the Microsoft Access application

Amazon Ex. 1008.0041

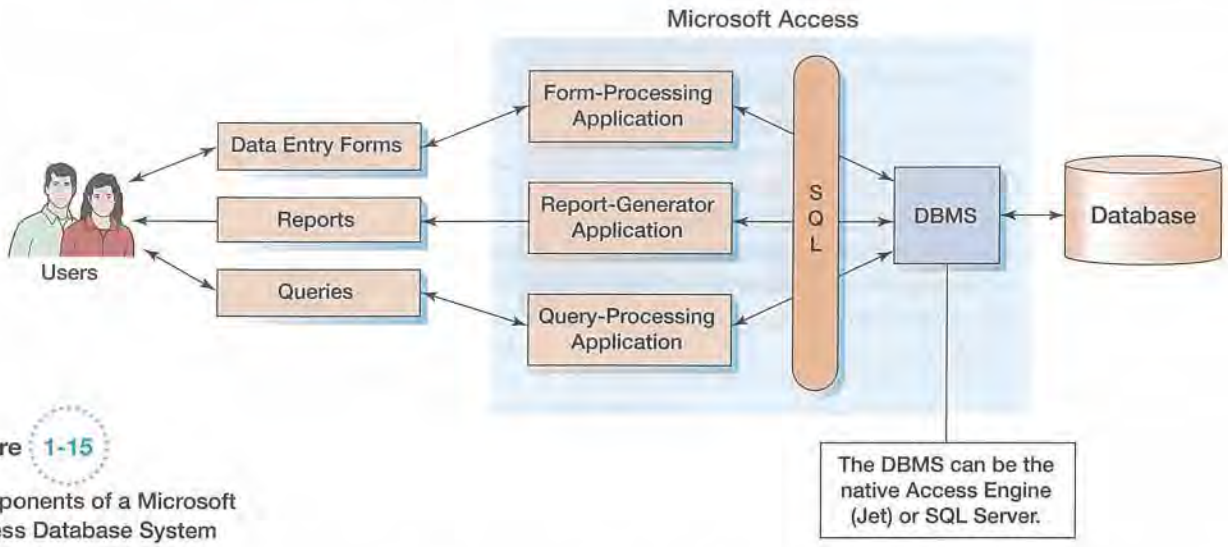


Figure 1-15

Components of a Microsoft Access Database System

generator. The components of Microsoft Access are shown in Figure 1-15, which illustrates that the Microsoft Access form, report, and query applications create SQL statements and pass them to the DBMS for processing.

Microsoft Access is a low-end product intended for individuals and small workgroups. As such, Microsoft has done all that it can to hide the underlying database technology from the user. Users interact with the application through data entry forms like the one shown in Figure 1-9. They also request reports and perform queries against the database data. Microsoft Access then processes the forms, produces the reports, and runs the queries. Internally, the application components hidden under the Access cover use SQL to call the DBMS, which is also hidden under that cover. At Microsoft, the DBMS engine within Access is called Jet. You seldom hear about Jet because Microsoft does not sell Jet as a separate product.

BY THE WAY Although Microsoft Access is the best known personal database system, it is not the only one. OpenOffice Base is a personal database system distributed as part of the OpenOffice.org software suite, which is available at *www.openoffice.org*.

Although hiding the technology is an effective strategy for beginners working on small databases, it won't work for database professionals who work with applications, such as most of those described in Figure 1-5. For larger, more complex databases, it is necessary to understand the technology and components that Microsoft hides.

Nonetheless, because Microsoft Access is included in the Microsoft Office suite, it is often the first DBMS used by students. In fact, you may have already learned to use Microsoft Access in other classes you have taken, and in this book we will provide some examples using Microsoft Access 2007. If you are not familiar with Microsoft Access 2007, you should work through Appendix A, "Getting Started with Microsoft Access 2007."

BY THE WAY With Microsoft Access 2000 and later versions, you can effectively replace Jet with Microsoft's enterprise-class DBMS product—SQL Server. You would do this if you wanted to process a large database or if you needed the advanced functions and features of SQL Server.

Amazon Ex. 1008.0042

What Is an Enterprise-Class Database System?

Figure 1-16 shows the components of an **enterprise-class database system**. Here, the applications and the DBMS are not under the same cover as they are in Microsoft Access. Instead, the applications are separate from each other and separate from the DBMS.

Database Applications in an Enterprise-Class Database System

Earlier in this chapter, we discussed the basic functions of an application program, and these functions are summarized in Figure 1-8. However, as exemplified by the list in Figure 1-5, dozens of different types of database applications are available, and database applications in an enterprise-class database system introduce functions and features beyond the basics. For example, Figure 1-16 shows applications that connect to the database over a corporate network. Such applications are sometimes called *client/server applications* because the application program is a client that connects to a database server. Client/server applications often are written in programming languages such as VB.NET, C++, or Java.

A second category of applications in Figure 1-16 is e-commerce and other applications that run on a Web server. Users connect to such applications via Web browsers such as Microsoft Internet Explorer, Mozilla Firefox, and Google Chrome. Common Web servers include Microsoft's Internet Information Server (IIS) and Apache. Common languages for Web server applications are PHP, Java, and the Microsoft .NET languages such as C# and VB.NET. We will discuss some of the technology for such applications in Chapter 13.

A third category of applications is reporting applications that publish the results of database queries on a corporate portal or other Web site. Such reporting applications are often created using third-party report generation and digital dashboard products from vendors such as IBM (Cognos) and MicroStrategy (MicroStrategy 9). We will describe these applications in Chapter 15.

The last category of applications is XML Web services. These applications use a combination of the XML markup language and other standards to enable program-to-program communication. In this way, the code that comprises an application is distributed over several different computers. Web services can be written in Java or any of the .NET languages. We will discuss this important new class of applications in Chapter 13.

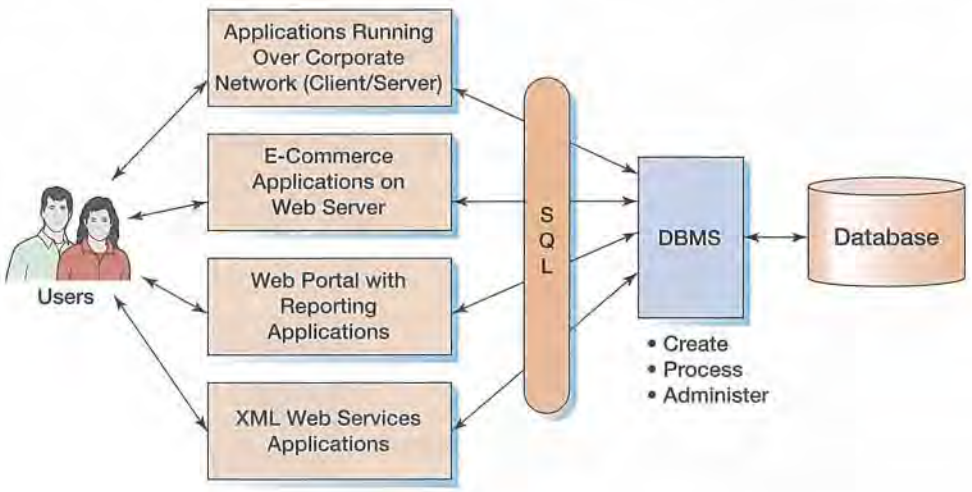
All of these database applications get and put database data by sending SQL statements to the DBMS. These applications may create forms and reports, or they may send their results to other programs. They also may implement application logic that goes beyond simple form and report processing. For example, an order entry application uses application logic to deal with out-of-stock items and backorders.

The DBMS in an Enterprise-Class Database Systems

As stated earlier, the DBMS manages the database. It processes SQL statements and provides other features and functions for creating, processing, and administering the database. Figure 1-17

Figure 1-16

Components of an Enterprise-class Database System



Amazon Ex. 1008.0043

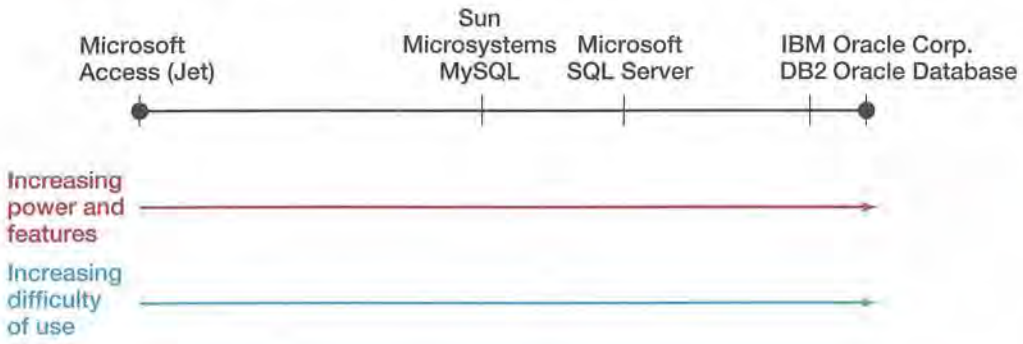


Figure 1-17
Common Professional View of DBMS Products

presents the five most prominent DBMS products. The products are shown in order of increasing power, features, and difficulty of use.

Access (really Jet) is the easiest to use and the least powerful. Sun Microsystems MySQL is a powerful, open source DBMS frequently chosen for Web applications. Microsoft SQL Server has far more power than its stable mate Microsoft Access—it can process larger databases, faster, and it includes features for multiuser control, backup and recovery, and other administrative functions. DB2 is a DBMS product from IBM. Most people would agree that it has faster performance than SQL Server, that it can handle larger databases, and that it is also more difficult to use. Finally, the fastest and most capable DBMS is Oracle Database from the Oracle Corporation. Oracle Database can be configured to offer very high performance on exceedingly large databases that operate 24/7, year after year. Oracle Database is also far more difficult to use and administer than Microsoft SQL Server.

Database Design

Database design is both difficult and important. Determining the proper structure of tables, the proper relationships among tables, the appropriate data constraints, and other structural components is challenging, and sometimes even daunting. Consequently, the world is full of poorly designed databases. Such databases do not perform well. They may require application developers to write overly complex and contrived SQL to get wanted data, they may be difficult to adapt to new and changing requirements, or they fail in some other way.

Because database design is both difficult and important, we will devote most of the first half of this text to the topic. As shown in Figure 1-18, there are three types of database design:

- Database design from existing data
- Database design for new systems development
- Database redesign of an existing database

Database Design from Existing Data

The first type of database design involves databases that are constructed from existing data, as shown in Figure 1-19. In some cases, a development team is given a set of spreadsheets or a set of text files with tables of data. The team is required to design a database and import the data from those spreadsheets and tables into a new database.

Alternatively, databases can be created from extracts of other databases. This alternative is especially common in business intelligence (BI) systems, which include reporting and data mining applications. For example, data from an operational database, such as a CRM or ERP database, may be copied into a new database that will be used only for studies and analysis. As you will learn in Chapter 13, such databases are used in facilities called **data warehouses** and **data marts**. The data warehouse and data mart databases store data specifically organized for research and reporting purposes, and these data often are exported to other analytical tools, such as SAS's Enterprise Miner, SPSS's Clementine, or Insightful Corporation's I-Miner.

Amazon Ex. 1008.0044

- From Existing Data (Chapters 3 and 4)
  - Analyze spreadsheets and other data tables
  - Extract data from other databases
  - Design using normalization principles
- New Systems Development (Chapters 5 and 6)
  - Create data model from application requirements
  - Transform data model into database design
- Database Redesign (Chapter 8)
  - Migrate databases to newer databases
  - Integrate two or more databases
  - Reverse engineer and design new databases using normalization principles and data model transformation

Note: Chapter 7 discusses database implementation using SQL. You need that knowledge before you can understand database redesign.

Figure 1-18

Three Types of Database Design

When creating a database from existing data, database developers must determine the appropriate structure for the new database. A common issue is how the multiple files or tables in the new database should be related. However, even the import of a single table can pose design questions. Figure 1-20 shows two different ways of importing a simple table of employees and their departments. Should this data be stored as one table or two?

Decisions such as this are not arbitrary. Database professionals use a set of principles, collectively called **normalization**, or **normal forms**, to guide and assess database designs. You will learn those principles and their role in database design in Chapter 3.

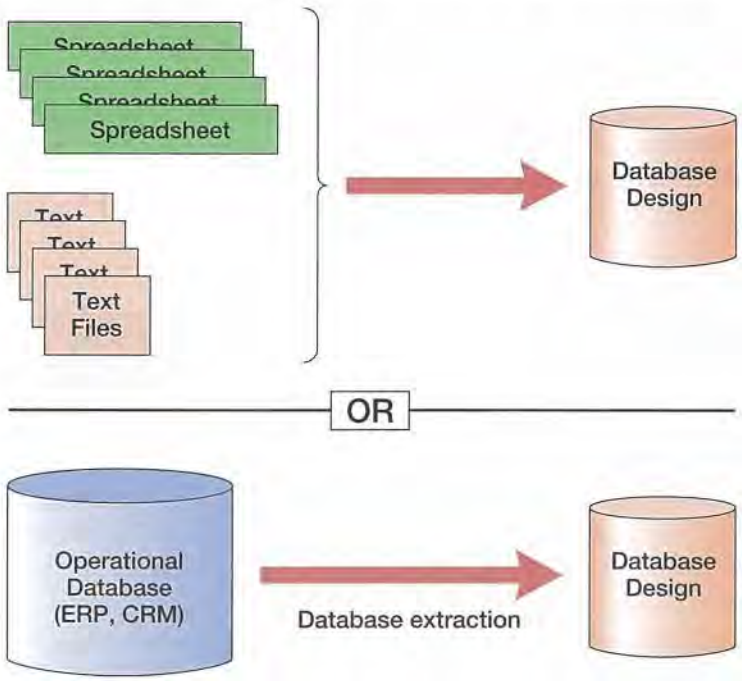
Database Design for New Systems Development

A second way that databases are designed is for the development of new information systems. As shown in Figure 1-21, requirements for a new system, such as desired data entry forms and reports, user requirements statements, use cases, and other requirements, are analyzed to create the database design.

In all but the simplest system development projects, the step from user requirements to database design is too big. Accordingly, the development team proceeds in two steps. First, the

Figure 1-19

Databases Originating from Existing Data



Amazon Ex. 1008.0045

EmpNum	EmpName	DeptNum	DeptName
100	Jones	10	Accounting
150	Lau	20	Marketing
200	McCauley	10	Accounting
300	Griffin	10	Accounting

(a) One-Table Design

OR?

DeptNum	DeptName
10	Accounting
20	Marketing

EmpNum	EmpName	DeptNum
100	Jones	10
150	Lau	20
200	McCauley	10
300	Griffin	10

(b) Two-Table Design

Figure 1-20
Data Import: One or Two Tables?

team creates a **data model** from the requirements statements and then transforms that data model into a database design. You can think of a data model as a blueprint that is used as a design aid on the way to a **database design**, which is the basis for constructing the actual database in a DBMS.

In Chapter 5, you will learn about the most popular data modeling technique—**entity-relationship (ER) data modeling**. You also will see how to use the entity-relationship model to represent a variety of common form and report patterns. Then, in Chapter 6, you will learn how to transform entity-relationship data models into database designs.

Database Redesign

Database redesign also requires that databases are designed. As shown in Figure 1-22, there are two common types of database redesign.

In the first, a database is adapted to new or changing requirements. This process sometimes is called **database migration**. In the migration process, tables may be created, modified, or removed; relationships may be altered; data constraints may be changed; and so forth.

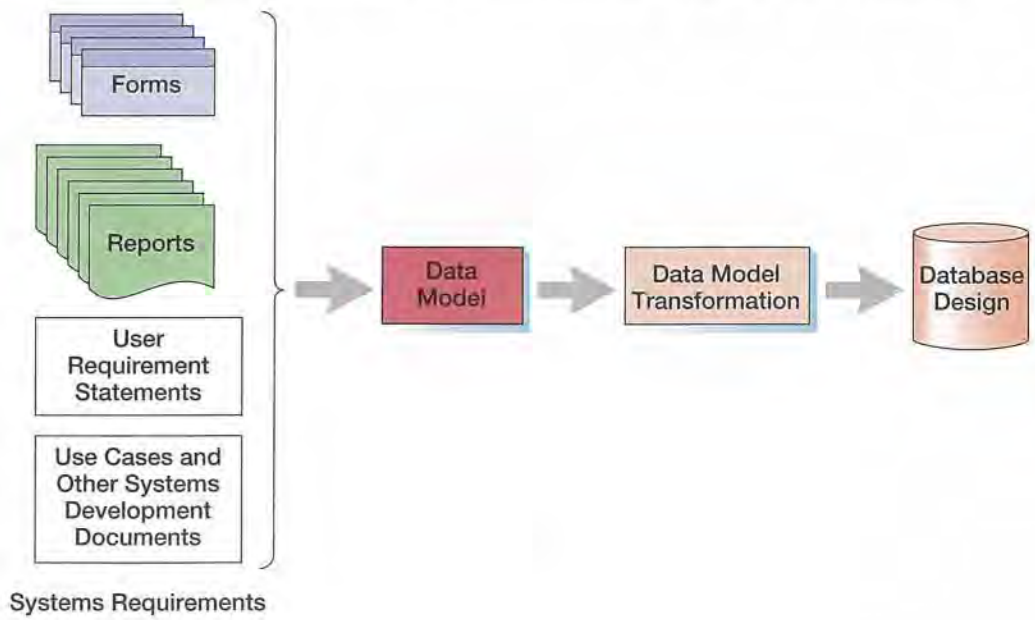


Figure 1-21
Databases Originating from New Systems Development

Amazon Ex. 1008.0046

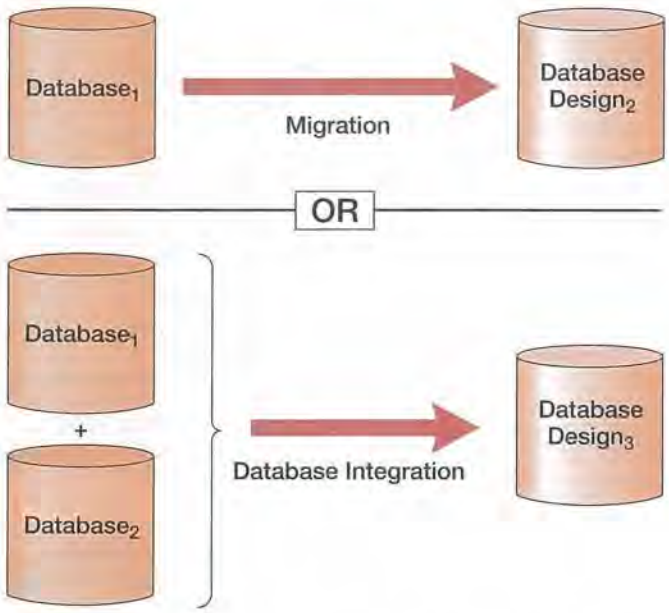


Figure 1-22
Databases Originating from Database Redesign

The second type of database redesign involves the integration of two or more databases. This type of redesign is common when adapting or removing legacy systems. It is also common for enterprise application integration, when two or more previously separate information systems are adapted to work with each other.

Database redesign is complicated. There is no getting around that fact. If this is your first exposure to database design, your instructor may skip this topic. If this is the case, after you have gained more experience, you should reread this material. In spite of its difficulty, database redesign is important.

To understand database redesign, you need to know SQL statements for defining database structures and more advanced SQL statements for querying and updating a database. Consequently, we will not address database redesign until after Chapter 7, which presents SQL statements and techniques for creating and altering the tables that make up a database.

What You Need to Learn

In your career, you may work with database technology as either a user or as a database administrator. As a user you may be a **knowledge worker** who prepares reports, mines data, and does other types of data analysis or you may be a **programmer** who writes applications that process the database. Alternatively, you might be a **database administrator** who designs, constructs, and manages the database itself. Users are primarily concerned with constructing SQL statements to get and put the data they want. Database administrators are primarily concerned with the management of the database. The domains for each of these roles are shown in Figure 1-23.

BY THE WAY The most exciting and interesting jobs in technology are always those on the leading edge. If you live in the United States and are concerned about outsourcing, a recent study by the Rand Corporation[3] indicates that the most secure jobs in the United States involve the adaptation of new technology to solve business problems in innovative ways.

Right now, the leading edge involves the integration of XML, Web services, and database processing. You will need all of the fundamentals presented in this book, especially the material in Chapter 12, to work in this exciting new area.

[3] Karoly, Lynn A., and Constantijn W. A. Panis. *The 21st Century at Work.* (Santa Monica, CA: The Rand Corporation, 2004).

Amazon Ex. 1008.0047

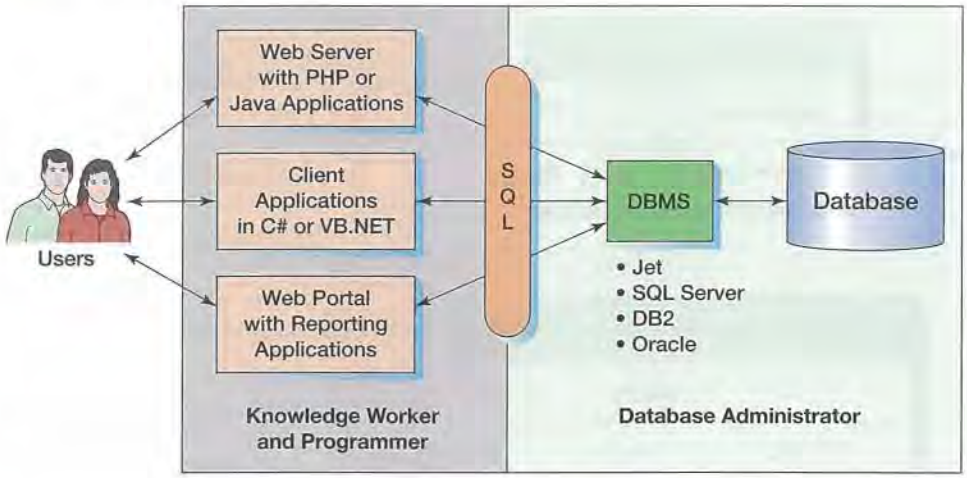


Figure 1-23

Working Domains of Knowledge Workers, Programmers, and Database Administrators

Topic	Chapter	Importance to Knowledge Worker and Programmer	Importance to Database Administrator
Basic SQL	Chapter 2	1	1
Design via normalization	Chapter 3	2	1
Data modeling	Chapter 4	1	1
Data model transformation	Chapter 5	2	1
DDL SQL	Chapter 6	2	1
Constraint enforcement	Chapter 7	3	1
Database redesign	Chapter 8	3	2, but 1 for senior DBA
Database administration	Chapter 9	2	1
SQL Server, Oracle, MySQL specifics	Chapters 10, 10A, 10B	3	1
Database application technology	Chapters 13, 14, 15	1	3

1 = Very important; 2 = Important; 3 = Less important

Warning: Opinions vary, ask your instructor for his or hers.

Figure 1-24

Priorities of What You Need to Know

Both users and database administrators need all of the knowledge in this text. However, the emphasis on each topic differs for the two groups. Figure 1-24 shows our opinion as to the relative importance of each topic to each group. Discuss this table with your instructor. He or she may have knowledge about your local job market that affects the relative importance of these topics.

A Brief History of Database Processing

Database processing emerged around 1970 and has been continuously evolving and changing since then. This continual change has made it a fascinating and thoroughly enjoyable field in which to work. Figure 1-25 summarizes the major eras of database processing.

The Early Years

Prior to 1970, all data were stored in separate files, most of which were kept on reels of magnetic tape. Magnetic disks and drums (magnetic cylinders that are no longer used) were exceedingly expensive and very small. Today's 1.44 megabyte floppy disk (which is now itself a technology declining in use) has more capacity than many disks of that era. Memory was expensive as well. In 1969, we were processing payroll on a computer that had just 32,000 *bytes* of memory, while the computer on which this history is being written has 2 *gigabytes* of memory.

Amazon Ex. 1008.0048

Era	Years	Important Products	Remarks
Predatabase	Before 1970	File managers	All data were stored in separate files. Data integration was very difficult. File storage space was expensive and limited.
Early database	1970–1980	ADABAS, System2000, Total, IDMS, IMS	First products to provide related tables. CODASYL DBTG and hierarchical data models (DL/I) were prevalent.
Emergence of relational model	1978–1985	DB2, Oracle	Early relational DBMS products had substantial inertia to overcome. In time, the advantages weighed out.
Microcomputer DBMS products	1982–1992+	dBase-II, R:base, Paradox, Access	Amazing! A database on a micro. All micro DBMS products were eliminated by Microsoft Access in the early 1990s.
Object-oriented DBMS	1985–2000	Oracle ODBMS and others	Never caught on. Required relational database to be converted. Too much work for perceived benefit.
Web databases	1995–present	IIS, Apache, PHP, ASP.NET, and Java	Stateless characteristic of HTTP was a problem at first. Early applications were simple one-stage transactions. Later, more complex logic developed.
Open source DBMS products	1995–present	MySQL, PostgresQL, and other products	Open source DBMS products provide much of the functionality and features of commercial DBMS products at reduced cost.
XML and Web services	1998–present	XML, SOAP, WSDL, UDDI, and other standards	XML provides tremendous benefits to Web-based database applications. Very important today. May replace relational databases during your career. See Chapter 12.

Figure 1-25
Database History

Integrated processing was an important but very difficult problem. An insurance company, for example, wanted to relate customer account data to customer claim data. Accounts were stored on one magnetic tape, and claims were stored on another. To process claims, the data on the two tapes had to be integrated somehow.

The need for data integration drove the development of the first database technology. By 1973, several commercial DBMS products had emerged. These products were in use by the mid-1970s. The first edition of this text, copyrighted 1977, featured the DBMS products ADABAS, System2000, Total, IDMS, and IMS. Of those five, only ADABAS and IMS are still in use, and neither of them has substantial market share today.

Those early DBMS products varied in the way that they structured data relationships. One method, called **Data Language/I (DL/I)** used hierarchies or trees (see Appendix D) to represent

Amazon Ex. 1008.0049

relationships. IMS, which was developed and licensed by IBM, was based on this model. IMS had success at many organizations, particularly among large manufacturers, and is still in limited use today.

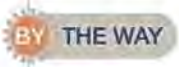
Another technique for structuring data relationships used data structures called *networks*. The CODASYL Committee (the group that developed the programming language COBOL) sponsored a subcommittee called the Database Task Group (DBTG). This subcommittee developed a standard data model that came to bear its name—the **CODASYL DBTG** model. It was an unnecessarily complicated model (everyone's favorite idea made it into the committee's design), but several successful DBMS products were developed using it. The most successful was IDMS, and its vendor, the Cullinane Corporation, was the first software company to be listed on the New York Stock Exchange. To the best of our knowledge, no IDMS database is in use today.

The Emergence and Dominance of the Relational Model

In 1970, a then little-known IBM engineer named E. F. Codd published a paper in the *Communications of the ACM*[4] in which he applied the concepts of a branch of mathematics called relational algebra to the problem of "shared data banks," as databases were then known. The results of this work are now the **relational model** for databases, and all relational database DBMS products are built on this model.

Codd's work was at first viewed as too theoretical for practical implementation. Practitioners argued that it was too slow and required so much storage that it would never be useful in the commercial world. However, the relational model and relational database DBMS products became adopted as the best way to create and manage databases.

The 1977 edition of this text featured a chapter on the relational model (which Codd himself reviewed). Many years later, Wayne Ratliff, the creator of the *dBase* series of products for personal computers, stated that he had the idea for *dBase* while reading that very chapter.[5]



Today, there are as many opportunities for innovation as there were for Wayne Ratliff in 1977. Perhaps you can read Chapter 12 and develop an innovative product that integrates XML and DBMS processing in a new way. Just as in 1977, no product has a lock on the future. Opportunity awaits you!

The relational model, relational algebra, and later, SQL made sense. They were not needlessly complicated; rather, they seemed to boil down the data integration problem to a few essential ideas. Over time, Codd convinced IBM management to develop relational-model DBMS products. The result was IBM's DB2 and its variants, which are still very popular today.

Meanwhile, other companies were considering the relational model as well, and by 1980 several more relational DBMS products had been released. The most prominent and important of those was Oracle Corporation's Oracle Database. Oracle achieved success for many reasons, one of which was that it would run on just about any computer and just about any operating system. (Some users complained, "Yes, and equally badly on all of them." Another, when asked "Should we sell it to communist Russia?" responded, "Only as long as they have to take the documentation with it.")

However, in addition to being able to run on many different types of machines, Oracle Database had, and continues to have, an elegant and efficient internal design. You will learn aspects of that design in the concurrency-control section in Chapter 10A. That excellent design, together with hard-driving and successful sales and marketing, has pushed Oracle to the top of the DBMS market.

Meanwhile, Gordon Moore and others were hard at work at Intel. By the early 1980s, personal computers were prevalent, and DBMS products were developed for them. Developers of microcomputer DBMS products saw the advantages of the relational model and developed their products around it. dBase was the most successful of the early products, but another product,

[4] Codd, E. F. "A Relational Model of Data for Large Shared Databanks," *Communications of the ACM*, June 1970, pp. 377–387.
[5] Ratliff, C. Wayne. "dStory: How I Really Developed dBASE," *Data Based Advisor*, March 1991, p. 94.

Amazon Ex. 1008.0050

R:base, was the first to implement true relational algebra and other operations on the PC. Later, another relational DBMS product named Paradox was developed for personal computers. Eventually, Paradox was acquired by Borland.

Alas, it all came to an end when Microsoft entered the picture. Microsoft released Access in 1991 and priced it at $99. No other PC DBMS vendor could survive at that price point. Access killed R:base and Paradox, and then Microsoft bought a dBase "work-alike" product called FoxPro and used it to eliminate dBase. Microsoft has now stopped upgrading Visual FoxPro, but will continue to support it until 2014. Microsoft Access is the only major survivor of that bloodbath of PC DBMS products.

The one main challenge to Microsoft Access actually comes from the Sun Microsystems and the open source software development community, who have taken over development of OpenOffice.org, a downloadable suite of free software products that includes the personal database OpenOffice.org Base (see *www.openoffice.org*).

Post-Relational Developments

In the mid-1980s, **object-oriented programming (OOP)** emerged, and its advantages over traditional structured programming were quickly recognized. By 1990, some vendors had developed **object-oriented DBMS (OODBMS or ODBMS)** products. These products were designed to make it easy to store the data encapsulated in OOP objects. Several special-purpose OODBMS products were developed, and Oracle added OOP constructs to Oracle to enable the creation of a hybrid called an **object-relational DBMS**.

OODBMS never caught on, and today that category of DBMS products is fading away. There were two reasons for their lack of acceptance. First, using an OODBMS required that the relational data be converted from relational format to object-oriented format. By the time OODBMS emerged, billions upon billions of bytes of data were stored in relational format in organizational databases. No company was willing to undergo the expensive travail of converting those databases to be able to use the new OODBMS.

Second, object-oriented databases had no substantial advantage over relational databases for most commercial database processing. As you will see in the next chapter, SQL is not object oriented. But it works, and thousands of developers have created programs that use it. Without a demonstrable advantage over relational databases, no organization was willing to take on the task of converting their data to OODBMS format.

Meanwhile, the Internet took off. By the mid-1990s, it was clear that the Internet was one of the most important phenomena in history. It changed, forever, the ways that customers and businesses relate to each other. Early Web sites were nothing more than online brochures, but within a few years dynamic Web sites that involved querying and processing databases began to appear.

However, one substantial problem existed. HTTP is a stateless protocol; a server receives a request from a user, processes the request, and then forgets about the user and the request. Many database interactions are multistage. A customer views products, adds one or more to a shopping cart, views more products, adds more to the shopping cart, and eventually checks out. A stateless protocol cannot be used for such applications.

Over time, capabilities emerged to overcome this problem. Web application developers learned to add SQL statements to their Web applications, and soon thousands of databases were being processed over the Web. You will learn more about such processing in Chapter 11. An interesting phenomenon was the emergence of open source DBMS products. Open source products generally make the source code widely available so that a group of programmers not bound to a single company can contribute to the program. Further, some form of these products are usually offered as free downloads, although other forms or product support must be purchased from the company that owns the product.

A good example of this is the MySQL DBMS (www.mysql.com). MySQL was originally released in 1995 by the Swedish company MySQL AB. In February 2008, Sun Microsystems bought MySQL AB,[6] but MySQL continues to be available as an open source product, and the free MySQL Community Server edition can be downloaded from the MySQL Web site. MySQL has proven to be especially popular with Web site developers who need to run Web page queries

[6] As this book goes to press, Oracle Corporation has concluded an agreement with Sun Microsystems to purchase Sun Microsystems. This will make Oracle the owner of both Oracle Database and MySQL. See http://www.sun.com/third-party/global/oracle.

Amazon Ex. 1008.0051

against an SQL DBMS on a Web server running the Linux operating system. We will work with MySQL in Chapter 10B.

MySQL is not the only open source DBMS product—in fact, as this is being written there are 47 listed on the Wikipedia category page *http://en.wikipedia.org/wiki/Category:Open_source_database_management_systems*.

One interesting outcome of the emergence of open source DBMS products is that companies that typically sell proprietary (closed source) DBMS products now offer free versions of their products. For example, Microsoft now offers SQL Server 2008 Express (*www.microsoft.com/express/sql/download*), and Oracle Corporation makes its Oracle Database 10g Express Edition available for free (*www.oracle.com/technology/products/database/xe/index.html*). Although neither of these products is as complete or as powerful (for example, in terms of maximum data storage allowed) as some other versions the companies sell, they are useful for projects that require a small database. They are also ideal for students learning to use databases and SQL.

The last row in Figure 1-25 brings us to the present. In the late 1990s, XML was defined to overcome the problems that occur when HTML is used to exchange business documents. The design of the XML family of standards not only solved the problems of HTML, it also meant that XML documents were superior for exchanging views of database data. In 2002, Bill Gates said that "XML is the lingua-franca of the Internet Age." As you will learn in Chapter 12, however, two key problems that remain are (1) getting data from a database and putting it into an XML document and (2) taking data from an XML document and putting it into a database. In fact, this is where future application programmers can enter the picture.

XML database processing was given a further boost with the definition of XML Web service standards such as SOAP (not an acronym), WSDL (Web Services Description Language), UDDI (Universal Description, Discovery, and Integration), and others. Using Web services, it is possible to expose nuggets of database processing to other programs that use the Internet infrastructure. This means, for example, that in a supply chain management application, a vendor can expose portions of its inventory application to its suppliers. Further, it can do so in a standardized way.

XML Web services bring us to the edge of the IT volcano, where the magma of new technology is just now oozing out of the ground. What happens next will be, in part, up to you.

Summary

The purpose of a database is to help people keep track of things. Databases store data in tables in which each table has data about a different type of thing. Instances of the thing are stored in the rows of tables, and the characteristics of those instances are stored in columns. In this text, table names are written in all capital letters; column names are written in initial capital letters. Databases store data and the relationships among the data. Databases store data, but they are structured so that information can be created from that data.

Figure 1-5 lists many important examples of database applications. Databases can be processed by a single user or by many users. Those that support many users require special concurrency-control mechanisms to ensure that one user's work does not conflict with a second user's work.

Some databases involve just a few users and thousands of rows of data in a few tables. At the other end of the spectrum, some large databases, such as those that support ERP applications, support thousands of users and include many millions of rows in several hundred different tables.

Some database applications support e-commerce activities. Some of the largest databases are those that track users' responses to Web pages and Web page components. These databases are used to analyze customers' responses to different Web-based marketing programs.

Digital dashboards, data mining applications, and other reporting applications use database data that is generated by transaction processing systems to help manage the enterprise. Digital dashboards and reporting systems assess past and current performance. Data mining applications predict future performance. The basic components of a database system are the database, the database management system (DBMS), one or more database applications, and users. Because Structured Query Language (SQL) is an internationally recognized language for processing databases, it can be considered a fifth component of a database system.

The functions of database applications are to create and process forms, to process user queries, and to create and process reports. Application programs also execute specific application logic and control the application. Users provide data and data changes and read data in forms, queries, and reports.

A DBMS is a large, complicated program used to create, process, and administer a database. DBMS products are

Amazon Ex. 1008.0052

almost always licensed from software vendors. Specific functions of a DBMS are summarized in Figure 1-12.

A database is a self-describing collection of integrated tables. A relational database is a self-describing collection of related tables. Tables are integrated because they store data about the relationships between rows of data. Tables are related by storing linking values of a common column. A database is self-describing because it contains a description of its contents within itself, which is known as metadata. Most DBMS products carry metadata in the form of tables. As shown in Figure 1-14, databases also contain indexes, triggers, stored procedures, security features, and backup and recovery data.

Microsoft Access is not a DBMS but rather an application generator plus a DBMS. The application generator consists of applications components that create and process forms, reports, and queries. The default Access DBMS product is called Jet, which is not licensed as a separate product. SQL Server can be substituted for Jet to support larger databases.

Enterprise database systems do not combine applications and the DBMS as Access does. Instead, applications are programs separate from each other and from the DBMS. Figure 1-16 shows four categories of database applications: client/server applications, Web applications, reporting applications, and XML Web services applications.

The five most popular DBMS products, in order of power, features, and difficulty of use, are Access (really Jet), MySQL, SQL Server, DB2, and Oracle. Jet and SQL Server are licensed by Microsoft, MySQL is licensed by Sun Microsystems, DB2 is licensed by IBM, and Oracle Database is licensed by the Oracle Corporation.

Database design is both difficult and important. Most of the first half of this text concerns database design. New databases arise in three ways: from existing data, from new systems development, and from database redesign. Normalization is used to guide the design of databases from existing data. Data models are used to create a blueprint from system requirements. The blueprint is later transformed into a database design. Most data models are created using the entity-relationship model. Database redesign occurs when an existing database is adapted to support new or changed requirements or when two or more databases are integrated.

With regards to database processing, you can have one of two roles: user or database administrator. You may be a *user* of a database/DBMS as a knowledge worker or as an application programmer. Alternatively, you might be a *database administrator* who designs, constructs, and manages the database itself. The domains of each role are shown in Figure 1-23, and the priorities as to what you need to know for each role are shown in Figure 1-24.

The history of database processing is summarized in Figure 1-25. In the early years, prior to 1970, database processing did not exist, and all data was stored in separated files. The need for integrated processing drove the development of early DBMS products. The CODASYL DBTG and DL/I data models were prevalent. Of the DBMS products used at that time, only ADABAS and IMS are still in use.

The relational model rose to prominence in the 1980s. At first, the relational model was judged to be impractical, but over time relational products such as DB2 and Oracle achieved success. During this time, DBMS products were developed for personal computers as well. dBase, R:base, and Paradox were all PC DBMS products that were eventually consumed by the success of Microsoft Access.

Object-oriented DBMS products were developed in the 1990s but never achieved commercial success. More recently, Web-based databases have been developed to support e-commerce. Open source DBMS products are readily available, forcing commercial DBMS vendors to offer limited-capacity free versions of their enterprise products. Features and functions have been implemented to overcome the stateless nature of HTTP. XML and XML Web services databases are at the leading edge of database processing.

Key Terms

CODASYL DBTG
column
concurrency
data
Data Language/I (DL/I)
data marts
data model
data warehouse
database
database administrator
database application
database design
database management system (DBMS)
database migration
enterprise-class database system
entity-relationship (ER) data modeling
foreign key
information
instance
integrated tables
knowledge worker
metadata
normal forms
normalization
object-oriented DBMS (OODBMS or ODBMS)
object-oriented programming (OOP)
object-relational DBMS
personal database system

Amazon Ex. 1008.0053

primary key
programmer
record
referential integrity constraints
relational database
relational model
relationship
row
self-describing
Structured Query Language (SQL)
table
User

Review Questions

1.1 What is the purpose of a database?

1.2 What is the most commonly used type of database?

1.3 Give an example of two related tables other than one in this book. Use the STUDENT and GRADE tables in Figure 1-3 as an example pattern for your tables. Name the tables and columns using the conventions in this book.

1.4 For the tables you created in Review Question 1.3, what are the primary keys of each table?

1.5 Explain how the two tables you provided in Review Question 1.3 are related. Which table contains the foreign key, and what is the foreign key?

1.6 Show your two tables from Review Question 1.3 without the columns that represent the relationships. Explain how the value of your two tables is diminished without the relationships.

1.7 Define the terms *data* and *information*. Explain how the two terms differ.

1.8 Give an example of information that could be determined using the two tables you provided in your answer to Review Question 1.3.

1.9 Give examples of a single-user database application and a multiuser database application other than the ones shown in Figure 1-5.

1.10 What problem can occur when a database is processed by more than one user?

1.11 Give an example of a database application that has hundreds of users and a very large and complicated database. Use an example other than one in Figure 1-5.

1.12 What is the purpose of the largest databases at e-commerce companies such as Amazon.com?

1.13 How do the e-commerce companies use these databases?

1.14 How do digital dashboard and data mining applications differ from transaction processing applications?

1.15 Explain why a small database is not necessarily simpler than a large one.

1.16 Explain the components in Figure 1-7.

1.17 What are the functions of application programs?

1.18 What is Structured Query Language (SQL), and why is it important?

1.19 What does DBMS stand for?

1.20 What are the functions of the DBMS?

1.21 Name four vendors of DBMS products.

1.22 Define the term *database*.

1.23 Why is a database considered to be self-describing?

Amazon Ex. 1008.0054

1.24 What is metadata? How does this term pertain to a database?

1.25 What advantage is there in storing metadata in tables?

1.26 List the components of a database other than user tables and metadata.

1.27 Is Microsoft Access a DBMS? Why or why not?

1.28 Describe the components shown in Figure 1-15.

1.29 What is the function of the application generator in Access?

1.30 What is the name of the DBMS engine within Access? Why do we rarely hear about that engine?

1.31 Why does Microsoft Access hide important database technology?

1.32 Why would someone choose to replace the native Access DBMS engine with SQL Server?

1.33 Name the components of an enterprise-class database system.

1.34 Name and describe the four categories of database applications that would use an enterprise-class database system.

1.35 How do database applications get and put database data?

1.36 Name the five DBMS products described in this chapter, and compare them in terms of power, features, and ease of use.

1.37 List several consequences of a poorly designed database.

1.38 Explain two ways that a database can be designed from existing data.

1.39 What is a data warehouse? What is a data mart?

1.40 Describe the general process of designing a database for a new information system.

1.41 Explain two ways that databases can be redesigned.

1.42 What does the term *database migration* mean?

1.43 Summarize the various ways that you might work with database technology.

1.44 What job functions does a knowledge worker perform?

1.45 What job functions does a database administrator perform?

1.46 Explain the meaning of the domains in Figure 1-23.

1.47 What need drove the development of the first database technology?

1.48 What are Data Language/I and CODASYL DBTG?

1.49 Who was E. F. Codd?

1.50 What were the early objections to the relational model?

1.51 Name two early relational DBMS products.

1.52 What are some of the reasons for Oracle's success?

1.53 Name three early personal computer DBMS products.

1.54 What happened to the products in your answer to Review Question 1.53?

1.55 What was the purpose of OODBMS products? State two reasons that OODBMS products were not successful.

1.56 What characteristic of HTTP was a problem for database processing applications?

1.57 What is an open source DBMS product?

Amazon Ex. 1008.0055

1.58 Which of the five DBMS products that you named in answering Review Question 1.36 is historically an open source DBMS product?

1.59 What has been the response of companies that sell proprietary DBMS products to the open source DBMS products? Include two examples in your answer.

1.60 What comment did Bill Gates make regarding XML?

Project Questions

To perform the following projects you will need a computer that has Microsoft Access installed. If you have no experience working with Access, read Appendix A before you proceed.

For this set of project questions, we will create a Microsoft Access database for the Wedgewood Pacific Corporation (WPC). Founded in 1957 in Seattle, Washington, WPC has grown into an internationally recognized organization. The company is located in two buildings. One building houses the Administration, Accounting, Finance, and Human Resources departments, and the second houses the Production, Marketing, and Information Systems departments. The company database contains data about company employees, departments, company projects, company assets (for example, computer equipment), and other aspects of company operations.

In the following project questions, we will start by creating the WPC.accdb database with the following two tables:

DEPARTMENT (DepartmentName, BudgetCode, OfficeNumber, Phone)

EMPLOYEE (EmployeeNumber, FirstName, LastName, *Department*, Phone, Email)

1.61 Create a Microsoft Access database namedWPC.accdb.

1.62 Figure 1-26 shows the column characteristics for the WPC DEPARTMENT table. Using the column characteristics, create the DEPARTMENT table in the WPC.accdb database.

1.63 Figure 1-27 shows the data for the WPC DEPARTMENT table. Using Datasheet view, enter the data shown in Figure 1-27 into your DEPARTMENT table.

1.64 Figure 1-28 shows the column characteristics for the WPC EMPLOYEE table. Using the column characteristics, create the EMPLOYEE table in the WPC.accdb database.

1.65 Create the relationship and referential integrity constraint between DEPARTMENT and EMPLOYEE. Enable enforcing of referential integrity and cascading of data updates, but do *not* enable cascading of data from deleted records.

Figure 1-26
Database Column Characteristics for the DEPARTMENT Table

DEPARTMENT

Column Name	Type	Key	Required	Remarks
DepartmentName	Text (35)	Primary Key	Yes	
BudgetCode	Text (30)	No	Yes	
OfficeNumber	Text (15)	No	Yes	
Phone	Text (12)	No	Yes	

Amazon Ex. 1008.0056

DepartmentName	BudgetCode	OfficeNumber	Phone
Administration	BC-100-10	BLDG01-300	360-285-8100
Legal	BC-200-10	BLDG01-200	360-285-8200
Accounting	BC-300-10	BLDG01-100	360-285-8300
Finance	BC-400-10	BLDG01-140	360-285-8400
Human Resources	BC-500-10	BLDG01-180	360-285-8500
Production	BC-600-10	BLDG02-100	360-287-8600
Marketing	BC-700-10	BLDG02-200	360-287-8700
InfoSystems	BC-800-10	BLDG02-270	360-287-8800

Figure 1-27
WPC DEPARTMENT Data

1.66 Figure 1-29 shows the data for the WPC EMPLOYEE table. Using Datasheet view, enter the first three rows of the data shown in Figure 1-29 into your EMPLOYEE table.

1.67 Using the Microsoft Access form wizard, create a data input form for the EMPLOYEE table and name it WPC Employee Data Form. Make any adjustments necessary to the form so that all data display properly. Use this form to enter the rest of the data in the EMPLOYEE table shown in Figure 1-29 into your EMPLOYEE table.

1.68 Using the Access report wizard, create a report named Wedgewood Pacific Corporation Employee Report that presents the data contained in your EMPLOYEE table sorted first by employee last name and then by employee first name. Make any adjustments necessary to the report so that all headings and data display properly. Print a copy of this report.

1.69 Using the Microsoft Access form wizard, create a form that has all of the data from both tables. When asked how you want to view your data, select *by DEPARTMENT*. Choose the default options for other questions that the wizard asks. Open your form and page through your departments.

1.70 Using the Access report wizard, create a report that has all of the data from both tables. When asked how you want to view your data, select *by DEPARTMENT*. For the data contained in your EMPLOYEE table in the report, specify that it will be sorted first by

Figure 1-28
Database Column Characteristics for the EMPLOYEE Table

EMPLOYEE

Column Name	Type	Key	Required	Remarks
EmployeeNumber	AutoNumber	Primary Key	Yes	Surrogate Key
FirstName	Text (25)	No	Yes	
LastName	Text (25)	No	Yes	
Department	Text (35)	No	Yes	
Phone	Text (12)	No	No	
Email	Text (100)	No	Yes	

Amazon Ex. 1008.0057

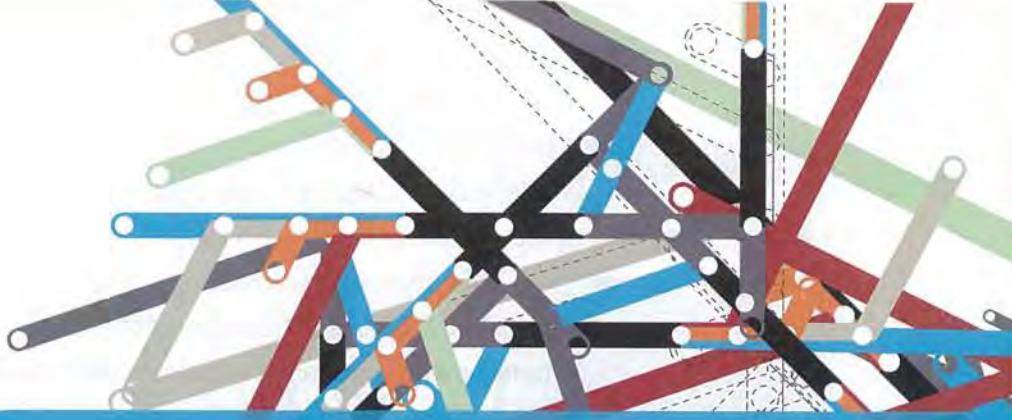
EmployeeNumber	FirstName	LastName	Department	Phone	Email
[AutoNumber]	Mary	Jacobs	Administration	360-285-8110	MJacobs@WPC.com
[AutoNumber]	Rosalie	Jackson	Administration	360-285-8120	RJackson@WPC.com
[AutoNumber]	Richard	Bandalone	Legal	360-285-8210	RBandalone@WPC.com
[AutoNumber]	Tom	Caruthers	Accounting	360-285-8310	TCaruthers@WPC.com
[AutoNumber]	Heather	Jones	Accounting	360-285-8320	HJones@WPC.com
[AutoNumber]	Mary	Abernathy	Finance	360-285-8410	MAbernathy@WPC.com
[AutoNumber]	George	Smith	Human Resources	360-285-8510	GSmith@WPC.com
[AutoNumber]	Tom	Jackson	Production	360-287-8610	TJackson@WPC.com
[AutoNumber]	George	Jones	Production	360-287-8620	GJones@WPC.com
[AutoNumber]	Ken	Numoto	Marketing	360-287-8710	KNumoto@WPC.com
[AutoNumber]	James	Nestor	InfoSystems		JNestor@WPC.com
[AutoNumber]	Rick	Brown	InfoSystems	360-287-8820	RBrown@WPC.com

Figure 1-29
WPC EMPLOYEE Data

employee last name and then by employee first name. Make any adjustments necessary to the report so that all headings and data display properly. Print a copy of this report.

1.71 Explain, to the level of detail in this chapter, what is going on within Microsoft Access in Project Questions 1.67, 1.68, 1.69, and 1.70. What subcomponent created the form and report? Where is the data stored? What role do you think SQL is playing?

Amazon Ex. 1008.0058



2 Introduction to Structured Query Language

Chapter Objectives

- To understand the use of extracted data sets.
- To understand the use of ad-hoc queries.
- To understand the history and significance of Structured Query Language (SQL).
- To understand the SQL SELECT/FROM/WHERE framework as the basis for database queries.
- To be able to write queries in SQL to retrieve data from a single table.
- To be able to write queries in SQL to use the SQL SELECT, FROM, WHERE, ORDER BY, GROUP BY, and HAVING clauses.
- To be able to write queries in SQL to use SQL DISTINCT, AND, OR, NOT, BETWEEN, LIKE, and IN keywords.
- To be able to use the SQL built-in functions of SUM, COUNT, MIN, MAX, and AVG with and without the use of a GROUP BY clause.
- To be able to write queries in SQL to retrieve data from a single table but restricting the data based upon data in another table (subquery).
- To be able to write queries in SQL to retrieve data from multiple tables using an SQL join.

In today's business environment, users typically use data stored in databases to produce information that can help them make business decisions. In Chapter 13, we will take an in-depth look at **business intelligence (BI) systems**, which are used to support management by producing information for assessment, analysis, planning, and control. In this chapter, we will see how users use **ad-hoc queries**, which are essentially questions that can be answered using database data. For example, in English an ad-hoc query would be "How many

Amazon Ex. 1008.0059

customers in Portland, Oregon, bought our green baseball cap?" These queries are called *ad-hoc* because they are created by the user as needed, rather than programmed into an application.

This approach to database querying has become important enough that some companies produce dedicated applications to help users who are not familiar with database structures create ad-hoc queries. One example is Open Text's LiveLink ECM BI Query product (*www.hummingbird.com/products/bi/bi_query.html*), which uses a user-friendly **graphical user interface (GUI)** to simplify the creation of ad-hoc queries. Personal databases such as Microsoft Access also have ad-hoc query tools available. Microsoft Access uses a GUI style called **query by example (QBE)** to simplify ad-hoc queries.

However, **Structured Query Language (SQL)**—the universal query language of relational DBMS products—is always behind the user-friendly GUIs. In this chapter, we will introduce SQL by learning how to write and run SQL queries. We will then return to SQL in Chapter 7 to learn how to use it for other purposes, such as how to create and add data to the databases themselves.

Cape Codd Outdoor Sports

For our work in this chapter, we will use data from Cape Codd Outdoor Sports (although based on a real outdoor retail equipment vendor, Cape Codd Outdoor Sports is a fictitious company). Cape Codd sells recreational outdoor equipment in 15 retail stores across the United States and Canada. It also sells merchandise over the Internet from a Web storefront application and via mail order. All retail sales are recorded in a sales database managed by the Oracle DBMS, as shown in Figure 2-1.

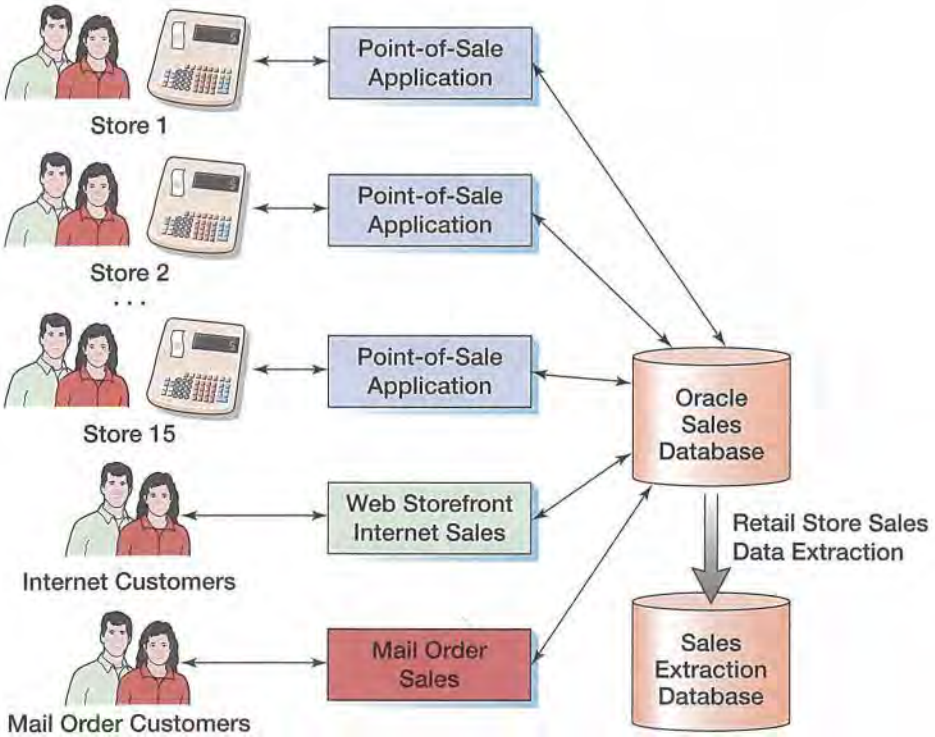


Figure 2-1
Cape Codd Retail Sales Data Extraction

Amazon Ex. 1008.0060

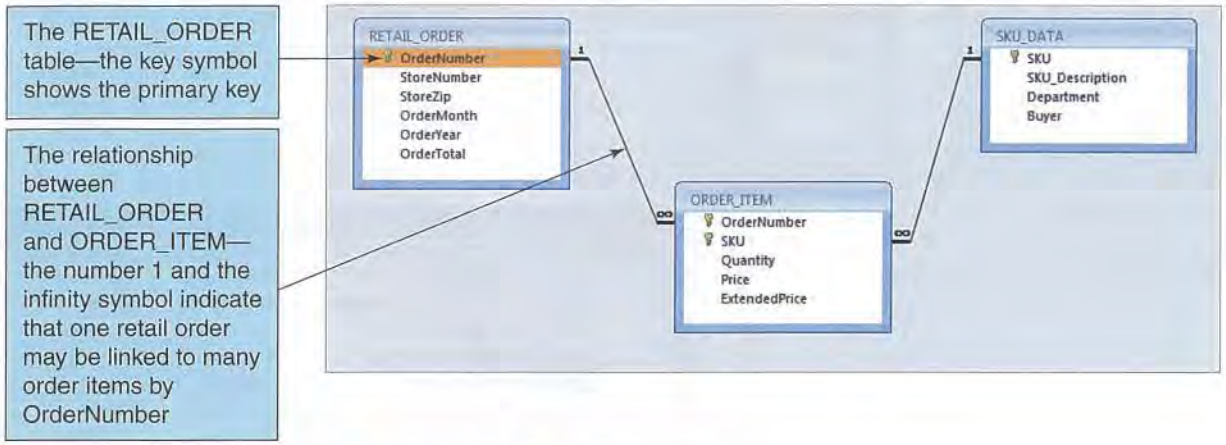


Figure 2-2

Cape Codd Extracted Retail Sales Data Database Tables and Relationships

The Retail Sales Data Extraction

Cape Codd's marketing department wants to perform an analysis of in-store sales. Accordingly, marketing analysts ask the IT department to extract retail sales data from the operational database. To perform the marketing study, they do not need all of the order data. They want just the tables and columns shown in Figure 2-2. The data types for the columns in the tables is shown in Figure 2-3.

As shown in Figures 2-2 and 2-3, three tables are needed: RETAIL_ORDER, ORDER_ITEM, and SKU_DATA. The RETAIL_ORDER table has data about each retail sales order, the ORDER_ITEM table has data about each item in an order, and the SKU_DATA table has data about each **stock-keeping unit (SKU)**. SKU is a unique identifier for each particular item that Cape Codd sells. The data stored in the tables is shown in Figure 2-4.

RETAIL_ORDER Data

As shown in Figures 2-2 and 2-3, the RETAIL_ORDER table has columns for OrderNumber, StoreNumber, StoreZip (the zip code of the store selling the order), OrderMonth, OrderYear, and OrderTotal:

RETAIL_ORDER (OrderNumber, StoreNumber, StoreZip, OrderMonth, OrderYear, OrderTotal)

Figure 2-3

Cape Codd Extracted Retail Sales Data Format

Table	Column	Date Type
RETAIL_ORDER	OrderNumber	Integer
	StoreNumber	Integer
	StoreZip	Character (9)
	OrderMonth	Character (12)
	OrderYear	Integer
	OrderTotal	Currency
ORDER_ITEM	OrderNumber	Integer
	SKU	Integer
	Quantity	Integer
	Price	Currency
	ExtendedPrice	Currency
SKU_DATA	SKU	Integer
	SKU_Description	Character (35)
	Department	Character (30)
	Buyer	Character (30)

Amazon Ex. 1008.0061

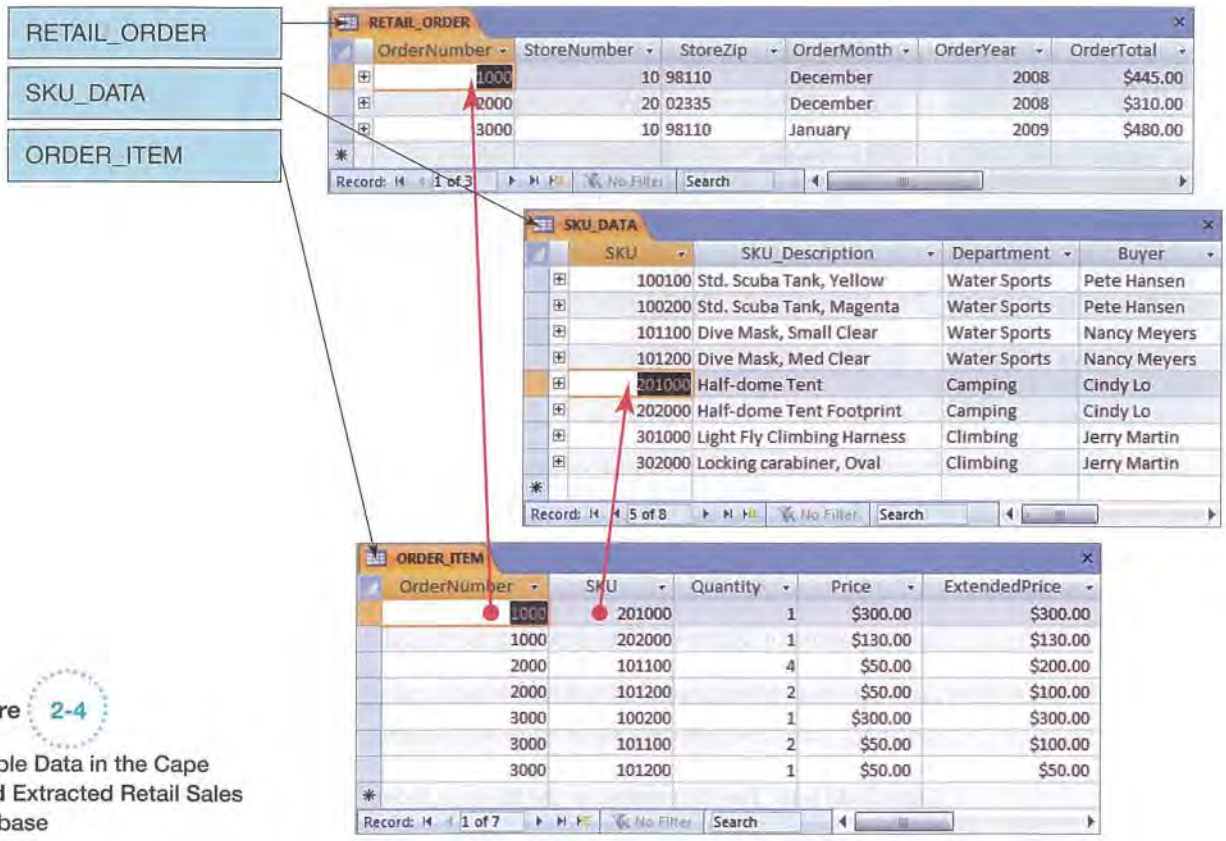


Figure 2-4

Sample Data in the Cape Codd Extracted Retail Sales Database

Sample data for RETAIL_ORDER is shown in Figure 2-4. This extract only includes data for retail store sales, and operational data for other types of sales (and returns and other sales-related transactions) is not copied during the extraction process. Further, the data extraction process selects only a few columns of the operational data—the Point of Sale (POS) and other sales applications process far more data than that shown here. The operational database also stores the data in a different format. For example, the order data in the Oracle operational database contains a column named OrderDate that stores the data in the date format MM/DD/YYYY (e.g., 10/22/2008 for October 22, 2008). The extraction program used to populate the retail sales extracted data database converts OrderDate into two separate values of OrderMonth and OrderYear. This is done because this is the data format that marketing wants. Such filtering and data transformation are typical of a data extraction process.

ORDER_ITEM Data

As shown in Figures 2-2 and 2-3, the ORDER_ITEM table has columns for OrderNumber, SKU, Quantity, Price, and ExtendedPrice (which equals Quantity × Price):

ORDER_ITEM (OrderNumber, *SKU*, Quantity, Price, ExtendedPrice)

Thus, the ORDER_ITEM table stores an extract of the items purchased in each order. There is one row in the table for each item in an order, and this item is identified by its SKU. To understand this table, think about a sales receipt you get from a retail store. That receipt has data for one order. It includes basic order data such as the date and order total, and it has one line for each item you purchase. The rows in the ORDER_ITEM table correspond to the lines on such an order receipt.

The OrderNumber Column in ORDER_ITEM relates each row in ORDER_ITEM to the corresponding OrderNumber in the RETAIL_ORDER table. SKU indentifies the actual item purchased by its stock-keeping unit number. Further, the SKU column in ORDER_ITEM relates each row in ORDER_ITEM to its corresponding SKU in the SKU_DATA table (discussed in the

Amazon Ex. 1008.0062

next section). Quantity is the number of items of that SKU purchased in that order. Price is the price of each item, and ExtendedPrice is equal to Quantity × Price.

ORDER_ITEM data are shown in the bottom part of Figure 2-4. The first row relates to order 1000 and to SKU 201000. For SKU 201000, one item was purchased for $300.00, and the ExtendedPrice was $300.00. The second row shows the second item in order 1000. There, 1 of item 202000 was purchased for $50.00, and the ExtendedPrice is 1 × $50.00, or $50.00. This table structure of an ORDER table related to an ORDER_ITEM table is typical for sales system with many items in one order. We will discuss it in detail in Chapters 5 and 6, where we will create a data model of a complete order and then design the database for that data model.

BY THE WAY You would expect the total of ExtendedPrice for all rows for a given order to equal OrderTotal in the RETAIL_ORDER table. They do not. For order 1000, for example, the sum of ExtendedPrice in the relevant rows of ORDER_ITEM is $300.00 + $130.00 = $430.00. However, the OrderTotal for order 1000 is $445.00. The difference occurs because OrderTotal includes tax, shipping, and other charges that do not appear in the data extract.

SKU_DATA Table

As shown in Figures 2-2 and 2-3, the SKU_DATA table has columns SKU, SKU_Description, Department, and Buyer:

SKU_DATA (SKU, SKU_Description, Department, Buyer)

SKU is an integer value that identifies a particular product that Cape Codd sells. For example, SKU 100100 indentifies a yellow standard-size SCUBA tank, whereas SKU 100200 identifies the magenta version of the same tank. SKU_Description contains a brief text description of each item. Department and Buyer identify the department and individual who is responsible for purchasing the product. As with the other tables, these columns are a subset of the SKU data stored in the operational database.

Data Extracts Are Common

Before we continue, realize that the data extraction process described here is not just an academic exercise. To the contrary, such extraction processes are realistic, common, and important BI system operations. Right now, hundreds of businesses worldwide are using their BI systems to create extract databases just like the one created by Cape Codd.

In the next sections of this chapter, you will learn how to write SQL statements to process the extracted data. This knowledge is exceedingly valuable and practical. Again, right now, as you read this paragraph, hundreds of people are writing SQL to create information from extracted data. The SQL you will learn in this chapter will be an essential asset to you as a knowledge worker, application programmer, or database administrator. Invest the time to learn SQL—the investment will pay great dividends later in your career.

SQL Background

SQL was developed by the IBM Corporation in the late 1970s. It was endorsed as a national standard by the **American National Standards Institute (ANSI)** in 1986 and by the **International Organization for Standardization (ISO)** (and no, that's not a typo—the acronym is *ISO*, not *IOS*!) in 1987. Subsequent versions of SQL were adopted in 1989 and 1992. The 1992 version is sometimes referred to as SQL-92, or sometimes as ANSI-92 SQL. In 1999, SQL:1999 (also referred to as SQL3), which incorporated some object-oriented concepts, was released. This was followed by the release of SQL:2003 in 2003; SQL:2006 in 2006; and, most recently, SQL:2008 in 2008. Each of these added new features or extended

Amazon Ex. 1008.0063

existing SQL features, the most important of which for us is SQL support for **Extensible Markup Language (XML)**. (XML is discussed in Chapter 12.) Our discussion in this chapter and in Chapter 7 focuses on common language features that have been in SQL since SQL-92, and some discussion of the SQL XML features is included in Chapter 12.

SQL is not a complete programming language, like Java or C#. Instead, it is called a **data sublanguage**, because it has only those statements needed for creating and processing database data and metadata. You can use SQL statements in many different ways. You can submit them directly to the DBMS for processing. You can embed SQL statements into client/server application programs. You can embed them into Web pages, and you can use them in reporting and data extraction programs. You also can execute SQL statements directly from Visual Studio. NET and other development tools.

SQL statements are commonly divided into categories, two of which are of interest to us here:

- **Data definition language (DDL)** statements, which are used for creating tables, relationships, and other structures
- **Data manipulation language (DML)** statements, which are used for querying, inserting, modifying, and deleting data

This chapter considers only DML statements for querying data. The remaining DML statements for inserting, modifying, and deleting data are discussed in Chapter 7, where we will also discuss SQL DDL statements.

SQL is ubiquitous, and SQL programming is a critical skill. Today, all DBMS products process SQL. Enterprise-class DBMSs such as Oracle Database, DB2, SQL Server, and MySQL require that you know SQL. With these products, all data manipulation is expressed using SQL.

As explained in Chapter 1, if you have used Microsoft Access, you have used SQL, even if you didn't know it. Every time you process a form, create a report, or run a query, Microsoft Access generates SQL and sends that SQL to Access' internal Jet DBMS engine. To do more than elementary database processing, you need to uncover the SQL hidden by Microsoft Access. Further, once you know SQL, you will find it easier to write a query statement in SQL rather than fight with the graphical forms, buttons, and other paraphernalia that you must use to create queries with the Microsoft Access query-by-example style GUI.

The SQL SELECT/FROM/WHERE Framework

This section introduces the fundamental statement framework for SQL query statements. After we discuss this basic structure, you will learn how to submit SQL statements to Access, SQL Server, Oracle, and MySQL. If you choose, you can then follow along with the text and process additional SQL statements as they are explained in the rest of this chapter. The basic form of SQL queries uses the **SQL SELECT/FROM/WHERE framework**. In this framework:

- The **SQL SELECT clause** specifies which *columns* are to be listed in the query results.
- The **SQL FROM clause** specifies which *tables* are to be used in the query.
- The **SQL WHERE clause** specifies which *rows* are to be listed in the query results.

Let's work through some examples so that this framework makes sense to you.

Reading Specified Columns from a Single Table

We begin very simply. Suppose we want to obtain just the values of the Department and Buyer columns of the SKU_DATA table. A SQL statement to read that data is the following:

```
SELECT    Department, Buyer
FROM      SKU_DATA;
```

Amazon Ex. 1008.0064

Using the data in Figure 2-4, when the DBMS processes this statement the result will be:

	Department	Buyer
1	Water Sports	Pete Hansen
2	Water Sports	Pete Hansen
3	Water Sports	Nancy Meyers
4	Water Sports	Nancy Meyers
5	Camping	Cindy Lo
6	Camping	Cindy Lo
7	Climbing	Jerry Martin
8	Climbing	Jerry Martin

When SQL statements are executed, the statements transform tables. SQL statements start with a table, process that table in some way, and then place the results in another table structure. Even if the result of the processing is just a single number, that number is considered to be a table with one row and one column. As you will learn at the end of this chapter, some SQL statements process multiple tables. Regardless of the number of input tables, though, the result of every SQL statement is a single table.

Notice that SQL statements terminate with a semicolon (;) character. The semicolon is required by the SQL-92 standard. Although some DBMS products will allow you to omit the semicolon, some will not, so develop the habit of terminating SQL statements with a semicolon.

The order of the column names in the SELECT phrase determines the order of the columns in the results table. Thus, if we switch Buyer and Department in the SELECT phrase, they will be switched in the output table as well. Hence, the SQL statement:

```
SELECT      Buyer, Department
FROM        SKU_DATA;
```

produces the following result table:

	Buyer	Department
1	Pete Hansen	Water Sports
2	Pete Hansen	Water Sports
3	Nancy Meyers	Water Sports
4	Nancy Meyers	Water Sports
5	Cindy Lo	Camping
6	Cindy Lo	Camping
7	Jerry Martin	Climbing
8	Jerry Martin	Climbing

Notice that some rows are duplicated in these results. The data in the first and second row, for example, is identical. We can eliminate duplicates by using the **SQL DISTINCT keyword** as follows:

```
SELECT      DISTINCT Buyer, Department
FROM        SKU_DATA;
```

The result of this statement, where all of the duplicate rows have been removed, is:

	Buyer	Department
1	Cindy Lo	Camping
2	Jerry Martin	Climbing
3	Nancy Meyers	Water Sports
4	Pete Hansen	Water Sports

Amazon Ex. 1008.0065



The reason that SQL does not automatically eliminate duplicate rows is that it can be very time consuming to do so. To determine if any rows are duplicates, every row must be compared with every other row. If there are 100,000 rows in a table, that checking will take a long time. Hence, by default, duplicates are not removed. However, it is always possible to force their removal using the DISTINCT keyword.

Suppose that we want to view all of the columns of the SKU_DATA table. To do so, we can name each column in the SELECT statement as follows:

```
SELECT      SKU, SKU_Description, Department, Buyer
FROM        SKU_DATA;
```

The result will be a table with all of the rows and all four of the columns in SKU_DATA:

	SKU	SKU_Description	Department	Buyer
1	100100	Std. Scuba Tank, Yellow	Water Sports	Pete Hansen
2	100200	Std. Scuba Tank, Magenta	Water Sports	Pete Hansen
3	101100	Dive Mask, Small Clear	Water Sports	Nancy Meyers
4	101200	Dive Mask, Med Clear	Water Sports	Nancy Meyers
5	201000	Half-dome Tent	Camping	Cindy Lo
6	202000	Half-dome Tent Footprint	Camping	Cindy Lo
7	301000	Light Fly Climbing Harness	Climbing	Jerry Martin
8	302000	Locking carabiner, Oval	Climbing	Jerry Martin

However, SQL provides a shorthand notation for querying all of the columns of a table. The shorthand is to use an asterisk (*) to indicate that we want all the columns to be displayed:

```
SELECT      *
FROM        SKU_DATA;
```

The result will again be a table with all rows and all four of the columns in SKU_DATA:

	SKU	SKU_Description	Department	Buyer
1	100100	Std. Scuba Tank, Yellow	Water Sports	Pete Hansen
2	100200	Std. Scuba Tank, Magenta	Water Sports	Pete Hansen
3	101100	Dive Mask, Small Clear	Water Sports	Nancy Meyers
4	101200	Dive Mask, Med Clear	Water Sports	Nancy Meyers
5	201000	Half-dome Tent	Camping	Cindy Lo
6	202000	Half-dome Tent Footprint	Camping	Cindy Lo
7	301000	Light Fly Climbing Harness	Climbing	Jerry Martin
8	302000	Locking carabiner, Oval	Climbing	Jerry Martin

Reading Specified Rows from a Single Table

Suppose we want all of the *columns* of the SKU_DATA table, but we want only the *rows* for the Water Sports department. We can obtain that result by using the SQL WHERE clause as follows:

```
SELECT      *
FROM        SKU_DATA
WHERE       Department='Water Sports';
```

Amazon Ex. 1008.0066

The result of this statement will be:

	SKU	SKU_Description	Department	Buyer
1	100100	Std. Scuba Tank, Yellow	Water Sports	Pete Hansen
2	100200	Std. Scuba Tank, Magenta	Water Sports	Pete Hansen
3	101100	Dive Mask, Small Clear	Water Sports	Nancy Meyers
4	101200	Dive Mask, Med Clear	Water Sports	Nancy Meyers

In an SQL WHERE clause, if the column contains text or date data, the comparison values must be enclosed in single quotation marks (' '). If the column contains numeric data, however, the comparison values need not be in quotes. Thus, to find all of the SKU rows with a value greater than 200,000, we would use the SQL statement (note that no comma is provided in the numeric value code):

```
SELECT      *
FROM        SKU_DATA
WHERE       SKU > 200000;
```

The result is:

	SKU	SKU_Description	Department	Buyer
1	201000	Half-dome Tent	Camping	Cindy Lo
2	202000	Half-dome Tent Footprint	Camping	Cindy Lo
3	301000	Light Fly Climbing Harness	Climbing	Jerry Martin
4	302000	Locking carabiner, Oval	Climbing	Jerry Martin

BY THE WAY SQL is very fussy about single quotes. It wants the plain, nondirectional quotes found in basic text editors. The fancy directional quotes produced by many word processors will produce errors. For example, the data value 'Water Sports' is correctly stated, but ‘Water Sports’ is not. Do you see the difference?

Reading Specified Columns and Rows from a Single Table

So far, we have selected certain columns and all rows and we have selected all columns and certain rows. We can combine these operations to select certain columns and certain rows by naming the columns we want and then using the SQL WHERE clause. For example, to obtain the SKU_Description and Department of all products in the Climbing department, we use the SQL query:

```
SELECT      SKU_Description, Department
FROM        SKU_DATA
WHERE       Department='Climbing';
```

The result is:

	SKU_Description	Department
1	Light Fly Climbing Harness	Climbing
2	Locking carabiner, Oval	Climbing

SQL does not require that the column used in the WHERE clause also appear in the SELECT clause column list. Thus, we can specify:

```
SELECT      SKU_Description, Buyer
FROM        SKU_DATA
WHERE       Department='Climbing';
```

Amazon Ex. 1008.0067

where the qualifying column, Department, does not appear in the SELECT clause column list. The result is:

	SKU_Description	Buyer
1	Light Fly Climbing Harness	Jerry Martin
2	Locking carabiner, Oval	Jerry Martin

BY THE WAY Standard practice is to write SQL statements with the SELECT, FROM, and WHERE clauses on separate lines. This practice is just a coding convention, however, and SQL parsers do not require it. You could code the last SQL statement all on one line as:

```
SELECT SKU_Description, Buyer FROM SKU_DATA WHERE Department=
'Climbing';
```

All DBMS products would process the statement written in this fashion. However, the standard multiline coding convention makes SQL easier to read, and we encourage you to write your SQL according to it.

Submitting SQL Statements to the DBMS

Before continuing the explanation of SQL, it will be useful for you to learn how to submit SQL statements to specific DBMS products. That way, you can work along with the text by keying and running SQL statements as you read the discussion. The particular means by which you submit SQL statements depends on the DBMS. Here we will describe the process for Access, SQL Server, Oracle Database, and MySQL.

BY THE WAY You can learn SQL without running the queries in a DBMS, so if for some reason you do not have Access, SQL Server, Oracle Database, or MySQL readily available, do not despair. You can learn SQL without them. Chances are your instructor, like a lot of us in practice today, learned SQL without a DBMS. It is just that SQL statements are easier to understand and remember if you can run the SQL while you read. However, given that there are freely downloadable versions of Microsoft SQL Server 2008 Express Edition, Oracle Database 10g Express Edition, and Sun Microsystems MySQL Server Community Edition, you can have an installed DBMS to run these SQL examples even if you haven't purchased Microsoft Access. See Chapters 10, 10A, and 10B for specific instructions, for creating databases using each of these products. The SQL scripts needed to create the Cape Codd Outdoor Sports database used in this chapter are available at *www.pearsonhighered.com/kroenke*.

Using SQL in Microsoft Access 2007

Before you can execute SQL statements, you need a computer that has Microsoft Access installed, and you need a Microsoft Access database that contains the tables and sample data in Figure 2-4. Microsoft Access is part of many versions of the Microsoft Office suite, so it should not be too difficult to find a computer that has it.

Because Microsoft Access is commonly used in classes that use this book as a textbook, we will look how to use SQL in Microsoft Access in some detail. Before we proceed, however, we need to discuss a specific peculiarity of Microsoft Access—the limitations of the default version of SQL used in Access.

Amazon Ex. 1008.0068

"Does Not Work with Microsoft Access ANSI-89 SQL"

As mentioned previously, our discussion of SQL is based on SQL features present in SQL standards since the ANSI SQL-92 standard (which Microsoft refers to as ANSI-92 SQL). Unfortunately, Microsoft Access 2007 still defaults to the earlier SQL-89 version—Microsoft calls it ANSI-89 SQL or Microsoft Jet SQL (after the Microsoft Jet DBMS engine used by Access). ANSI-89 SQL differs significantly from SQL-92, and, therefore, some features of the SQL-92 language will not work in Access.

Microsoft Access 2007 (and the earlier Microsoft Access 2003 version) does contain a setting that allows you to use SQL-92 instead of the default ANSI-89 SQL. Microsoft included this option to allow Access tools such as forms and reports to be used in application development for Microsoft SQL Server, which supports newer SQL standards. To set the option in Microsoft Access 2007, click the **Microsoft Office Button** and then click the **Access Options** button to open the Access Options dialog box. In the Access Options dialog box, click the **Object Designers** button to display the Access Options Object Designers page, as shown in Figure 2-5.

As shown in Figure 2-5, the **SQL Server Compatible Syntax (ANSI 92)** options control which version of SQL is used in an Access 2007 database. If you check the **This database** check box, you will use SQL-92 syntax in the current database. Or, you can check the **Default for new databases** check box to make SQL-92 syntax the default for all new databases you create. When you click the **OK** button to save the changed SQL syntax option, the SQL-Syntax Information dialog box shown in Figure 2-6 will be displayed. Read the information, and then click the **OK** button to close the dialog box.

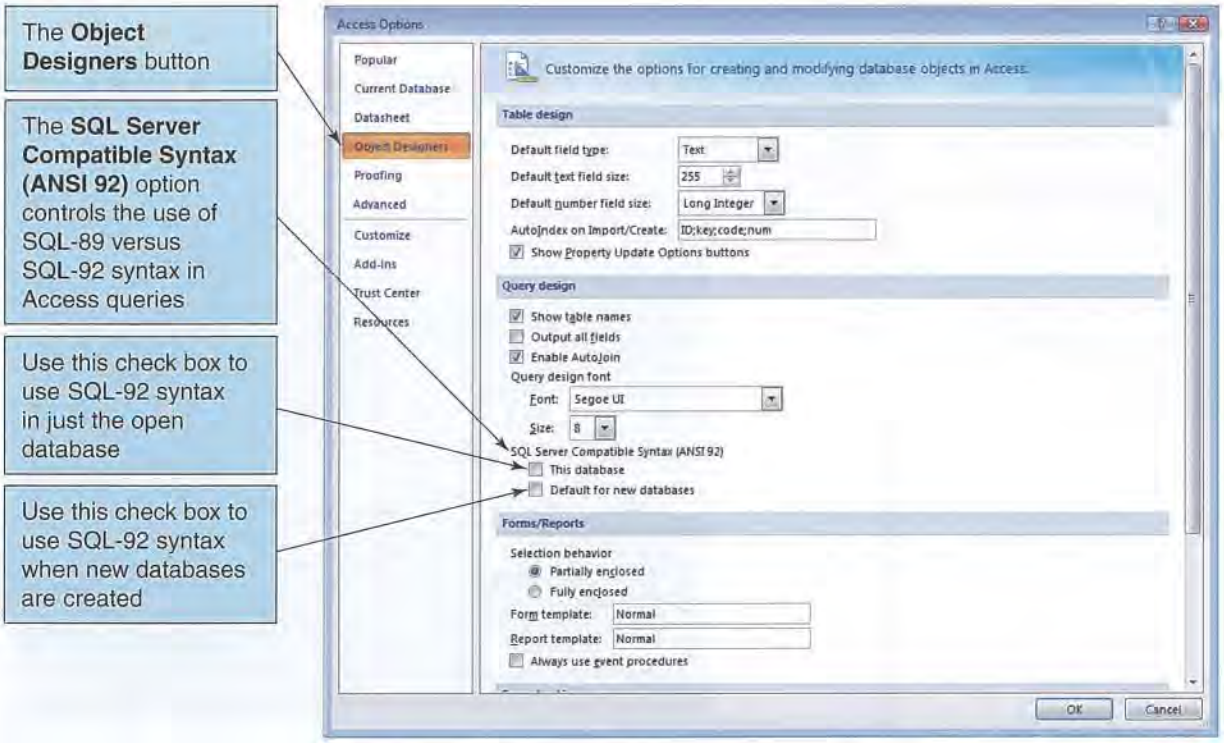
Unfortunately, very few Microsoft Access users or organizations using Access are likely to set the Access SQL version to the SQL-92 option, and, in this chapter, we assume that Access is running in the default ANSI-89 SQL mode. One advantage of doing so is that it will help you understand the limitations of Access ANSI-89 SQL and how to cope with them.

In the discussion that follows, we use "Does Not Work with Microsoft Access ANSI-89 SQL" boxes to identify SQL commands and SQL clauses that do not work in Access ANSI-89 SQL. We also identify any workarounds that are available. Remember that the one *permanent* workaround is to choose to use the SQL-92 syntax option in the databases you create!

Nonetheless, two versions of the Microsoft Access 2007 Cape Codd Outdoor Sports database are available at *www.pearsonhighered.com/kroenke* for your use with this chapter.

Figure 2-5

The Microsoft Access 2007 Options Object Designers Page



Amazon Ex. 1008.0069

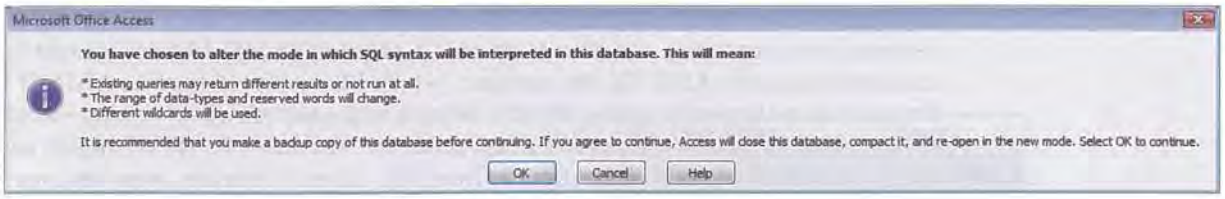


Figure 2-6

The Microsoft Access 2007 SQL-Syntax Information Dialog Box

The Access database file named *Cape-Codd.accdb* is set to use Microsoft Access ANSI-89, whereas the Access database file name *Cape-Codd-SQL-92.accdb* is set to use Microsoft Access SQL-92. Choose the one you want to use (or use them both and compare the results!). Note that these files contain two additional tables (INVENTORY and WAREHOUSE) that we will not use in this chapter, but that you will need for the Review Questions at the end of the chapter.

Alternatively, of course, you can create your own Access database and then add the tables and data in Figures 2-3 and 2-4, as described in Appendix A. If you create your own database, look at the Review Questions at the end of the chapter and create the INVENTORY and WAREHOUSE tables shown there in addition to the RETAIL-ORDER, ORDER_ITEM and SKU tables shown in the chapter discussion. This will make sure that what you see on your monitor matches the screenshots in this chapter.

Whether you download the database file or build it yourself, you will need to do one or the other before you can proceed.

Processing SQL Statements in Microsoft Access 2007

To process an SQL statement in Microsoft Access 2007, first open the database in Access as described in Appendix A and then create a new tabbed Query window.

Opening an Access Query Window in Design View

1. Click the **Create** command tab to display the Create command groups, as shown in Figure 2-7.
2. Click the **Query Design** button.
3. The Query1 tabbed document window is displayed in Design view, along with the Show Table dialog box, as shown in Figure 2-8.

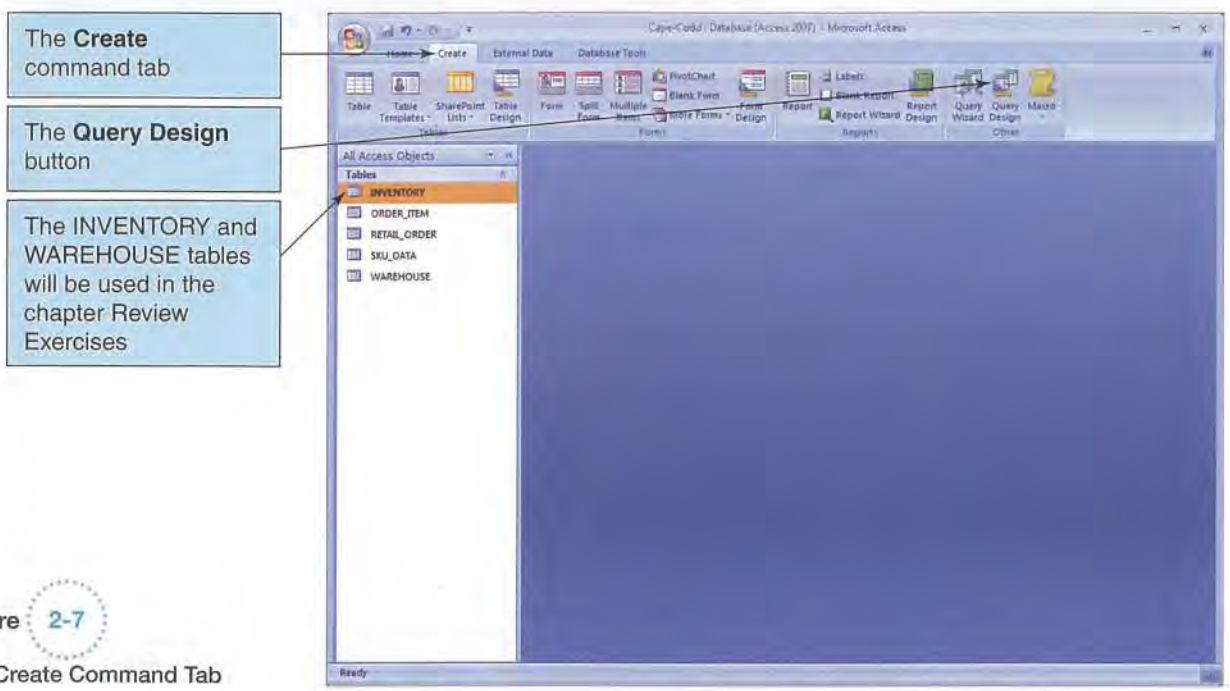


Figure 2-7

The Create Command Tab

Amazon Ex. 1008.0070

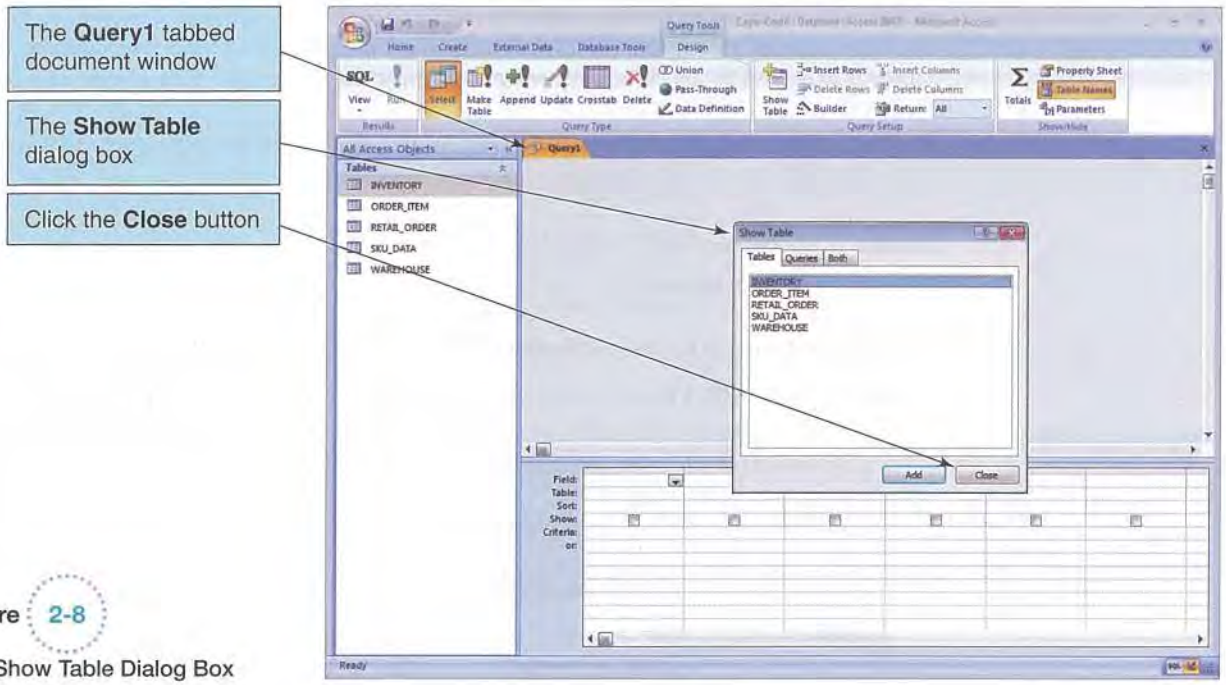


Figure 2-8

The Show Table Dialog Box

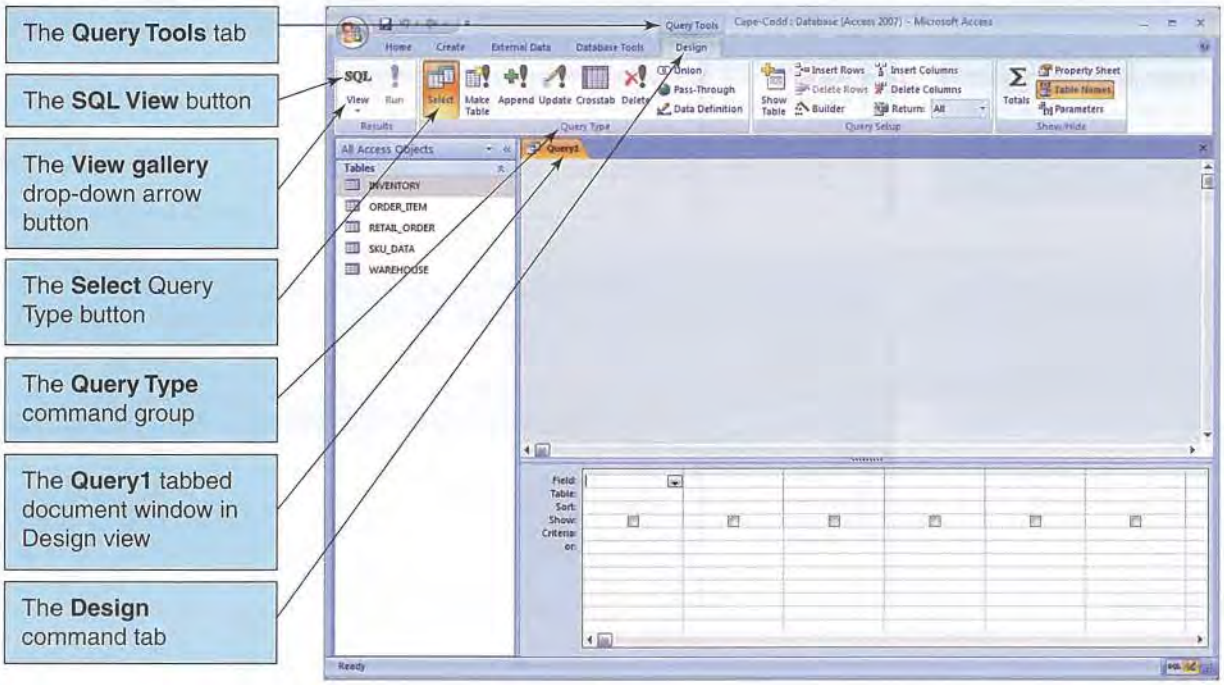
4. Click the **Close** button on the Show Table dialog box. The Query1 document window now looks as shown in Figure 2-9. This window is used for creating and editing Access queries in Design view and is used with Access QBE.

Note that in Figure 2-9, the Select button is selected in the Query Type group on the Design tab. You can tell this is so because active or selected buttons are always shown in color on the Ribbon. This indicates that we are creating a query that is the equivalent of an SQL SELECT statement.

Also note that in Figure 2-9 the View gallery is available in the Results group of the Design tab. We can use this gallery to switch between Design view and SQL view. However, we can also

Figure 2-9

The Query Tools Contextual Command Tab



Amazon Ex. 1008.0071

just use the displayed SQL View button to switch to SQL view. The SQL View button is being displayed because Access considers that to be the view you would most likely choose in the gallery if you used it. Access always presents a "most likely needed" view choice as a button above the View gallery.

For our example SQL query in Microsoft Access, we will use the same query that we used as the first SQL query earlier in our discussion:

```
SELECT      Department, Buyer
FROM        SKU_DATA;
```

Opening an Access SQL Query Window and Running an Access SQL Query

1. Click the **SQL View** button in the Results group on the Design tab. The Query1 window switches to the SQL view, as shown in Figure 2-10. Note the basic SQL command **SELECT;** that's shown in the window. This is an incomplete command, and running it will not produce any results.
2. Edit the SQL SELECT command to read:

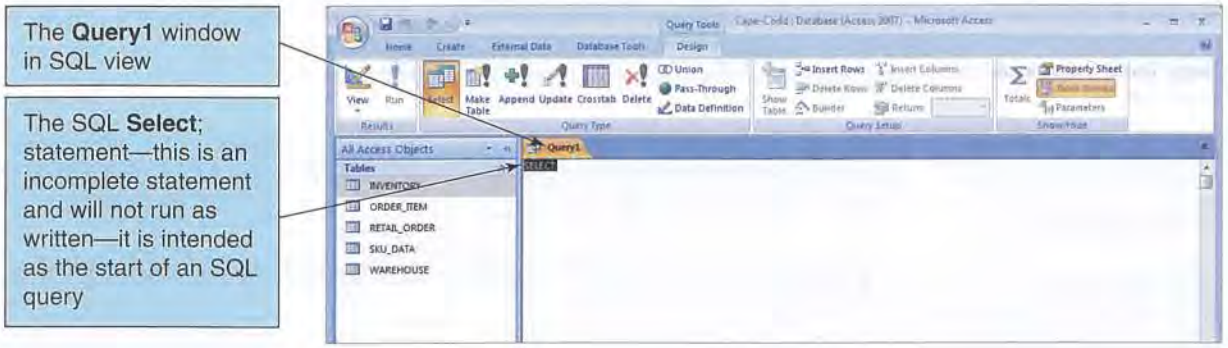
```
SELECT      Department, Buyer
FROM        SKU_DATA;
```

   as shown in Figure 2-11.
3. Click the **Run** button on the Design tab. The query results appear, as shown in Figure 2-12. Compare the results shown in Figure 2-12 to the SQL query results shown on page 37.

Because Microsoft Access is a personal database and includes an application generator, we can save Access queries for future use. Enterprise-level DBMS products generally do not allow us to save queries (although they do allow us to save SQL Views within the database and SQL query scripts as separate files—we will discuss these methods later).

Figure 2-10

The Query1 Window in SQL View



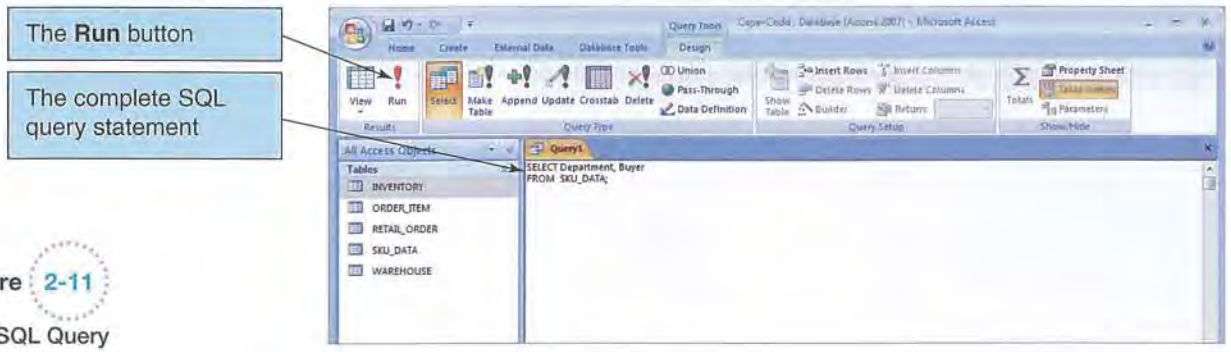


Figure 2-11

The SQL Query

Amazon Ex. 1008.0072

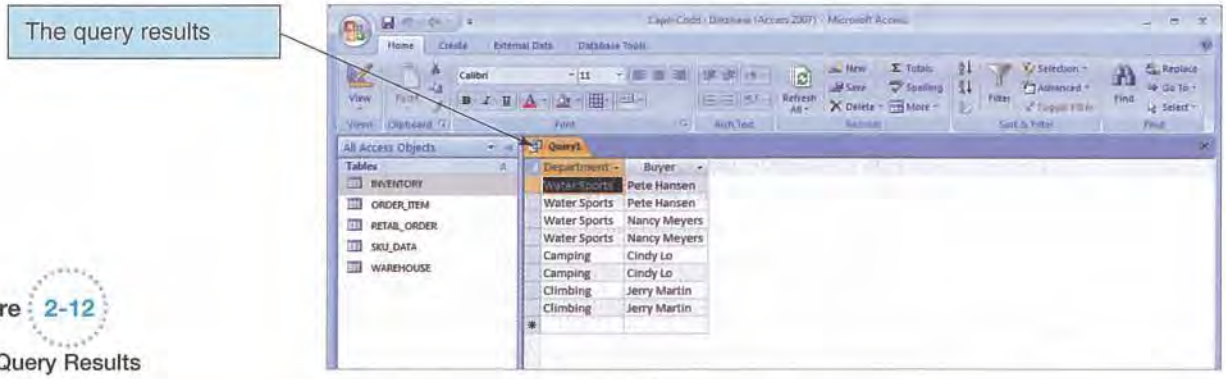


Figure 2-12
The Query Results

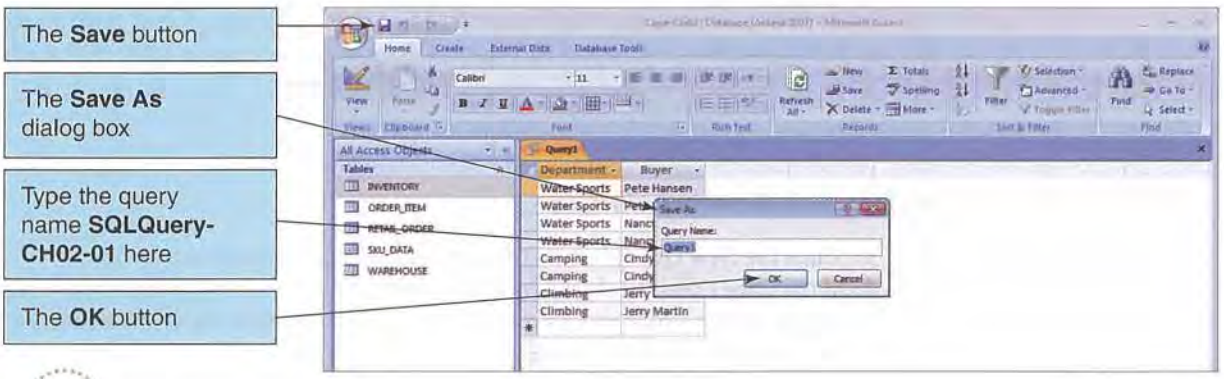


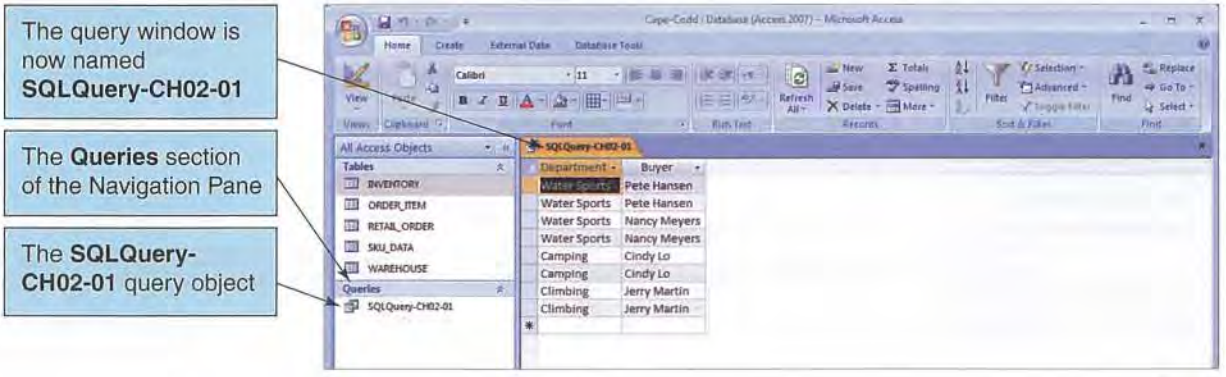
Figure 2-13
The Query Results

Saving an Access SQL Query

1. To save the query, click the **Save** button on the Quick Access Toolbar. The Save As dialog box appears, as shown in Figure 2-13.
2. Type in the query name **SQLQuery-CH02-01** and then click the **OK** button. The query is saved, and the window is renamed with the query name. As shown in Figure 2-14, the query document window is now named SQLQuery-CH02-01, and a newly created SLQQuery-CH02-01 query object appears in a Queries section of the Navigation Pane.
3. Close the SQLQuery-CH02-01 window by clicking the document window's **Close** button.
4. If Access displays a dialog box, asking whether you want to save changes to the design of the query SQLQuery-CH02-01, click the **Yes** button.

At this point, you should work through each of the other nine queries in the preceding discussion of the SQL SELECT/FROM/WHERE framework. Save each query as SQLQuery-CH02-##, where ## is a sequential number from 02 to 09.

Figure 2-14
The Named and Saved Query



Amazon Ex. 1008.0073

Using SQL in Microsoft SQL Server 2008

Before you can use SQL statements with Microsoft SQL Server, you need access to a computer that has SQL Server installed and that has a database with the tables and data shown in Figures 2-3 and 2-4. Your instructor may have installed SQL Server in your computer lab and entered the data for you. If so, follow his or her instructions for accessing that database. Otherwise, you will need to obtain a copy of SQL Server 2008 and install it on your computer. Read the appropriate sections of Chapter 11 about obtaining and installing SQL Server 2008.

After you have SQL Server 2008 installed, you will need to read the introductory discussion for using SQL Server in Chapter 10, starting on page 353, and create the Cape Codd database. SQL Server scripts for creating and populating the Cape Codd database tables are available on our Web site at *www.pearsonhighered.com/kroenke*.

SQL Server 2008 uses the Microsoft SQL Server 2008 Management Studio as the GUI tool for managing the SQL Server DBMS and the databases controlled by the DBMS. The Microsoft SQL Server 2008 Management Studio is installed as part of the SQL Server 2008 installation process and is discussed in Chapter 10. Figure 2-15 shows the execution of the SQL statement:

```
SELECT      Department, Buyer
FROM        SKU_DATA;
```

Running an SQL Server SQL Query

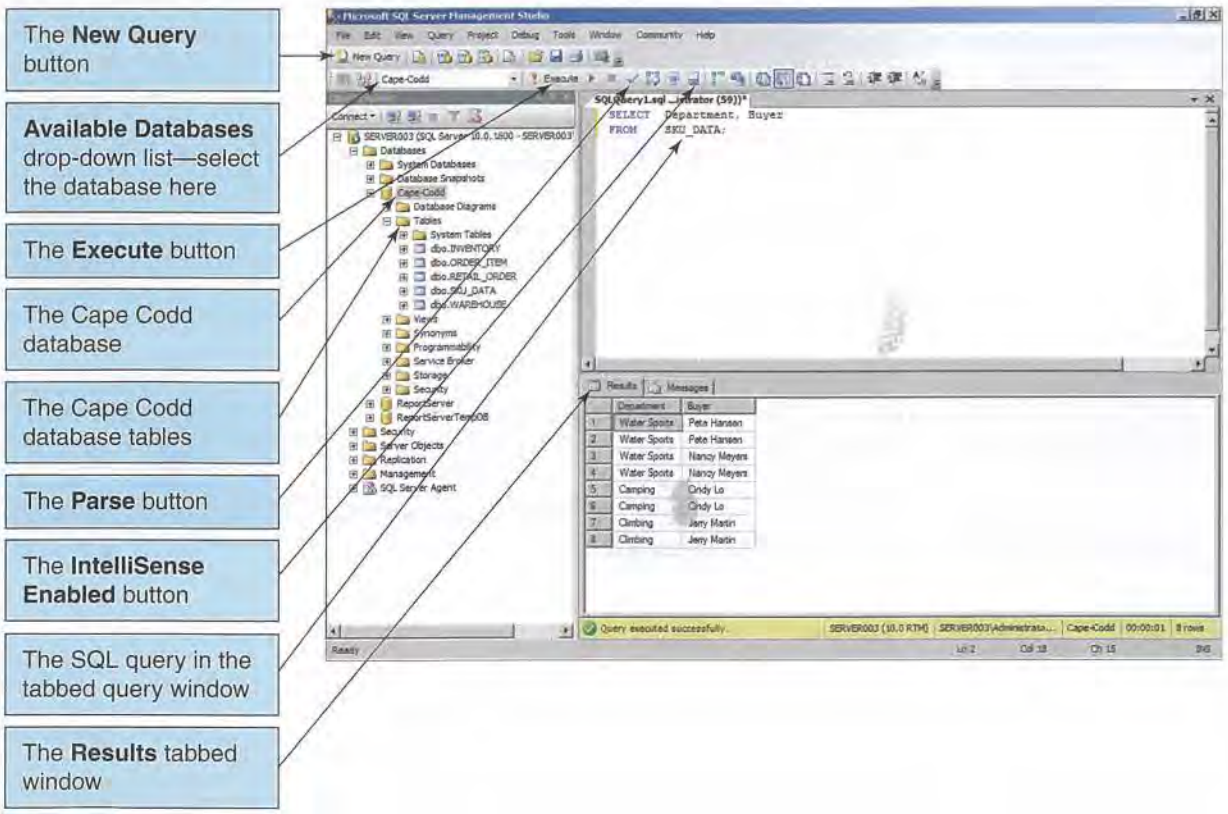
1. Click the **New Query** button to display a new tabbed query window.
2. If the Cape Codd database is not displayed in the Available Database box, select it in the Available Databases drop-down list, and then click the **Intellisense Enabled** button to *disable* Intellisense.
3. Type the SQL SELECT command:

```
SELECT      Department, Buyer
FROM        SKU_DATA;
```

in the query window, as shown in Figure 2-15.

Figure 2-15

Running an SQL Query in SQL Server Management Studio



Amazon Ex. 1008.0074

4. At this point you can check the SQL command syntax before actually running the command by clicking the **Parse** button. A Results window will be displayed in the same location shown in Figure 2-15, but with the message "Command(s) completed successfully" if the SQL command syntax is correct or with an error message if there is a problem with the syntax.
5. Click the Execute button to run the query. The results are displayed in a results window, as shown in Figure 2-15.

Note that in Figure 2-15 the Cape Codd database object in the Object Browser in the left side window of the SQL Server Management Studio has been expanded to show the tables in the Cape Codd database. Many of the functions of the SQL Server Management Studio are associated with the objects in the Object Browser and are often accessed by right-clicking the object to display a shortcut menu.

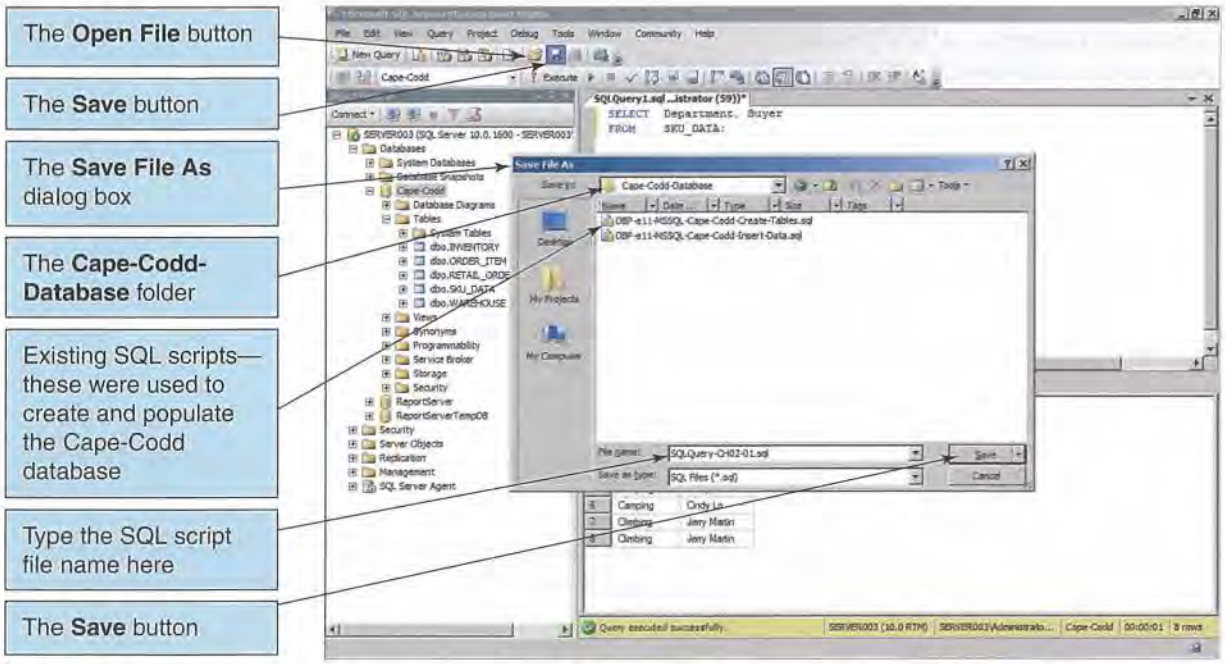
BY THE WAY We are using SQL Server 2008 Enterprise edition running in Microsoft Server 2008. When we give specific sequences of steps to follow in the text or figures in this book, we use the command terminology used by SQL Server 2008 and associated utility programs in Microsoft Server 2008. If you are running a workstation operating system such as Microsoft XP or Microsoft Vista, the terminology may vary somewhat.

SQL Server 2008 is an enterprise-class DBMS product, and, as is typical of such products, does not store queries within the DBMS (it does store SQL Views, which can be considered a type of query, and we will discuss SQL Views later in this chapter). However, you can save queries as SQL script files. An **SQL script file** is a separately stored plain text file, and it usually uses a file name extension of **.sql*. An SQL script can be opened and run as an SQL command (or set of commands). Often used to create and populate databases, scripts can also be used to store a query or set of queries. Figure 2-16 shows the SQL query being saved as an SQL script.

Note that in Figure 2-16 the SQL scripts are shown in a folder named *Cape-Codd-Database*. When the Microsoft SQL Server 2008 Management Studio is installed, a new folder named *SQL Server Management Studio* is created in your Documents (My Documents in Windows XP) folder, with *Projects* as a subfolder. The Projects folder is the default location used by SQL Server 2008 for SQL script files.

Figure 2-16

Saving an SQL Query as an SQL Script in SQL Server Management Studio



Amazon Ex. 1008.0075

We recommend that you create a folder for each database in the Projects folder. We have created a folder named *Cape-Code-Database* to store the script files associated with the Cape Codd database.

Saving an SQL Server Query as an SQL Script

1. Click the **Save** button shown in Figure 2-16. The Save File As dialog appears, as shown in Figure 2-16.
2. Browse to the */SQL Server Management Studio/Project/Cape-Codd-Database* folder.
3. Note that there are already two SQL script names displayed in the dialog box. These are the SQL scripts that were used to create and populate the Cape Codd database tables, and they are available on our Web site at *www.pearsonhighered.com/kroenke*.
4. In the File Name text box, type the SQL script file name *SQLQuery-CH02-01*.
5. Click the **Save** button.

To rerun the saved query, you would click the **Open File** button shown in Figure 2-16 to open the Open File dialog box, open the query, and then click the **Execute** button.

At this point, you should work through each of the other nine queries in the preceding discussion of the SQL SELECT/FROM/WHERE framework. Save each query as SQLQuery-CH02-##, where ## is a sequential number from 02 to 09.

Using SQL in Oracle Database 11g

Before you can enter SQL statements to Oracle Database 11g, you need access to a computer that has Oracle installed and that has a database with the tables and data shown in Figure 2-4. Your instructor may have installed Oracle on a computer in the lab and entered the data for you. If so, follow his or her instructions for accessing that database. Otherwise, you will need to obtain a copy of Oracle Database 11g and install it on your computer. Read the appropriate sections of Chapter 10A about obtaining and installing Oracle Database 11g.

After you have installed Oracle 11g, you will need to read the introductory discussion for Oracle in Chapter 10A, starting on page 10A-5, and create the Cape Codd database. Oracle scripts for creating and populating the Cape Codd database tables are available on our Web site at *www.pearsonhighered.com/kroenke*.

Although Oracle users have been dedicated to the Oracle SQL*Plus command line tool, professionals are moving to the new Oracle SQL Developer GUI tool. This application is installed as part of the Oracle installation, and updated versions are available for free download at *www.oracle.com/technology/software/products/sql/index.html*. We will use it as our standard GUI tool for managing the databases created by the Oracle DBMS. Figure 2-17 shows the execution of the SQL statement:

```
SELECT      Department, Buyer
FROM        SKU_DATA;
```

Running an SQL Server SQL Query

1. Click the **New Connection** button and open the Cape Codd database.
2. In the tabbed SQL Worksheet, type the SQL SELECT command:

   ```
   SELECT      Department, Buyer
   FROM        SKU_DATA;
   ```

   as shown in Figure 2-17.
3. Click the **Execute** button to run the query. The results are displayed in a results window, as shown in Figure 2-17.

Note that in Figure 2-17, the Cape Codd database object in the Object Browser in the left side Connection object browser of the Oracle SQL Developer has been expanded to show the tables

Amazon Ex. 1008.0076

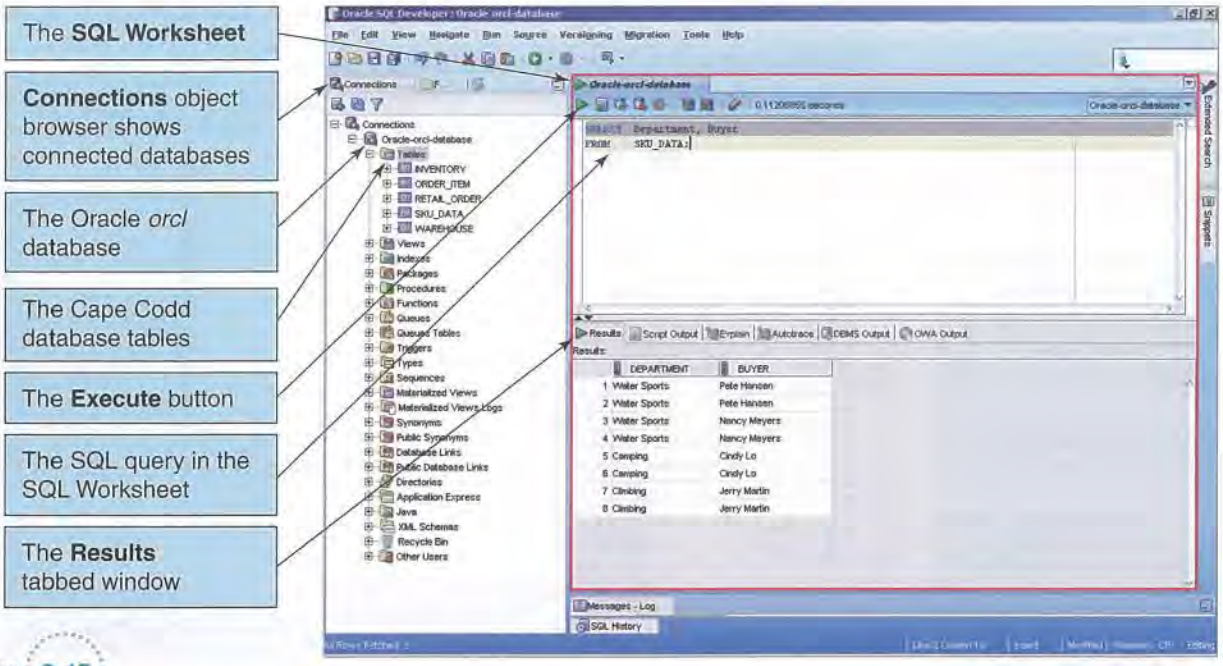


Figure 2-17

Running an SQL Query in Oracle SQL Developer

in the Cape Codd database. Many of the functions of SQL Developer are associated with the objects in the Connections object browser and are often accessed by right-clicking the object to display a shortcut menu.

> **BY THE WAY** We are using Oracle Database 11g Enterprise edition running in Microsoft Server 2008. When we give specific sequences of steps to follow in the text or figures in this book, we use the command terminology used by Oracle Database 11g and associated utility programs in Microsoft Server 2008. If you are running a workstation operating system such as Microsoft XP, Microsoft Vista, or Linux, the terminology may vary somewhat.

Oracle 11g is an enterprise-class DBMS product, and, as is typical of such products, does not store queries within the DBMS (it does store SQL Views, which can be considered a type of query, and we will discuss SQL Views later in this chapter). However, you can save queries as SQL script files. An SQL script file is a separately stored plain text file and it usually uses a file name extension of **.sql*. An SQL script can be opened and run as an SQL command (or set of commands). Often used to create and populate databases, scripts can also be used to store a query or set of queries. Figure 2-18 shows the SQL query being saved as an SQL script.

Note that in Figure 2-18 the SQL scripts are shown in a folder named *{UserName}\Documents\Oracle Workspace\Cape-Codd-Database*. By default, Oracle SQL Developer stores *.sql files in an obscure location within its own application files. We recommend that you create a subfolder in your Documents folder (My Documents in Windows XP) named Oracle Workspace, and then create a subfolder for each database in the Oracle Workspace folder. We have created a folder named *Cape-Code-Database* to store the script files associated with the Cape Codd database.

When we did this, a *Cape-Codd-Database Folder* button was added to the Save dialog box shown in Figure 2-18. You can easily move to the Cape-Codd-Database folder by clicking this button.

Amazon Ex. 1008.0077

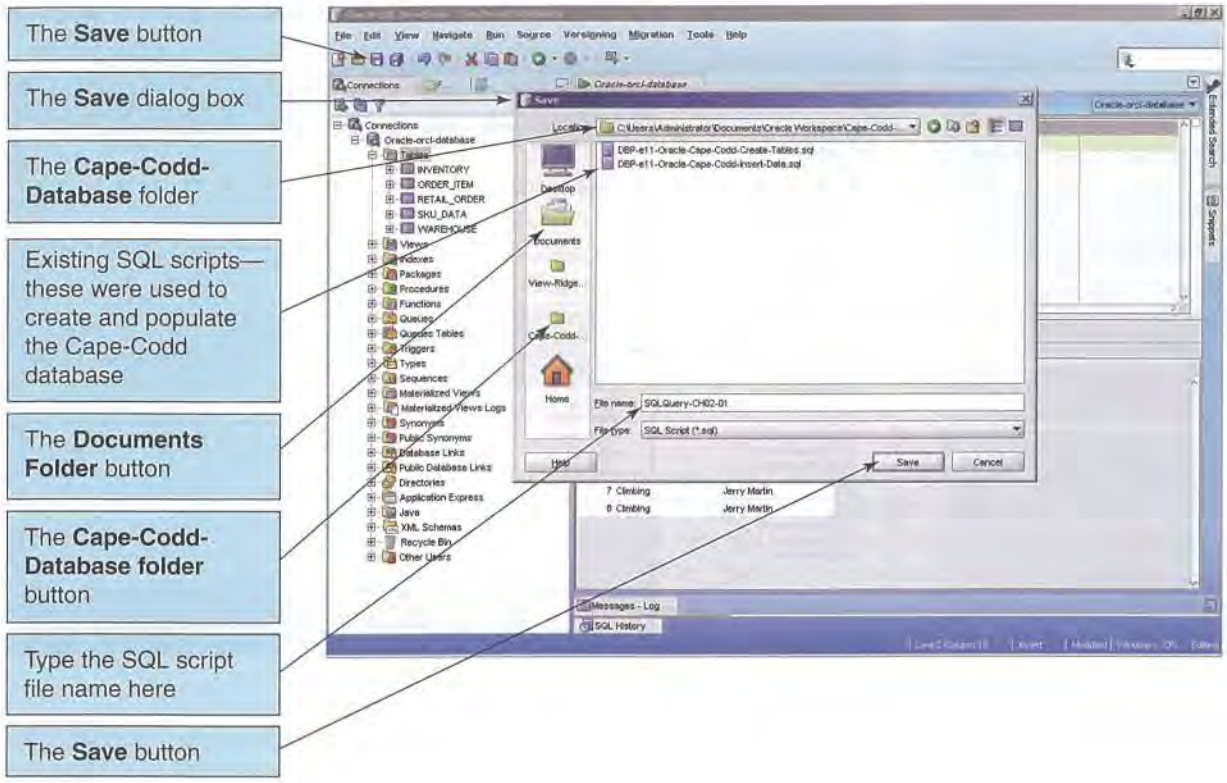


Figure 2-18

Saving an SQL Query as an SQL Script in Oracle SQL Developer

Saving an SQL Script in Oracle SQL Developer

1. Click the **Save** button shown in Figure 2-18. The Save dialog appears, as shown in Figure 2-18.
2. Click the **Cape-Codd-Database Folder** button on the Save dialog box to move to the Cape-Code-Database folder. Alternatively, you can click the **Documents** button on the Save dialog box to move to the Documents folder, and then browse to the Cape-Codd-Database folder.
3. Note that there are already two SQL script names displayed in the dialog box. These are the SQL scripts that were used to create and populate the Cape Codd database tables, and they are available on our Web site at *www.pearsonhighered.com/kroenke*.
4. In the File Name text box, type the SQL script file name *SQLQuery-CH02-01.sql*.
5. Click the **Save** button.

To rerun the saved query, you would click the SQL Developer **Open File** button to open the Open File dialog box, browse to the query file, open the query file, and then click the **Execute** button.

At this point, you should work through each of the other nine queries in the preceding discussion of the SQL SELECT/FROM/WHERE framework. Save each query as SQLQuery-CH02-##, where ## is a sequential number from 02 to 09.

Using SQL in Sun Microsystems MySQL 5.1

Before you can use SQL statements with Sun Microsystems MySQL, you need access to a computer that has MySQL installed and that has a database with the tables and data shown in Figure 2-4. Your instructor may have installed MySQL in your computer lab and entered the data for you. If so, follow his or her instructions for accessing that database. Otherwise, you will need to obtain a copy of MySQL Server 5.1 and install it on your computer. Read the appropriate sections of Chapter 10B about obtaining and installing SQL Server 2008.

After you have MySQL installed, you will need to read the introductory discussion for MySQL Server in Chapter 10B, starting on page 10B-4, and create the Cape Codd database. MySQL scripts for creating and populating the Cape Codd database tables are available on our Web site at *www.pearsonhighered.com/kroenke*.

Amazon Ex. 1008.0078

MySQL uses two tools, the MySQL Administrator and the MySQL Query Browser, as the GUI tools for managing the MySQL DBMS and the databases controlled by the DBMS. These tools must be installed separately from the MySQL DBMS, and this is discussed in Chapter 10B. SQL statements are created and run in the MySQL Query Browser, and Figure 2-19 shows the execution of the SQL statement:

```
SELECT      Department, Buyer
FROM        SKU_DATA;
```

Running an SQL Query in the MySQL Query Browser

1. To make the Cape Codd database the default schema (active database), right-click the Cape Codd schema (database) object to display the shortcut menu and then click the **Make Default** command.
2. Type the SQL SELECT command:

```
SELECT      Department, Buyer
FROM        SKU_DATA;
```

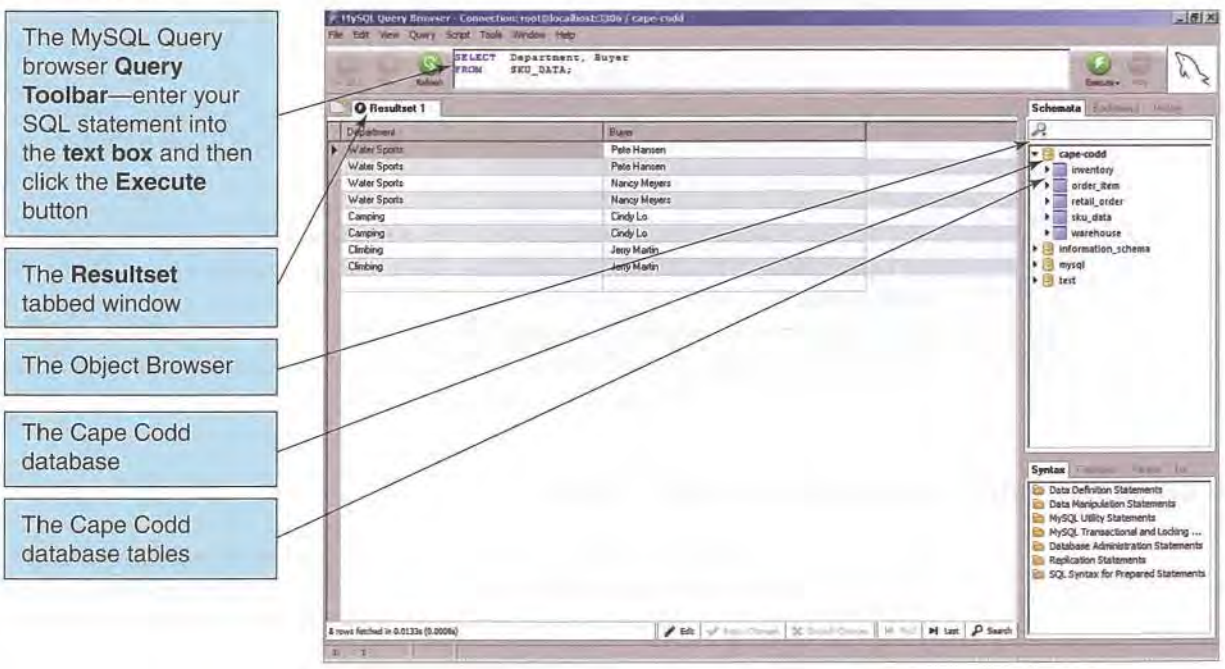
   in the **Query Toolbar text box,** as shown in Figure 2-19.
3. Click the **Execute** button to run the query. The results are displayed in a Resultset window, as shown in Figure 2-19.

Note that in Figure 2-19 the Cape Codd database object in the Object Browser in the right-side window of the MySQL Query Browser has been expanded to show the tables in the Cape Codd database. Many of the functions of the MySQL Query Browser are associated with the objects in the Object Browser and are often accessed by right-clicking the object to display a shortcut menu.

BY THE WAY We are using MySQL 5.1 Community Server running in Microsoft Server 2008. When we give specific sequences of steps to follow in the text or figures in this book, we use the command terminology used for MySQL 5.1 and associated utility programs in Microsoft Server 2008. If you are running a workstation operating system such as Microsoft XP, Microsoft Vista, or Linux, the terminology may vary somewhat.

Figure 2-19

Running an SQL Query in MySQL Query Browser



Amazon Ex. 1008.0079

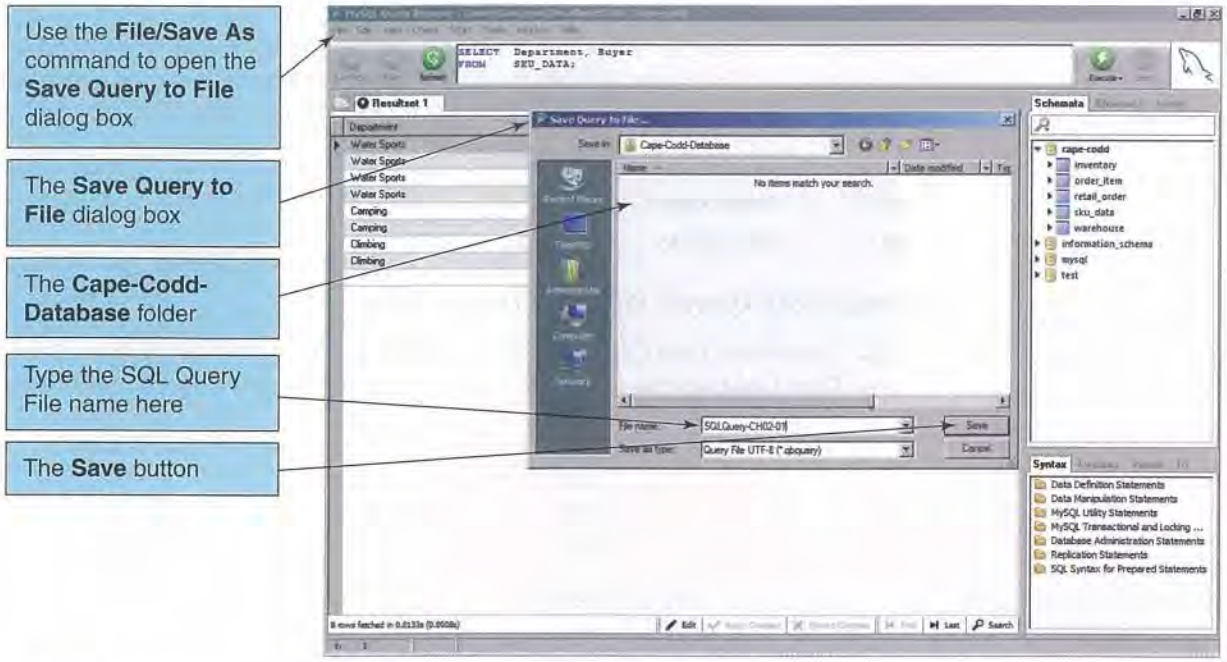


Figure 2-20

Saving an SQL Query as a MySQL Query File in MySQL Query Browser

MySQL 5.1 is an enterprise-class DBMS product, and, as is typical of such products, does not store queries within the DBMS (it does store SQL Views, which can be considered a type of query, and we will discuss SQL Views later in this chapter). However, you can save MySQL queries as **MySQL query files**. A MySQL query file is a separately stored plain text file, and it usually uses a file name extension of **.qbquery*. A MySQL query file can be opened and run as an SQL command. Figure 2-20 shows the SQL query being saved as a MySQL query file.

Note that in Figure 2-20 the query will be saved in a folder named *{UserName}\Documents\MySQL Workspace\Cape-Codd-Database*. By default, MySQL Query Browser stores files in the user's Documents folder (My Documents in Windows XP). We recommend that you create a subfolder in your Documents folder named MySQL Workspace, and then create a subfolder for each database in the MySQL Workspace folder. We have created a folder named *Cape-Code-Database* to store the script files associated with the Cape Codd database.

Saving a MySQL Query

1. Use the **File | Save as** command as shown in Figure 2-20. The Save Query to File dialog appears, as shown in Figure 2-20.
2. Browse to the *Documents\MySQL Workspace\Cape-Codd-Database* folder.
3. In the File Name text box, type the SQL query file name *SQLQuery-CH02-01*.
4. Click the **Save** button.

To rerun the saved query, you would click the **File | Open Query** menu command to open the **Open Query from File** dialog box, then select and open the query, and finally click the **Execute** button.

At this point, you should work through each of the other nine queries in the preceding discussion of the SQL SELECT/FROM/WHERE framework. Save each query as SQLQuery-CH02-##, where ## is a sequential number from 02 to 09.

SQL Enhancements for Querying a Single Table

We started our discussion of SQL queries with SQL statements for processing a single table, and now we will add an additional SQL feature to those queries. As we proceed, you will begin to see how powerful SQL can be for querying databases and for creating information from existing data.

Amazon Ex. 1008.0080

BY THE WAY The SQL results shown in this chapter were generated using Microsoft SQL Server 2008. Query results from other DBMS products will be similar, but may vary a bit.

Sorting the Results

The order of the rows produced by an SQL statement is arbitrary and determined by programs in the bowels of each DBMS. If you want the DBMS to display the rows in a particular order, you can use the **SQL ORDER BY clause**. For example, the SQL statement:

```
SELECT      *
FROM        ORDER_ITEM
ORDER BY    OrderNumber;
```

will generate the following results:

	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	1000	201000	1	300.00	300.00
2	1000	202000	1	130.00	130.00
3	2000	101100	4	50.00	200.00
4	2000	101200	2	50.00	100.00
5	3000	101200	1	50.00	50.00
6	3000	101100	2	50.00	100.00
7	3000	100200	1	300.00	300.00

We can sort by two columns by adding a second column name. For example, to sort first by OrderNumber and then by Price within OrderNumber, we use the following SQL query:

```
SELECT      *
FROM        ORDER_ITEM
ORDER BY    OrderNumber, Price;
```

The result for this query is:

	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	1000	202000	1	130.00	130.00
2	1000	201000	1	300.00	300.00
3	2000	101100	4	50.00	200.00
4	2000	101200	2	50.00	100.00
5	3000	101200	1	50.00	50.00
6	3000	101100	2	50.00	100.00
7	3000	100200	1	300.00	300.00

If we want to sort the data by Price and then by OrderNumber, we would simply reverse the order of those columns in the ORDER BY clause as follows:

```
SELECT      *
FROM        ORDER_ITEM
ORDER BY    Price, OrderNumber;
```

Amazon Ex. 1008.0081

with the results:

	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	2000	101100	4	50.00	200.00
2	2000	101200	2	50.00	100.00
3	3000	101200	1	50.00	50.00
4	3000	101100	2	50.00	100.00
5	1000	202000	1	130.00	130.00
6	1000	201000	1	300.00	300.00
7	3000	100200	1	300.00	300.00



Note to Microsoft Access users: Unlike the SQL Server output shown here, Microsoft Access displays dollar signs in the output of currency data.

By default, rows are sorted in ascending order. To sort in descending order, add the **SQL DESC keyword** after the column name. Thus, to sort first by Price in descending order and then by OrderNumber in ascending order, we use the SQL query:

```
SELECT      *
FROM        ORDER_ITEM
ORDER BY    Price DESC, OrderNumber ASC;
```

The result is:

	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	1000	201000	1	300.00	300.00
2	3000	100200	1	300.00	300.00
3	1000	202000	1	130.00	130.00
4	2000	101100	4	50.00	200.00
5	2000	101200	2	50.00	100.00
6	3000	101200	1	50.00	50.00
7	3000	101100	2	50.00	100.00

Because the default order is ascending, it is not necessary to specify ASC in the last SQL statement. Thus, the following SQL statement is equivalent to the previous SQL query:

```
SELECT      *
FROM        ORDER_ITEM
ORDER BY    Price DESC, OrderNumber;
```

and produces the same results:

	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	1000	201000	1	300.00	300.00
2	3000	100200	1	300.00	300.00
3	1000	202000	1	130.00	130.00
4	2000	101100	4	50.00	200.00
5	2000	101200	2	50.00	100.00
6	3000	101200	1	50.00	50.00
7	3000	101100	2	50.00	100.00

Amazon Ex. 1008.0082

SQL WHERE Clause Options

SQL includes a number of SQL WHERE clause options that greatly expand SQL's power and utility. In this section, we consider three options: compound clauses, ranges, and wildcards.

Compound WHERE Clauses

SQL WHERE clauses can include multiple conditions by using the SQL AND, OR, IN, and NOT IN operators. For example, to find all of the rows in SKU_DATA that have a Department named Water Sports and a Buyer named Nancy Meyers, we can use the **SQL AND operator** in our query code:

```
SELECT      *
FROM        SKU_DATA
WHERE       Department='Water Sports'
     AND    Buyer='Nancy Meyers';
```

The results of this query are:

	SKU	SKU_Description	Department	Buyer
1	101100	Dive Mask, Small Clear	Water Sports	Nancy Meyers
2	101200	Dive Mask, Med Clear	Water Sports	Nancy Meyers

Similarly, to find all of the rows of SKU_DATA for either the Camping or Climbing departments, we can use the **SQL OR operator** in the SQL query:

```
SELECT      *
FROM        SKU_DATA
WHERE       Department='Camping'
     OR     Department='Climbing';
```

which gives us the following results:

	SKU	SKU_Description	Department	Buyer
1	201000	Half-dome Tent	Camping	Cindy Lo
2	202000	Half-dome Tent Footprint	Camping	Cindy Lo
3	301000	Light Fly Climbing Harness	Climbing	Jerry Martin
4	302000	Locking carabiner, Oval	Climbing	Jerry Martin

Three or more AND and OR conditions can be combined, but in such cases the **SQL IN operator** and the **SQL NOT IN operator** are easier to use. For example, suppose we want to obtain all of the rows in SKU_DATA for buyers Nancy Meyers, Cindy Lo, and Jerry Martin. We could construct a WHERE clause with two ANDs, but an easier way to do this is to use the IN operator, as illustrated in the SQL query:

```
SELECT      *
FROM        SKU_DATA
WHERE       Buyer IN ('Nancy Meyers', 'Cindy Lo', 'Jerry Martin');
```

Amazon Ex. 1008.0083

In this format, a set of values is enclosed in parentheses. A row is selected if Buyer is equal to any one of the values provided. The result is:

	SKU	SKU_Description	Department	Buyer
1	101100	Dive Mask, Small Clear	Water Sports	Nancy Meyers
2	101200	Dive Mask, Med Clear	Water Sports	Nancy Meyers
3	201000	Half-dome Tent	Camping	Cindy Lo
4	202000	Half-dome Tent Footprint	Camping	Cindy Lo
5	301000	Light Fly Climbing Harness	Climbing	Jerry Martin
6	302000	Locking carabiner, Oval	Climbing	Jerry Martin

Similarly, if we want to find rows of SKU_DATA for which the buyer is someone *other* than Nancy Meyers, Cindy Lo, or Jerry Martin, we would use the SQL query:

```
SELECT      *
FROM        SKU_DATA
WHERE       Buyer NOT IN ('Nancy Meyers', 'Cindy Lo', 'Jerry Martin');
```

The result is:

	SKU	SKU_Description	Department	Buyer
1	100100	Std. Scuba Tank, Yellow	Water Sports	Pete Hansen
2	100200	Std. Scuba Tank, Magenta	Water Sports	Pete Hansen

Observe an important difference between IN and NOT IN. A row qualifies for an IN condition if the column is equal to *any* of the values in the parentheses. However, a row qualifies for a NOT IN condition if it is not equal to *all* of the items in the parentheses.

Ranges in SQL WHERE Clauses

SQL WHERE clauses can specify ranges of data values by using the **SQL BETWEEN keyword**. For example, the following SQL statement:

```
SELECT      *
FROM        ORDER_ITEM
WHERE       ExtendedPrice BETWEEN 100 AND 200;
```

will produce the following results:

	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	2000	101100	4	50.00	200.00
2	3000	101100	2	50.00	100.00
3	2000	101200	2	50.00	100.00
4	1000	202000	1	130.00	130.00

Notice that both the ends of the range, 100 and 200, are included in the resulting table. The preceding SQL statement is equivalent to the SQL query:

```
SELECT      *
FROM        ORDER_ITEM
WHERE       ExtendedPrice >=100
     AND    ExtendedPrice >=200;
```

Amazon Ex. 1008.0084

And which, of course, produces identical results:

	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	2000	101100	4	50.00	200.00
2	3000	101100	2	50.00	100.00
3	2000	101200	2	50.00	100.00
4	1000	202000	1	130.00	130.00

Wildcards in SQL WHERE Clauses

The **SQL LIKE keyword** can be used in SQL WHERE clauses to specify matches on portions of column values. For example, suppose we want to find the rows in the SKU_DATA table for all buyers whose first name is *Pete*. To find such rows, we use the SQL keyword LIKE with the **SQL percent sign (%) wildcard character**, as shown in the SQL query:

```
SELECT      *
FROM        SKU_DATA
WHERE       Buyer LIKE 'Pete%';
```

When used as an SQL wildcard character, the percent symbol (%) stands for any sequence of characters. When used with the SQL LIKE keyword, the character string 'Pete%' means any sequence of characters that start with the letters *Pete*. The result of this query is:

	SKU	SKU_Description	Department	Buyer
1	100100	Std. Scuba Tank, Yellow	Water Sports	Pete Hansen
2	100200	Std. Scuba Tank, Magenta	Water Sports	Pete Hansen



Does Not Work with MS Access ANSI-89 SQL

Microsoft Access ANSI-89 SQL uses wildcards, but not the SQL-92 standard wildcards. Microsoft Access uses the **Access asterisk (*) wildcard character** instead of a percent sign to represent multiple characters.

Solution: Use the Microsoft Access asterisk (*) wildcard in place of the SQL-92 percent sign (%) wildcard in Microsoft Access ANSI-89 SQL statements. Thus, the preceding SQL query would be written as follows for Microsoft Access:

```
SELECT      *
FROM        SKU_DATA
WHERE       Buyer LIKE 'Pete*';
```

Suppose we want to find the rows in SKU_DATA for which the SKU_Description includes the word *Tent* somewhere in the description. Because the word *Tent* could be at the front, the end, or in the middle, we need to place a wildcard on both ends of the LIKE phrase, as follows:

```
SELECT      *
FROM        SKU_DATA
WHERE       SKU_Description LIKE '%Tent%';
```

This query will find rows in which the word *Tent* occurs in any place in the SKU_Description. The result is:

	SKU	SKU_Description	Department	Buyer
1	201000	Half-dome Tent	Camping	Cindy Lo
2	202000	Half-dome Tent Footprint	Camping	Cindy Lo

Amazon Ex. 1008.0085

Sometimes we need to search for a particular value in a particular location in the column. For example, assume SKU values are coded such that a 2 in the third position from the right has some particular significance, maybe it means that the product is a variation of another product. For whatever reason, assume that we need to find all SKUs that have a 2 in the third column from the right. Suppose we try the SQL query:

```
SELECT      *
FROM        SKU_DATA
WHERE       SKU LIKE '%2%';
```

The result is

	SKU	SKU_Description	Department	Buyer
1	100200	Std. Scuba Tank, Magenta	Water Sports	Pete Hansen
2	101200	Dive Mask, Med Clear	Water Sports	Nancy Meyers
3	201000	Half-dome Tent	Camping	Cindy Lo
4	202000	Half-dome Tent Footprint	Camping	Cindy Lo
5	302000	Locking carabiner, Oval	Climbing	Jerry Martin

This is *not* what we wanted. We mistakenly retrieved all rows that had a 2 in *any* position in the value of SKU. To find the products we want, we cannot use the SQL wildcard character %. Instead, we must use the **SQL underscore (_) wildcard character**, which represents a single, unspecified character in a specific position. The following SQL statement will find all SKU_DATA rows with a value of 2 in the third position from the right:

```
SELECT      *
FROM        SKU_DATA
WHERE       SKU LIKE '%2__';
```

Observe that there are *two* underscores in this SQL query— one for the first position on the right and another for the second position on the right. This query gives us the result that we want:

	SKU	SKU_Description	Department	Buyer
1	100200	Std. Scuba Tank, Magenta	Water Sports	Pete Hansen
2	101200	Dive Mask, Med Clear	Water Sports	Nancy Meyers



Does Not Work with MS Access ANSI-89 SQL

Microsoft Access ANSI-89 SQL uses wildcards, but not the SQL-92 standard wildcards. Microsoft Access uses the **Access question mark (?) wildcard character** instead of an underscore (_) to represent a single character.

Solution: Use the Microsoft Access question mark (?) wildcard in place of the SQL-92 underscore (_) wildcard in Microsoft Access ANSI-89 SQL statements. Thus, the preceding SQL query would be written as follows for Microsoft Access:

```
SELECT      *
FROM        SKU_DATA
WHERE       SKU LIKE '*2??';
```

Furthermore, Microsoft Access can sometimes be fussy about stored trailing spaces in a text field. You may have problems with a WHERE clause like this:

```
WHERE       SKU LIKE '10?200';
```

Solution: Use a trailing asterisk (*), which allows for the trailing spaces:

```
WHERE       SKU LIKE '10?200*';
```

Amazon Ex. 1008.0086

BY THE WAY The SQL wildcard percent sign (%) and underscore (_) characters are specified in the SQL-92 standard. They are accepted by all DBMS products *except Microsoft Access*. So, why does Microsoft Access use the asterisk (*) character instead of the percent sign (%) and the question mark (?) instead of the underscore? This difference probably exists because the designers of Access chose to use the same wildcard characters that Microsoft was already using in the Microsoft MS-DOS operating system.

Combing the SQL WHERE Clause and the SQL ORDER BY Clause

If we want to sort the results generated by these enhanced SQL WHERE clauses, we simply combine the SQL ORDER BY clause with the WHERE clause. This is illustrated by the following SQL query:

```
SELECT      *
FROM        ORDER_ITEM
WHERE       ExtendedPrice BETWEEN 100 AND 200
ORDER BY    OrderNumber DESC;
```

which will produce the following result:

	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	3000	101100	2	50.00	100.00
2	2000	101200	2	50.00	100.00
3	2000	101100	4	50.00	200.00
4	1000	202000	1	130.00	130.00

Performing Calculations in SQL Queries

It is possible to perform certain types of calculations in SQL query statements. One group of calculations involves the use of SQL built-in functions. Another group involves simple arithmetic operations on the columns in the SELECT statement. We will consider each in turn.

Using SQL Built-in Functions

There are five **SQL built-in functions** for performing arithmetic on table columns: **SUM**, **AVG**, **MIN**, **MAX**, and **COUNT**. Some DBMS products extend these standard built-in functions by providing additional functions. Here, we will focus only on the five standard SQL built-in functions.

Suppose we want to know the sum of OrderTotal for all of the orders in RETAIL_ORDER. We can obtain that sum by using the following SQL query:

```
SELECT      SUM(OrderTotal)
FROM        RETAIL_ORDER;
```

The result will be:

	(No column name)
1	1235.00

Recall that the result of an SQL statement is always a table. In this case, the table has one cell (the intersection of one row and one column that contains the sum of OrderTotal). But because the OrderTotal sum is not a column in a table, the DBMS has no column name to provide.

Amazon Ex. 1008.0087

The preceding result was produced by Microsoft SQL Server 2008, and it names the column '(No column name)'. Other DBMS products take other, equivalent actions.

This result is ugly. We would prefer to have a meaningful column name, and SQL allows us to assign one using the **SQL AS keyword**. If we use the AS keyword in the query as follow:

```
SELECT      SUM(OrderTotal) AS OrderSum
FROM        RETAIL_ORDER;
```

The result of this modified query will be:

	OrderSum
1	1235.00

This result has a much more meaningful column label. The name *OrderSum* is arbitrary—we are free to pick any name that we think would be meaningful to the user of the result. We could pick *OrderTotal_Total*, *OrderTotalSum*, or any other label that we think would be useful.

The utility of the built-in functions increases when you use them with an SQL WHERE clause. For example, we can write the SQL query:

```
SELECT      SUM(ExtendedPrice) AS Order3000Sum
FROM        ORDER_ITEM
WHERE       OrderNumber=3000;
```

The result of this query is:

	Order3000Sum
1	450.00

The SQL built-in functions can be mixed and matched in a single statement. For example, we can create the following SQL statement:

```
SELECT      SUM(ExtendedPrice) AS OrderItemSum,
            AVG(ExtendedPrice) AS OrderItemAvg,
            MIN(ExtendedPrice) AS OrderItemMin,
            MAX(ExtendedPrice) AS OrderItemMax
FROM        ORDER_ITEM;
```

The result of this query is:

	OrderItemSum	OrderItemAvg	OrderItemMin	OrderItemMax
1	1180.00	168.5714	50.00	300.00

The SQL built-in COUNT function sounds similar to the SUM function, but it produces very different results. The COUNT function *counts* the number of rows, whereas the SUM function *adds* the values in a column. If we write the SQL statement:

```
SELECT      COUNT(*) AS NumberOfRows
FROM        ORDER_ITEM;
```

The result of this query is:

	NumberOfRows
1	7

Amazon Ex. 1008.0088

This result indicates that there are seven rows in the ORDER_ITEM table. Notice that we need to provide an asterisk (*) after the COUNT function when we want to count rows. COUNT is the only built-in function that requires an asterisk. The COUNT function is also unique because it can be used on any type of data, but the SUM, AVG, MIN, and MAX functions can only be used with numeric data.

The COUNT function can produce some surprising results. For example, suppose you want to count the number of departments in the SKU_DATA table. If we use the following query:

```
SELECT      COUNT(Department) AS DeptCount
FROM        SKU_DATA;
```

The result is:

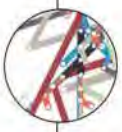
	DeptCount
1	8

which is the number of rows in the SKU_DATA table, *not* the number of unique values of Department, as shown in Figure 2-4. If we want to count the unique values of Department, we need to use the SQL DISTINCT keyword, as follows:

```
SELECT      COUNT(DISTINCT Department) AS DeptCount
FROM        SKU_DATA;
```

The result of this query is:

	DeptCount
1	3



Does Not Work with MS Access ANSI-89 SQL

Microsoft Access does not support the DISTINCT keyword as part of the COUNT expression, so while the SQL command with COUNT(Department) will work, the SQL command with COUNT(DISTINCT Department) will fail.

Solution: Use an SQL subquery structure (discussed later in this chapter) with the DISTINCT keyword in the subquery itself. This SQL query works:

```
SELECT    COUNT(*) AS DeptCount
FROM      (SELECT DISTINCT Department
           FROM SKU_DATA) AS DEPT;
```

Note that this query is a bit different from the other SQL queries using subqueries we show in this text because this subquery is in the FROM clause instead of (as you'll see) the WHERE clause. Basically, this subquery builds a new temporary table named DEPT containing only distinct Department values, and the query counts the number of those values.

You should be aware of two limitations to SQL built-in functions. First, except for grouping (defined later), you *cannot* combine a table column name with an SQL built-in function. For example, what happens if we run the following SQL query?

```
SELECT      Department, COUNT(*)
FROM        SKU_DATA;
```

Amazon Ex. 1008.0089

The result in SQL Server 2008 is:

```
Msg 8120, Level 16, State 1, Line 1
Column 'SKU_DATA.Department' is invalid in the select list because it is not contained
in either an aggregate function or the GROUP BY clause.
```

This is the specific SQL Server 2008 error message. However, you will receive an equivalent message from Access, Oracle, DB2, or MySQL.

The second problem with the SQL built-in functions that you should understand is that you cannot use them in an SQL WHERE clause. Thus, you cannot use the following SQL statement:

```
SELECT      *
FROM        RETAIL_ORDER
WHERE       OrderTotal > AVG(OrderTotal);
```

An attempt to use such a statement will also result in an error statement from the DBMS:

```
Msg 147, Level 15, State 1, Line 3
An aggregate may not appear in the WHERE clause unless it is in a subquery contained
in a HAVING clause or a select list, and the column being aggregated is an outer reference.
```

Again, this is the specific SQL Server 2008 error message, but other DBMS products will give you an equivalent error message. In Chapter 7, you will learn how to obtain the desired result of the above query using a sequence of SQL views.

SQL Expressions in SQL SELECT Statements

It is possible to do basic arithmetic in SQL statements. For example, suppose we want to compute the values of extended price, perhaps because we want to verify the accuracy of the data in the ORDER_ITEM table. To compute the extended price, we can use the SQL expression Quantity * Price in the SQL query:

```
SELECT      Quantity * Price AS EP
FROM        ORDER_ITEM;
```

The result is:

	EP
1	300.00
2	200.00
3	100.00
4	100.00
5	50.00
6	300.00
7	130.00

An **SQL expression** is basically a formula or set of values that determines the exact results of an SQL Query. We can think of an SQL expression as anything that follows an actual or implied equal to (=) character (or any other relational operator, such as greater than (>), less than (<), and so on) or that follows certain SQL keywords such as LIKE and BETWEEN. Thus, the SELECT

Amazon Ex. 1008.0090

clause in the preceding query includes the implied equal to (=) sign as *EP = Quantity * Price*. For another example, in the WHERE clause:

```
WHERE       Buyer IN ('Nancy Meyers', 'Cindy Lo', 'Jerry Martin');
```

the SQL expression consists of the three text values following the IN keyword.

Now that we know how to calculate the value of extended price, we can compare this computed value to the stored value of ExtendedPrice by using the SQL query:

```
SELECT      Quantity * Price AS EP, ExtendedPrice
FROM        ORDER_ITEM;
```

The result of this statement now allows us to visually compare the two values to ensure that the stored data are correct:

	EP	ExtendedPrice
1	300.00	300.00
2	200.00	200.00
3	100.00	100.00
4	100.00	100.00
5	50.00	50.00
6	300.00	300.00
7	130.00	130.00

Another use for SQL expressions in SQL statements is to perform string manipulation. Suppose we want to combine the Buyer and Department columns into a single column named Sponsor. If we use the SQL statement:

```
SELECT      Buyer+' in '+Department AS Sponsor
FROM        SKU_DATA;
```

the result will include a column named Sponsor that contains the combined text values:

	Sponsor
1	Pete Hansen in Water Sports
2	Pete Hansen in Water Sports
3	Nancy Meyers in Water Sports
4	Nancy Meyers in Water Sports
5	Cindy Lo in Camping
6	Cindy Lo in Camping
7	Jerry Martin in Climbing
8	Jerry Martin in Climbing

This particular result is also ugly. We can eliminate the extra spaces by using more advanced functions. The syntax and use of such functions vary from one DBMS to another, however, and a discussion of the features of each product will take us away from the point of this discussion. To learn more, search on *string functions* in the documentation for your specific DBMS. Just to illustrate the possibilities, however, here is an SQL Server 2008 statement that strips the tailing blanks off the right-hand side of Buyer and Department:

```
SELECT      DISTINCT RTRIM(Buyer)+' in '+RTRIM(Department) AS Sponsor
FROM        SKU_DATA;
```

Amazon Ex. 1008.0091

The result of this query is much more visually pleasing:

	Sponsor
1	Cindy Lo in Camping
2	Jerry Martin in Climbing
3	Nancy Meyers in Water Sports
4	Pete Hansen in Water Sports

Grouping in SQL SELECT Statements

In SQL queries, rows can be grouped according to common values using the **SQL GROUP BY clause**. For example, if you specify GROUP BY Department in a SELECT statement on the SKU_DATA table, the DBMS will first sort all rows by Department and then combine all of the rows having the same value into a group for that department. A grouping will be formed for each unique value of Department. For example, when we use the GROUP BY clause in the SQL query:

```
SELECT      Department, COUNT(*) AS Dept_SKU_Count
FROM        SKU_DATA
GROUP BY    Department;
```

We get the result:

	Department	Dept_SKU_Count
1	Camping	2
2	Climbing	2
3	Water Sports	4

To obtain this result, the DBMS first sorts the rows according to Department and then counts the number of rows having the same value of Department.

Here is another example of an SQL query using GROUP BY:

```
SELECT      SKU, AVG(ExtendedPrice) AS AvgEP
FROM        ORDER_ITEM
GROUP BY    SKU;
```

The result for this query is:

	SKU	AvgEP
1	100200	300.00
2	101100	150.00
3	101200	75.00
4	201000	300.00
5	202000	130.00

Here the rows have been sorted and grouped by SKU and the average ExtendedPrice for each group of SKU items has been calculated.

We can include more than one column in a GROUP BY expression. For example, the SQL statement:

```
SELECT      Department, Buyer, COUNT(*) AS Dept_Buyer_SKU_Count
FROM        SKU_DATA
GROUP BY    Department, Buyer;
```

Amazon Ex. 1008.0092

groups rows according to the value of Department first, then according to Buyer, and then counts the number of rows for each combination of Department and Buyer. The result is:

	Department	Buyer	Dept_Buyer_SKU_Count
1	Camping	Cindy Lo	2
2	Climbing	Jerry Martin	2
3	Water Sports	Nancy Meyers	2
4	Water Sports	Pete Hansen	2

When using the GROUP BY clause, only the column or columns in the GROUP BY expression and the SQL built-in functions can be used in the expressions in the SELECT clause. The following expressions will result in an error:

```
SELECT      SKU, Department, COUNT(*) AS Dept_SKU_Count
FROM        SKU_DATA
GROUP BY    Department;
```

```
Msg 8120, Level 16, State 1, Line 1
Column 'SKU_DATA.SKU' is invalid in the select list because it is not contained
in either an aggregate function or the GROUP BY clause.
```

This is the specific SQL Server 2008 error message, but other DBMS products will give you an equivalent error message. Statements like this one are invalid because there are many values of SKU for each Department group. The DBMS has no place to put those multiple values in the result. If you do not understand the problem, try to process this statement by hand. It cannot be done.

Of course, the SQL WHERE and ORDER BY clauses can also be used with SELECT statements, as shown in the following query:

```
SELECT      Department, COUNT(*) AS Dept_SKU_Count
FROM        SKU_DATA
WHERE       SKU <>302000
GROUP BY    Department
ORDER BY    Dept_SKU_Count;
```

The result is:

	Department	Dept_SKU_Count
1	Climbing	1
2	Camping	2
3	Water Sports	4

Notice that one of the rows of the Climbing department has been removed from the count because it did not meet the WHERE clause condition. Without the ORDER BY clause, the rows would be presented in arbitrary order of Department. With it, the order is as shown. In general, to be safe, always place the WHERE clause before the GROUP BY clause. Some DBMS products do not require that placement, but others do.



Does Not Work with MS Access ANSI-89 SQL

Microsoft Access does not properly recognize the alias *Dept_SKU_Count* in the ORDER BY clause and creates a parameter query that requests an input value of as yet nonexistent Dept_SKU_Count! However, it doesn't matter whether you enter parameter values or not—click the OK button and the query will run. The results will be basically correct, but they will not be sorted correctly.

(*continued*)

Amazon Ex. 1008.0093

Solution: Use the Access QBE GUI to modify the query structure. The correct QBE structure is shown in Figure 2-21. The resulting Access ANSI-89 SQL is:

```
SELECT      SKU_DATA.Department, Count(*) AS Dept_SKU_Count
FROM        SKU_DATA
WHERE       (((SKU_DATA.SKU)<>302000))
GROUP BY    SKU_DATA.Department
ORDER BY    Count(*);
```

which can be edited down to:

```
SELECT      Department, Count(*) AS Dept_SKU_Count
FROM        SKU_DATA
WHERE       SKU<>302000
GROUP BY    Department
ORDER BY    Count(*);
```

SQL provides one more GROUP BY clause feature that extends its functionality even further. The **SQL HAVING clause** restricts the groups that are presented in the result. We can restrict the previous query to display only groups having more than one row by using the SQL query:

```
SELECT      Department, COUNT(*) AS Dept_SKU_Count
FROM        SKU_DATA
WHERE       SKU <>302000
GROUP BY    Department
HAVING      COUNT (*) > 1
ORDER BY    Dept_SKU_Count;
```

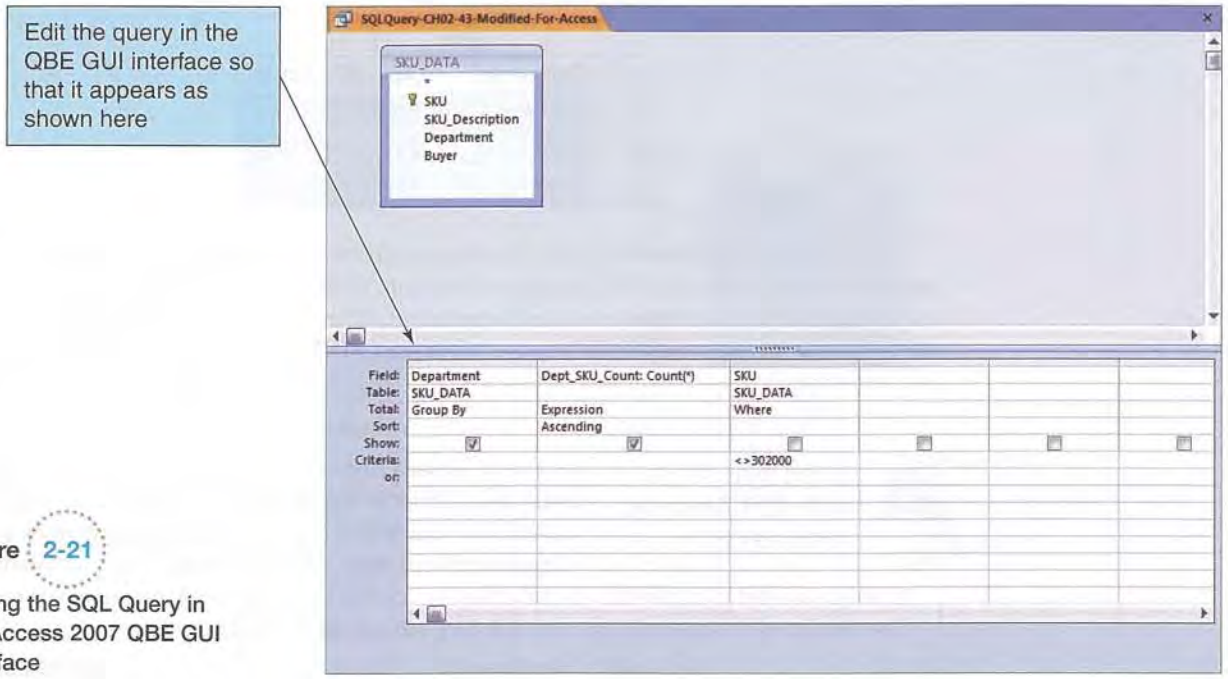


Figure 2-21

Editing the SQL Query in the Access 2007 QBE GUI Interface

Amazon Ex. 1008.0094

The result of this modified query is:

	Department	Dept_SKU_Count
1	Camping	2
2	Water Sports	4

Comparing this result with the previous one, the row for Climbing (which has a count of 1) has been eliminated.



Does Not Work with MS Access ANSI-89 SQL

This query fails in Microsoft Access ANSI-89 SQL for the same reason as the previous query.

Solution: See the solution described in the previous "Does Now Work with Microsoft Access ANSI-89 SQL" box. The correct Access ANSI-89 SQL for this query is:

```
SELECT      Department, Count(*) AS Dept_SKU_Count
FROM        SKU_DATA
WHERE       SKU<>302000
GROUP BY    Department
HAVING      Count(*)>1
ORDER BY    Count(*);
```

SQL built-in functions can be used in the HAVING clause. For example, the following is a valid SQL query:

```
SELECT      COUNT(*) AS SKU_Count, SUM(Price) AS TotalRev, SKU
FROM        ORDER_ITEM
GROUP BY    SKU
HAVING      SUM(Price)=100;
```

The results for this query are:

	SKU_Count	TotalRev	SKU
1	2	100.00	101100
2	2	100.00	101200

Be aware that there is an ambiguity in statements that include both WHERE and HAVING clauses. The results vary depending on whether the WHERE condition is applied before or after the HAVING. To eliminate this ambiguity, the WHERE clause is *always* applied before the HAVING clause.

Looking for Patterns in NASDAQ Trading

Before we continue our discussion of SQL, consider an example problem that will illustrate the power of the SQL just described.

Suppose that a friend tells you that she suspects the stock market tends to go up on certain days of the week and down on others. She asks you to investigate past trading data to determine if this is true. Specifically, she wants to trade an index fund called the NASDAQ 100, which is a stock fund of the 100 top companies traded on the NASDAQ stock exchange. She gives you a dataset with 20 years (1985–2004) of NASDAQ 100 trading data for analysis. Assume she gives you the data in the form of a table named NDX containing 4611 rows of data for use with a relational database (this dataset is available on the text's Web site at *www.pearsonhighered.com/kroenke*).

Amazon Ex. 1008.0095

Investigating the Characteristics of the Data

Suppose you first decide to investigate the general characteristics of the data. You begin by seeing what columns are present in the table by issuing the SQL query:

```
SELECT      *
FROM        NDX;
```

The first five rows of that query are as follows:

	TClose	PriorClose	ChangeClose	Volume	TMonth	TDayOfMonth	TYear	TDayOfWeek	TQuarter
1	1520.46	1530.65	-10.1900000000001	24827600	January	9	2004	Friday	1
2	1530.65	1514.26	16.3900000000001	26839500	January	8	2004	Thursday	1
3	1514.26	1501.26	13	22942800	January	7	2004	Wednesday	1
4	1501.26	1496.58	4.68000000000006	22732200	January	6	2004	Tuesday	1
5	1496.58	1463.57	33.01	23629100	January	5	2004	Monday	1

Assume that you learn that the first column has the value of the fund at the close of a trading day, the second column has the value of the fund at the close of the prior trading day, and the third row has the difference between the current day's close and the prior day's close. Volume is the number of shares traded, and the rest of the data concerns the trading date.

Next, you decide to investigate the change of the stock price by issuing the SQL query:

```
SELECT      AVG(ChangeClose) AS AverageChange,
            MAX(ChangeClose) AS MaxGain,
            MIN(ChangeClose) AS MaxLoss
FROM        NDX;
```

The result of this query is:

	AverageChange	MaxGain	MaxLoss
1	0.281167028199584	399.6	-401.03

BY THE WAY DBMS products have many functions for formatting query results to reduce the number of decimal points displayed, to add currency characters such as $ or £, or to make other formatting changes. However, these functions are DBMS-dependent. Search the documentation of your DBMS for the term *formatting results* to learn more about such functions.

Just out of curiosity, you decide to determine which days had the maximum and minimum change. To avoid having to key in the long string of decimal places that would be required to make an equal comparison, you use a greater than and less than comparison with values that are close:

```
SELECT      ChangeClose, TMonth, TDayOfMonth, TYear
FROM        NDX
WHERE       ChangeClose > 398
      OR    ChangeClose < -400;
```

Amazon Ex. 1008.0096

The result is:

	ChangeClose	TMonth	TDayOfMonth	TYear
1	-401.03	January	3	1994
2	399.6	January	3	2001

This result is surprising! Is there some reason that both the greatest loss and the greatest gain both occurred on January 3? You begin to wonder if your friend might have a promising idea.

Searching for Patterns in Trading by Day of Week

You want to determine if there is a difference in the average trade by day of week. Accordingly, you create the SQL query:

```
SELECT      TDayOfWeek, AVG(ChangeClose) AS AvgChange
FROM        NDX
GROUP BY    TDayOfWeek;
```

The result is:

	TDayOfWeek	AvgChange
1	Wednesday	0.777940552017005
2	Monday	-1.03577929465299
3	Friday	0.146021739130452
4	Thursday	2.17412972972975
5	Tuesday	-0.711440677966085

Indeed, there does seem to be a difference according to the day of the week. The NASDAQ 100 appears to go down on Monday and Tuesday and then go up on the other three days of the week. Thursday, in particular, seems to be a good day to trade long.

But, you begin to wonder, is this pattern true for each year? To answer that question, you use the query:

```
SELECT      TDayOfWeek, TYear, AVG(ChangeClose) AS AvgChange
FROM        NDX
GROUP BY    TDayOfWeek, TYear
ORDER BY    TDayOfWeek, TYear DESC;
```

Because there are 20 years of data, this query results in 100 rows, of which the first 12 are shown in the following results:

	TDayOfWeek	TYear	AvgChange
1	Friday	2004	-7.2700000000001
2	Friday	2003	-2.48499999999996
3	Friday	2002	-2.19419999999997
4	Friday	2001	-19.5944
5	Friday	2000	8.8980392156863
6	Friday	1999	13.9656000000001
7	Friday	1998	5.2640816326531
8	Friday	1997	-0.194799999999989
9	Friday	1996	0.819019607843153
10	Friday	1995	0.691372549019617
11	Friday	1994	0.123725490196082
12	Friday	1993	-0.899399999999989

Amazon Ex. 1008.0097

To simplify your analysis, you decide to restrict the number of rows to the most recent 5 years (2000–2004):

```
SELECT      TDayOfWeek, TYear, AVG(ChangeClose) AS AvgChange
FROM        NDX
WHERE       TYear > '1999'
GROUP BY    TDayOfWeek, TYear
ORDER BY    TDayOfWeek, TYear DESC;
```

Partial results from this query are as follows:

	TDayOfWeek	TYear	AvgChange
1	Friday	2004	-7.2700000000001
2	Friday	2003	-2.48499999999996
3	Friday	2002	-2.19419999999997
4	Friday	2001	-19.5944
5	Friday	2000	8.8980392156863
6	Monday	2004	33.01
7	Monday	2003	3.77416666666668
8	Monday	2002	-2.60229166666664
9	Monday	2001	-3.7527083333333
10	Monday	2000	-19.899574468085
11	Thursday	2004	16.3900000000001
12	Thursday	2003	5.70700000000002

Alas, it does not appear that day of week is a very good predictor of gain or loss. At least, not for this fund over this period of time. We could continue this discussion to further analyze this data, but by now you should understand how useful SQL can be for analyzing and processing a table. Suggested additional NDX analysis exercises are included in the SQL problems at the end of this chapter.

Querying Two or More Tables with SQL

So far in this chapter we've worked with only one table. Now we will conclude by describing SQL statements for querying two or more tables.

Suppose that you want to know the revenue generated by SKUs managed by the Water Sports department. We can compute revenue as the sum of ExtendedPrice, but we have a problem. ExtendedPrice is stored in the ORDER_ITEM table, and Department is stored in the SKU_DATA table. We need to process data in two tables, and all of the SQL presented so far operates on a single table at a time.

SQL provides two different techniques for querying data from multiple tables: subqueries and joins. Although both work with multiple tables, they are used for slightly different purposes, as you will learn.

Querying Multiple Tables with Subqueries

How can we obtain the sum of ExtendedPrice for items managed by the Water Sports department? If we somehow knew the SKU values for those items, we could use a WHERE clause with the IN keyword.

Amazon Ex. 1008.0098

For the data in Figure 2-4, the SKU values for items in Water Sports are 100100, 100200, 101100, and 101200. Knowing those values, we can obtain the sum of their ExtendedPrice with the following SQL query:

```
SELECT      SUM(ExtendedPrice) AS Revenue
FROM        ORDER_ITEM
WHERE       SKU IN (100100, 100200, 101100, 101200);
```

The result is:

	Revenue
1	750.00

But, in general, we do not know the necessary SKU values ahead of time. However, we do have a way to obtain them from an SQL query on the data in the SKU_DATA table. To obtain the SKU values for the Water Sports department, we use the SQL statement:

```
SELECT      SKU
FROM        SKU_DATA
WHERE       Department='Water Sports'
```

The result of this SQL statement is:

	SKU
1	100100
2	100200
3	101100
4	101200

which is, indeed, the desired list of SKU values.

Now we need only combine the last two SQL statements to obtain the result we want. We replace the list of values in the WHERE clause of the first SQL query with the second SQL statement as follows:

```
SELECT      SUM(ExtendedPrice) AS Revenue
FROM        ORDER_ITEM
WHERE       SKU IN
            (SELECT     SKU
            FROM        SKU_DATA
            WHERE       Department='Water Sports');
```

The result of the query is:

	Revenue
1	750.00

which is the same result we obtained before when we know the values of SKU to use.

In the preceding SQL query, the second SELECT statement, the one enclosed in parentheses, is called a **subquery**. We can use multiple subqueries to process three or even more tables. For example, suppose we want to know the name of the buyers who manage any product purchased in January 2004. First, note that Buyer data is stored in the SKU_DATA table and OrderMonth and OrderYear data are stored in the RETAIL_ORDER table.

Amazon Ex. 1008.0099

Now, we can use an SQL query with two subqueries to obtain the desired data as follows:

```
SELECT      Buyer
FROM        SKU_DATA
WHERE       SKU IN
            (SELECT      SKU
             FROM        ORDER_ITEM
             WHERE       OrderNumber  IN
                         (SELECT      OrderNumber
                          FROM        RETAIL_ORDER
                          WHERE       OrderMonth='January'
                               AND    OrderYear=2009));
```

The result of this statement is:

	Buyer
1	Pete Hansen
2	Nancy Meyers
3	Nancy Meyers

To understand this statement, work from the bottom up. The bottom SELECT statement obtains the list of OrderNumbers of orders sold in January 2009. The middle SELECT statement obtains the SKU values for items sold in orders in January 2009. Finally, the top-level SELECT query obtains Buyer for all of the SKUs found in the middle SELECT statement.

Any parts of the SQL language that you have learned earlier in this chapter can be applied to a table generated by a subquery, regardless of how complicated the SQL looks. For example, we can apply the DISTINCT keyword on the results to eliminate duplicate rows. Or, we can apply the GROUP BY and ORDER BY clauses as follows:

```
SELECT      Buyer, COUNT(*) AS NumberSold
FROM        SKU_DATA
WHERE       SKU IN
            (SELECT      SKU
             FROM        ORDER_ITEM
             WHERE       OrderNumber IN
                         (SELECT      OrderNumber
                          FROM        RETAIL_ORDER
                          WHERE       OrderMonth='January'
                               AND    OrderYear=2009))
GROUP BY    Buyer
ORDER BY    NumberSold DESC;
```

The result is:

	Buyer	NumberSold
1	Nancy Meyers	2
2	Pete Hansen	1

Amazon Ex. 1008.0100



Does Not Work with MS Access ANSI-89 SQL

This query fails in Microsoft Access ANSI-89 SQL for the same reason previously described on pages 65–66.

Solution: See the solution described in the "Does Now Work with Microsoft Access ANSI-89 SQL" box on pages 65–66. The correct Access ANSI-89 SQL statement for this query is:

```
SELECT      Buyer, Count(*) AS NumberSold
FROM        SKU_DATA
WHERE       SKU IN
            (SELECT   SKU
             FROM     ORDER_ITEM
             WHERE    OrderNumber IN
                      (SELECT      OrderNumber
                       FROM        RETAIL_ORDER
                       WHERE       OrderMonth='January'
                            AND    OrderYear=2009))
GROUP BY    Buyer
ORDER BY    Count(*) DESC;
```

Querying Multiple Tables with Joins

Subqueries are very powerful, but they do have a serious limitation. The selected data can only come from the top-level table. We cannot use a subquery to obtain data that arise from more than one table. To do so, we must use a join instead.

The **SQL join operator** is used to combine two or more tables by concatenating (sticking together) the rows of one table with the rows of another table. Consider how we might combine the data in the RETAIL_ORDER and ORDER_ITEM tables. We can concatenate the rows of one table with the rows of the second table with the following SQL statement:

```
SELECT      *
FROM        RETAIL_ORDER, ORDER_ITEM;
```

This statement will just stick every row of one table together with every row of the second table. For the data in Figure 2-4, the result is

	OrderNumber	StoreNumber	StoreZip	OrderMonth	OrderYear	OrderTotal	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	1000	10	98110	December	2008	445.00	3000	100200	1	300.00	300.00
2	1000	10	98110	December	2008	445.00	2000	101100	4	50.00	200.00
3	1000	10	98110	December	2008	445.00	3000	101100	2	50.00	100.00
4	1000	10	98110	December	2008	445.00	2000	101200	2	50.00	100.00
5	1000	10	98110	December	2008	445.00	3000	101200	1	50.00	50.00
6	1000	10	98110	December	2008	445.00	1000	201000	1	300.00	300.00
7	1000	10	98110	December	2008	445.00	1000	202000	1	130.00	130.00
8	2000	20	02335	December	2008	310.00	3000	100200	1	300.00	300.00
9	2000	20	02335	December	2008	310.00	2000	101100	4	50.00	200.00
10	2000	20	02335	December	2008	310.00	3000	101100	2	50.00	100.00
11	2000	20	02335	December	2008	310.00	2000	101200	2	50.00	100.00
12	2000	20	02335	December	2008	310.00	3000	101200	1	50.00	50.00
13	2000	20	02335	December	2008	310.00	1000	201000	1	300.00	300.00
14	2000	20	02335	December	2008	310.00	1000	202000	1	130.00	130.00
15	3000	10	98110	January	2009	480.00	3000	100200	1	300.00	300.00
16	3000	10	98110	January	2009	480.00	2000	101100	4	50.00	200.00
17	3000	10	98110	January	2009	480.00	3000	101100	2	50.00	100.00
18	3000	10	98110	January	2009	480.00	2000	101200	2	50.00	100.00
19	3000	10	98110	January	2009	480.00	3000	101200	1	50.00	50.00
20	3000	10	98110	January	2009	480.00	1000	201000	1	300.00	300.00
21	3000	10	98110	January	2009	480.00	1000	202000	1	130.00	130.00

Amazon Ex. 1008.0101

Because there are three rows of retail order and seven rows of order items, there are 3 times 7, or 21, rows in this table. Notice that the retail order with OrderNumber 1000 has been combined with all seven of the rows in ORDER_ITEM_GRADE, the retail order with OrderNumber has been combined with all seven of the same rows, and, finally, that the retail order with OrderNumber 3000 has again been combined with all seven rows.

This is illogical—what we need to do is to select only those rows for which the OrderNumber of RETAIL_ORDER matches the OrderNumber in ORDER_ITEM. This is easy to do: we simply add an SQL WHERE clause to the query:

```
SELECT      *
FROM        RETAIL_ORDER, ORDER_ITEM
WHERE       RETAIL_ORDER.OrderNumber=ODRER_ITEM.OrderNumber;
```

The result is:

	OrderNumber	StoreNumber	StoreZip	OrderMonth	OrderYear	OrderTotal	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	3000	10	98110	January	2009	480.00	3000	100200	1	300.00	300.00
2	2000	20	02335	December	2008	310.00	2000	101100	4	50.00	200.00
3	3000	10	98110	January	2009	480.00	3000	101100	2	50.00	100.00
4	2000	20	02335	December	2008	310.00	2000	101200	2	50.00	100.00
5	3000	10	98110	January	2009	480.00	3000	101200	1	50.00	50.00
6	1000	10	98110	December	2008	445.00	1000	201000	1	300.00	300.00
7	1000	10	98110	December	2008	445.00	1000	202000	1	130.00	130.00

This is technically correct, but it will be easier to read if we sort the results using an ORDER BY clause:

```
SELECT      *
FROM        RETAIL_ORDER, ORDER_ITEM
WHERE       RETAIL_ORDER.OrderNumber=ORDER_ITEM.OrderNumber
ORDER BY    RETAIL_ORDER.OrderNumber, ORDER_ITEM.SKU;
```

The result is:

	OrderNumber	StoreNumber	StoreZip	OrderMonth	OrderYear	OrderTotal	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	1000	10	98110	December	2008	445.00	1000	201000	1	300.00	300.00
2	1000	10	98110	December	2008	445.00	1000	202000	1	130.00	130.00
3	2000	20	02335	December	2008	310.00	2000	101100	4	50.00	200.00
4	2000	20	02335	December	2008	310.00	2000	101200	2	50.00	100.00
5	3000	10	98110	January	2009	480.00	3000	100200	1	300.00	300.00
6	3000	10	98110	January	2009	480.00	3000	101100	2	50.00	100.00
7	3000	10	98110	January	2009	480.00	3000	101200	1	50.00	50.00

If you compare this result with the data in Figure 2-4, you will see that only the appropriate order items are associated with each retail order. You also can tell that this has been done by noticing that in each row the value of OrderNumber from RETAIL_ORDER (the first column) equals the value of OrderNumber from ORDER_ITEM (the seventh column). This was not true for our first result.

You may have noticed that we introduced a new variation in SQL statement syntax in the previous two queries, where the terms RETAIL_ORDER.OrderNumber, ORDER_ITEM.OrderNumber, and ORDER_ITEM.SKU were used. The new syntax is simply **TableName.ColumnName**, and it is used to specify exactly which table each column is linked to. RETAIL_ORDER.OrderNumber simply means the OrderNumber from the RETAIL_ORDER table. Similarly, ORDER_ITEM.OrderNumber refers to the OrderNumber in the ORDER_ITEM table, and ORDER_ITEM.SKU refers to the SKU column in the ORDER_ITEM table. You can always qualify a column name with the name of its table like this. We have not done so previously because we were working with only one table, but the SQL statements shown previously would have worked just as well with syntax like SKU_DATA.Buyer rather than just Buyer or ORDER_ITEM.Price instead of Price.

Amazon Ex. 1008.0102

The table that is formed by concatenating two tables is called a **join**. The process of creating such a table is called **joining the two tables**, and the associated operation is called a **join operation**. When the tables are joined using an equal condition (like the one on OrderNumber), this join is called an **equijoin**. When people say *join*, 99.99999 percent of the time they mean an equijoin. This type of join is also referred to as an **inner join**.

We can use a join to obtain data from two or more tables. For example, using the data in Figure 2-4, suppose we want to show the name of the Buyer and the ExtendedPrice of the sales of all items managed by that Buyer. The following SQL query will obtain that result:

```
SELECT      Buyer, ExtendedPrice
FROM        SKU_DATA, ORDER_ITEM
WHERE       SKU_DATA.SKU=ORDER_ITEM.SKU;
```

The result is:

	Buyer	ExtendedPrice
1	Pete Hansen	300.00
2	Nancy Meyers	200.00
3	Nancy Meyers	100.00
4	Nancy Meyers	100.00
5	Nancy Meyers	50.00
6	Cindy Lo	300.00
7	Cindy Lo	130.00

Again, the result of every SQL statement is just a single table, so we can apply any of the SQL you learned for a single table to this result. For example, we can use the GROUP BY and ORDER BY clauses to obtain the total revenue associated with each buyer as shown in the following SQL query:

```
SELECT      Buyer, SUM(ExtendedPrice) AS BuyerRevenue
FROM        SKU_DATA, ORDER_ITEM
WHERE       SKU_DATA.SKU=ORDER_ITEM.SKU
GROUP BY    Buyer
ORDER BY    BuyerRevenue DESC;
```

The result is:

	Buyer	BuyerRevenue
1	Nancy Meyers	450.00
2	Cindy Lo	430.00
3	Pete Hansen	300.00



Does Not Work with MS Access ANSI-89 SQL

This query fails in Microsoft Access ANSI-89 SQL for the same reason previously described on pages 65–66.

Solution: See the solution described in the "Does Now Work with Microsoft Access ANSI-89 SQL" box on pages 65–66. The correct Access ANSI-89 SQL statement for this query is:

```
SELECT      Buyer, Sum(ORDER_ITEM.ExtendedPrice) AS BuyerRevenue
FROM        SKU_DATA, ORDER_ITEM
WHERE       SKU_DATA.SKU=ORDER_ITEM.SKU
GROUP BY    Buyer
ORDER BY    Sum(ExtendedPrice) DESC;
```

Amazon Ex. 1008.0103

We can extend this syntax to join three or more tables. For example, suppose we want to obtain the Buyer and the ExtendedPrice and OrderMonth for all purchases of items managed by each buyer. To retrieve that data, we need to join all three tables together as shown in this SQL query:

```
SELECT      Buyer, ExtendedPrice, OrderMonth
FROM        SKU_DATA, ORDER_ITEM, RETAIL_ORDER
WHERE       SKU_DATA.SKU=ORDER_ITEM.SKU
     AND    ORDER_ITEM.OrderNumber=RETAIL_ORDER.OrderNumber;
```

The result is:

	Buyer	ExtendedPrice	OrderMonth
1	Pete Hansen	300.00	January
2	Nancy Meyers	200.00	December
3	Nancy Meyers	100.00	January
4	Nancy Meyers	100.00	December
5	Nancy Meyers	50.00	January
6	Cindy Lo	300.00	December
7	Cindy Lo	130.00	December

We can improve this result by sorting with the ORDER BY clause and grouping by Buyer with the GROUP BY clause:

```
SELECT      Buyer, OrderMonth, SUM(ExtendedPrice) AS BuyerRevenue
FROM        SKU_DATA, ORDER_ITEM, RETAIL_ORDER
WHERE       SKU_DATA.SKU=ORDER_ITEM.SKU
     AND    ORDER_ITEM.OrderNumber=RETAIL_ORDER.OrderNumber
GROUP BY    Buyer, OrderMonth
ORDER BY    Buyer, OrderMonth DESC;
```

The result is:

	Buyer	OrderMonth	BuyerRevenue
1	Cindy Lo	December	430.00
2	Nancy Meyers	December	300.00
3	Nancy Meyers	January	150.00
4	Pete Hansen	January	300.00

Joins also can be written using another syntax, the **SQL JOIN . . . ON syntax**, and there is a bit more for you to learn about joins when values are missing, but this chapter is long enough. We will finish the discussion of joins in Chapter 7. If you just cannot wait, turn to pages 260–265 for the rest of the join story.

Comparing Subqueries and Joins

Subqueries and joins both process multiple tables, but they differ slightly. As mentioned earlier, a subquery can only be used to retrieve data from the top table. A join can be used to obtain data from any number of tables. Thus, a join can do everything a subquery can do, and more. So why learn subqueries? For one, if you just need data from a single table, you might use a subquery because it is easier to write and understand. This is especially true when processing multiple tables.

Amazon Ex. 1008.0104

In Chapter 8, however, you will learn about a type of subquery called a **correlated subquery**. A correlated subquery can do work that is not possible with joins. Thus, it is important for you to learn about both joins and subqueries, even though right now it appears that joins are uniformly superior. If you're curious, ambitious, and courageous, jump ahead and read the discussion of correlated subqueries on pages 297–301.

Summary

Wow! That was a full chapter!

Structured Query Language (SQL) was developed by IBM and has been endorsed by the ANSI SQL-92 and following standards. SQL is a data sublanguage that can be embedded into full programming languages or submitted directly to the DBMS. Knowing SQL is critical for knowledge workers, application programmers, and database administrators.

All DBMS products process SQL. Access hides SQL, but SQL Server, Oracle, DB2, and MySQL require that you use it

There are basically two kinds of SQL statements: DML and DDL statements. DML statements include statements for querying data and for inserting, updating, and deleting data. This chapter addresses only DML query statements.

The examples in this chapter are based on three tables extracted from the operational database at Cape Codd Outdoor Sports. Such database extracts are common and important. Sample data for the three tables is shown in Figure 2-4.

The basic structure of an SQL query statement is SELECT/FROM/WHERE. The columns to be selected are listed after SELECT, the table(s) to process is listed after FROM, and any restrictions on data values are listed after WHERE. In a WHERE clause, character and date data values must be enclosed in single quotes. Numeric data need not be enclosed in quotes. You can submit SQL statements directly to Access, SQL Server, Oracle, and MySQL, as described in this chapter.

This chapter explained the use of the following SQL clauses: SELECT, FROM, WHERE, ORDER BY, GROUP BY, and HAVING. This chapter explained the use of the following SQL keywords: DISTINCT, DESC, ASC, AND, OR, IN, NOT IN, BETWEEN, LIKE, % (* for Access), _ (? for Access), SUM, AVG, MIN, MAX, COUNT, AS. You should know how to mix and match these features to obtain the results you want. By default, the WHERE clause is applied before the HAVING clause.

You can query multiple tables using subqueries and joins. Subqueries are nested queries that use the SQL keywords IN and NOT IN. An SQL SELECT expression is placed inside parentheses. Using a subquery, you can display data from the top table only. A join is created by specifying multiple table names in the FROM clause. An SQL WHERE clause is used to obtain an equijoin. In most cases, equijoins are the most sensible option. Joins can display data from multiple tables. In Chapter 8, you will learn another type of subquery that can perform work that is not possible with joins.

Key Terms

ad-hoc queries
American National Standards Institute (ANSI)
Access asterisk (*) wildcard character
Access question mark (?) wildcard character
AVG
business intelligence (BI) systems
correlated subquery
COUNT
data definition language (DDL)
data manipulation language (DML)
data sublanguage
equijoin
Extensible Markup Language (XML)
graphical user interface (GUI)
inner join
International Organization for Standardization (ISO)
join
join operation
joining the two tables
MAX
MIN
MySQL query files
query by example (QBE)
SQL AND operator
SQL AS keyword
SQL BETWEEN keyword
SQL built-in functions
SQL DESC keyword
SQL DISTINCT keyword
SQL expression
SQL FROM clause
SQL GROUP BY clause
SQL HAVING clause
SQL IN operator
SQL join operat

,000
130,000

Amazon Ex. 1008.0105

SQL JOIN...ON syntax
SQL LIKE keyword
SQL NOT IN operator
SQL OR operator
SQL ORDER BY clause
SQL percent sign (%) wildcard character
SQL script file
SQL SELECT clause
SQL SELECT/FROM/WHERE framework
SQL Server Compatible Syntax (ANSI 92)
SQL underscore (_) wildcard character
SQL WHERE clause
stock-keeping unit (SKU)
Structured Query Language (SQL)
subquery
SUM
TableName.ColumnName syntax

Review Questions

2.1 What is a business intelligence (BI) system?

2.2 What is an ad-hoc query?

2.3 What does SQL stand for, and what is SQL?

2.4 What does SKU stand for? What is an SKU?

2.5 Summarize how data were altered and filtered in creating the Cape Codd data extraction.

2.6 Explain, in general terms, the relationships among the RETAIL_ORDER, ORDER_ITEM, and SKU_DATA tables.

2.7 Summarize the background of SQL.

2.8 What is SQL-92? How does it relate to the SQL statements in this chapter?

2.9 What features have been added to SQL in versions subsequent to the SQL-92?

2.10 Why is SQL described as a data sublanguage?

2.11 What does DML stand for? What are DML statements?

2.12 What does DDL stand for? What are DDL statements?

2.13 What is the SQL SELECT/FROM/WHERE framework?

2.14 Explain how Access uses SQL.

2.15 Explain how enterprise-class DBMS products use SQL.

The Cape Codd Outdoor Sports sale extraction database has been modified to include two additional tables, the INVENTORY table and the WAREHOUSE table. The table schemas for these tables, together with the SKU table, are as follows:

SKU_DATA (SKU, SKU_Description, Department, Buyer)
INVENTORY (*SKU*, *Warehouse*, SKU_Description, QuantityOnHand, QuantityOnOrder)
WAREHOUSE (Warehouse, Manager, Squarefeet)

The column characteristics for the WAREHOUSE table are shown in Figure 2-22, and the column characteristics for the INVENTORY table are shown in Figure 2-23. The data for the WAREHOUSE table are shown in Figure 2-24, and the data for the INVENTORY table is shown in Figure 2-25.

If at all possible, you should run your SQL solutions to the following questions against an actual database. A Microsoft Access database named Cape-Codd.accdb is available on our Web site (*www.pearsonhighered.com/kroenke*) that contains all the tables and data for the Cape Codd Outdoor Sports sales data extract database.

Amazon Ex. 1008.0106

WAREHOUSE

Column Name	Type	Key	Required	Remarks
Warehouse	Text (30)	Primary Key	Yes	
Manager	Text (25)	No	No	
SquareFeet	Integer	No	No	

Figure 2-22
Column Characteristics for the WAREHOUSE Table

Also available on our Web site are SQL scripts for creating and populating the tables for the Cape Codd database in SQL Server, Oracle, and MySQL.

2.16 There is an intentional flaw in the design of the INVENTORY table used in these exercises. This flaw was purposely included in the INVENTORY tables so that you can answer some of the following questions using only that table. Compare the SKU and INVENTORY tables, and determine what design flaw is included in INVENTORY. Specifically, why did we include it?

Use *only* the INVENTORY table to answer Review Questions 2.17 through 2.46:

2.17 Write an SQL statement to display SKU and SKU_Description.

2.18 Write an SQL statement to display SKU_Description and SKU.

2.19 Write an SQL statement to display Warehouse.

2.20 Write an SQL statement to display Warehouse with no duplications.

2.21 Write an SQL statement to display all of the columns without using *.

Figure 2-23
Column Characteristics for the INVENTORY Table

WAREHOUSE

Column Name	Type	Key	Required	Remarks
SKU	Integer	Primary Key, Foreign Key	Yes	Surrogate Key
Warehouse	Text (30)	Primary Key, Foreign Key	Yes	
SKU_Description	Text (35)	No	Yes	
QuantityOnHand	Integer	No	No	
QuantityOnOrder	Integer	No	No	

Warehouse	Manager	SquareFeet
Atlanta	Jones	125,000
Chicago	Smith	100,000
New Jersey	Evans	150,000
Seattle	Rogers	130,000

Figure 2-24
Cape Codd Outdoor Sports WAREHOUSE Data

Amazon Ex. 1008.0107

SKU	Warehouse	SKU_Description	QuantityOnHand	QuantityOnOrder
100100	Atlanta	Std. Scuba Tank, Yellow	250	0
100100	Chicago	Std. Scuba Tank, Yellow	100	50
100100	New Jersey	Std. Scuba Tank, Yellow	100	0
100100	Seattle	Std. Scuba Tank, Yellow	200	0
100200	Atlanta	Std. Scuba Tank, Magenta	200	30
100200	Chicago	Std. Scuba Tank, Magenta	75	75
100200	New Jersey	Std. Scuba Tank, Magenta	100	100
100200	Seattle	Std. Scuba Tank, Magenta	250	0
101100	Atlanta	Dive Mask, Small Clear	0	500
101100	Chicago	Dive Mask, Small Clear	0	500
101100	New Jersey	Dive Mask, Small Clear	300	200
101100	Seattle	Dive Mask, Small Clear	450	0
101200	Atlanta	Dive Mask, Med Clear	100	500
101200	Chicago	Dive Mask, Med Clear	50	500
101200	New Jersey	Dive Mask, Med Clear	475	0
101200	Seattle	Dive Mask, Med Clear	250	250
201000	Atlanta	Half-Dome Tent	2	100
201000	Chicago	Half-Dome Tent	10	250
201000	New Jersey	Half-Dome Tent	250	0
201000	Seattle	Half-Dome Tent	0	250
202000	Atlanta	Half-Dome Tent Footprint	10	250
202000	Chicago	Half-Dome Tent Footprint	1	250
202000	New Jersey	Half-Dome Tent Footprint	100	0
202000	Seattle	Half-Dome Tent Footprint	0	200
301000	Atlanta	Light Fly Climbing Harness	300	250
301000	Chicago	Light Fly Climbing Harness	250	250
301000	New Jersey	Light Fly Climbing Harness	0	250
301000	Seattle	Light Fly Climbing Harness	0	250
302000	Atlanta	Locking Carabiner	1000	0
302000	Chicago	Locking Carabiner	1250	0
302000	New Jersey	Locking Carabiner	500	500
302000	Seattle	Locking Carabiner	0	1000

Figure 2-25

Cape Codd Outdoor Sports
INVENTORY Data

Amazon Ex. 1008.0108

2.22 Write an SQL statement to display all of the columns using *.

2.23 Write an SQL statement to display all data on products having a QuantityOnHand greater than 0.

2.24 Write an SQL statement to display the SKU and SKU_Description for products having QuantityOnHand equal to 0.

2.25 Write an SQL statement to display the SKU, SKU_Description, and Warehouse for products having QuantityOnHand equal to 0. Sort the results in ascending order by Warehouse.

2.26 Write an SQL statement to display the SKU, SKU_Description, and Warehouse on products having QuantityOnHand equal to 0. Sort the results in descending order by Warehouse and in ascending order by SKU.

2.27 Write an SQL statement to display the SKU, SKU_Description, and Warehouse on products having QuantityOnHand equal to 0. Sort the resuts in descending order by Warehouse and in ascending order by SKU.

2.28 Write an SQL statement to display SKU and SKU_Description for all products that have a QuantityOnHand equal to 0 or a QuantityOnOrder equal to 0.

2.29 Write an SQL statement to display the SKU and SKU_Description of all items stored in the Seattle, Chicago, or New Jersey warehouse. Do not use the IN keyword.

2.30 Write an SQL statement to display the SKU and SKU_Description of all items stored in the Seattle, Chicago, or New Jersey warehouse. Use the IN keyword.

2.31 Write an SQL statement to display the SKU and SKU_Description of all items not stored in the Seattle, Chicago, or New Jersey warehouse. Do not use the NOT IN keyword.

2.32 Write an SQL statement to display the SKU and SKU_Description of all items not stored in the Seattle, Chicago, or New Jersey warehouse. Use the NOT IN keyword.

2.33 Write an SQL statement to display the SKU, SKU_Description, and QuantityOnHand for all products having a QuantityOnHand greater than 1 and less than 10. Do not use the BETWEEN keyword.

2.34 Write an SQL statement to display the SKU, SKU_Description, and QuantityOnHand for all products having a QuantityOnHand greater than 1 and less than 10. Use the BETWEEN keyword.

2.35 Write an SQL statement to show SKU and SKU_Description for all products having an SKU description starting with 'Half-dome'.

2.36 Write an SQL statement to show SKU and SKU_Description for all products having a description that includes the word 'Foot'.

2.37 Write an SQL statement to show SKU and SKU_Description for all products having a description that includes the word 'Foot'.

2.38 Write an SQL statement that uses all of the SQL built-in functions on the QuantityOnHand column. Include meaningful column names in the result.

2.39 Explain the difference between the SQL built-in functions COUNT and SUM.

2.40 Write an SQL statement to produce a single column called ItemLocation that combines the SKU_Description, the phrase "is located in," and Warehouse for all products that have a QuantityOnHand greater than 0. Do not be concerned with removing leading or trailing blanks.

2.41 Write an SQL statement to display the Warehouse and a count of QuantityOnHand, grouped by Warehouse. Name the count TotalItemsOnHand and display the results in descending order of TotalItemsOnHand.

2.42 Write an SQL statement to display the Warehouse and a count of QuantityOnHand, grouped by Warehouse. Omit all items that have a count greater than 2. Name the count TotalItemsOnHand and display the results in descending order of TotalItemsOnHand.

Amazon Ex. 1008.0109

2.43 Write an SQL statement to display the Warehouse and a count of QuantityOnHand grouped by Warehouse. Omit all items that have a count greater than 2. Show only groups having fewer than 2 item counts. Name the count TotalItemsOnHand and display the results in descending order of TotalItemsOnHand.

2.44 In your answer to Review Question 2.44, was the WHERE clause or the HAVING clause applied first? Why?

2.45 Write an SQL statement to display the Warehouse, the sum of QuantityOnOrder and sum of QuantityOnHand, grouped by Warehouse and QuantityOnOrder. Omit all items that have a count greater than 2. Name the count of QuantityOnOrder as TotalItemsOnOrder and the count of QuantityOnHand as TotalItemsOnHand.

Use *both* the INVENTORY and WAREHOUSE tables to answer Review Questions 2.46 through 2.52:

2.46 Write an SQL statement to show the SKU and SKU_Description for all items stored in a warehouse managed by 'Smith'. Use a subquery.

2.47 Write an SQL statement to show the SKU and SKU_Description for all items stored in a warehouse managed by 'Smith'. Use a join.

2.48 Write an SQL statement to show the Warehouse and average QuantityOnHand of all items stored in a warehouse managed by 'Smith'. Use a subquery.

2.49 Write an SQL statement to show the Warehouse and average QuantityOnHand of all items stored in a warehouse managed by 'Smith'. Use a join.

2.50 Write an SQL statement to show the Warehouse, Manager, and QuantityOnHand of all items stored in a warehouse managed by 'Smith'. Use a join.

2.51 Explain why you cannot use a subquery in your answer to Review Question 2.51.

2.52 Explain how subqueries and joins differ.

Project Questions

For this set of project questions, we will continue creating a Microsoft Access database for the Wedgewood Pacific Corporation (WPC). Founded in 1957 in Seattle, Washington, WPC has grown into an internationally recognized organization. The company is located in two buildings. One building houses the Administration, Accounting, Finance, and Human Resources departments, and the second houses the Production, Marketing, and Information Systems departments. The company database contains data about company employees, departments, company projects, company assets such as computer equipment, and other aspects of company operations.

In the following project questions, we have already created the WPC.accdb database with the following two tables:

DEPARTMENT (DepartmentName, BudgetCode, OfficeNumber, Phone)

EMPLOYEE (EmployeeNumber, FirstName, LastName, *Department*, Phone, Email)

Now we will add in the following two tables:

PROJECT (ProjectID, Name, *Department*, MaxHours, StartDate, EndDate)

ASSIGNMENT (*ProjectID*, *EmployeeNumber*, HoursWorked)

Amazon Ex. 1008.0110

Column Name	Type	Key	Required	Remarks
ProjectID	Number	Primary Key	Yes	Long Integer
Name	Text (50)	No	Yes	
Department	Text (35)	Foreign Key	Yes	
MaxHours	Number	No	Yes	Double
StartDate	Date/Time	No	No	
EndDate	Date/Time	No	No	

Figure 2-26

Database Column Characteristics for the PROJECT Table

2.53 Figure 2-26 shows the column characteristics for the WPC PROJECT table. Using the column characteristics, create the PROJECT table in the WPC.accdb database.

2.54 Create the relationship and referential integrity constraint between PROJECT and DEPARTMENT. Enable enforcing of referential integrity and cascading of data updates, but do *not* enable cascading of data from deleted records.

2.55 Figure 2-27 shows the data for the WPC PROJECT table. Using the Datasheet view, enter the data shown in Figure 2-27 into your PROJECT table.

2.56 Figure 2-28 shows the column characteristics for the WPC ASSIGNMENT table. Using the column characteristics, create the ASSIGNMENT table in the WPC.accdb database.

2.57 Create the relationship and referential integrity constraint between ASSIGNMENT and EMPLOYEE. Enable enforcing of referential integrity, but do *not* enable either cascading updates or the cascading of data from deleted records.

2.58 Create the relationship and referential integrity constraint between ASSIGNMENT and PROJECT. Enable enforcing of referential integrity and cascading of deletes, but do *not* enable cascading updates.

2.59 Figure 2-29 shows the data for the WPC ASSIGNMENT table. Using the Datasheet view, enter the data shown in Figure 2-29 into your ASSIGNMENT table.

2.60 In Review Questions 2.55 and 2.59, the table data was entered after referential integrity constraints were created in Review Questions 2.54, 2.57 and 2.58. Why was the data entered after the referential integrity constraints were created instead of before the constraints were created?

2.61 Using Access SQL, create and run queries to answer the following questions. Save each query using the query name format SQL-Query-02-##, where the ## sign is replaced by the letter designator of the question. For example, the first query will be saved as SQL-Query-02-A.

Figure 2-27

Sample Data for the PROJECT Table

ProjectID	Name	Department	MaxHours	StartDate	EndDate
1000	2008 Q3 Product Plan	Marketing	135	05/10/08	06/15/08
1100	2008 Q3 Portfolio Analysis	Finance	120	07/05/08	07/25/08
1200	2008 Q3 Tax Preparation	Accounting	145	08/10/08	10/25/08
1300	2008 Q4 Product Plan	Marketing	150	08/10/08	09/15/08
1400	2008 Q4 Portfolio Analysis	Finance	140	10/05/08	

Amazon Ex. 1008.0111

Column Name	Type	Key	Required	Remarks
ProjectID	Number	Primary Key, Foreign Key	Yes	Long Integer
EmployeeNumber	Number	Primary Key, Foreign Key	Yes	Long Integer
HoursWorked	Number	No	No	Double

Figure 2-28

Database Column Characteristics for the ASSIGNMENT Table

A. What projects are in the PROJECT table? Show all information for each project.

B. What are the ProjectID, Name, StartDate, and EndDate values of projects in the PROJECT table?

C. What projects in the PROJECT table started before August 1, 2008? Show all the information for each project.

D. What projects in the PROJECT table have not been completed? Show all the information for each project.

E. Who are the employees assigned to each project? Show ProjectID, EmployeeNumber, LastName, FirstName, and Phone.

ProjectID	EmployeeNumber	HoursWorked
1000	1	30.0
1000	8	75.0
1000	10	55.0
1100	4	40.0
1100	6	45.0
1100	1	25.0
1200	2	20.0
1200	4	45.0
1200	5	40.0
1300	1	35.0
1300	8	80.0
1300	10	50.0
1400	4	15.0
1400	5	10.0
1400	6	27.5

Figure 1-29

Sample Data for the ASSIGNMENT Table

Amazon Ex. 1008.0112

F. Who are the employees assigned to each project? The ProjectID, Name, and Department. Show EmployeeNumber, LastName, FirstName, and Phone.

G. Who are the employees assigned to each project? Show ProjectID, Name, Department, and Department Phone. Show EmployeeNumber, LastName, FirstName, and Employee Phone. Sort by ProjectID, in ascending order.

H. Who are the employees assigned to projects run by the marketing department? Show ProjectID, Name, Department, and Department Phone. Show EmployeeNumber, LastName, FirstName, and Employee Phone. Sort by ProjectID, in ascending order.

I. How many projects are being run by the marketing department? Be sure to assign an appropriate column name to the computed results.

J. What is the total MaxHours of projects being run by the marketing department? Be sure to assign an appropriate column name to the computed results.

K. What is the average MaxHours of projects being run by the marketing department? Be sure to assign an appropriate column name to the computed results.

L. How many projects are being run by each department? Be sure to display each DepartmentName and to assign an appropriate column name to the computed results.

2.62 Using Access QBE, create and run new queries to answer the questions in exercise 2.61. Save each query using the query name format QBE-Query-02-##, where the ## sign is replaced by the letter designator of the question. For example, the first query will be saved as QBE-Query-02-A.

The following questions refer to the NDX table of data as described starting on page 67. You can obtain a copy of this data in the Access database, DBP-e11-NDX.accdb, from the text's Web site (*www.pearsonhighered.com/kroenke*).

2.63 Write SQL queries to produce the following results:

A. The ChangeClose on Fridays.

B. The minimum, maximum, and average ChangeClose on Fridays.

C. The average ChangeClose grouped by TYear. Show TYear.

D. The average ChangeClose grouped by TYear and TMonth. Show TYear and TMonth.

E. The average ChangeClose grouped by TYear, TQuarter, TMonth shown in descending order of the average (you will have to give a name to the average in order to sort by it). Show TYear, TQuarter, and TMonth. Note that months appear in alphabetical and not calendar order. Explain what you need to do to obtain months in calendar order.

F. The difference between the maximum ChangeClose and the minimum ChangeClose grouped by TYear, TQuarter, TMonth shown in descending order of the difference (you will have to give a name to the difference in order to sort by it). Show TYear, TQuarter, and TMonth.

G. The average ChangeClose grouped by TYear shown in descending order of the average (you will have to give a name to the average in order to sort by it). Show only groups for which the average is positive.

H. Display a single field with the date in the form day/month/year. Do not be concerned with trailing blanks.

2.64 It is possible that volume (the number of shares traded) has some correlation with the direction of the stock market. Use the SQL you have learned in this chapter to investigate this possibility. Develop at least five different SQL statements in your investigation.

Amazon Ex. 1008.0113



Marcia's Dry Cleaning is an upscale dry cleaner in a well-to-do suburban neighborhood. Marcia makes her business stand out from the competition by providing superior customer service. She wants to keep track of each of her customers and their orders. Ultimately, she wants to notify them that their clothes are ready via e-mail. To provide this service, she has developed an initial database with several tables. Three of those tables are the following:

CUSTOMER (CustomerID, FirstName, LastName, Phone, Email,)

ORDER (InvoiceNumber, *CustomerNumber*, DateIn, DateOut, TotalAmount)

ORDER_ITEM (*InvoiceNumber*, ItemNumber, Item, Quantity, UnitPrice)

In the database schema above, the primary keys are underlined and the foreign keys are shown in italics.

The database is named MDC. The column characteristics for the tables are shown in Figures 2-30, 2-31, and 2-32. The data for these tables are shown in Figures 2-33, 2-34, and 2-35.

We recommend that you create an Access 2007 database named MDC-Ch02.accdb using the database characteristics and data above, and then use this database to test your solutions to the questions in this section.

Write SQL statements and show the results based on the MDC data for each of the following:

A. Show all data in each of the tables.

B. List the Phone and LastName of all customers.

C. List the Phone and LastName for all customers with a FirstName of 'Nikki'.

D. List the Phone, DateIn, and DateOut of all orders in excess of $100.00.

E. List the Phone and FirstName of all customers whose first name starts with 'B'.

Figure 2-30

Column Characteristics for the CUSTOMER Table

Column Name	Type	Key	Required	Remarks
CustomerID	Number	Primary Key	Yes	Long Integer
FirstName	Text (25)	No	Yes	
LastName	Text (25)	No	Yes	
Phone	Text (12)	No	No	
Email	Text (100)	No	No	

Figure 2-31

Column Characteristics for the ORDER Table

Column Name	Type	Key	Required	Remarks
InvoiceNumber	Number	Primary Key	Yes	Long Integer
CustomerNumber	Number	Foreign Key	Yes	Long Integer
DateIn	Date/Time	No	Yes	
DateOut	Date/Time	No	No	
TotalAmount	Currency	No	No	Two Decimal Places

Amazon Ex. 1008.0114

Column Name	Type	Key	Required	Remarks
InvoiceNumber	Number	Primary Key, Foreign Key	Yes	Long Integer
ItemNumber	Number	Primary Key	Yes	Long Integer
Item	Text (50)	No	Yes	
Quantity	Number	No	Yes	Long Integer
UnitPrice	Currency	No	Yes	Two Decimal Places

Figure 2-32
Column Characteristics for the ORDER_ITEM Table

CustomerID	FirstName	LastName	Phone	Email
1	Nikki	Kaccaton	723-543-1233	NKaccaton@somewhere.com
2	Brenda	Catnazaro	723-543-2344	BCatnazaro@somewhere.com
3	Bruce	LeCat	723-543-3455	BLeCat@somewhere.com
4	Betsy	Miller	723-654-3211	BMiller@somewhere.com
5	George	Miller	723-654-4322	GMiller@somewhere.com
6	Kathy	Miller	723-514-9877	KMiller@somewhere.com
7	Betsy	Miller	723-514-8766	BMiller@elsewhere.com

Figure 2-33
Sample Data for the CUSTOMER Table

F. List the Phone and FirstName of all customers whose last name includes the characters 'cat'.

G. List the Phone, FirstName, and LastName for all customers whose second and third numbers of their phone number are 23.

InvoiceNumber	CustomerID	DateIn	DateOut	TotalAmount
2009001	1	04-Oct-09	06-Oct-09	$158.50
2009002	2	04-Oct-09	06-Oct-09	$25.00
2009003	1	06-Oct-09	08-Oct-09	$49.00
2009004	4	06-Oct-09	08-Oct-09	$17.50
2009005	6	07-Oct-09	11-Oct-09	$12.00
2009006	3	11-Oct-09	13-Oct-09	$152.50
2009007	3	11-Oct-09	13-Oct-09	$7.00
2009008	7	12-Oct-09	14-Oct-09	$140.50
2009009	5	12-Oct-09	14-Oct-09	$27.00

Figure 2-34
Sample Data for the ORDER Table

Amazon Ex. 1008.0115

InvoiceNumber	ItemNumber	Item	Quantity	UnitPrice
2009001	1	Blouse	2	$3.50
2009001	2	Dress Shirt	5	$2.50
2009001	3	Formal Gown	2	$10.00
2009001	4	Slacks-Mens	10	$5.00
2009001	5	Slacks-Womens	10	$6.00
2009001	6	Suit-Mens	1	$9.00
2009002	1	Dress Shirt	10	$2.50
2009003	1	Slacks-Mens	5	$5.00
2009003	2	Slacks-Womens	4	$6.00
2009004	1	Dress Shirt	7	$2.50
2009005	1	Blouse	2	$3.50
2009005	2	Dress Shirt	2	$2.50
2009006	1	Blouse	5	$3.50
2009006	2	Dress Shirt	10	$2.50
2009006	3	Slacks-Mens	10	$5.00
2009006	4	Slacks-Womens	10	$6.00
2009007	1	Blouse	2	$3.50
2009008	1	Blouse	3	$3.50
2009008	2	Dress Shirt	12	$2.50
2009008	3	Slacks-Mens	8	$5.00
2009008	4	Slacks-Womens	10	$6.00
2009009	1	Suit-Mens	3	$9.00

Figure 2-35
Sample Data for the ORDER_ITEM Table

H. Determine the maximum and minimum TotalAmount.

I. Determine the average TotalAmount.

J. Count the number of customers.

K. Group customers by LastName and then by FirstName.

L. Count the number of customers having each combination of LastName and FirstName.

M. Show the FirstName and LastName of all customers who have had an order with TotalAmount greater than $100.00. Use a subquery. Present the results sorted by LastName in ascending order and then FirstName in descending order.

Amazon Ex. 1008.0116

N. Show the FirstName and LastName of all customers who have had an order with TotalAmount greater than $100.00. Use a join. Present results sorted by LastName in ascending order and then FirstName in descending order.

O. Show the FirstName and LastName, of all customers who have had an order with an Item named 'Dress Shirt'. Use a subquery. Present results sorted by LastName in ascending order and then FirstName in descending order.

P. Show the FirstName and LastName of all customers who have had an order with an Item named 'Dress Shirt'. Use a join. Present results sorted by LastName in ascending order and then FirstName in descending order.

Q. Show the FirstName, LastName, and TotalAmount of all customers who have had an order with an Item named 'Dress Shirt'. Use a join with a subquery. Present results sorted by LastName in ascending order and then FirstName in descending order.

Morgan Importing



Morgan Importing purchases antiques and home furnishings in Asia and ships those items to a warehouse facility in Los Angeles. Mr. Morgan uses a database to keep a list of items purchased, shipments, and shipment items. His database includes the following tables:

SHIPMENT (ShipmentID, ShipperName, ShipperInvoiceNumber, DepartureDate, ArrivalDate, InsuredValue)

SHIPMENT_ITEM (*ShipmentID*, ShipmentItemID, *ItemID*, Value)

ITEM (ItemID, Description, PurchaseDate, Store, City, Quantity, LocalCurrencyAmt, ExchangeRate)

In the database schema above, the primary keys are underlined and the foreign keys are shown in italics.

The database is named MI. The column characteristics for the tables are shown in Figures 2-36, 2-37, and 2-38. The data for these tables are shown in Figures 2-39, 2-40, and 2-41.

We recommend that you create an Access 2007 database named MI-Ch02.accdb using the database characteristics and data above, and then use this database to test your solutions to the questions in this section.

Write SQL statements and shown the results based on the MDC data for each of the following:

A. Show all data in each of the tables.

B. List the ShipmentID, ShipperName, and ShipperInvoiceNumber of all shipments.

Figure 2-36

Column Characteristics for the SHIPMENT Table

Column Name	Type	Key	Required	Remarks
ShipmentID	Number	Primary Key	Yes	Long Integer
ShipperName	Text (35)	No	Yes	
ShipperInvoiceNumber	Number	No	Yes	Long Integer
DepartureDate	Date/Time	No	No	
ArrivalDate	Date/Time	No	No	
InsuredValue	Currency	No	No	Two Decimal Places

Amazon Ex. 1008.0117

Column Name	Type	Key	Required	Remarks
ShipmentID	Number	Primary Key, Foreign Key	Yes	Long Integer
ShipmentItemID	Number	Primary Key	Yes	Long Integer
ItemID	Number	Foreign Key	Yes	Long Integer
Quantity	Number	No	Yes	Long Integer
Value	Currency	No	Yes	Two Decimal Places

Figure 2-37
Column Characteristics for the SHIPMENT_ITEM Table

C. List the ShipmentID, ShipperName, and ShipperInvoiceNumber for all shipments that have an insured value greater than $10,000.00.

D. List the ShipmentID, ShipperName, and ShipperInvoiceNumber of all shippers whose name starts with 'AB'.

E. Assume DepartureDate and ArrivalDate are in the format MM/DD/YY. List the ShipmentID, ShipperName, ShipperInvoiceNumber, and ArrivalDate of all shipments that departed in December.

F. Assume DepartureDate and ArrivalDate are in the format MM/DD/YY. List the ShipmentID, ShipperName, ShipperInvoiceNumber, and ArrivalDate of all shipments that departed on the tenth day of any month.

G. Determine the maximum and minimum InsuredValue.

H. Determine the average InsuredValue.

I. Count the number of shipments.

J. Show ItemID, Description, Store, and a calculated column named USCurrencyAmount that is equal to LocalCurrencyAmt multiplied by the ExchangeRate for all rows of ITEM.

K. Group item purchases by City and Store.

L. Count the number of purchases having each combination of City and Store.

Figure 2-38
Column Characteristics for the ITEM Table

Column Name	Type	Key	Required	Remarks
ItemID	Number	Primary Key	Yes	Long Integer
Description	Text (255)	No	Yes	Long Integer
PurchaseDate	Date/Time	No	Yes	
Store	Text (50)	No	Yes	
City	Text (35)	No	Yes	
Quantity	Number	No	Yes	Long Integer
LocalCurrencyAmt	Number	No	Yes	Decimal, 18 Auto
ExchangeRate	Number	No	Yes	Decimal, 12 Auto

Amazon Ex. 1008.0118

ShipmentID	ShipperName	ShipperInvoiceNumber	DepartureDate	ArrivalDate	InsuredValue
1	ABC Trans-Oceanic	2008651	10-Dec-08	15-Mar-09	$15,000.00
2	ABC Trans-Oceanic	2009012	10-Jan-09	20-Mar-09	$12,000.00
3	Worldwide	49100300	05-May-09	17-Jun-09	$27,500.00
4	International	399400	02-Jun-09	17-Jul-09	$7,500.00
5	Worldwide	84899440	10-Jul-09	28-Jul-09	$25,000.00
6	International	488955	05-Aug-09	11-Sep-09	$18,000.00

Figure 2-39

Sample Data for the SHIPMENT Table

M. Show the ShipperName and DepartureDate of all shipments that have an item with a value of $1,000.00 or more. Use a subquery. Present results sorted by ShipperName in ascending order and then DepartureDate in descending order.

N. Show the ShipperName and DepartureDate of all shipments that have an item with a value of $1,000.00 or more. Use a join. Present results sorted by ShipperName in ascending order and then DepartureDate in descending order.

O. Show the ShipperName and DepartureDate of all shipments that have an item that was purchased in Singapore. Use a subquery. Present results sorted by ShipperName in ascending order and then DepartureDate in descending order.

P. Show the ShipperName and DepartureDate of all shipments that have an item that was purchased in Singapore. Use a join. Present results sorted by ShipperName in ascending order and then DepartureDate in descending order.

Q. Show the ShipperName, DepartureDate of shipment, and Value for items that were purchased in Singapore. Use a combination of a join and a subquery. Present results sorted by ShipperName in ascending order and then DepartureDate in descending order.

Figure 2-40

Sample Data for the SHIPMENT_ITEM Table

ShipmentID	ShipmentItemID	ItemID	Quantity	Value
4	1	4	40	$1,200.00
4	2	3	8	$9,500.00
4	3	2	75	$4,500.00

Figure 2-41

Sample Data for the ITEM Table

ItemID	Description	PurchaseDate	Store	City	Quantity	LocalCurrencyAmt	ExchangeRate
1	QE Dining Set	07-Apr-09	Eastern Treasures	Manila	2	403405	0.01774
2	Willow Serving Dishes	15-Jul-09	Jade Antiques	Singapore	75	102	0.5903
3	Large Bureau	17-Jul-09	Eastern Sales	Singapore	8	2000	0.5903
4	Brass Lamps	20-Jul-09	Jade Antiques	Singapore	40	50	0.5903

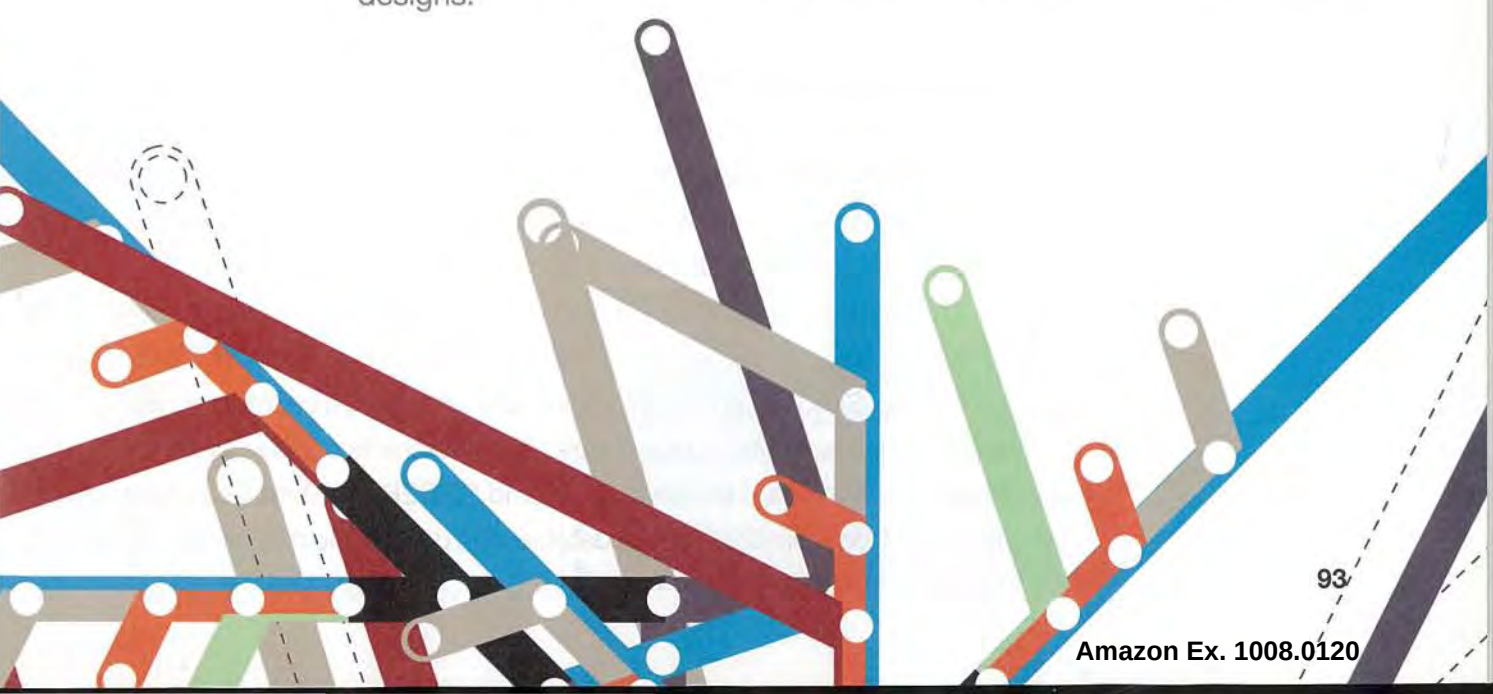
Amazon Ex. 1008.0119

Part 2

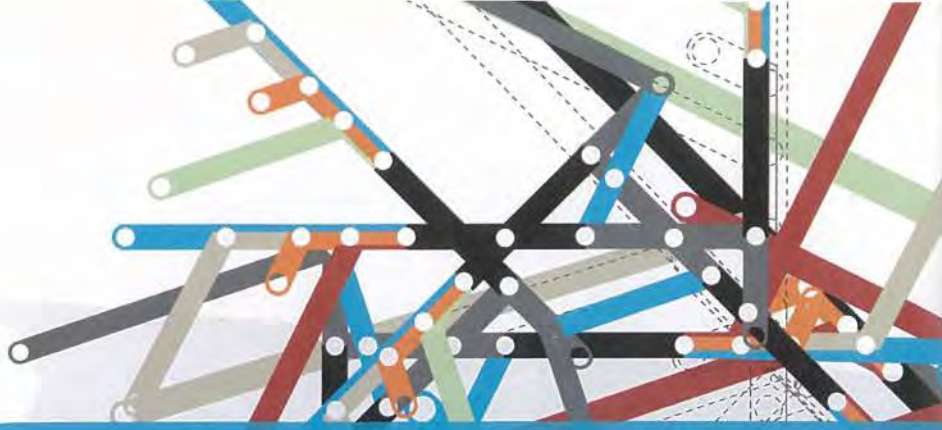
Database Design

The four chapters in Part 2 discuss database design principles and techniques. Chapters 3 and 4 describe the design of databases that arise from existing data sources, such as spreadsheets, text files, and database extracts. We begin in Chapter 3 by defining the relational model and discussing normalization, a process that transforms relations with modification problems. Then, in Chapter 4, we use normalization principles to guide the design of databases from existing data.

Chapters 5 and 6 examine the design of databases that arise from the development of new information systems. Chapter 5 describes the entity-relationship data model, a tool used to create plans for constructing database designs. As you will learn, such data models are developed by analysis of forms, reports, and other information systems requirements. Chapter 6 concludes this part by describing techniques for transforming entity-relationship data models into relational database designs.



Amazon Ex. 1008.0120



3 The Relational Model and Normalization

Chapter Objectives

- To understand basic relational terminology.
- To understand the characteristics of relations.
- To understand alternative terminology used in describing the relational model.
- To be able to identify functional dependencies, determinants, and dependent attributes.
- To identify primary, candidate, and composite keys.
- To be able to identify possible insertion, deletion, and update anomalies in a relation.
- To be able to place a relation into BCNF normal form.
- To understand the special importance of domain/key normal form.
- To be able to identify multivalued dependencies.
- To be able to place a relation in fourth normal form.

As we discussed in Chapter 1, databases arise from three sources: from existing data, from the development of new information systems, and from the redesign of existing databases. In this chapter and the next, we consider the design of databases from existing data, such as data from spreadsheets or extracts of existing databases.

The premise of Chapters 3 and 4 is that you have received one or more tables of data from some source that are to be stored in a new database. The question is: Should this data be stored as is, or should it be transformed in some way before it is stored? For example, consider the two tables in the top part of Figure 3-1. These are the SKU_DATA and ORDER_ITEM tables extracted from the Cape Codd Outdoor Sports database as used in the database in Chapter 2.

Amazon Ex. 1008.0121

ORDER_ITEM

	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	1000	201000	1	300.00	300.00
2	1000	202000	1	130.00	130.00
3	2000	101100	4	50.00	200.00
4	2000	101200	2	50.00	100.00
5	3000	100200	1	300.00	300.00
6	3000	101100	2	50.00	100.00
7	3000	101200	1	50.00	50.00

SKU_DATA

	SKU	SKU_Description	Department	Buyer
1	100100	Std. Scuba Tank, Yellow	Water Sports	Pete Hansen
2	100200	Std. Scuba Tank, Magenta	Water Sports	Pete Hansen
3	101100	Dive Mask, Small Clear	Water Sports	Nancy Meyers
4	101200	Dive Mask, Med Clear	Water Sports	Nancy Meyers
5	201000	Half-dome Tent	Camping	Cindy Lo
6	202000	Half-dome Tent Footprint	Camping	Cindy Lo
7	301000	Light Fly Climbing Harness	Climbing	Jerry Martin
8	302000	Locking carabiner, Oval	Climbing	Jerry Martin

SKU_ITEM

	OrderNumber	SKU	Quantity	Price	SKU_Description	Department	Buyer
1	1000	201000	1	300.00	Half-dome Tent	Camping	Cindy Lo
2	1000	202000	1	130.00	Half-dome Tent Footprint	Camping	Cindy Lo
3	2000	101100	4	50.00	Dive Mask, Small Clear	Water Sports	Nancy Meyers
4	2000	101200	2	50.00	Dive Mask, Med Clear	Water Sports	Nancy Meyers
5	3000	100200	1	300.00	Std. Scuba Tank, Magenta	Water Sports	Pete Hansen
6	3000	101100	2	50.00	Dive Mask, Small Clear	Water Sports	Nancy Meyers
7	3000	101200	1	50.00	Dive Mask, Med Clear	Water Sports	Nancy Meyers

Figure 3-1
How Many Tables?

You can design the new database to store this data as two separate tables, or you can join the tables together and design the database with just one table. Each alternative has advantages and disadvantages. When you make the decision to use one design, you obtain certain advantages at the expense of certain costs. The purpose of this chapter is to help you understand those advantages and costs.

Such questions do not seem difficult, and you may be wondering why we need two chapters to answer them. In truth, even a single table can have surprising complexity. Consider, for example, the table in Figure 3-2, which shows sample data extracted from a corporate database. This simple table has three columns: the buyer's name, the SKU of the products that the buyer purchases, and the names of the buyer's college major(s). Buyers manage more than one SKU, and they can have multiple college majors.

PRODUCT_BUYER

	BuyerName	SKU_Managed	CollegeMajor
1	Nancy Meyers	101100	Art
2	Nancy Meyers	101100	Info Systems
3	Nancy Meyers	101200	Art
4	Nancy Meyers	101200	Info Systems
5	Pete Hansen	100100	Business
6	Pete Hansen	100200	Business

Figure 3-2
PRODUCT_BUYER—A Very Strange Table

Amazon Ex. 1008.0122

To understand why this is an odd table, suppose that Nancy Meyers is assigned a new SKU, say 101300. What addition should we make to this table? Clearly, we need to add a row for the new SKU, but if we add just one row, say the row ('Nancy Meyers', 101300, 'Art'), it will appear that she manages product 101300 as an Art major, but not as an Info Systems major. To avoid such an illogical state, we need to add two rows: ('Nancy Meyers', 101300, 'Art') and ('Nancy Meyers', 101300, 'Info Systems').

This is a strange requirement. Why should we have to add two rows of data simply to record the fact that a new SKU has been assigned to a buyer? Further, if we assign the product to Pete Hansen instead, we would only have to add one row, but if we assigned the product to a buyer who had four majors, we would have to add four new rows.

The more one thinks about the table in Figure 3-2, the more strange it becomes. What changes should we make if SKU 101100 is assigned to Pete Hansen? What changes should we make if SKU 100100 is assigned to Nancy Meyers? What should we do if all the SKU values in Figure 3-2 are deleted? Later in this chapter, you will learn that these problems arise because this table has a problem called a *multivalued dependency.* Even better, you will learn how to remove that problem.

Tables can have many different patterns; some patterns are susceptible to serious problems and other patterns are not. Before we can address this question, however, you need to learn some basic terms.

Relational Model Terminology

Figure 3-3 lists the most important terms used by the relational model. By the time you finish Chapters 3 and 4, you should be able to define each of these terms and explain how each pertains to the design of relational databases. Use this list of terms as a check on your comprehension.

Relations

So far, we have used the terms *table* and *relation* interchangeably. In fact, a relation is a special case of a table. This means that all relations are tables, but not all tables are relations. Codd

Figure 3-3
Important Relational Model Terms

- Relation
- Functional dependency
- Determinant
- Candidate key
- Composite key
- Primary key
- Surrogate key
- Foreign key
- Referential integrity constraint
- Normal form
- Multivalued dependency

Amazon Ex. 1008.0123

Characteristics of Relations
Rows contain data about an entity.
Columns contain data about attributes of the entities.
All entries in a column are of the same kind.
Each column has a unique name.
Cells of the table hold a single value.
The order of the columns is unimportant.
The order of the rows is unimportant.
No two rows may be identical.

Figure 3-4
Characteristic of Relations

defined the characteristics of a relation in his 1970 paper that laid the foundation for the relational model.[1] Those characteristics are summarized in Figure 3-4.

BY THE WAY In Figure 3-4 and in this discussion, we use the term **entity** to mean some identifiable thing. A customer, a salesperson, an order, a part, and a lease are all examples of what we mean by an entity. When we introduce the entity-relationship model in Chapter 5, we will make the definition of entity more precise. For now, just think of an entity as some identifiable thing that users want to track.

Characteristics of Relations

A **relation** has a specific definition as shown in Figure 3-4. and for a table to be a relation the criteria of this definition must be met. First, the rows of the table must store data about an entity and the columns of the table must store data about the characteristics of those entities. Further, in a relation, all of the values in a column are of the same kind. If, for example, the second column of the first row of a relation has FirstName, then the second column of every row in the relation has FirstName. Also, the names of the columns are unique; no two columns in the same relation may have the same name. The EMPLOYEE table shown in Figure 3-5 meets these criteria and is a relation.

Figure 3-5
Sample EMPLOYEE Relation

EmployeeNumber	FirstName	LastName	Department	Email	Phone
100	Jerry	Johnson	Accounting	JJ@somewhere.com	834-1101
200	Mary	Abernathy	Finance	MA@somewhere.com	834-2101
300	Liz	Smathers	Finance	LS@somewhere.com	834-2102
400	Tom	Caruthers	Accounting	TC@somewhere.com	834-1102
500	Tom	Jackson	Production	TJ@somewhere.com	834-4101
600	Eleanore	Caldera	Legal	EC@somewhere.com	834-3101
700	Richard	Bandalone	Legal	RB@somewhere.com	834-3102

[1] Codd, E. F. "A Relational Model of Data for Large Shared Databanks," *Communications of the ACM*, June 1970, pp. 377–387.

Amazon Ex. 1008.0124

BY THE WAY Columns in different relations may have the same name. In Chapter 2, for example, two relations had a column named SKU. When there is risk of confusion, we precede the column name with the relation name followed by a period. Thus, the name of the SKU column in the SKU_DATA relation is SKU_DATA.SKU, and column C1 of relation R1 is named R1.C1. Because relation names are unique within a database, and because column names are unique within a relation, the combination of relation name and column name uniquely identifies every column in the database.

Each cell of a relation has only a single value or item; multiple entries are not allowed. The table in Figure 3-6 is *not* a relation, because the Phone values of employees Caruthers and Bandalone store multiple phone numbers.

In a relation, the order of the rows and the order of the columns are immaterial. No information can be carried by the ordering of rows or columns. The table in Figure 3-7 is not a relation, because the entries for employees Caruthers and Caldera require a particular row arrangement. If the rows in this table were rearranged, we would not know which employee has the indicated Fax and Home numbers.

Finally, according to the last characteristic in Figure 3-4, for a table to be a relation, no two rows can be identical. As you learned in Chapter 2, some SQL statements do produce tables with duplicate rows. In such cases, you can use the DISTINCT keyword to force uniqueness. Such row duplication only occurs as a result of SQL manipulation. Tables that you design to be stored in the database should never contain duplicate rows.

BY THE WAY Do not fall into a common trap. Even though every cell of a relation must have a single value, this does not mean that all values must have the same length. The table in Figure 3-8 is a relation even though the length of the Comment column varies from row to row. It is a relation because, even though the comments have different lengths, there is only *one* comment per cell.

Alternative Terminology

As defined by Codd, the columns of a relation are called **attributes**, and the rows of a relation are called **tuples** (rhymes with "couples"). Most practitioners, however, do not use these academic-sounding terms and use the terms *column* and *row*, instead. Also, even though a table is not necessarily a relation, most practitioners mean *relation* when they say *table*. Thus, in most

Figure 3-6
Nonrelational Table—Multiple Entries per Cell

EmployeeNumber	FirstName	LastName	Department	Email	Phone
100	Jerry	Johnson	Accounting	JJ@somewhere.com	834-1101
200	Mary	Abernathy	Finance	MA@somewhere.com	834-2101
300	Liz	Smathers	Finance	LS@somewhere.com	834-2102
400	Tom	Caruthers	Accounting	TC@somewhere.com	834-1102, 834-1191, 834-1192
500	Tom	Jackson	Production	TJ@somewhere.com	834-4101
600	Eleanore	Caldera	Legal	EC@somewhere.com	834-3101
700	Richard	Bandalone	Legal	RB@somewhere.com	834-3102, 834-3191

Amazon Ex. 1008.0125

EmployeeNumber	FirstName	LastName	Department	Email	Phone
100	Jerry	Johnson	Accounting	JJ@somewhere.com	834-1101
200	Mary	Abernathy	Finance	MA@somewhere.com	834-2101
300	Liz	Smathers	Finance	LS@somewhere.com	834-2102
400	Tom	Caruthers	Accounting	TC@somewhere.com	834-1102
				Fax:	834-9911
				Home:	723-8795
500	Tom	Jackson	Production	TJ@somewhere.com	834-4101
600	Eleanore	Caldera	Legal	EC@somewhere.com	834-3101
				Fax:	834-9912
				Home:	723-7654
700	Richard	Bandalone	Legal	RB@somewhere.com	834-3102

Figure 3-7

Nonrelational Table—Order of Rows Matters and Kind of Column Entries Differs in Email

conversations the terms *relation* and *table* are synonymous. In fact, for the rest of this book *table* and *relation* will be used synonymously.

Additionally, a third set of terminology also is used. Some practitioners use the terms *file*, *field*, and *record* for the terms *table*, *column*, and *row*, respectively. These terms arose from traditional data processing and are common in connection with legacy systems. Sometimes, people mix and match these terms. You might hear someone say, for example, that a relation has a certain column and contains 47 records. These three sets of terms are summarized in Figure 3-9.

Figure 3-8

Relation with Variable—Length Column Values

EmployeeNumber	FirstName	LastName	Department	Email	Phone	Comment
100	Jerry	Johnson	Accounting	JJ@somewhere.com	834-1101	Joined the Accounting Department in March after completing his MBA. Will take the CPA exam this fall.
200	Mary	Abernathy	Finance	MA@somewhere.com	834-2101	
300	Liz	Smathers	Finance	LS@somewhere.com	834-2102	
400	Tom	Caruthers	Accounting	TC@somewhere.com	834-1102	
500	Tom	Jackson	Production	TJ@somewhere.com	834-4101	
600	Eleanore	Caldera	Legal	EC@somewhere.com	834-3101	
700	Richard	Bandalone	Legal	RB@somewhere.com	834-3102	Is a full time consultant to Legal on a retainer basis.

Amazon Ex. 1008.0126

Table	Column	Row
Relation	Attribute	Tuple
File	Field	Record

Figure 3-9
Three Sets of Equivalent Terms

Functional Dependencies

Functional dependencies are the heart of the database design process, and it is vital for you to understand them. We first explain the concept in general terms and then examine two examples. We begin with a short excursion into the world of algebra. Suppose you are buying boxes of cookies and someone tells you that each box costs $5.00. With this fact, you can compute the cost of several boxes with the formula:

$$\text{CookieCost} = \text{NumberOfBoxes} \times \$5$$

A more general way to express the relationship between CookieCost and NumberOfBoxes is to say that CookieCost *depends upon* NumberOfBoxes. Such a statement tells us the character of the relationship between CookieCost and NumberOfBoxes, even though it doesn't give us the formula. More formally, we can say that CookieCost is **functionally dependent** on NumberOfBoxes. Such a statement can be written as:

NumberOfBoxes → CookieCost

This expression can be read as "NumberOfBoxes *determines* CookieCost." The variable on the left, here NumberOfBoxes, is called the **determinant**.

Using another formula, we can compute the extended price of a part order by multiplying the quantity of the item times its unit price, or:

$$\text{ExtendedPrice} = \text{Quantity} \times \text{UnitPrice}$$

In this case, we say that ExtendedPrice is functionally dependent on Quantity and UnitPrice, or:

(Quantity, UnitPrice) → ExtendedPrice

Here, the determinant is the composite (Quantity, UnitPrice).

Functional Dependencies That Are Not Equations

In general, a functional dependency exists when the value of one or more attributes determines the value of another attribute. Many functional dependencies exist that do not involve equations.

Consider an example. Suppose you know that a sack contains either red, blue, or yellow objects. Further, suppose you know that the red objects weigh 5 pounds, the blue objects weigh 3 pounds, and the yellow objects weigh 7 pounds. If a friend looks into the sack, sees an object, and tells you the color of the object, you can tell her the weight of the object. We can formalize this as:

ObjectColor → Weight

Thus, we can say that Weight is functionally dependent on ObjectColor and that ObjectColor determines Weight. The relationship here does not involve an equation, but the functional dependency holds. Given a value for ObjectColor, you can determine the object's weight.

If we also know that the red objects are balls, the blue objects are cubes, and the yellow objects are cubes, we can also say:

ObjectColor → Shape

Amazon Ex. 1008.0127

Thus, ObjectColor determines Shape. We can put these two together to state:

ObjectColor → (Weight, Shape)

Thus, ObjectColor determines Weight and Shape.

Another way to represent these facts is to put them into a table:

Object Color	Weight	Shape
Red	5	Ball
Blue	3	Cube
Yellow	7	Cube

This table meets all of the conditions listed in Figure 3-4, and therefore it is a relation. You may be thinking that we performed a trick or sleight of hand to arrive at this relation, but, in truth, the only reason for having relations is to store instances of functional dependencies. If there were a formula by which we could take ObjectColor and somehow compute Weight and Shape, then we would not need the table. We would just make the computation. Similarly, if there were a formula by which we could take EmployeeNumber and compute EmployeeName and HireDate, then we would not need an EMPLOYEE relation. However, because there is no such formula, we must store the combinations of EmployeeNumber, EmployeeName, and HireDate in the rows of a relation.

Composite Functional Dependencies

The determinant of a functional dependency can consist of more than one attribute. For example, a grade in a class is determined by both the student and the class, or:

(StudentNumber, ClassNumber) → Grade

In this case, the determinant is called a **composite determinant**.

Notice that both the student and the class are needed to determine the grade. In general, if (A, B) → C, then neither A nor B will determine C by itself. However, if A → (B, C), then it is true that A → B and A → C. Work through examples of your own for both of these cases so that you understand why this is true.

Finding Functional Dependencies

To fix the idea of functional dependency in your mind, consider what functional dependencies exist in the SKU_DATA and ORDER_ITEM tables in Figure 3-1.

Functional Dependencies in the SKU_DATA Table

To find functional dependencies in a table, we must ask "Does any column determine the value of another column?" For example, consider the values of the SKU_DATA table in Figure 3-1:

	SKU	SKU_Description	Department	Buyer
1	100100	Std. Scuba Tank, Yellow	Water Sports	Pete Hansen
2	100200	Std. Scuba Tank, Magenta	Water Sports	Pete Hansen
3	101100	Dive Mask, Small Clear	Water Sports	Nancy Meyers
4	101200	Dive Mask, Med Clear	Water Sports	Nancy Meyers
5	201000	Half-dome Tent	Camping	Cindy Lo
6	202000	Half-dome Tent Footprint	Camping	Cindy Lo
7	301000	Light Fly Climbing Harness	Climbing	Jerry Martin
8	302000	Locking carabiner, Oval	Climbing	Jerry Martin

Amazon Ex. 1008.0128

Consider the last two columns. If we know the value of Department, can we determine a unique value of Buyer? No, we cannot, because a Department may have more than one Buyer. In this sample data, 'Water Sports' is associated with Pete Hansen and Nancy Meyers. Therefore, Department does not functionally determine Buyer.

What about the reverse? Does Buyer determine Department? In every row, for a given value of Buyer, do we find the same value of Department? Every time Jerry Martin appears, for example, is he paired with the same department? The answer is yes. Further, every time Cindy Lo appears, she is paired with the same department. The same is true for the other buyers. Therefore, assuming that these data are representative, Buyer does determine Department, and we can write:

Buyer → Department

Does Buyer determine any other column? If we know the value of Buyer, do we know the value of SKU? No, we do not, because a given buyer has many SKUs assigned to him or her. Does Buyer determine SKU_Description? No, because a given value of Buyer occurs with many values of SKU_Description.



As stated, for the Buyer → Department functional dependency a Buyer is paired with one and only one value of Department. Notice that a buyer can appear more than once in the table, but, if so, that buyer is always paired with the same department. This is true for all functional dependencies. If A → B, then each value of A will be paired with one and only one value of B. A particular value of A may appear more than once in the relation, but, if so, it is always paired with the same value of B. Note, too, that the reverse is not necessarily true. If A → B, then a value of B may be paired with many values of A.

What about the other columns? It turns out that if we know the value of SKU, we also know the values of all of the other columns. In other words:

SKU → SKU_Description

because a given value of SKU will have just one value of SKU_Description. Next,

SKU → Department

because a given value of SKU will have just one value of Department. And, finally,

SKU → Buyer

because a given value of SKU will have just one value of Buyer.

We can combine these three statements as:

SKU → (SKU_Description, Department, Buyer)

For the same reasons, SKU_Description determines all of the other columns, and we can write:

SKU_Description → (SKU, Department, Buyer)

In summary, the functional dependencies in the SKU_DATA table are:

SKU → (SKU_Description, Department, Buyer)

SKU_Description → (SKU, Department, Buyer)

Buyer → Department

Amazon Ex. 1008.0129

BY THE WAY You cannot always determine functional dependencies from sample data. You may not have any sample data, or you may have just a few rows that are not representative of all of the data conditions. In such cases, you must ask the users who are experts in the application that creates the data. For the SKU_DATA table, you would ask questions such as, "Is a Buyer always associated with the same Department?" and "Can a Department have more than one Buyer?" In most cases, answers to such questions are more reliable than sample data. When in doubt, trust the users.

Functional Dependencies in the ORDER_ITEM Table

Now consider the ORDER_ITEM table in Figure 3-1. For convenience, here is a copy of the data in that table:

	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	1000	201000	1	300.00	300.00
2	1000	202000	1	130.00	130.00
3	2000	101100	4	50.00	200.00
4	2000	101200	2	50.00	100.00
5	3000	100200	1	300.00	300.00
6	3000	101100	2	50.00	100.00
7	3000	101200	1	50.00	50.00

What are the functional dependencies in this table? Start on the left. Does OrderNumber determine another column? It does not determine SKU, because several SKUs are associated with a given order. For the same reasons, it does not determine Quantity, Price, or ExtendedPrice.

What about SKU? SKU does not determine OrderNumber because several OrderNumbers are associated with a given SKU. It does not determine Quantity or ExtendedPrice for the same reason.

What about SKU and Price? From this data, it does appear that

SKU → Price

but that might not be true in general. In fact, we know that prices can change after an order has been processed. Further, an order might have special pricing due to a sale or promotion. To keep an accurate record of what the customer actually paid, we need to associate a particular SKU price with a particular order. Thus:

(OrderNumber, SKU) → Price

Considering the other columns, Quantity, Price, and ExtendedPrice do not determine anything else. You can decide this by looking at the sample data. You can reinforce this conclusion by thinking about the nature of sales. Would a Quantity of 2 ever determine an OrderNumber or a SKU? This makes no sense. At the grocery store, if I tell you I bought two of something, you have no reason to conclude that my OrderNumber was 1010022203466 or that I bought carrots. Quantity does not determine OrderNumber or SKU.

Similarly, if I tell you that the price of an item was $3.99, there is no logical way to conclude what my OrderNumber was or that I bought a jar of green olives. Thus, Price does not determine OrderNumber or SKU. Similar comments pertain to ExtendedPrice. It turns out that no single column is a determinant in the ORDER_ITEM table.

What about pairs of columns? We already know that

(OrderNumber, SKU) → Price

Examining the data, (OrderNumber, SKU) determines the other two columns as well. Thus:

(OrderNumber, SKU) → (Quantity, Price, ExtendedPrice)

This functional dependency makes sense. It means that given a particular order and a particular item on that order, there is only one quantity, one price, and one extended price.

Amazon Ex. 1008.0130

Notice, too, that because ExtendedPrice is computed from the formula ExtendedPrice = (Quantity * Price) we have:

(Quantity, Price) → ExtendedPrice

In summary, the functional dependencies in ORDER_ITEM are:

(OrderNumber, SKU) → (Quantity, Price, ExtendedPrice)

(Quantity, Price) → ExtendedPrice

No single skill is more important for designing databases than the ability to identify functional dependencies. Make sure you understand the material in this section. Work problems 3.58 and 3.59 and the Marcia's Dry Cleaning and Morgan Importing projects at the end of the chapter. Ask your instructor for help if necessary. You *must* understand functional dependencies and be able to work with them.

When Are Determinant Values Unique?

In the previous section, you may have noticed an irregularity. Sometimes the determinants of a functional dependency are unique in a relation, and sometimes they are not. Consider the SKU_DATA relation, with determinants SKU, SKU_Description, and Buyer. In SKU_DATA, the values of both SKU and SKU_Description are unique in the table. For example, the SKU value 100100 appears just once. Similarly, the SKU_Description value 'Half-dome Tent' occurs just once. From this, it is tempting to conclude that values of determinants are always unique in a relation. However, this is *not* true.

For example, Buyer is a determinant, but it is not unique in SKU_DATA. The buyer 'Cindy Lo' appears in two different rows. In fact, for this sample data all of the buyers occur in two different rows.

In truth, a determinant is unique in a relation only if it determines every other column in the relation. For the SKU_DATA relation, SKU determines all of the other columns. Similarly, SKU_Description determines all of the other columns. Hence, they both are unique. Buyer, however, only determines the Department column. It does not determine SKU or SKU_Description.

The determinants in ORDER_ITEM are (OrderNumber, SKU) and (Quantity, Price). Because (OrderNumber, SKU) determines all of the other columns, it will be unique in the relation. The composite (Quantity and Price) only determines ExtendedPrice. Therefore, it will not be unique in the relation.

This fact means that you cannot find the determinants of all functional dependencies simply by looking for unique values. Some of the determinants will be unique, but some will not be. Instead, to determine if column A determines column B, look at the data and ask, "Every time that a value of column A appears is it matched with the same value of Column B?" If so, it can be a determinant of B. Again, however, sample data can be incomplete, so the best strategies are to think about the nature of the business activity from which the data arise and to ask the users.

Keys

The relational model has more keys than a locksmith. There are candidate keys, composite keys, primary keys, surrogate keys, and foreign keys. In this section, we will define each of these types of keys. Because key definitions rely on the concept of functional dependency, make sure you understand that concept before reading on.

In general, a **key** is a combination of one or more columns that is used to identify particular rows in a relation. Keys that have two columns or more are called **composite keys**.

Candidate Keys

A **candidate key** is a determinant that determines all of the other columns in a relation. The SKU_DATA relation has two candidate keys: SKU and SKU_Description. Buyer is a determinant, but it is not a candidate key because it only determines Department.

The ORDER_ITEM table has just one candidate key: (OrderNumber, SKU). The other determinant in this table, (Quantity, Price), is not a candidate key because it determines only ExtendedPrice.

Amazon Ex. 1008.0131

Candidate keys identify a unique row in a relation. Given the value of a candidate key, we can find one and only one row in the relation that has that value. For example, given the SKU value of 100100, we can find one and only one row in SKU_DATA. Similarly, given the OrderNumber and SKU values (2000, 101100), we can find one and only one row in ORDER_ITEM.

Primary Keys

When designing a database, one of the candidate keys is selected to be the **primary key**. This term is used because this key will be defined to the DBMS, and the DBMS will use it as its primary means for finding rows in a table. A table has only one primary key. The primary key can have one column or it can be a composite.

In this text, to clarify discussions we will sometimes indicate table structure by showing the name of a table followed by the names of the table's columns enclosed in parentheses. When we do this, we will underline the column(s) that comprise the primary key. For example, we can show the structure of SKU_DATA and ORDER_ITEM as follows:

SKU_DATA (SKU, SKU_Description, Department, Buyer)

ORDER_ITEM (OrderNumber, SKU, Quantity, Price, ExtendedPrice)

This notation indicates that SKU is the primary key of SKU_DATA and that (OrderNumber, SKU) is the primary key of ORDER_ITEM.

BY THE WAY What do you do if a table has no candidate keys? In that case, define the primary key as the collection of all of the columns in the table. Because there are no duplicate rows in a stored relation, the combination of all of the columns of the table will always be unique. Again, while tables generated by SQL manipulation may have duplicate rows, the tables that you design to be stored should never be constructed to have duplication. Thus, the combination of all columns is always a candidate key.

Surrogate Keys

A **surrogate key** is an artificial column that is added to a table to serve as the primary key. The DBMS assigns a unique value to a surrogate key when the row is created. The assigned value never changes. Surrogate keys are used when the primary key is large and unwieldy. For example, consider the relation RENTAL_PROPERTY:

RENTAL_PROPERTY (Street, City, State/Province, Zip/PostalCode, Country, Rental_Rate)

The primary key of this table is (Street, City, State/Province, Zip/PostalCode, Country). As you will learn in Chapter 6, for good performance a primary key should be short and, if possible, numeric. The primary key of RENTAL_PROPERTY is neither.

In this case, the designers of the database would likely create a surrogate key. The structure of the table would then be:

RENTAL_PROPERTY (PropertyID, Street, City, State/Province, Zip/PostalCode, Country, Rental_Rate)

The DBMS will assign a numeric value to PropertyID when a row is created. Using that key will result in better performance than using the original key. Note that surrogate key values are artificial and have no meaning to the users. In fact, surrogate key values are normally hidden in forms and reports.

Foreign Keys

A **foreign key** is a column or composite of columns that is the primary key of a table other than the one in which it appears. The term arises because it is a key of a table *foreign* to the one in which it appears. In the following two tables, DEPARTMENT.DepartmentName is the primary

Amazon Ex. 1008.0132

key of DEPARTMENT, and EMPLOYEE.DepartmentName is a foreign key. In this text, we will show foreign keys in italics:

DEPARTMENT (DepartmentName, BudgetCode, ManagerName)

EMPLOYEE (EmployeeNumber, EmployeeLastName, EmployeeFirstName, *DepartmentName*)

Foreign keys express relationships between rows of tables. In this example, the foreign key EMPLOYEE.DepartmentName stores the relationship between an employee and his or her department.

Consider the SKU_DATA and ORDER_ITEM tables. SKU_DATA.SKU is the primary key of SKU_DATA, and ORDER_ITEM.SKU is a foreign key.

SKU_DATA (SKU, SKU_Description, Department, Buyer)

ORDER_ITEM (OrderNumber, *SKU*, Quantity, Price, ExtendedPrice)

Notice that ORDER_ITEM.SKU is both a foreign key and also part of the primary key of ORDER_ITEM. This condition sometimes occurs, but it is not required. In the example above, EMPLOYEE.DepartmentName is a foreign key, but is not part of the EMPLOYEE primary key. You will see some uses for foreign keys later in this chapter and the next, and you will study them at length in Chapter 6.

In most cases, we need to ensure that the values of a foreign key match a valid value of a primary key. For the SKU_DATA and ORDER_ITEM tables, we need to ensure that all of the values of ORDER_ITEM.SKU match a value of SKU_DATA.SKU. To accomplish this, we create a **referential integrity constraint**, which is a statement that limits the values of the foreign key. In this case, we create the constraint:

SKU in ORDER_ITEM must exist in SKU in SKU_DATA

This constraint stipulates that every value of SKU in ORDER_ITEM must match a value of SKU in SKU_DATA.

Normal Forms

All relations are not equal. Some are easy to process, and others are problematic. Relations are categorized into **normal forms** based on the kinds of problems that they have. Knowledge of these normal forms will help you create appropriate database designs. To understand normal forms, we need first to define modification anomalies.

Modification Anomalies

Consider the EQUIPMENT_REPAIR relation in Figure 3-10, which stores data about manufacturing equipment and equipment repairs. Suppose we delete the data for repair number 2100. When we delete this row (the second one in Figure 3-10), we remove not only data about the repair, but also data about the machine itself. We will no longer know, for example, that the machine was a Lathe and that its AcquisitionPrice was 4750.00. When we delete one row, the structure of this table forces us to lose facts about two different things, a machine and a repair, This condition is called a **deletion anomaly**.

Figure 3-10

The EQUIPMENT_REPAIR Table

	ItemNumber	Type	AcquisitionCost	RepairNumber	RepairDate	RepairCost
1	100	Drill Press	3500.00	2000	2009-05-05 ...	375.00
2	200	Lathe	4750.00	2100	2009-05-07 ...	255.00
3	100	Drill Press	3500.00	2200	2009-06-19 ...	178.00
4	300	Mill	27300.00	2300	2009-06-19 ...	1875.00
5	100	Drill Press	3500.00	2400	2009-07-05 ...	0.00
6	100	Drill Press	3500.00	2500	2009-08-17 ...	275.00

Amazon Ex. 1008.0133

	ItemNumber	Type	AcquisitionCost	RepairNumber	RepairDate	RepairCost
1	100	Drill Press	3500.00	2000	2009-05-05 ...	375.00
2	200	Lathe	4750.00	2100	2009-05-07 ...	255.00
3	100	Drill Press	3500.00	2200	2009-06-19 ...	178.00
4	300	Mill	27300.00	2300	2009-06-19 ...	1875.00
5	100	Drill Press	3500.00	2400	2009-07-05 ...	0.00
6	100	Drill Press	5500.00	2500	2009-08-17 ...	275.00

Figure 3-11

The EQUIPMENT_REPAIR Table After an Incorrect Update

Now suppose we want to enter the first repair for a piece of equipment. To enter repair data, we need to know not just RepairNumber, RepairDate, and RepairCost, but also ItemNumber, Type, and AcquisitionCost. If we work in the repair department, this is a problem, because we are unlikely to know the value of AcquisitionCost. The structure of this table forces us to enter facts about two entities when we just want to enter facts about one. This condition is called an **insertion anomaly**.

Finally, suppose we want to change existing data. If we alter a value of RepairNumber, RepairDate, or RepairCost, there is no problem. But if we alter a value of ItemNumber, Type, or AcquisitionCost, we may create a data inconsistency. To see why, suppose we update the last row of the table in Figure 3-10 using the data (100, 'Drill Press', 5500, 2500, '08/17/09', 275).

Figure 3-11 shows the table after this erroneous update. The drill press has two different AcquisitionCosts. Clearly, this is an error. Equipment cannot be acquired at two different costs. If there were, say, 10,000 rows in the table, however, it might be very difficult to detect this error. This condition is called an **update anomaly**.

BY THE WAY Notice that the EQUIPMENT_REPAIR table in Figures 3-10 and 3-11 duplicates data. For example, the AcquisitionCost of the same item of equipment appears several times. Any table that duplicates data is susceptible to update anomalies like the one in Figure 3-11. A table that has such inconsistencies is said to have **data integrity problems**.

As you will learn in Chapter 4, to improve query speed we sometimes design a table to have duplicated data. Be aware, however, that any time we design a table this way we open the door to data integrity problems.

A Short History of Normal Forms

When Codd defined the relational model, he noticed that some tables had modification anomalies. In his second paper,[2] he defined first normal form , second normal form, and third normal form. He defined **first normal form (1NF)** as the *set of conditions for a relation* shown in Figure 3-4. Any table meeting the conditions in Figure 3-4 is therefore a relation in 1NF. Codd also noted that some tables (or, interchangeably in this book, relation) in 1NF had modification anomalies. He found that he could remove some of those anomalies by applying certain conditions. A relation that met those conditions, which we will discuss later in this chapter, was said to be in **second normal form (2NF)**. He also observed, however, that relations in 2NF could also have anomalies, and so he defined **third normal form (3NF)**, which is a set of conditions that removes even more anomalies, and which we will also discuss later in this chapter. As time went by, other researchers found still other ways that anomalies can occur, and the conditions for **Boyce-Codd Normal Form (BCNF)** was defined.

These normal forms are defined so that a relation in BCNF is in 3NF, a relation in 3NF is in 2NF, and a relation in 2NF is in 1NF. Thus, if you put a relation into BCNF, it is automatically in the lesser normal forms.

[2] Codd, E. F. and A. L. Dean. "Proceedings of 1971 ACM-SIGFIDET Workshop on Data Description," *Access and Control*, San Diego, California, November 11–12, 1971 ACM 1971.

Amazon Ex. 1008.0134

Normal forms 2NF through BCNF concern anomalies that arise from functional dependencies. Other sources of anomalies were found later. They led to the definition of **fourth normal form (4NF)** and **fifth normal form (5NF)**, both of which we will discuss later in this chapter. So it went, with researchers chipping away at modification anomalies, each one improving on the prior normal form.

In 1982, Fagin published a paper that took a different tack.[3] Instead of looking for just another normal form, Fagin asked, "What conditions need to exist for a relation to have no anomalies?" In that paper, he defined **domain/key normal form (DK/NF)**. Fagin ended the search for normal forms by showing that a relation in DK/NF has no modification anomalies and, further, that a relation that has no modification anomalies is in DK/NF. DK/NF is discussed in more detail later in this chapter.

Normalization Categories

As shown in Figure 3-12, normalization theory can be divided into three major categories. Some anomalies arise from functional dependencies, some arise from multivalued dependencies, and some arise from data constraints and odd conditions.

BCNF, 3NF, and 2NF, are all concerned with anomalies that are caused by functional dependencies. A relation that is in BCNF has no modification anomalies from functional dependencies. It is also automatically in 2NF and 3NF, and, therefore, we will focus on transforming relations into BCNF. However, it is instructive to work through the progression of normal forms from 1NF to BCNF in order to understand how each normal form deals with specific anomalies, and we will do this later in this chapter.[4]

As shown in the second row of Figure 3-12, some anomalies arise because of another kind of dependency called a multivalued dependency. Those anomalies can be eliminated by placing each multivalued dependency in a relation of its own, a condition known as 4NF. You will see how to do that in the last section of this chapter.

The third source of anomalies is esoteric. These problems involve specific, rare, and even strange data constraints. Accordingly, we will not discuss them in this text.

From First Normal Form to Boyce-Codd Normal Form

Any table that meets the definition of a relation in Figure 3-4 is defined as being in 1NF. This means that the following must hold: The cells of a table must be a single value, and neither repeating groups nor arrays are allowed as values; all entries in a column must be of the same data type; each column must have a unique name, but the order of the columns in the table is not significant; no two rows in a table may be identical, but the order of the rows is not significant.

Second Normal Form

When Codd discovered anomalies in 1NF tables, he defined 2NF to eliminate some of these anomalies. A relation is 2NF if and only if *it is in 1NF* and *all non-key attributes are determined by the entire primary key*. This means that if the primary key is a composite primary key, then

Figure 3-12
Summary of Normalization Theory

Source of Anomaly	Normal Forms	Design Principles
Functional dependencies	1NF, 2NF, 3NF, BCNF	BCNF: Design tables so that every determinant is a candidate key.
Multivalued dependencies	4NF	4NF: Move each multivalued dependency to a table of its own.
Data constraints and oddities	5NF, DK/NF	DK/NF: Make every constraint a logical consequence of candidate keys and domains.

[3] Fagin, R. "A Normal Form for Relational Databases That Is Based on Domains and Keys," *ACM Transactions on Database Systems*, September 1981, pp. 387–414.
[4] See Date, C. J. *Database Systems*, 8th ed. (New York: Addison-Wesley, 2003) for a complete discussion of normal forms.

Amazon Ex. 1008.0135

no non-key attribute can be determined by an attribute or set of attributes that make up only part of the key. Thus, if you have a relation **R (A, B, N, O, P)** with the composite key **(A, B)**, then none of the non-key attributes N, O, or P can be determined by just A or just B.

Note that the only way a non-key attribute can be dependent on part of the primary key is if there is a composite primary key. This means that relations with single-attribute primary keys are automatically in 2NF.

For example, consider the relation STUDENT_ACTIVITY (StudentID, Activity, Fee). The composite primary key is (StudentID, Activity), which allows us to determine the fee a particular student will have to pay for a particular activity. However, since fees are determined by activities, Fee is also functionally dependent on just Activity itself, and we can say that Fee is **partially dependent** on the key of the table. Thus, there is a non-key attribute determined by part of the composite primary, and the STUDENT_ACTIVITY relation is not in 2NF.

What do we do in this case? We will have to move the columns of this functional dependency into a separate relation while leaving the determinant in the original relation as a foreign key. We will end up with two relations: STUDENT_ACTIVITY (StudentID, *Activity*) and ACTIVITY_FEE (Activity, Fee). The Activity column in STUDENT_ACTIVITY becomes a foreign key.

Third Normal Form

However, the conditions necessary for 2NF do not eliminate all anomalies. To deal with additional anomalies, Codd defined 3NF. A relation is in 3NF if and only if *it is in 2NF* and *there are no non-key attributes determined by another non-key attribute.* The technical name for a non-key attribute determined by another non-key attribute is **transitive dependency**. We can therefore restate the definition of 3NF as a relation is in 3NF if and only if *it is in 2NF* and *it has no transitive dependencies.* Thus, in order for our relation **R (A, B, N, O, P)** to be in 3NF, none of the non-key attributes **N, O**, or **P** can be determined by **N, O**, or **P**.

For example, consider the relation STUDENT_HOUSING (StudentID, Building, Fee). In this case, the building fee is independent of which student is housed in the building. However, the same fee is charged for every room in a building, and therefore Building determines Fee. Thus, a non-key attribute (Fee) is functionally determined by another non-key attribute (Building).

Again, we will have to move the columns of the functional dependency into a separate relation while leaving the determinant in the original relation as a foreign key. We will end up with two relations: STUDENT_HOUSING (StudentID, *Building*) and HOUSING_FEE (Building, Fee). The Building column in STUDENT_HOUSING becomes a foreign key.

Boyce-Codd Normal Form

Some database designers normalize their relations to 3NF. Unfortunately, there are still anomalies due to functional dependences in 3NF. Together with Raymond Boyce, Codd defined BCNF to fix this situation. A relation is in BCNF if and only if *it is in 3NF* and *every determinant is a candidate key.*

For example, consider the relation STUDENT_ADVISIOR (StudentID, Major, AdvisorName), where a student (StudentID) can have one or more majors (Major), a major can have one or more faculty advisors (AdvisorName), and a faculty member advises in only one major area.

Since students can have several majors, StudentID does not determine Major. Moreover, since students can have several advisers, StudentID does not determine AdvisorName. Therefore, StudentID by itself cannot be a key. However, the composite key (StudentID, Major) determines AdvisorName, and the composite key (StudentID, AdvisorName) determines Major. This gives us (StudentID, Major) and (StudentId, AdvisorName) as two candidate keys. We can select either of these as the primary key for the relation.

Note that STUDENT_ADVISOR is in 2NF because it has no non-key attributes in the sense that every attribute is a part of at least one candidate key. This is a subtle condition, based on the fact that technically the definition of 2NF states that no *non-prime attribute* can be partially dependent on a candidate key, where a **non-prime attribute** is an attribute that is not contained in any candidate key. STUDENT_ADVISOR is in 3NF because there are no transitive dependencies in the relation.

The two candidate keys for this relation are **overlapping candidate keys**, because they share the attribute StudentID. When a table in 3NF has overlapping candidate keys, it can still have modification anomalies based on functional dependencies. In the STUDENT_ADVISOR relation, there will be modification anomalies because there is one other functional dependency in the relation: because a faculty member can be an advisor for only one major area, AdvisorName determines Major. Therefore, AdvisorName is a determinant, but not a candidate key.

Amazon Ex. 1008.0136

Suppose that we have a student (StudentID = 300) majoring in psychology (Major = Psychology) with faculty advisor Perls (AdvisorName = Perls). Further, assume that this row is the only one in the table with the AdvisorName value of Perls. If we delete this row, we will lose all data about Perls. This is a deletion anomaly. Similarly, we cannot insert the data to represent the Economics advisor Keynes until a student majors in Economics. This is an insertion anomaly. Situations like this led to the development of BCNF.

What do we do with the STUDENT_ADVISOR relation? As before, we move the functional dependency creating the problem to another relation while leaving the determinant in the original relation as a foreign key. In this case, we will create STUDENT_ADVISOR (StudentID, *AdvisorName*) and ADVISOR_MAJOR (AdvisorName, Major). The AdvisorName column in STUDENT_ADVISOR is the foreign key.

Eliminating Anomalies from Functional Dependencies

Most modification anomalies occur because of problems with functional dependencies. You can eliminate these problems by progressively testing a relation for 1NF, 2NF, 3NF and the BCNF using the definitions of these normal forms given previously. Or, you can eliminate such problems by simply designing (or redesigning) your tables so that every determinant is a candidate key. This condition, which, of course, is the definition of BCNF, will eliminate all anomalies due to functional dependencies. We prefer this strategy, and will use it exclusively in this book.

The general normalization method is summarized in Figure 3-13. Identify every functional dependency in the relation, then identify the candidate keys. If there are determinants that are not candidate keys, then the relation is not in BCNF, and it is subject to modification anomalies. To put the relation into BCNF, follow the procedure in step 3. To fix this procedure in your mind, we will illustrate it with five different examples.

BY THE WAY Our process rule that a relation is in BCNF if and only if every determinant is a candidate key is summed up in a widely known phrase:

I swear to construct my tables so that all non-key columns are dependent on the key, the whole key and nothing but the key, so help me Codd!

Figure 3-13
Process for Putting a Relation into BCNF

Process for Putting a Relation into BCNF
1. Identify every functional dependency.
2. Identify every candidate key.
3. If there is a functional dependency that has a determinant that is not a candidate key: A. Move the columns of that functional dependency into a new relation. B. Make the determinant of that functional dependency the primary key of the new relation. C. Leave a copy of the determinant as a foreign key in the original relation. D. Create a referential integrity constraint between the original relation and the new relation.
4. Repeat step 3 until every determinant of every relation is a candidate key.

Note: In step 3, if there is more than one such functional dependency, start with the one with the most columns.

Amazon Ex. 1008.0137

Normalization Example 1

Consider the SKU_DATA table:

SKU_DATA (SKU, SKU_Description, Department, Buyer)

As discussed earlier, this table has three functional dependencies:

SKU → (SKU_Description, Department, Buyer)

SKU_Description → (SKU, Department, Buyer)

Buyer → Department

SKU and SKU_Description determine all of the columns in the table, so they are candidate keys. Buyer is a determinant, but it does not determine all of the other columns, and hence is not a candidate key. Hence, SKU_DATA has a determinant that is not a candidate key and is therefore not in BCNF. It will have modification anomalies.

To remove such anomalies, in step 3A we move the columns of functional dependency whose determinant is not a candidate key into a new table. In this case, we place Buyer and Department into a new table:

BUYER (Buyer, Department)

Next, in step 3B, we make the primary key of the new table the determinant of the functional dependency. In this case, Buyer becomes the primary key:

BUYER (Buyer, Department)

Next, following step 3C, we leave a copy of the determinant as a foreign key in the original relation. Thus, SKU_DATA becomes SKU_DATA_2:

SKU_DATA_2 (SKU, SKU_Description, *Buyer*)

The resulting tables are thus:

BUYER (Buyer, Department)

SKU_DATA_2 (SKU, SKU_Description, *Buyer*)

where SKU_DATA_2.Buyer is a foreign key to the BUYER table.

Both of these tables are now in BCNF and will have no anomalies due to functional dependencies. For the data in these tables to be consistent, however, we also need to define the referential integrity constraint in step 3D:

SKU_DATA_2.Buyer must exist in BUYER.Buyer

This statement means that every value in the Buyer column of SKU_DATA_2 must also exist as a value in the Buyer column of BUYER. Sample data for the resulting tables are shown in Figure 3-14.

Normalization Example 2

Now consider the EQUIPMENT_REPAIR relation in Figure 3-10. The structure of the table is:

EQUIPMENT_REPAIR (ItemNumber, Type, AcquisitionCost, RepairNumber, RepairDate, RepairCost)

Examining the data in Figure 3-8, the functional dependencies are:

ItemNumber → (Type, AcquisitionCost)

RepairNumber → (ItemNumber, Type, AcquisitionCost, RepairDate, RepairCost)

Both ItemNumber and RepairNumber are determinants, but only RepairNumber is a candidate key. Accordingly, EQUIPMENT_REPAIR is not in BCNF and is subject to modification anomalies. Following the procedure in Figure 3-13, we place the columns of the problematic functional dependency into a separate table, as follows:

ITEM (ItemNumber, Type, AcquisitionCost)

Amazon Ex. 1008.0138

BUYER

	Buyer	Department
1	Cindy Lo	Camping
2	Jerry Martin	Climbing
3	Nancy Meyers	Water Sports
4	Pete Hansen	Water Sports

SKU_DATA_2

	SKU	SKU_Description	Buyer
1	100100	Std. Scuba Tank, Yellow	Pete Hansen
2	100200	Std. Scuba Tank, Magenta	Pete Hansen
3	101100	Dive Mask, Small Clear	Nancy Meyers
4	101200	Dive Mask, Med Clear	Nancy Meyers
5	201000	Half-dome Tent	Cindy Lo
6	202000	Half-dome Tent Footprint	Cindy Lo
7	301000	Light Fly Climbing Harness	Jerry Martin
8	302000	Locking carabiner, Oval	Jerry Martin

Figure 3-14
The Normalized BUYER and SKU_DATA_2 Relations

and remove all but ItemNumber from EQUIPMENT_REPAIR (and rearrange the columns so that the primary key RepairNumber is the first column in the relation) to create:

REPAIR (RepairNumber, *ItemNumber*, RepairDate, RepairCost)

We also need to create the referential integrity constraint:

REPAIR.ItemNumber must exist in ITEM.ItemNumber

Data for these two new relations are shown in Figure 3-15.

BY THE WAY There is another, more intuitive way to think about normalization. Do you remember your eighth grade English teacher? She said that every paragraph should have a single theme. If you write a paragraph that has two themes, you should break it up into two paragraphs, each with a single theme.

The problem with the EQUIPMENT_REPAIR relation is that it has two themes: one about repairs and a second about items. We eliminated modification anomalies by breaking that single table with two themes into two tables, each with a single theme. Sometimes, it is helpful to look at a table and ask, "How many themes does it have?" If it has more than one, then redefine the table so that it has a single theme.

ITEM

	ItemNumber	Type	AcquisitionCost
1	100	Drill Press	3500.00
2	200	Lathe	4750.00
3	300	Mill	27300.00

REPAIR

	RepairNumber	ItemNumber	RepairDate	RepairCost
1	2000	100	2009-05-05	375.00
2	2100	200	2009-05-07	255.00
3	2200	100	2009-06-19	178.00
4	2300	300	2009-06-19	1875.00
5	2400	100	2009-07-05	0.00
6	2500	100	2009-07-05	275.00

Figure 3-15
The Normalized ITEM and REPAIR Relations

Amazon Ex. 1008.0139

Normalization Example 3

Consider now the Cape Codd database ORDER_ITEM relation with the structure:

ORDER_ITEM (OrderNumber, SKU, Quantity, Price, ExtendedPrice)

with functional dependencies:

(OrderNumber, SKU) → (Quantity, Price, ExtendedPrice)

(Quantity, Price) → ExtendedPrice

This table is not in BCNF because the determinant (Quantity, Price) is not a candidate key. We can follow the same normalization practice as illustrated in examples 1 and 2, but in this case, because the second functional dependency arises from the formula ExtendedPrice = (Quantity * Price), we reach a silly result.

To see why, we follow the procedure in Figure 3-13 to create tables such that every determinant is a candidate key. This means that we move the columns Quantity, Price, and ExtendedPrice to tables of their own, as follows:

EXTENDED_PRICE (Quantity, Price, ExtendedPrice)

ORDER_ITEM (OrderNumber, SKU, *Quantity, Price*)

Notice that we left both Quantity and Price in the original relation as a composite foreign key. These two tables are in BCNF, but the values in the EXTENDED_PRICE table are ridiculous. They are just the results of multiplying Quantity by Price. The simple fact is that we do not need to create a table to store these results. Instead, any time we need to know ExtendedPrice we will just compute it. In fact, we can define this formula to the DBMS and let the DBMS compute the value of ExtendedPrice when necessary. You will see how to do this with Oracle, SQL Server, and MySQL in Chapters 10, 11, and 12, respectively.

Using the formula, we can remove ExtendedPrice from the table. The resulting table is in BCNF:

ORDER_ITEM (OrderNumber, SKU, Quantity, Price)

Note that Quantity and Price are no longer foreign keys. The ORDER_ITEM table with sample data now appears as shown in Figure 3-16.

Normalization Example 4

Consider the following table that stores data about student activities:

STUDENT_ACTIVITY (SID, Name, Club, Cost, AmountPaid)

where SID is a student identifier, Name is student name, Club is the name of a club, Cost is the cost of joining a club, and AmountPaid is the amount the student has paid to that club. Figure 3-17 shows sample data for this table.

SID is a unique student identifier, so we know that:

SID → Name

Figure 3-16
The Normalized ORDER_ITEM Relation

ORDER_ITEM

	OrderNumber	SKU	Quantity	Price
1	1000	201000	1	300.00
2	1000	202000	1	130.00
3	2000	101100	4	50.00
4	2000	101200	2	50.00
5	3000	100200	1	300.00
6	3000	101100	2	50.00
7	3000	101200	1	50.00

Amazon Ex. 1008.0140

STUDENT_ACTIVITY

	SID	Name	Club	Cost	AmountPaid
1	100	Jones	Scuba	400.00	0.00
2	200	Chau	Scuba	400.00	400.00
3	200	Chau	Skiing	550.00	550.00
4	300	Garrett	Climbing	150.00	150.00
5	400	Jones	Skiing	550.00	550.00

Figure 3-17
Sample Data for the STUDENT_ACTIVITY Relation

However, does the functional dependency

SID → Club

exist? It does if a student belongs to just one club, but it does not if a student belongs to more than one club. Looking at the data, student Chau with SID 200 belongs to two different clubs, so SID does not determine Club. SID does not determine Cost or AmountPaid, either.

Now consider the Name column. Does Name determine SID? Is, for example, the value 'Jones' always paired with the same value of SID? No, there are two students named 'Jones', and they have different SID values. Name does not determine any other column in this table, either.

Considering the next column, Club, we know that many students can belong to a club. Therefore, Club does not determine SID or Name. Does Club determine Cost? Is the value 'Scuba', for example, always paired with the same value of Cost? From this data, it appears so, and using just this sample data, we can conclude that Club determines Cost.

However, this data is just a sample. Logically, it is possible for students to pay different costs, perhaps because they select different levels of club membership. If that were the case, then we would say that

(SID, Club) → Cost

To find out, we need to check with the users. Here, assume that all students pay the same cost for a given club. The last column is AmountPaid, and it does not determine anything.

So far, we have two functional dependencies:

SID → Name

Club → Cost

Are there other functional dependencies with composite determinants? No single column determines AmountPaid, so consider possible composite determinants for it. AmountPaid is dependent on both the student and the club the student has joined. Therefore, it is determined by the combination of the determinants Student and Club. Thus, we can say

(SID, Club) → AmountPaid

So far we have three determinants: SID, Club, and (SID, Club). Are any of these candidate keys? Do any of these determinants identify a unique row? From the data, it appears that (SID, Club) identifies a unique row and is a candidate key. Again, in real situations, we would need to check this assumption out with the users.

STUDENT_ACTIVITY is not in BCNF because columns SID and Club are both determinants, but neither is a candidate key. SID and Club are only part of the candidate key (SID, Club).

BY THE WAY Both SID and Club are *part of* the candidate key (SID, Club). This, however, is not good enough. A determinant must have all of the same columns to be the same as a candidate key. Remember, as we stated above:

I swear to construct my tables so that all non-key columns are dependent on the key, the whole key, and nothing but the key, so help me Codd."

Amazon Ex. 1008.0141

To normalize this table, we need to construct tables so that every determinant is a candidate key. We can do this by creating a separate table for each functional dependency as we did before. The result is:

STUDENT (SID, Name)

CLUB (Club, Cost)

PAYMENT (*SID*, *Club*, AmountPaid)

with referential integrity constraints:

PAYMENT.SID must exist in STUDENT.SID

and

PAYMENT.Club must exist in CLUB.Club

These tables are in BCNF and will have no anomalies from functional dependencies. The sample data for the normalized tables are shown in Figure 3-18.

Normalization Example 5

Now consider a normalization process that requires two iterations of step 3 in the procedure in Figure 3-13. To do this, we will extend the SKU_DATA relation by adding the budget code of each department. We call the revised relation SKU_DATA_3 and define it as follows:

SKU_DATA_3 (SKU, SKU_Description, Department, DeptBudgetCode, Buyer)

Sample data for this relation are shown in Figure 3-19.

SKU_DATA_3 has the following functional dependencies:

SKU → (SKU_Description, Department, DeptBudgetCode, Buyer)

SKU_Description → (SKU, Department, DeptBudgetCode, Buyer)

Buyer → (Department, DeptBudgetCode)

Department → DeptBudgetCode

Of the four determinants, both SKU and SKU_Description are candidate keys, but neither Department nor Buyer is a candidate key. Therefore, this relation is not in BCNF.

Figure 3-18

The Normalized STUDENT, CLUB, and PAYMENT Relations

STUDENT

	SID	Name
1	100	Jones
2	200	Chau
3	300	Garrett
4	400	Jones

CLUB

	Club	Cost
1	Climbing	150.00
2	Scuba	400.00
3	Skiing	550.00

PAYMENT

	SID	Club	AmountPaid
1	100	Scuba	0.00
2	200	Scuba	400.00
3	200	Skiing	550.00
4	300	Climbing	150.00
5	400	Skiing	550.00

Amazon Ex. 1008.0142

SKU_DATA_3

	SKU	SKU_Description	Department	DeptBudgetCode	Buyer
1	100100	Std. Scuba Tank, Yellow	Water Sports	BC-100	Pete Hansen
2	100200	Std. Scuba Tank, Magenta	Water Sports	BC-100	Pete Hansen
3	101100	Dive Mask, Small Clear	Water Sports	BC-100	Nancy Meyers
4	101200	Dive Mask, Med Clear	Water Sports	BC-100	Nancy Meyers
5	201000	Half-dome Tent	Camping	BC-200	Cindy Lo
6	202000	Half-dome Tent Footprint	Camping	BC-200	Cindy Lo
7	301000	Light Fly Climbing Harness	Climbing	BC-300	Jerry Martin
8	302000	Locking carabiner, Oval	Climbing	BC-300	Jerry Martin

Figure 3-19
Sample Data for the SKU_DATA_3 Relation

To normalize this table, we must transform this table into two or more tables that are in BCNF. In this case, there are two problematic functional dependencies. According to the note at the end of the procedure in Figure 3-13, we take the functional dependency whose determinant is not a candidate key and has the largest number of columns first. In this case, we take the columns of

Buyer → (Department, Dept_BudgetCode)

and place them in a table of their own.

Next, we make the determinant the primary key of the new table, remove all columns except Buyer from SKU_DATA_3, and make Buyer a foreign key of the new version of SKU_DATA_3, which we will name SKU_DATA_4. The results are:

BUYER (Buyer, Department, DeptBudgetCode)

SKU_DATA_4 (SKU, SKU_Description, *Buyer*)

We also create the referential integrity constraint:

SKU_DATA_4.Buyer must exist in BUYER.Buyer

The functional dependencies from SKU_DATA_4 are:

SKU → (SKU_Description, Buyer)

SKU_Description → (SKU, Buyer)

Because every determinant of SKU_DATA_4 is also a candidate key, the relationship is now in BCNF. Looking at the functional dependencies from BUYER we find:

Buyer → (Department, DeptBudgetCode)

Department → DeptBudgetCode

BUYER is not in BCNF because Department is a determinant that is not a candidate key. In this case, we must move (Department, DeptBudgetCode) into a table of its own. Following the procedure in Figure 3-13 and breaking BUYER into two tables (DEPARTMENT and BUYER_2) gives us a set of three tables:

DEPARTMENT (Department, DeptBudgetCode)

BUYER_2 (Buyer, *Department*)

SKU_DATA_4 (SKU, SKU_Description, *Buyer*)

with referential integrity constraints:

SKU_DATA_4.Buyer must exist in BUYER_2.Buyer

BUYER_2.Department must exist in DEPARTMENT.Department

The functional dependencies from all three of these tables are:

Department → DeptBudgetCode

Buyer → Department

SKU → (SKU_Description, Buyer)

SKU_Description → SKU, Buyer

Amazon Ex. 1008.0143

DEPARTMENT

	Department	DeptBudgetCode
1	Camping	BC-200
2	Climbing	BC-300
3	Water Sports	BC-100

BUYER_2

	Buyer	Department
1	Cindy Lo	Camping
2	Jerry Martin	Climbing
3	Nancy Meyers	Water Sports
4	Pete Hansen	Water Sports

SKU_DATA_4

	SKU	SKU_Description	Buyer
1	100100	Std. Scuba Tank, Yellow	Pete Hansen
2	100200	Std. Scuba Tank, Magenta	Pete Hansen
3	101100	Dive Mask, Small Clear	Nancy Meyers
4	101200	Dive Mask, Med Clear	Nancy Meyers
5	201000	Half-dome Tent	Cindy Lo
6	202000	Half-dome Tent Footprint	Cindy Lo
7	301000	Light Fly Climbing Harness	Jerry Martin
8	302000	Locking carabiner, Oval	Jerry Martin

Figure 3-20

The Normalized DEPARTMENT, BUYER_2, and SKU_DATA_4 Relations

At last, every determinant is a candidate key and all three of the tables are in BCNF. The resulting relations form these operations are shown in Figure 3-20.

Eliminating Anomalies from Multivalued Dependencies

All of the anomalies in the last section were due to functional dependencies. Anomalies can also arise from another kind of dependency—the multivalued dependency. A **multivalued dependency** occurs when a determinant is matched with a particular *set* of values.

Examples of multivalued dependencies are:

Employee →→ Degree

Employee →→ Sibling

PartKit →→ Part

Such expressions are read as "Employee multi-determines Degree" and "Employee multi-determines Sibling" and "PartKit multi-determines Part." Note that multideterminants are shown with a double arrow rather than a single arrow.

In each case, the determinant is associated with a set of values. Example data for each of these multidependencies are shown in Figure 3-21. Employee Jones, for example, has degrees AA and BS. Employee Greene has degrees BS, MS, and PhD. Employee Chau has just one degree, BS. Similarly, Employee Jones has siblings (brothers and sisters) Fred, Sally, and Frank. Employee Greene has sibling Nikki, and Employee Chau has siblings Jonathan and Eileen. Finally, PartKit Bike Repair has parts Wrench, Screwdriver, and Tube Fix. Other kits have parts as shown in Figure 3-21.

Unlike functional dependencies, the determinant of a multivalued dependency can never be the primary key. In all three of the tables in Figure 3-21, the primary key consists of the

Amazon Ex. 1008.0144

EMPLOYEE_DEGREE

	Employee	Degree
1	Chau	BS
2	Greene	BS
3	Greene	MS
4	Greene	PhD
5	Jones	AA
6	Jones	BS

EMPLOYEE_SIBLING

	Employee	Sibling
1	Chau	Eileen
2	Chau	Jonathan
3	Greene	Nikki
4	Jones	Frank
5	Jones	Fred
6	Jones	Sally

PARTKIT_PART

	PartKit	Part
1	Bike Repair	Screwdriver
2	Bike Repair	Tube Fix
3	Bike Repair	Wrench
4	First Aid	Aspirin
5	First Aid	Bandaids
6	First Aid	Elastic Band
7	First Aid	Ibuprofin
8	Vice	Extension Screw
9	Vice	Handle
10	Vice	Vice Jaw

Figure 3-21

Three Examples of Multivalued Dependencies

composite of the two columns in each table. The primary key of the EMPLOYEE_DEGREE table is the composite (Employee, Degree).

Multivalued dependencies pose no problem as long as they exist in tables of their own. None of the tables in Figure 3-21 have modification anomalies. However, if A $\rightarrow\!\!\rightarrow$ B, then any relation that contains A, B, and one or more additional columns will have modification anomalies.

For example, consider the situation if we combine the employee data in Figure 3-21 into a single EMPLOYEE_DEGREE_SIBLING table with three columns (Employee, Degree, Sibling), as shown in Figure 3-22.

Now, what actions need to be taken if student Jones earns an MBA? We must add three rows to the table. If we do not, if we only add the row ('Jones', 'MBA', 'Fred'), it will appear as if Jones is an MBA with her brother Fred, but not with her sister Sally or her other brother Frank. However, suppose Greene earns an MBA. Then we need only add one row ('Greene', 'MBA', 'Nikki'). But, if Chau earns an MBA, we need to add two rows. These are insertion anomalies. There are equivalent modification and deletion anomalies as well.

In Figure 3-22, we combined two multivalued dependencies into a single table and obtained modification anomalies. Unfortunately, we will also get anomalies if we combine a multivalued dependency with any other column, even if that other column has no multivalued dependency.

Figure 3-23 shows what happens when we combine the multivalued dependency

PartKit $\rightarrow\!\!\rightarrow$ Part

Amazon Ex. 1008.0145

EMPLOYEE_DEGREE_SIBLING

	Employee	Degree	Sibling
1	Chau	BS	Eileen
2	Chau	BS	Jonathan
3	Greene	BS	Nikki
4	Greene	MS	Nikki
5	Greene	PhD	Nikki
6	Jones	AA	Frank
7	Jones	BS	Frank
8	Jones	AA	Fred
9	Jones	BS	Fred
10	Jones	AA	Sally
11	Jones	BS	Sally

Figure 3-22

EMPLOYEE_DEGREE_SIBLING Relation with Two Multivalued Dependencies

with the functional dependency

PartKit →→ Price

For the data to be consistent, we must repeat the value of price for as many rows as each kit has parts. For this example, we must add three rows for the Bike Repair kit and four rows for the First Aid kit. The result is duplicated data that can cause data integrity problems.

Now you also know the problem with the relation in Figure 3-2. Anomalies exist in that table because it contains two multivalued dependencies:

BuyerName →→ SKU_Managed

BuyerName →→ CollegeMajor

Fortunately, it is easy to deal with multivalued dependencies: Put them into a table of their own. None of the tables in Figure 3-21 has modification anomalies, because each table consists of only the columns in a single, multivalued dependency. Thus, to fix the table in Figure 3-2, we must move BuyerName and SKU_Managed into one table and BuyerName and CollegeMajor into a second table:

PRODUCT_BUYER_SKU (BuyerName, SKU_Managed)

PRODUCT_BUYER_MAJOR (BuyerName, CollegeMajor)

The results are shown in Figure 3-24. If we want to maintain strict equivalence between these tables, we would also add the referential integrity constraint:

PRODUCT_BUYER_SKU.BuyerName must be identical to PRODUCT_BUYER_MAJOR.BuyerName

PARTKIT_PART_PRICE

	PartKit	Part	Price
1	Bike Repair	Screwdriver	14.95
2	Bike Repair	Tube Fix	14.95
3	Bike Repair	Wrench	14.95
4	First Aid	Aspirin	24.95
5	First Aid	Bandaids	24.95
6	First Aid	Elastic Band	24.95
7	First Aid	Ibuprofin	24.95
8	Vice	Extension Screw	125.00
9	Vice	Handle	125.00
10	Vice	Vice Jaw	125.00

Figure 3-23

PARTKIT_PART_PRICE Relation with a Functional Dependency and a Multivalued Dependency

Amazon Ex. 1008.0146

PRODUCT_BUYER_SKU

	BuyerName	SKU_Managed
1	Nancy Meyers	101100
2	Nancy Meyers	101200
3	Pete Hansen	100100
4	Pete Hansen	100200

PRODUCT_BUYER_MAJOR

	BuyerName	CollegeMajor
1	Nancy Meyers	Art
2	Nancy Meyers	Info Systems
3	Pete Hansen	Business

Figure 3-24

Placing the Two Multivalued Dependencies in Figure 3-2 into Separate Relations

This referential integrity constraint may not be necessary, depending on the requirements of the application.

Notice that when you put multivalued dependencies into a table of their own, they disappear. The result is just a table with two columns, and the primary key (and sole candidate key) is the composite of those two columns. When multivalued dependencies have been isolated in this way, the table is said to be in fourth normal form (4NF).

The hardest part of multivalued dependencies is finding them. Once you know they exist in a table, just move them into a table of their own. Whenever you encounter tables with odd anomalies, especially anomalies that require you to insert, modify, or delete different numbers of rows to maintain integrity, check for multivalued dependencies.

BY THE WAY You will sometimes hear people use the term *normalize* in phrases like, "that table has been normalized" or "check to see if those tables are normalized." Unfortunately, not everyone means the same thing with these words. Some people do not know about BCNF, and they will use it to mean tables in 3NF, which is a lesser form of normalization, one that allows for anomalies from functional dependencies that BCNF does not allow. Others use it to mean tables that are both BCNF and 4NF. Others may mean something else. The best choice is to use the term *normalize* to mean tables that are in both BCNF and 4NF.

Fifth Normal Form

There is a fifth normal form (5NF), also known as **Project-Join Normal Form (PJ/NF)**, which involves an anomaly where a table can be split apart but not correctly joined back together. However, the conditions under which this happens are complex, and generally if a relation is in 4NF, it is in 5NF. We will not deal with 5NF in this book. For more information about 5NF, start with the work by date cited earlier in this chapter, and the Wikipedia article at http://en.wikipedia.org/wiki/Fifth_normal_form.

Domain/Key Normal Form

As we have previously discussed in this chapter, in 1982, R. Fagin published a paper that that defined domain/key normal form (DK/NF). Fagin asked, "What conditions need to exist for a relation to have no anomalies?" He showed that a relation in DK/NF has no modification anomalies and, further, that a relation that has no modification anomalies is in DK/NF.

But what does this mean? Basically, DK/NF requires that all the constraints on the data values be logical implications of the definitions of domains and keys. To the level of detail in this text, and to the level of detail experienced by 99 percent of all database practitioners, this can

Amazon Ex. 1008.0147

be restated as follows: Every determinant of a functional dependency must be a candidate key. This, of course, is simply our definition of BCNF, and, for practical purposes, relations in BCNF are in DK/NF as well.

Summary

Databases arise from three sources: from existing data, from new systems development, and from the redesign of existing databases. This chapter and the next are concerned with databases that arise from existing data. Even though a table is a simple concept, certain tables can lead to surprisingly difficult processing problems. This chapter uses the concept of normalization to understand and possibly solve those problems. Figure 3-3 lists terms you should be familiar with.

A relation is a special case of a table; all relations are tables, but not all tables are relations. Relations are tables that have the properties listed in Figure 3-4. Three sets of terms are used to describe relation structure: (relation, attribute, tuple); (table, column, row); and (file, field, and record). Sometimes these terms are mixed and matched. In practice, the terms *table* and *relation* are commonly used synonymously, and we will do so for the balance of this text.

In a functional dependency, the value of one attribute, or attributes, determines the value of another. In the functional dependency A → B, attribute A is called the determinant. Some functional dependencies arise from equations, but many others do not. The purpose of a database is, in fact, to store instances of functional dependencies that do not arise from equations.

Determinants that have more than one attribute are called composite determinants. If A → (B, C), then A → B and A → C. However, if (A, B) → C, then, in general, neither A → C nor B → C.

If A → B, the values of A may or may not be unique in a relation. However, every time a given value of A appears, it will be paired with the same value of B. A determinant is unique in a relation only if it determines every other attribute of the relation. You cannot always rely on determining functional dependencies from sample data. The best idea is to verify your conclusions with the users of the data.

A key is a combination of one or more columns used to identify one or more rows. A composite key is a key with two or more attributes. A determinant that determines every other attribute is called a candidate key. A relation may have more than one candidate key. One of them is selected to be used by the DBMS for finding rows and is called the primary key. A surrogate key is an artificial attribute used as a primary key. The value of a surrogate key is supplied by the DBMS and has no meaning to the user. A foreign key is a key in one table that references the primary key of a second table. A referential integrity constraint is a limitation on data values of a foreign key that ensures that every value of the foreign key has a match to a value of a primary key.

The three kinds of modification anomalies are insert, update, and delete. Codd and others defined normal forms for describing different table structures that lead to anomalies. A table that meets the conditions listed in Figure 3-4 is in 1NF. Some anomalies arise from functional dependencies. Three forms, 2NF, 3NF, and BCNF, are used to treat such anomalies.

In this text, we are only concerned with the best of these forms, BCNF. If a relation is in BCNF, then no anomalies from functional dependencies can occur. A relation is in BCNF if every determinant is a candidate key.

Some anomalies arise from multivalued dependencies. A multidetermines B, or A →→ B, if A determines a set of values. If A multidetermines B, then any relation that contains A, B, and one or more other columns will have modification anomalies. Anomalies due to multivalued dependencies can be eliminated by placing the multivalued dependency in a table of its own. Such tables are in 4NF.

There is a 5NF, but generally tables in 4NF are in 5NF. DK/NF has been defined, but in practical terms the definition of DK/NF is the same as the definition of BCNF.

Key Terms

attribute
Boyce-Codd Normal Form (BCNF)
candidate key
composite determinant
composite key
data integrity problems
deletion anomaly
determinant
domain/key normal form (DK/NF)
entity
fifth normal form (5NF)
first normal form (1NF)
foreign key
fourth normal form (4NF)
functionally dependent
insertion anomaly
key
multivalued dependency

Amazon Ex. 1008.0148

non-prime attribute
normal forms
overlapping candidate key
partially dependent
primary key
Project-Join Normal Form (PJ/NF)
referential integrity constraint
relation
second normal form (2NF)
surrogate key
third normal form (3NF)
transitive dependency
tuple
update anomaly

Review Questions

3.1 Name three sources for databases.

3.2 What is the basic premise of this and the next chapter?

3.3 Explain what is wrong with the table in Figure 3-2.

3.4 Define each of the terms listed in Figure 3-3.

3.5 Describe the characteristics of a table that make it a relation.

3.6 Give an example of two tables that are not relations.

3.7 Suppose that two columns in two different tables have the same column name. What convention is used to give each a unique name?

3.8 Must all the values in the same column of a relation have the same length?

3.9 Explain the three different sets of terms used to describe tables, columns, and rows.

3.10 Explain the difference between functional dependencies that arise from equations and those that do not.

3.11 Intuitively, what is the meaning of the functional dependency

PartNumber → PartWeight

3.12 Explain the following statement: "The only reason for having relations is to store instances of functional dependencies."

3.13 Explain the meaning of the expression:

(FirstName, LastName) → Phone

3.14 What is a composite determinant?

3.15 If (A, B) → C , then can we also say that A → C?

3.16 If A → (B, C), then can we also say that A → B?

3.17 For the SKU_DATA table in Figure 3-1, explain why Buyer determines Department, but Department does not determine Buyer.

3.18 For the SKU_DATA table in Figure 3-1, explain why SKU_Description → (SKU, Department, Buyer).

3.19 If it is true that

PartNumber → PartWeight

does that mean that PartNumber will be unique in a relation?

3.20 Under what conditions will a determinant be unique in a relation?

3.21 What is the best test for determining whether a determinant is unique?

3.22 What is a composite key?

Amazon Ex. 1008.0149

3.23 What is a candidate key?

3.24 What is a primary key?

3.25 Explain the difference between a candidate key and a primary key.

3.26 What is a surrogate key?

3.27 Where does the value of a surrogate key come from?

3.28 When would you use a surrogate key?

3.29 What is a foreign key?

3.30 The term *domestic key* is not used. If it were used, however, what do you think it would mean?

3.31 What is a normal form?

3.32 Illustrate deletion, modification, and insertion anomalies on the STUDENT_ACTIVITY relation in Figure 3-17.

3.33 Explain why duplicated data leads to data integrity problems.

3.34 What relations are in 1NF?

3.35 Which normal forms are concerned with functional dependencies?

3.36 What conditions are required for a relation to be in 2NF?

3.37 What conditions are required for a relation to be in 3NF?

3.38 What conditions are required for a relation to be in BCNF?

3.39 If a relation is in BCNF, what can we say about it with regard to 2NF and 3NF?

3.40 What normal form is concerned with multivalued dependencies?

3.41 What is the premise of Fagin's work on DK/NF?

3.42 Summarize the three categories of normalization theory.

3.43 In general, how can you transform a relation not in BCNF into ones that are in BCNF?

3.44 What is a referential integrity constraint?

3.45 Explain the role of referential integrity constraints in normalization.

3.46 Why is an un-normalized relation like a paragraph with multiple themes?

3.47 In normalization example 3, why is the EXTENDED_PRICE relation "silly"?

3.48 In normalization example 4, under what conditions is

(SID, Club) → Cost

more accurate than

Club → Cost

3.49 If a determinant is part of a candidate key, is that good enough for BCNF?

3.50 In normalization example 5, why are the following two tables not correct?

DEPARTMENT (Department, DeptBudgetCode, Buyer)

SKU_DATA_4 (SKU, SKU_Description, *Department*)

3.51 How does a multivalued dependency differ from a functional dependency?

3.52 Consider the relation:

PERSON (Name, Sibling, ShoeSize)

Amazon Ex. 1008.0150

Assume that the following functional dependencies exist:

Name → Sibling
Name → ShoeSize

Describe deletion, modification, and insertion anomalies for this relation.

3.53 Place the PERSON relation into 4NF.

3.54 Consider the relation:

PERSON_2 (Name, Sibling, ShoeSize, Hobby)

Assume that the following functional dependencies exist:

Name → Sibling
Name → ShoeSize
Name → Hobby

Describe deletion, modification, and insertion anomalies for this relation.

3.55 Place the PERSON_2 relation into 4NF.

3.56 What is 5NF?

3.57 How do the conditions for DK/NF correspond to the conditions for BCNF?

Project Questions

3.58 Consider the table:

STAFF_MEETING (EmployeeName, ProjectName, Date)

The rows of this table record the fact that an employee from a particular project attended a meeting on a given date. Assume that a project meets at most once per day. Also, assume that only one employee represents a given project, but that employees can be assigned to multiple projects.

A. State the functional dependencies in STAFF_MEETING.

B. Transform this table into one or more tables in BCNF. State the primary keys, candidate keys, foreign keys, and referential integrity constraints.

C. Is your design in part B an improvement over the original table? What advantages and disadvantages does it have?

3.59 Consider the table:

STUDENT (Number, Name, Dorm, RoomType, DormCost, Club, ClubCost, Sibling, Nickname)

Assume that students pay different dorm costs, depending on the type of room they have, but that all members of a club pay the same cost. Assume that students can have multiple nicknames.

A. State any multivalued dependencies in STUDENT.

B. State the functional dependencies in STUDENT.

C. Transform this table into two or more tables such that each table is in BCNF and in 4NF. State the primary keys, candidate keys, foreign keys, and referential integrity constraints.

Amazon Ex. 1008.0151



A. Assume that Marcia keeps a table of data about her customers. Consider just the following part of that table:

CUSTOMER (Phone, FirstName, LastName)

Explain the conditions under which each of the following are true:

1. **Phone → (FirstName, LastName)**
2. **(Phone, FirstName) → LastName**
3. **(Phone, LastName) → FirstName**
4. **(LastName, FirstName) → Phone**
5. **Phone →→ LastName**
6. **Phone →→ FirstName**
7. **Phone →→ (FirstName, LastName)**

B. Is condition A.7 the same as conditions A.5 and A.6? Why or why not?

C. State an appropriate referential integrity constraint for the tables:

CUSTOMER (Phone, FirstName, LastName)

ORDER (OrderNumber, DateIn, DateOut, *Phone*)

D. Consider the tables:

CUSTOMER (Phone, FirstName, LastName)

ORDER (OrderNumber, DateIn, DateOut, *FirstName, LastName*)

What does the following referential integrity constraint mean?

ORDER.(FirstName, LastName) must be in CUSTOMER.(FirstName, LastName)

Is this constraint the same as the set of referential integrity constraints:

ORDER.FirstName must be in CUSTOMER.FirstName

ORDER.LastName must be in CUSTOMER.LastName

Explain why or why not.

E. Do you prefer the design in C or the design in D? Explain your reasoning.

F. Transform the following table into two or more tables in BCNF and 4NF. Indicate the primary keys, candidate keys, foreign keys, and referential integrity constraints. Make and state assumptions as necessary.

ORDER (CustomerNumber, FirstName, LastName, Phone, OrderNumber, DateIn, DateOut, ItemType, Quantity, ItemPrice, ExtendedPrice, SpecialInstructions)

G. Explain how your answer to question E changes depending on whether you assume that

CustomerNumber → (FirstName, LastName)

or

CustomerNumber →→ (FirstName, LastName)

Amazon Ex. 1008.0152



A. Morgan keeps a table of data about the stores from which he purchases. The stores are located in different countries and have different specialties. Consider the following relation:

STORE (Name, City, Country, Owner, Specialty)

Explain the conditions under which each of the following are true:

1. **Name → City**
2. **City → Name**
3. **City → Country**
4. **(Name, Country) → (City, Owner)**
5. **(City, Specialty) → Name**
6. **Owner →→ Name**
7. **Name →→ Specialty**

B. With regard to the relation in part A:

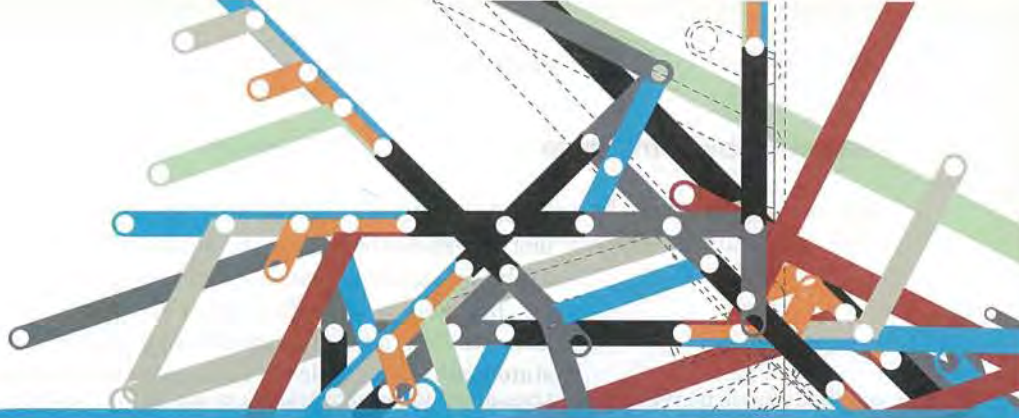
1. Specify which of the dependencies in part A seem most appropriate for a small import–export business.
2. Given your assumptions in B.1, transform the STORE table into a set of tables that are in both 4NF and BCNF. Indicate the primary keys, candidate keys, foreign keys, and referential integrity constraints.

C. Consider the relation:

SHIPMENT (ShipmentNumber, VendorName, VendorContact, VendorFax, DepartureDate, ArrivalDate, CountryOfOrigin, Destination, ShipmentCost, InsuranceValue, Insurer)

1. Write a functional dependency that expresses the fact that the cost of a shipment between two cities is always the same.
2. Write a functional dependency that expresses the fact the insurance value is always the same for a given vendor.
3. Write a functional dependency that expresses the fact the insurance value is always the same for a given vendor and country of origin.
4. Describe two possible multivalued dependencies in SHIPMENT.
5. State what you believe are reasonable functional dependencies for the SHIPMENT relation for a small import–export business.
6. State what you believe are reasonable multivalued dependencies for the SHIPMENT relation.
7. Using your assumptions in 5 and 6, transform SHIPMENT into a set of tables in BCNF and 4NF. Indicate the primary keys, candidate keys, foreign keys, and referential integrity constraints.

Amazon Ex. 1008.0153



4 Database Design Using Normalization

Chapter Objectives

- To design updatable databases to store data received from another source.
- To use SQL to access table structure.
- To understand the advantages and disadvantages of normalization.
- To understand denormalization.
- To design read-only databases to store data from updateable databases.
- To recognize and be able to correct common design problems:
  - The multivalue, multicolumn problem.
  - The inconsistent values problem.
  - The missing values problem.
  - The general-purpose remarks column problem

In Chapter 3, we defined the relational model, described modification anomalies, and discussed normalization using BCNF and 4NF. In this chapter, we apply those concepts to the design of databases that are created from existing data.

The premise of this chapter is that you have received, from some source, one or more tables of data that are to be stored in a new database. The question is, should that data be stored as is, or should it be transformed in some way before it is stored? Normalization theory plays an important role, as you will see.

Amazon Ex. 1008.0154

Assess Table Structure

When someone gives you a set of tables and asks you to construct a database to store them, your first step should be to assess the tables' structure and content. General guidelines for assessing a table's structure are summarized in Figure 4-1.

Begin by counting the number of rows in each table using the **SQL COUNT(*) function**. Then, to determine the number and type of the table's columns, use an **SQL SELECT * statement**. If your table has thousands or millions of rows, however, a full query will take considerable time. One way to limit the results of this query is to use the **SQL TOP keyword**. For example, to obtain all columns for the first 10 rows of the SKU_DATA table, you would code:

```
SELECT      TOP 10 *
FROM        SKU_DATA;
```

This query will show you all columns and data for 10 rows. If you want the top 50 rows, just use TOP 50 instead of TOP 10, and so on.

As shown in Figure 4-1, you should examine the data and determine the functional dependencies, multivalued dependencies, candidate keys, and each table's primary key. Also, look for possible foreign keys. Again, you can base your conclusions on sample data, but that data may not have all of the possible data cases. Therefore, verify your assumptions and conclusions with the users.

With regard to foreign keys, it is risky to assume that referential integrity constraints have been enforced on the data. Instead, check it yourself. For example, suppose you receive data for the following SKU_DATA and BUYER tables:

SKU_DATA (SKU, SKU_Description, Department, Buyer)

BUYER (BuyerName, Department)

After investigation, you learn that SKU is the primary key of SKU_DATA, and that BuyerName is the primary key of BUYER. You also think that SKU_DATA.Buyer is likely a foreign key to BUYER.BuyerName. The question is whether the following referential integrity constraint holds:

SKU_DATA.Buyer must exist in BUYER.BuyerName

You can use SQL to determine whether this is true. The following query will return any values of the foreign key that violate the constraint:

```
SELECT      Buyer
FROM        SKU_DATA
WHERE       Buyer NOT IN
            (SELECT     Buyer
             FROM       SKU_DATA, BUYER
             WHERE      SKU_DATA.Buyer = BUYER.BuyerName);
```

The subquery finds all values of Buyer for which there is a match between SKU_DATA.Buyer and BUYER.BuyerName. If there is any value of Buyer that is not in this subquery, then that

Figure 4-1

Guidelines for Assessing Table Structure

- Count rows and examine columns
- Examine data values and interview users to determine:
  - Multivalued dependencies
  - Functional dependencies
  - Candidate keys
  - Primary keys
  - Foreign keys
- Assess validity of assumed referential integrity constraints

Amazon Ex. 1008.0155

value will be displayed in the top query. All such values violate the referential integrity constraint.

After you have assessed the input tables, your next steps depend on whether you are creating an updatable database or a read-only database. We will consider updatable databases first.

Designing Updatable Databases

If you are constructing an updatable database, then you need to be concerned about modification anomalies and inconsistent data. Consequently, you must carefully consider normalization principles. Before we begin, let's first review the advantages and disadvantages of normalization.

Advantages and Disadvantages of Normalization

Figure 4-2 summarizes the advantages and disadvantages of normalization. On the positive side, normalization eliminates modification anomalies and reduces data duplication. Reduced data duplication eliminates the possibility of data integrity problems due to inconsistent data values. It also saves file space.

BY THE WAY Why do we say *reduce* data duplication rather than *eliminate* data duplication? The answer is that we cannot eliminate all duplicated data because we must duplicate data in foreign keys. We cannot eliminate Buyer, for example, from the SKU_DATA table, because we would then not be able to relate BUYER and SKU_DATA rows. Values of Buyer are thus duplicated in the BUYER and SKU_DATA tables.

This observation leads to a second question: If we only *reduce* data duplication, how can we claim to *eliminate* inconsistent data values? Data duplication in foreign keys will not cause inconsistencies, because referential integrity constraints prohibit them. As long as we enforce such constraints, the duplicate foreign key values will cause no inconsistencies.

On the negative side, normalization requires application programmers to write more complex SQL. To recover the original data, they must write subqueries and joins to connect data stored in separate tables. Also, with normalized data, the DBMS must read two or more tables, and this can mean slower application processing.

Functional Dependencies

As you learned in Chapter 3, we can eliminate anomalies due to functional dependencies by placing all tables in BCNF. Most of the time, the problems of modification anomalies are so great that you should put your tables into BCNF. There are exceptions, however, as you will see.

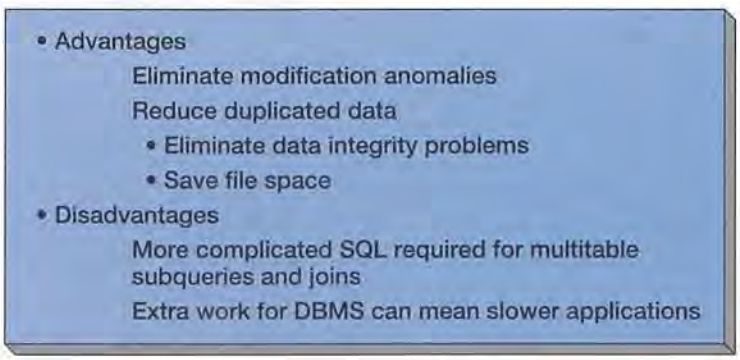


Figure 4-2
Advantages and Disadvantages of Normalization

Amazon Ex. 1008.0156

EQUIPMENT_REPAIR

	ItemNumber	Type	AcquisitionCost	RepairNumber	RepairDate	RepairCost
1	100	Drill Press	3500.00	2000	2009-05-05	375.00
2	200	Lathe	4750.00	2100	2009-05-07	255.00
3	100	Drill Press	3500.00	2200	2009-06-19	178.00
4	300	Mill	27300.00	2300	2009-06-19	1875.00
5	100	Drill Press	3500.00	2400	2009-07-05	0.00
6	100	Drill Press	5500.00	2500	2009-08-17	275.00

Figure 4-3
The EQUIPMENT_REPAIR Table

Normalizing with SQL

As we discussed in Chapter 3, a table is in BCNF if all determinants are candidate keys. If any determinant is not a candidate key, we must break the table into two or more tables. Consider an example. Suppose you are given the EQUIPMENT_REPAIR table in Figure 4-3 (the same table shown in Figure 3-10). In Chapter 3, we found that ItemNumber is a determinant, but not a candidate key. Consequently, we created the ITEM and REPAIR tables shown in Figure 4-4. In these tables, ItemNumber is a determinant and a candidate key of ITEM, and RepairNumber is a determinant and primary key of REPAIR; thus both tables are in BCNF.

Now, as a practical matter, how do we transform the data in the format in Figure 4-3 to that in Figure 4-4? To answer that question, we need to use the **SQL INSERT statement**. You will learn the particulars of the INSERT statement in Chapter 7. For now, we will jump ahead and use one version of it to illustrate the practical side of normalization.

First, we need to create the structure for the two new tables in Figure 4-4. If you are using Microsoft Access, you can follow the procedure in Appendix A to create the tables. Later, in Chapter 7, you will learn how to create tables using SQL, a process that works for all DBMS products.

Once the tables are created, you can fill them using the SQL INSERT command. To fill the ITEM table, we use:

```
INSERT INTO ITEM
   SELECT     DISTINCT ItemNumber, Type, AcquisitionCost
   FROM       EQUIPMENT_REPAIR;
```

Notice that we must use the DISTINCT keyword because the combination (ItemNumber, Type, AcquisitionCost) is not unique in the EQUIPMENT_REPAIR table. Once we have

Figure 4-4
The Normalized ITEM and REPAIR Relations

ITEM

	ItemNumber	Type	AcquisitionCost
1	100	Drill Press	3500.00
2	200	Lathe	4750.00
3	300	Mill	27300.00

REPAIR

	ItemNumber	RepairNumber	RepairDate	RepairCost
1	100	2000	2009-05-05	375.00
2	200	2100	2009-05-07	255.00
3	100	2200	2009-06-19	178.00
4	300	2300	2009-06-19	1875.00
5	100	2400	2009-07-05	0.00
6	100	2500	2009-08-17	275.00

Amazon Ex. 1008.0157

created the rows in ITEM, we can then use the following INSERT command to fill the rows of REPAIR:

```
INSERT INTO REPAIR
    SELECT    ItemNumber, RepairNumber, RepairDate, RepairAmount
    FROM      EQUIPMENT_REPAIR;
```

As you can see, the SQL statements for normalizing tables are relatively simple. After this transformation, we should probably remove the EQUIPMENT_REPAIR table. For now, you can do this using the graphical tools in Access, SQL Server, Oracle, or MySQL. In Chapter 7, you will learn how to remove tables using the **SQL DROP statement**. You will also learn how to use SQL to create the referential integrity constraint:

ItemNumber in REPAIR exists in ItemNumber in ITEM

If you want to try out this example, download the Microsoft Access 2007 database Equipment-Repair-Database.accdb from the text's Web site at www.pearsonhighered.com/kroenke. This database has the EQUIPMENT_REPAIR table with data. Create the new tables (see Appendix A) and then do the normalization by executing the SQL INSERT statements illustrated.

This process can be extended to any number of tables. We will consider richer examples of it in Chapter 7. For now, however, you should have the gist of the process.

Choosing Not to Use BCNF

Although in most cases the tables in an updatable database should be placed in BCNF, in some situations BCNF is just too pure. The classic example of unneeded normalization involves zip and similar postal codes. Consider the following table for customers in the United States:

CUSTOMER (CustomerID, LastName, FirstName, Street, City, State, Zip)

The functional dependencies of this table are:

CustomerID → (LastName, FirstName, Street, City, State, Zip)

Zip → (City, State)

This table is not in BCNF because Zip is a determinant that is not a candidate key. We can normalize this table as follows:

CUSTOMER_2 (CustomerID, LastName, FirstName, Street, *Zip*)

ZIP_CODE (Zip, City, State)

with referential integrity constraint:

CUSTOMER_2.Zip must exist in ZIP_CODE.Zip

The tables CUSTOMER_2 and ZIP_CODE are in BCNF, but consider these tables in light of the advantages and disadvantages of normalization listed in Figure 4-2. Normalization eliminates modification anomalies, but how often does zip code data change? How often does the post office change the city and state assigned to a zip code value? Almost never. The consequences on every business and person would be too severe. So, even though the design allows anomalies to occur, in practice, they will not occur because the data never change. Consider the second advantage: Normalization reduces data duplication, and hence improves data integrity. In fact, data integrity problems can happen in the single-table example if someone enters the wrong value for City, State, or Zip. In that case, the database will have inconsistent Zip values. But, normal business processes will cause zip code errors to be noticed, and they will be corrected without problem.

Now consider the disadvantages of normalization. Two separate tables require application programs to write more complex SQL. They also require the DBMS to process two tables, which may make the applications slow. Weighing the advantages and disadvantages, most practitioners would say that the normalized data are just too pure. Zip code data would therefore be left in the original table.

Amazon Ex. 1008.0158

In summary, when you design an updatable database from existing tables, examine every table to determine if it is in BCNF. If it is not, then the table is susceptible to modification anomalies and inconsistent data. In almost all cases, transform the table into tables that are in BCNF. However, if the data are never modified and if data inconsistencies will be easily corrected via the normal operation of business activity, then you may choose not to place the table into BCNF.

Multivalued Dependencies

Unlike functional dependencies, the anomalies from multivalued dependencies are so serious that multivalued dependencies should always be eliminated. Unlike BCNF, there is no gray area. Just place the columns of a multivalued dependency in tables of their own.

As shown in the last section, normalization is not difficult. It does mean that application programmers will have to write subqueries and joins to re-create the original data. Writing subqueries and joins, however, is *nothing* compared with the complexity of code that must be written to handle the anomalies due to multivalued dependencies.

Some experts might object to such a hard and fast rule, but it is justifiable. Although there may be a few rare, obscure, and weird cases in which multivalued dependencies are not problematic, such cases are not worth remembering. Until you have years of database design experience, always eliminate multivalued dependencies from any updatable table.

Designing Read-Only Databases

In the course of your career, you will likely be given tables of data and asked to create a read-only database. In fact, this task is commonly assigned to beginning database administrators.

Read-only databases are used in business intelligence (BI) systems for querying, reporting, and data mining applications, as you will learn in Chapter 13. Because such databases are never updated, the design guidelines and design priorities are different than those for updatable databases.

For several reasons, normalization is seldom an advantage for a read-only database. For one, if a database is never updated, then no modification anomalies can occur. Hence, considering Figure 4-2, the only reason to normalize a read-only database is to reduce data duplication. However, with no update activity, there is no risk of data integrity problems, so the only remaining reason to avoid duplicated data is to save file space.

Today, however, file space is exceedingly cheap, nearly free. So unless the database is enormous, the cost of storage is minimal. It is true that the DBMS will take longer to find and process data in large tables, so data might be normalized to speed up processing. But even that advantage is not clear-cut. If data are normalized, then data from two or more tables may need to be read, and the time required for the join may overwhelm the time savings of searching in small tables. In almost all cases, normalization of the tables in a read-only database is a bad idea.

Denormalization

Often the data for a read-only database are extracted from operational databases. Because such databases are updatable, they are probably normalized. Hence, you will likely receive the extracted data in normalized form. In fact, if you have a choice, ask for normalized data. For one, normalized data are smaller in size and can be transmitted to you more quickly. Also, if the data are normalized, it will be easier for you to reformat the data for your particular needs.

According to the last section, you probably do not want to leave the data in normalized form for a read-only database. If that is the case, you will need to **denormalize**, or join, the data prior to storage.

Consider the example in Figure 4-5. This is a copy of the normalized Student-Club-Payment data in Figure 3-18. Suppose that you are creating a read-only database that will be used to report amounts due for student club payments. If you store the data in this three-table form, every time someone needs to compare AmountPaid with Cost, he or she must join the three tables together.

Amazon Ex. 1008.0159

STUDENT

	SID	Name
1	100	Jones
2	200	Chau
3	300	Garrett
4	400	Jones

REPAIR

	Club	Cost
1	Climbing	150.00
2	Scuba	400.00
3	Skiing	550.00

PAYMENT

	SID	Club	AmountPaid
1	100	Scuba	0.00
2	200	Scuba	400.00
3	200	Skiing	550.00
4	300	Climbing	150.00
5	400	Skiing	550.00

Figure 4-5

The Normalized STUDENT, CLUB, and PAYMENT Relations

To do this, that person will need to know how to write a three-table join, and the DBMS will need to perform the join every time the report is prepared.

You can reduce the complexity of the SQL required to read these data and also reduce DBMS processing by joining the tables once and storing the joined result as a single table. The following SQL statement will join the three tables together and store them in a new table named PAYMENT_DATA:

```
INSERT INTO PAYMENT_DATA
   SELECT  STUDENT.SID, Name, CLUB.Club, Cost, AmtPaid
   FROM    STUDENT, PAYMENT, CLUB
   WHERE   STUDENT.SID = PAYMENT.SID
     AND   PAYMENT.Club = CLUB.Club;
```

As shown in Figure 4-6, the result of this join is the same as the original STUDENT_PAYMENT table.

As you can see, denormalization is simple. Just join the data together and store the joined result as a table. By doing this when you place the data into the read-only database, you save the application programmers from having to code joins for each application, and you also save the DBMS from having to perform joins and subqueries every time the users run a query or create a report.

Figure 4-6

The Denormalized STUDENT_ACTIVITY_PAYMENT_DATA Relation

STUDENT_ACTIVITY_PAYMENT_DATA

	SID	Name	Club	Cost	AmountPaid
1	100	Jones	Scuba	400.00	0.00
2	200	Chau	Scuba	400.00	400.00
3	200	Chau	Skiing	550.00	550.00
4	300	Garrett	Climbing	150.00	150.00
5	400	Jones	Skiing	550.00	550.00

Amazon Ex. 1008.0160



Figure 4-7
Columns in the PRODUCT Table

Customized Duplicated Tables

Because there is no danger of data integrity problems in a read-only database, and because the cost of storage today is miniscule, read-only databases are often designed with many copies of the same data, each copy customized for a particular application.

For example, suppose a company has a large PRODUCT table with the columns listed in Figure 4-7. The columns in this table are used by different business processes. Some are used for purchasing, some are used for sales analysis, some are used for displaying parts on a Web site, some are used for marketing, and some are used for inventory control.

The values of some of these columns, such as those for the picture images, are large. If the DBMS is required to read all of these data for every query, processing is likely to be slow. Accordingly, the organization might create several customized versions of this table for use by different applications. In an updatable database, so much duplicated data would risk severe data integrity problems, but for a read-only database there is no such risk.

Suppose for this example that the organization designs the following tables:

PRODUCT_PURCHASING (SKU, SKU_Description, VendorNumber, VendorName, VendorContact_1, VendorContact_2, VendorStreet, VendorCity, VendorState, VendorZip)

PRODUCT_USAGE (SKU, SKU_Description, QuantitySoldPastYear, QuantitySoldPastQuarter, QuantitySoldPastMonth)

PRODUCT_WEB (SKU, DetailPicture, ThumbnailPicture, MarketingShortDescription, MarketingLongDescription, PartColor)

PRODUCT_INVENTORY (SKU, PartNumber, SKU_Description, UnitsCode, BinNumber, ProductionKeyCode)

You can create these tables using the graphical design facilities of Access or another DBMS. Once the tables are created, they can be filled using INSERT commands similar to those already discussed. The only tricks are to watch for duplicated data and to use DISTINCT where necessary. See Review Question 4.10.

Amazon Ex. 1008.0161

Common Design Problems

Although normalization and denormalization are the primary considerations when designing databases from existing data, there are four additional practical problems to consider. These are summarized in Figure 4-8.

The Multivalue, Multicolumn Problem

The table in Figure 4-7 illustrates the first common problem. Notice the columns VendorContact_1 and VendorContact_2. These columns store the names of two contacts at the part vendor. If the company wanted to store the names of three or four contacts using this strategy, it would add columns VendorContact_3, VendorContact_4, and so forth.

Consider another example for an employee parking application. Suppose the EMPLOYEE_AUTO table includes basic employee data plus columns for license numbers for up to three cars. The following is the typical table structure:

EMPLOYEE (EmpNumber, Name, Email, Auto1_LicenseNumber, Auto2_LicenseNumber, Auto3_LicenseNumber)

Other examples of this strategy are to store employees' children's names in columns such as Child_1, Child_2, Child_3, and so forth, for as many children as the designer of the table thinks appropriate. Or, to store a picture of a house in a real estate application in columns labeled Picture_1, Picture_2, Picture_3, and so forth.

Storing multiple values in this way is convenient, but it has two serious disadvantages. The more obvious one is that the number of possible items is fixed. What if there are three contacts at a particular vendor? Where do we put the third name if only columns VendorContact_1 and VendorContact_2 are available? Or, if there are only three columns for child names, where do we put the name of the fourth child? And so forth.

The second disadvantage occurs when querying the data. Suppose we want to know the names of employees who have a child named Gretchen. If there are three child name columns, we must write:

```
SELECT      *
FROM        EMPLOYEE
WHERE       Child_1 = 'Gretchen'
   OR       Child_2 = 'Gretchen'
   OR       Child_3 = 'Gretchen';
```

Of course, if there are seven child names . . . well, you get the picture.

These problems can be eliminated by using a second table to store the multivalued attribute. For the employee–child case, the tables are:

EMPLOYEE (EmpNumber, EmpLastName, EmpFirstName, Email, . . . {other data})

CHILD (ChildFirstName, *EmpNumber*, . . . {other data})

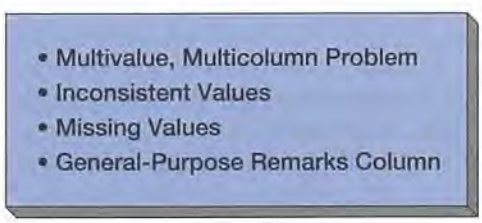


Figure 4-8 Practical Problems in Designing Databases from Existing Data

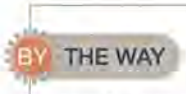
Amazon Ex. 1008.0162

Using this second structure, employees can have an unlimited number of children, and storage space will be saved for employees who have no children at all. Additionally, to find all of the employees who have a child named Gretchen, we can code:

```
SELECT      *
FROM        EMPLOYEE
WHERE       EmpNumber IN
            (SELECT     EmpNumber
             FROM       CHILD
             WHERE      ChildFirstName = 'Gretchen');
```

This second query is easier to write and understand and will work regardless of the number of children that an employee has.

The alternate design does require the DBMS to process two tables, and if the tables are large and performance is a concern one can argue that the original design is better. In such cases, storing multivalues in multiple columns may be preferred. Another, less valid objection to the two-table design is as follows: "We only need space for three cars because university policy restricts each employee to registering no more than three cars." The problem with this statement is that databases often outlive policies. Next year that policy may change, and, if it does, the database will need to be redesigned. As you will learn in Chapter 8, database redesign is tricky, complex, and expensive. It is better to avoid the need for a database redesign.



A few years ago, people argued that only three phone number columns are needed per person: Home, Office, and Fax. Later they said, "Well, OK, maybe we need four: Home, Office, Fax, and Mobile." Today, who would want to guess the maximum number of phone numbers a person might have? Rather than guess, just store Phone in a separate table; such a design will allow each person to have from none to an unlimited number of phone numbers.

You are likely to encounter the multivalue, multicolumn problem when creating databases from nondatabase data. It is particularly common in spreadsheet and text data files. Fortunately, the preferred two-table design is easy to create, and the SQL for moving the data to the new design is easy to write.



The multivalue, multicolumn problem is just another form of a multivalued dependency. For the parking application, for example, rather than store multiple rows in EMPLOYEE for each auto, multiple named columns are created in the table. The underlying problem is the same, however.

Inconsistent Values

Inconsistent values are a serious problem when creating databases from existing data. It occurs because different users or different data sources may use slightly different forms of the same data value. These slight differences may be hard to detect and will create inconsistent and erroneous information.

One of the hardest such problems occurs when different users have coded the same entries differently. One user may have coded a SKU_Description as Corn, Large Can; another may have coded the same item as Can, Corn, Large; and another may have coded the entry as Large Can Corn. Those three entries all refer to the same SKU, but they will be exceedingly difficult to

Amazon Ex. 1008.0163

reconcile. These examples are not contrived; such problems frequently occur, especially when combining data from different database, spreadsheet, and file sources.

A related, but simpler, problem occurs when entries are misspelled. One user may enter *Coffee*, another may enter *Coffeee*. They will appear as two separate products.

Inconsistent data values are particularly problematic for primary and foreign key columns. Relationships will be missing or wrong when foreign key data is coded inconsistently or misspelled.

Two techniques can be used to find such problems. One is the same as the check for referential integrity shown on page 128. This check will find values for which there is no match and will find misspellings and other inconsistencies.

Another technique is to use GROUP BY on the suspected column. For example, if we suspect that there are inconsistent values on SKU_Description in the SKU_DATA table, we can code:

```
SELECT      SKU_Description, COUNT(*) as NameCount
FROM        SKU_DATA
GROUP BY    SKU_Description;
```

The result of this query for the SKU_DATA values we have been using is:

	SKU_Description	NameCount
1	Dive Mask, Med Clear	1
2	Dive Mask, Small Clear	1
3	Half-dome Tent	1
4	Half-dome Tent Footprint	1
5	Light Fly Climbing Harness	1
6	Locking carabiner, Oval	1
7	Std. Scuba Tank, Magenta	1
8	Std. Scuba Tank, Yellow	1

In this case, there are no inconsistent values, but if there were, they would stand out. If the list resulting from the select is too long, groups can be selected that have just one or two elements using HAVING. Neither check is foolproof. Sometimes, you just have to read the data.

When working with such data, it is important to develop an error reporting and tracking system to ensure that inconsistencies that users do find are recorded and fixed. Users grow exceedingly impatient with data errors that persist after they have been reported.

Missing Values

Missing values are a third problem that occurs when creating databases from existing data. A missing value, or **null value**, is a value that has never been provided. It is not the same as a blank value, because a blank value is a value that is known to be blank. A null value is not known to be anything.

The problem with null values is ambiguity. A null value can indicate one of three conditions: The value is inappropriate; the value is appropriate but unknown; or the value is appropriate and known, but no one has entered it into the database. Unfortunately, we cannot tell from a null value which of these conditions is true.

Consider, for example, a null value for the column DateOfLastChildbirth in a PATIENT table. If a row represents a male patient, then the null occurs because the value is inappropriate; a male cannot give birth. Alternatively, if the patient is a female, but the patient has never been asked for the data, then the value is appropriate, but unknown. Finally, the null value could also mean that a date value is appropriate and known, but no one has recorded it into the database.

Amazon Ex. 1008.0164

You can use the SQL term IS NULL to check for null values. For example, to find the number of null values of Quantity in the ORDER_ITEM table, you can code:

```
SELECT    COUNT (*) as QuantityNullCount
FROM      ORDER_ITEM
WHERE     Quantity IS NULL;
```

The result of this query for the ORDER_ITEM values we have been using is:

	QuantityNullCount
1	0

In this case, there are no NULL values, but if there were, we would know how many, and then we could use a SELECT * statement to find the data of any row that has a null value.

When creating a database from existing data, if you try to define a column that has null values as the primary key, the DBMS will generate an error message. You will have to remove the nulls before creating the primary key. Also, you can tell the DBMS that a given column is not allowed to have null values, and when you import the data, if any row has a null value in that column, the DBMS will generate an error message. The particulars depend on the DBMS in use. See Chapter 10 for SQL Server 2008, Chapter 10A for Oracle Database 11g, and Chapter 10B for MySQL 5.1. You should form the habit of checking for null values in all foreign keys. Any row with a null foreign key will not participate in the relationship. That may or may not be appropriate; you will need to ask the users to find out. Also, null values can be problematic when joining tables together. You will learn how to deal with this problem in Chapter 7.

The General-Purpose Remarks Column

The general-purpose remarks column problem is common, serious, and very difficult to solve. Columns with names such as Remarks, Comments, and Notes often contain important data that are stored in an inconsistent, verbal, and verbose manner. Learn to be wary of columns with any such names.

To see why, consider customer data for a company that sells expensive items such as airplanes, rare cars, boats, or paintings. In a typical setting, someone has used a spreadsheet to track customer data. That person used a spreadsheet not because it was the best tool for such a problem, but rather because he or she had a spreadsheet program and knew how to use it.

The typical spreadsheet has columns like LastName, FirstName, Email, Phone, Address, and so forth. It almost always includes a column entitled Remarks, Comments, Notes, or something similar. The problem is that needed data are usually buried in such columns and nearly impossible to dig out. Suppose you want to create a database for a customer contact application for an airplane broker. Assume your design contains the two tables:

CONTACT (ID, LastName, FirstName, Address, . . . {other data}, *PlaneModel*)

AIRPLANE_MODEL (Model, Type, Description, . . . {other airplane data})

where CONTACT.PlaneModel is a foreign key to AIRPLANE_MODEL.Model. You want to use this relationship to determine who owns, has owned, or is interested in buying a particular model of airplane.

In the typical situation, the data for the foreign key has been recorded in the Remarks column. If you read the Remarks data, you will find entries like: 'Wants to buy a Piper Seneca II', 'Owner of a Piper Seneca II', and 'Possible buyer for a turbo Seneca'. All three of these rows should have a value of PlaneModel equal to 'Piper Seneca II', but you will pull your hair out making that determination.

Another problem with general-purpose remarks columns is that they are used inconsistently and contain multiple data items. One user may have used it to store the name of the spouse of the contact, another may have used it to store airplane models as just described, and a third may have used it to store the date the customer was last contacted. Or, the same user may have used it for all three purposes at different times!

Amazon Ex. 1008.0165

The best solution in this case is to identify all of the different purposes of the remarks column, create new columns for each of those purposes, and then extract the data and store it into the new columns as appropriate. However, this solution can seldom be automated.

In practice, all solutions require patience and hours of labor. Learn to be wary of such columns, and don't take such jobs on a fixed-price basis!

Summary

When constructing a database from existing data, the first step is to assess the structure and content of the input tables. Count the number of rows and use the SQL SELECT TOP 10 * phrase to learn the columns in the data. Then, examine the data and determine functional dependencies, multivalued dependencies, candidate keys, each table's primary key, and foreign keys. Check out the validity of possible referential integrity constraints.

Design principles differ depending on whether an updatable or read-only database is being constructed. If the former, then modification anomalies and inconsistent data are concerns. The advantages of normalization are elimination of modification anomalies, reduced data duplication, and the elimination of data inconsistencies. The disadvantages are that more complex SQL will be required and application performance may be slower.

For updatable databases, most of the time the problems of modification anomalies are so great that all tables should be placed in BCNF. SQL for normalization is easy to write. In some cases, if the data will be updated infrequently and if inconsistencies are readily corrected by business processes, then BCNF may be too pure and the tables should be left unnormalized. The problems of multivalued dependencies are so great that they should always be removed.

Read-only databases are created for reporting, querying, and data mining applications. Creating such a database is a task commonly assigned to beginners. When designing read-only databases, normalization is less desired. If input data is normalized, it frequently needs to be denormalized by joining it together and storing the joined result. Also, sometimes many copies of the same data are stored in tables customized for particular applications.

Four common problems occur when creating databases from existing data. The multivalue, multicolumn design sets a fixed number of repeating values and stores each in a column of its own. Such a design limits the number of items allowed and results in awkward SQL query statements. A better design results from putting multiple values in a table of their own.

Inconsistent values result when data arise from different users and applications. Inconsistent foreign key values create incorrect relationships. Data inconsistencies can be detected using SQL statements, as illustrated in this chapter. A null value is not the same as a blank. A null value is not known to be anything. Null values are a problem because they are ambiguous. They can mean that a value is inappropriate; unknown; or known, but not yet been entered into the database.

The general-purpose remarks column is a column that is used for different purposes. It collects data items in an inconsistent and verbose manner. Such columns are especially problematic if they contain data needed for a foreign key. Even if they do not, they often contain data for several different columns. Automated solutions are not possible, and the correction requires patience and labor.

Key Terms

denormalize
null value
SQL COUNT(*) function
SQL DROP statement
SQL INSERT statement
SQL SELECT * statement
SQL TOP keyword

Review Questions

4.1 Summarize the premise of this chapter.

4.2 When you receive a set of tables, what steps should you take to assess their structure and content?

4.3 Show SQL statements to count the number of rows and to list the top 15 rows of the RETAIL_ORDER table.

Amazon Ex. 1008.0166

4.4 Suppose you receive the following two tables:

DEPARTMENT (DepartmentName, BudgetCode)

EMPLOYEE (EmpNumber, LastName, FirstName, Email, DepartmentName)

and you conclude that EMPLOYEE.DepartmentName is a foreign key to DEPARTMENT.DepartmentName. Show SQL for determining whether the following referential integrity constraint has been enforced:

DepartmentName in EMPLOYEE must exist in DepartmentName in DEPARTMENT

4.5 Summarize how database design principles differ with regards to the design of updatable databases and the design of read-only databases.

4.6 Describe two advantages of normalized tables.

4.7 Why do we say that data duplication is only reduced? Why is it not eliminated?

4.8 If data duplication is only reduced, how can we say that the possibility of data inconsistencies has been eliminated?

4.9 Describe two disadvantages of normalized tables.

4.10 Suppose you are given the table:

EMPLOYEE_DEPARTMENT (EmpNumber, LastName, FirstName, Email, DepartmentName, BudgetCode)

and you wish to transform this table into the two tables:

EMPLOYEE (EmpNumber, LastName, FirstName, Email, DepartmentName)

and

DEPARTMENT (DepartmentName, BudgetCode)

Write the SQL statements needed for filling the EMPLOYEE and DEPARTMENT tables with data from EMPLOYEE_DEPARTMENT.

4.11 Summarize the reasons explained in this chapter for not placing zip code values into BCNF.

4.12 Describe a situation, other than the one for zip codes, in which one would choose not to place tables into BCNF. Justify your decision not to use BCNF.

4.13 According to this text, under what situations should you choose not to remove multivalued dependencies from a relation?

4.14 Compare the difficulty of writing subqueries and joins with the difficulty of dealing with anomalies caused by multivalued dependencies.

4.15 Describe three uses for a read-only database.

4.16 How does the fact that a read-only database is never updated influence the reasons for normalization?

4.17 For read-only databases, how persuasive is the argument that normalization reduces file space?

4.18 What is denormalization?

4.19 Suppose you are given the DEPARTMENT and EMPLOYEE tables in Review Question 4.10 and asked to denormalize them into the EMPLOYEE_DEPARTMENT relation. Show the design of the EMPLOYEE_DEPARTMENT relation. Write an SQL statement to fill this table with data.

4.20 Summarize the reasons for creating customized duplicated tables.

Amazon Ex. 1008.0167

4.21 Why are customized duplicated tables not used for updatable databases?

4.22 List four common design problems when creating databases from existing data.

4.23 Give an example of a multivalue, multicolumn table other than one discussed in this chapter.

4.24 Explain the problems in your example in Review Question 4.23.

4.25 Show how to represent the relation in your answer to Review Question 4.23 with two tables.

4.26 Show how the tables in your answer to Review Question 4.25 solve the problems you identified in Review Question 4.22.

4.27 Explain the following statement: "The multivalue, multicolumn problem is just another form of multivalued dependency." Show how this is so.

4.28 Explain ways in which inconsistent values arise.

4.29 Why are inconsistent values in foreign keys particularly troublesome?

4.30 Describe two ways to identify inconsistent values. Are these techniques certain to find all inconsistent values? What other step can be taken?

4.31 What is a null value?

4.32 How does a null value differ from a blank value?

4.33 What are three interpretations of null values? Use an example in your answer.

4.34 Show SQL for determining the number of null values in the column Name of the table EMPLOYEE.

4.35 Describe the general-purpose remarks column problem.

4.36 Give an example in which the general-purpose remarks column makes it difficult to obtain values for a foreign key.

4.37 Give an example in which the general-purpose remarks column causes difficulties when multiple values are stored in the same column. How is this problem solved?

4.38 Why should one be wary of general-purpose remarks columns?

Project Questions

The Elliot Bay Sports Club owns and operates three sports club facilities in Houston, Texas. Each facility has a large selection of modern exercise equipment, weight rooms, and rooms for yoga and other exercise classes. Elliot Bay offers 3-month and 1-year memberships. Members can use the facilities at any of the three club locations.

Elliot Bay maintains a roster of personal trainers who operate as independent consultants. Approved trainers can schedule appointments with clients at Elliot Bay facilities, as long as their client is a member of the club. Trainers also teach yoga, Pilates, and other classes. Answer the following questions, assuming you have been provided the following three tables of data (PT stands for personal trainer):

PT_SESSION (Trainer, Phone, Email, Fee, ClientLastName, ClientFirstName, ClientPhone, ClientEmail, Date, Time)

CLUB_MEMBERSHIP (ClientNumber, ClientLastName, ClientFirstName, ClientPhone, ClientEmail, MembershipType, EndingDate, Street, City, State, Zip)

CLASS (ClassName, Trainer, StartDate, EndDate, Time, DayOfWeek, Cost)

Amazon Ex. 1008.0168

4.39 Identify possible multivalued dependencies in these tables.

4.40 Identify possible functional dependencies in these tables.

4.41 Determine whether each table is in either BCNF or in 4NF. State your assumptions.

4.42 Modify each of these tables so that every table is in BCNF and 4NF. Use the assumptions you made in your answer to question 4.41.

4.43 Using these tables and your assumptions, recommend a design for an updatable database.

4.44 Add a table to your answer to question 4.43 that would allow Elliot Bay to assign members to particular classes. Include an AmountPaid column in your new table.

4.45 Recommend a design for a read-only database that would support the following needs:

A. Enable trainers to ensure that their clients are members of the club.

B. Enable the club to assess the popularity of various trainers.

C. Enable the trainers to determine if they are assisting the same client.

D. Enable class instructors to determine if the attendees to their classes have paid.



Marcia Wilson, the owner of Marcia's Dry Cleaning, is in the process of creating databases to support the operation and management of her business. For the past year, she and her staff have been using a cash register system that collects the following data:

SALE (InvoiceNumber, DateIn, DateOut, Total, Phone, FirstName, LastName)

Unfortunately, during rush times, not all of the data are entered, and there are many null values in Phone, FirstName, and LastName. In some cases all three are null, in other cases one or two are null. InvoiceNumber, DateIn, and Total are never null. DateOut has a few null values. Also, occasionally during a rush, phone number and name data have been entered incorrectly. To help create her database, Marcia purchased a mailing list from a local business bureau. The mailing list includes the following data:

HOUSEHOLD (Phone, FirstName, LastName, Street, City, State, Zip, Apartment)

In some cases, a phone number has multiple names. The primary key is thus the composite (Phone, FirstName, LastName). There are no null values in Phone, FirstName, and LastName, but there are some null values in the address data.

There are many names in SALE that are not in HOUSEHOLD, and there are many names in HOUSEHOLD that are not in SALE.

A. Design an updatable database for storing customer and sales data. Explain how to deal with the problems of missing data. Explain how to deal with the problems of incorrect phone and name data.

B. Design a read-only database for storing customer and sales data. Explain how to deal with the problems of missing data. Explain how to deal with the problems of incorrect phone and name data.

Amazon Ex. 1008.0169



Phillip Morgan, the owner of Morgan Importing, makes periodic buying trips to various countries. During the trips, he keeps notes about the items he purchases and basic data about their shipments. He hired a college student as an intern, and she transformed his notes into the spreadsheets in Figure 4-9. This is just sample data. Phillip has purchased hundreds of items over the years, and they have been shipped in dozens of different shipments.

Phillip wants to enter the information age, thus he has decided to develop a database of his inventory. He wants to keep track of the items he has purchased, their shipments, and eventually customers and sales. To get started, he has asked you to create a database for the data in Figure 4-9.

A. Follow the procedure shown in Figure 4-1 to assess these data. List all of the multivalued dependencies, functional dependencies, candidate keys, primary keys, and foreign keys. State any assumptions you make.

B. List questions you would ask Phillip to verify your assumptions.

C. Create tables as necessary to eliminate multivalued dependencies, if any.

D. The relationship between shipment and item data could possibly be inferred by matching values in the From cells to values in the City cells. Describe two problems with that strategy.

E. Describe a change to this spreadsheet that does express the shipment–item relationship.

F. Assume that Phillip wishes to create an updatable database from these data. Design tables you think are appropriate. State all referential integrity constraints.

G. Assume that Phillip wishes to create a read-only database from these data. Design tables you think are appropriate. State all referential integrity constraints.

H. Do these data have the multivalue, multicolumn problem? If so, how will you deal with it?

I. Do these data have the inconsistent data problem? If so, how will you deal with it?

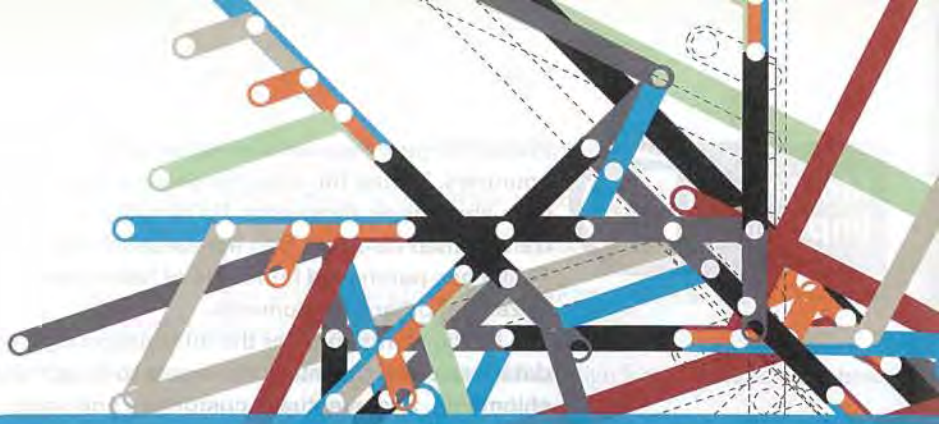
J. Do these data have a null value data problem? If so, how will you deal with it?

K. Do these data have the general-purpose remarks problem? If so, how will you deal with it?

Figure 4-9
Spreadsheets from Morgan Imports

	A	B	C	D	E	F	G	H	I
1	ShipmentNumber	Shipper	Phone	Contact	From	Departure	Arrival	Contents	InsuredValue
2	49100300	Wordwide	800-123-4567	Jose	Philippines	5/5/2009	6/17/1999	QE dining set, large bureau, porcelain lamps	$27,500
3	488955	Intenational	800-123-8898	Marilyn	Singapore	6/2/2009		Miscellaneous linen, large masks, 14 setting Willow design china	$7,500
4	84899440	Wordwide	800-123-4567	Jose	Peru	7/3/2009	7/28/2009	Woven goods, antique leather chairs	
5	399400	Intenational	800-123-8898	Marilyn	Singaporeee	8/5/2009	9/11/2009	Large bureau, brass lamps, willow design serving dishes	$18,000
6									
7									
8									
9		Item	Date	City	Store	Salesperson	Price		
10		Willow Serving Dishes	7/15/2009	Singapore	Jade Antiques	Swee Lai	$4,500		
11		Large bureau	7/17/2009	Singapore	Eastern Sales	Jeremey	$9,500		
12		Brass lamps	7/20/2009	Singapore	Jade Antiques	Mr. James	$1,200		
13		QE Dining Set	4/7/2009	Manila	E. Treasures	Gracielle	$14,300		
14									
15									

Amazon Ex. 1008.0170



5 Data Modeling with the Entity-Relationship Model

Chapter Objectives

- To understand the two-phase data modeling/database design process.
- To understand the purpose of the data modeling process.
- To understand entity-relationship (E-R) diagrams.
- To be able to determine entities, attributes, and relationships.
- To be able to create entity identifiers.
- To be able to determine minimum and maximum cardinalities.
- To understand variations of the E-R model.
- To understand and be able to use ID-dependent and other weak entities.
- To understand and be able to use supertype/subtype entities.
- To understand and be able to use strong entity patterns.
- To understand and be able to use the ID-dependent association pattern.
- To understand and be able to use the ID-dependent multivalued attribute pattern.
- To understand and be able to use the ID-dependent archetype/instance pattern.
- To understand and be able to use the line-item pattern.
- To understand and be able to use the for-use-by pattern.
- To understand and be able to use recursive patterns.
- To understand the iterative nature of the data modeling process.
- To be able to use the data modeling process.

In this chapter and the next, we consider the design of databases that arise from the development of new information systems. As you will learn, such databases are designed by analyzing requirements and creating a data model, or blueprint, of a database that will meet those requirements. The data model is then transformed into a database design.

Amazon Ex. 1008.0171

This chapter addresses the creation of data models using the entity-relationship data model, the most popular modeling technique. This chapter consists of three major sections. First, we explain the major elements of the entity-relationship model and briefly describe several variations on that model. Next, we examine a number of patterns in forms, reports, and data models that you will encounter when data modeling. We then illustrate the data modeling process using the example of a small database at a university. Before starting, however, you need to understand the purpose of a data model.

The Purpose of a Data Model

A **data model** is a plan, or blueprint, for a database design. By analogy, consider the construction of your dorm or apartment building. The contractor did not just buy some lumber, call for the concrete trucks, and start work. Instead, an architect constructed plans and blueprints for that building long before construction began. If, during the planning stage, it was determined that a room was too small or too large, the blueprint could be changed simply by redrawing the lines. If, however, the need for change occurs after the building is constructed, the walls, electrical system, plumbing, and so on will need to be rebuilt, at great expense and loss of time. It is easier, simpler, and faster to change the plan than it is to change a constructed building.

The same argument applies to data models and databases. Changing a relationship during the data modeling stage is just a matter of changing the diagram and related documentation. Changing a relationship after the database and applications have been constructed, however, is much more difficult. Data must be migrated to the new structure, SQL statements will need to be changed, forms and reports will need to be altered, and so forth.

The Entity-Relationship Model

Dozens of different tools and techniques for constructing data models have been defined over the years. They include the hierarchical data model, the network data model, the ANSI/SPARC data model, the entity-relationship data model, the semantic object model, and many others. Of these, the entity-relationship data model has emerged as the standard data model, and we will consider only that data model in this chapter.

The **entity-relationship (E-R) model** was first published by Peter Chen in 1976.[1] In this paper, Chen set out the basic elements of the model. Subtypes (discussed later) were added to the E-R model to create the **extended E-R model**,[2] and today it is the extended E-R model that most people mean when they use the term *E-R model*. In this text, we will use the extended E-R model.

Entities

An **entity** is something that users want to track. It is something that is readily identified in the users' work environment. Example entities are EMPLOYEE Mary Lai, CUSTOMER 12345, SALES-ORDER 1000, SALESPERSON Wally Smith, and PRODUCT A4200. Entities of a given type are grouped into an **entity class**. Thus, the EMPLOYEE entity class is the collection of all EMPLOYEE entities. In this text, entity classes are shown in capital letters.

It is important to understand the differences between an entity class and an entity instance. An entity class is a collection of entities and is described by the structure of the entities in that class. An **entity instance** of an entity class is the occurrence of a particular entity, such as CUSTOMER 12345. An entity class usually has many instances of an entity. For example, the

[1] Chen, Peter P. "The Entity-Relationship Model—Towards a Unified View of Data." *ACM Transactions on Database Systems*, January 1976, pp. 9–36.
[2] Teorey, T. J., D. Yang, and J. P. Fry. "A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model," *ACM Computing Surveys*, June 1986, pp. 197–222.

Amazon Ex. 1008.0172

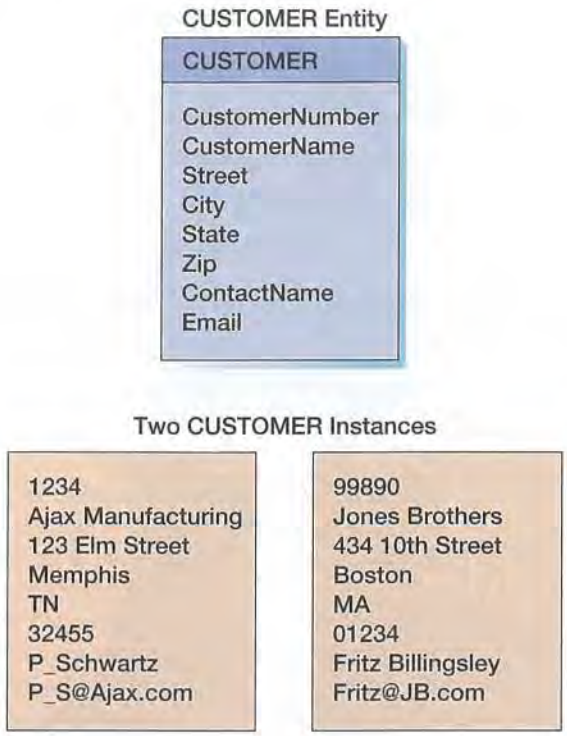


Figure 5-1
CUSTOMER Entity and Two Entity Instances

entity class CUSTOMER has many instances—one for each customer represented in the database. The CUSTOMER entity class and two of its instances are shown in Figure 5-1.

Attributes

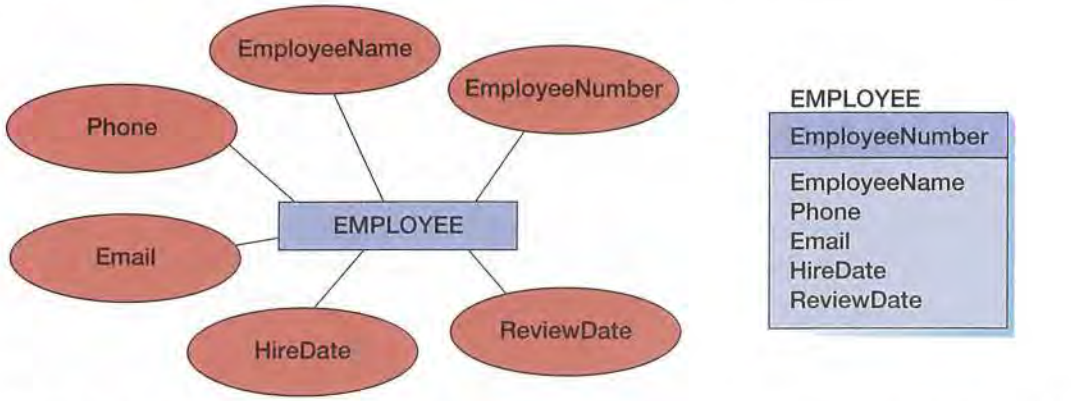
Entities have **attributes** that describe their characteristics. Examples of attributes are EmployeeNumber, EmployeeName, Phone, and Email. In this text, attributes are written in both uppercase and lowercase letters. The E-R model assumes that all instances of a given entity class have the same attributes.

Figure 5-2 shows two different ways of displaying the attributes of an entity. Figure 5-2(a) shows attributes in ellipses that are connected to the entity. This style was used in the original E-R model, prior to the advent of data modeling software products. Figure 5-2(b) shows the rectangle style that is commonly used by data modeling software products today.

Identifiers

Entity instances have **identifiers**, which are attributes that name, or identify, entity instances. For example, EMPLOYEE instances can be identified by EmployeeNumber, SocialSecurityNumber,

Figure 5-2
Variations on Entity Diagram Attribute Displays



(a) Attributes in Ellipses

(b) Attributes in Rectangle

Amazon Ex. 1008.0173

or EmployeeName. EMPLOYEE instances are not likely to be identified by attributes such as Salary or HireDate because these attributes are not normally used in a naming role. Similarly, customers can be identified by CustomerNumber or CustomerName, and sales orders can be identified by OrderNumber.

The identifier of an entity instance consists of one or more of the entity's attributes. Identifiers that consist of two or more attributes are called **composite identifiers**. Examples are (AreaCode, LocalNumber), (ProjectName, TaskName), and (FirstName, LastName, DateOfHire).

BY THE WAY Notice the correspondence of identifiers and keys. The term *identifier* is used in a data model, and the term *key* is used in a database design. Thus, entities have identifiers, and tables (or relations) have keys. Identifiers serve the same role for entities that keys serve for tables.

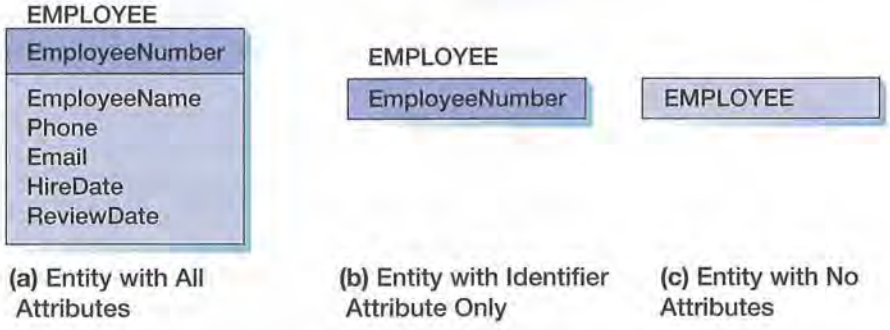
As shown in Figure 5-3, entities are portrayed in three levels of detail in a data model. As shown in Figure 5-3(a), sometimes the entity and all of its attributes are displayed. In such cases, the identifier of the attribute is shown at the top of the entity and a horizontal line is drawn after the identifier, However, in a large data model, so much detail can make the data model diagrams unwieldy. In those cases, the entity diagram is abbreviated by showing just the identifier, as in Figure 5-3(b), or by showing just the name of the entity in a rectangle, as shown in Figure 5-3(c). All three techniques are used in practice; the more abbreviated form in Figure 5-3(c) is used to show the big picture and overall entity relationships. The more detailed view in Figure 5-3(a) is frequently used during database design. Most data modeling software products have the ability to show all three displays.

Relationships

Entities can be associated with one another in **relationships**. The E-R model contains both relationship classes and relationship instances.[3] **Relationship classes** are associations among entity classes, and **relationship instances** are associations among entity instances. In the original E-R model, relationships could have attributes. Today, that feature is no longer used.

Relationships are given names that describe the nature of the relationship, as shown in Figure 5-4. In Figure 5-4(a), the Qualification relationship shows which employees have which skills. In Figure 5-4(b), the Assignment relationship shows which combinations of clients, architects, and projects have been created. To avoid unnecessary complexity, in this chapter, we will show the names of relationships only if there is a chance of ambiguity.

Figure 5-3
Variations on Level of Entity Attribute Displays



(a) Entity with All Attributes

(b) Entity with Identifier Attribute Only

(c) Entity with No Attributes

[3] For brevity, we sometimes drop the word *instance* when the context makes it clear that an instance rather than an entity class is involved.

Amazon Ex. 1008.0174

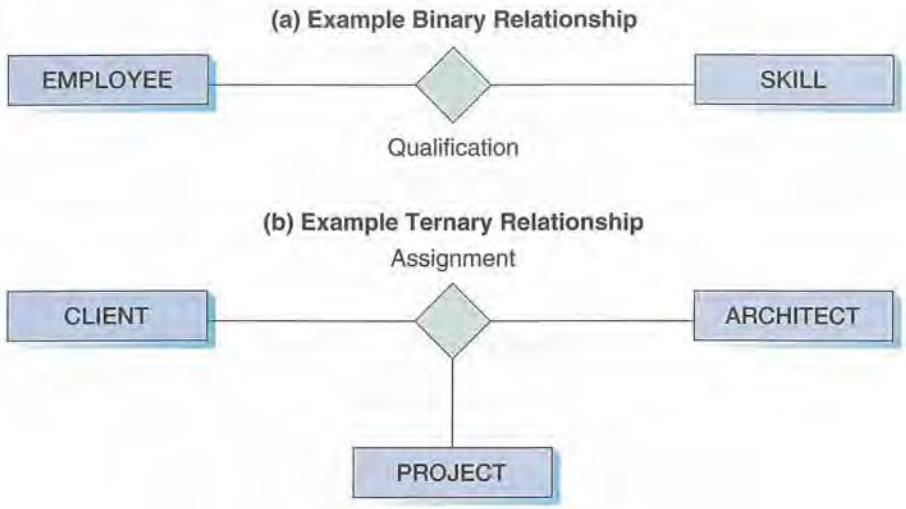


Figure 5-4
Binary Versus Ternary Relationships

BY THE WAY Your instructor may believe that it is important to always show the name of a relationship. If so, be aware that you can name a relationship from the perspective of either of the entities or both. For example, you can name the relationship between DEPARTMENT and EMPLOYEE as Department Consists Of; or you can name it as Employee Works In; or you can name it both ways, using a slash between the two names, Department Consists Of/Employee Works In. Relationship names are a necessity when there are two different relationships between the same two entities.

A relationship class can involve two or more entity classes. The number of entity classes in the relationship is the **degree** of the relationship. In Figure 5-4(a), the Qualification relationship is of degree two because it involves two entity classes: EMPLOYEE and SKILL. In Figure 5-4(a), the Assignment relationship is of degree three because it involves three entity classes: CLIENT, ARCHITECT, and PROJECT. Relationships of degree two are referred to as **binary relationships**. Similarly, relationships of degree three are called **ternary relationships**.

When transforming a data model into a relational design, relationships of all degrees are treated as combinations of binary relationships. The Assignment relationship in Figure 5-4(b), for example, is decomposed into three binary relationships (can you spot them?). Most of the time, this strategy is not a problem. However, some nonbinary relationships need additional work, as you will learn in Chapter 6. All data modeling software products require you to express relationships as binary relationships.

BY THE WAY At this point, you may be wondering, "What's the difference between an entity and a table?" So far, they seem like different terms for the same thing. *The principle difference between an entity and a table is you can express a relationship between entities without using foreign keys*. In the E-R model, you can specify a relationship just by drawing a line connecting two entities. Because you are doing logical data modeling and not physical database design, you need not worry about primary and foreign keys, referential integrity constraints, and the like. Most data modeling products will allow you to consider such details if you choose to, but they do not require it.

This characteristic makes entities easier to work with than tables, especially early in a project when entities and relationships are fluid and uncertain. You can show relationships between entities before you even know what the identifiers are. For example, you can say that a DEPARTMENT relates to many EMPLOYEEs before you know any of the attributes of either EMPLOYEE or DEPARTMENT. This characteristic enables you to work from the general to the specific. First identify the entities, then think about relationships, and finally, determine the attributes.

Amazon Ex. 1008.0175

In the entity-relationship model, relationships are classified by their **cardinality**, a word that means "count." The **maximum cardinality** is the maximum number of entity instances that can participate in a relationship instance. The **minimum cardinality** is the minimum number of entity instances that must participate in a relationship instance.

Maximum Cardinality

In Figure 5-5, the maximum cardinality is shown inside the diamond that represents the relationship. The three parts of this figure show the three basic maximum cardinalities in the E-R model.

Figure 5-5(a) shows a **one-to-one (abbreviated 1:1) relationship**. In a 1:1 relationship, an entity instance of one type is related to at most one entity instance of the other type. The Employee_Identity relationship in Figure 5-5(a) associates one EMPLOYEE instance with one BADGE instance. According to this diagram, no employee has more than one badge, and no badge is assigned to more than one employee.

The Computer_Assignment relationship in Figure 5-5(b) illustrates a **one-to-many (abbreviated 1:N) relationship**. Here, a single instance of EMPLOYEE can be associated with many instances of COMPUTER, but a COMPUTER instance is associated with just one instance of EMPLOYEE. According to this diagram, an employee can be associated with several computers, but a computer is assigned to just one employee.

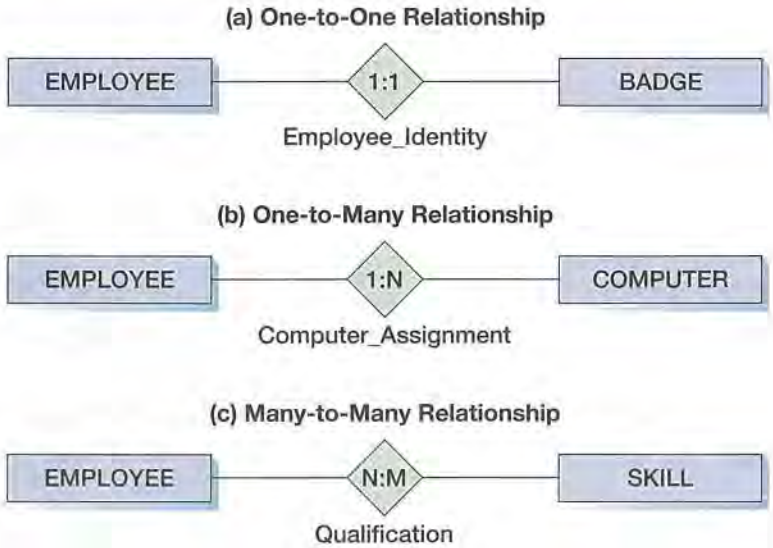
The positions of the 1 and the N are significant. The 1 is close to the line connecting EMPLOYEE, which means that the 1 refers to the EMPLOYEE side of the relationship. The N is close to the line connecting COMPUTER, which means that the N refers to the COMPUTER side of the relationship. If the 1 and the N were reversed and the relationship were written N:1, an EMPLOYEE would have one COMPUTER, and a COMPUTER would be assigned to many EMPLOYEEs.

When discussing one-to-many relationships, the terms **parent** and **child** are sometimes used. The *parent* is the entity on the 1 side of the relationship, and the *child* is the entity on the many side of the relationship. Thus, in a 1:N relationship between DEPARTMENT and EMPLOYEE, DEPARTMENT is the parent and EMPLOYEE is the child (one DEPARTMENT has many EMPLOYEEs).

Figure 5-5(c) shows a **many-to-many (abbreviated N:M) relationship**. According to the Qualification relationship, an EMPLOYEE instance can be associated with many SKILL instances, and a SKILL instance can be associated with many EMPLOYEE instances. This relationship documents that fact that an employee may have many skills, and a skill may be held by many employees.

Sometimes students wonder why we do not write many-to-many relationships as N:N or M:M. The reason is that cardinality in one direction may be different than the cardinality in the other direction. In other words, in an N:M relationship, N need not equal M. An EMPLOYEE can

Figure 5-5
Three Types of Maximum Cardinality



Amazon Ex. 1008.0176

have five skills, for example, but one of those skills can have three employees. Writing the relationship as N:M highlights the possibility that the cardinalities may be different.

Sometimes the maximum cardinality is an exact number. For example, for a sports team, the number of players on the roster is limited to some fixed number, say, 15. In that case, the maximum cardinality between TEAM and PLAYER would be set to 15 rather than to the more general N.

BY THE WAY Relationships like those in Figure 5-5 are sometimes called **HAS-A relationships**. This term is used because each entity instance has a relationship to a second entity instance. An employee has a badge, and a badge has an employee. If the maximum cardinality is greater than one, then each entity has a set of other entities. An employee has a set of skills, for example, and a skill has a set of employees who have that skill.

Minimum Cardinality

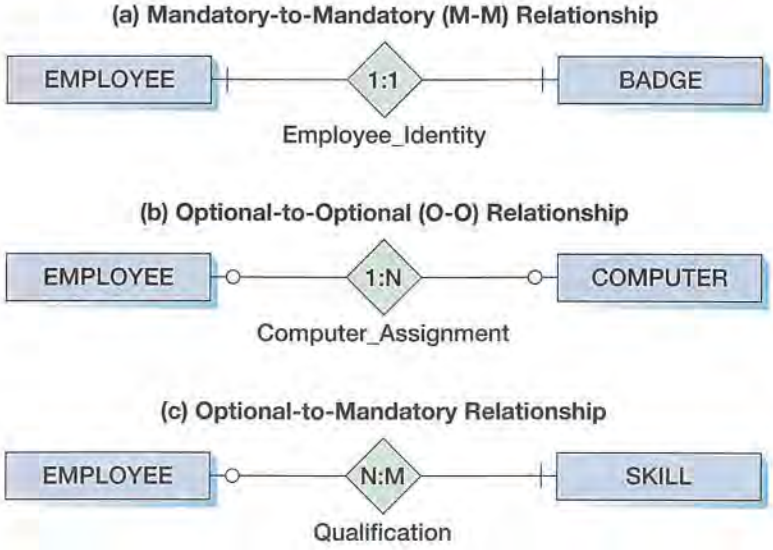
The minimum cardinality is the number of entity instances that must participate in a relationship. Generally, minimums are stated as either zero or one. If zero, then participation in the relationship is optional. If one, then at least one entity instance must participate in the relationship. In E-R diagrams, an optional relationship is represented by a small circle on the relationship line; a mandatory relationship is represented by a hash mark or line across the relationship line.

To better understand these terms, consider Figure 5-6. In the Employee_Identity relationship in Figure 5-6(a), the hash marks indicate that an EMPLOYEE is required to have a BADGE, and a BADGE must be allocated to an EMPLOYEE. Such a relationship is referred to as a **mandatory-to-mandatory (M-M) relationship**, because entities are required on both sides. The complete specification for the Employee_Identity relationship is that it is a 1:1, M-M relationship.

In Figure 5-6(b), the two small circles indicate that the Computer_Assignment relationship is an **optional-to-optional (O-O) relationship**. This means that an EMPLOYEE need not have a COMPUTER, and a COMPUTER need not be assigned to an EMPLOYEE. The Computer_Assignment relationship is thus a 1:N, O-O relationship.

Finally, in Figure 5-6(c) the combination of a circle and a hash mark indicates as an **optional-to-mandatory (O-M) relationship**. Here, an EMPLOYEE must be assigned to at least one SKILL, but a SKILL may not necessarily be related to any EMPLOYEE. The complete specification for the Qualification relationship is thus an N:M, O-M relationship. The position

Figure 5-6
Minimum Cardinality Examples



Amazon Ex. 1008.0177

of the circle and the hash mark are important. Because the circle is in front of EMPLOYEE, it means that the employee is optional in the relationship.

> **BY THE WAY** Sometimes when interpreting diagrams like Figure 5-6(c) students become confused about which entity is optional and which is required. An easy way to clarify this situation is to imagine that you are standing in the diamond on the relationship line. Imagine looking toward one of the entities. If you see an oval in that direction, then that entity is optional; if you see a hash mark, then that entity is required. Thus, in Figure 5-6(c), if you stand on the diamond and look toward SKILL, you see a hash mark. This means that SKILL is required in the relationship.

A fourth option, a **mandatory-to-optional (M-O) relationship** is not shown in Figure 5-6. But, if we exchange the circle and the hash mark in Figure 5-6(c), then Qualification becomes an M-O relationship. In that case, an EMPLOYEE need not have a SKILL, but a SKILL must have at least one EMPLOYEE.

As with maximum cardinalities, in rare cases the minimum cardinality is a specific number. To represent the relationship between PERSON and MARRIAGE, for example, the minimum cardinality would be 2:Optional.

Entity-Relationship Diagrams and Their Versions

The diagrams in Figures 5-5 and 5-6 are sometimes referred to as **entity-relationship (E-R) diagrams**. The original E-R model specified that such diagrams use diamonds for relationships, rectangles for entities, and connected ellipses for attributes, as shown in Figure 5-2. You may still see examples of such E-R diagrams, and it is important for you to be able to interpret them.

For two reasons, however, this original notation is seldom used today. First, there are a number of different versions of the E-R model, and these versions use different symbols. Second, data modeling software products use different techniques. For example, Computer Associates' ERwin product uses one set of symbols, and Microsoft Visio uses a second set.

Variations of the E-R Model

At least three different versions of the E-R model are in use today. One of them, **Information Engineering (IE)**, was developed by James Martin in 1990. This model uses crow's feet to show the many side of a relationship, and it is called the **IE Crow's foot model**. It is easy to understand, and we will use it throughout this text. In 1993, the National Institute of Standards and Technology announced another version of the E-R model as a national standard. This version is called **Integrated Definition 1, Extended (IDEF1X)**.[4] This standard incorporates the basic ideas of the E-R model, but uses different graphical symbols. Although this model is a national standard, it is difficult to understand and use. As a national standard, however, it is used in government, and thus it may become important to you. Therefore, the fundamentals of the IDEF1X model are described in Appendix B.

Meanwhile, to add further complication, a new object-oriented development methodology called the **Unified Modeling Language (UML)** adopted the E-R model, but introduced its own symbols while putting an object-oriented programming spin on the model. UML notation is summarized in Appendix C.

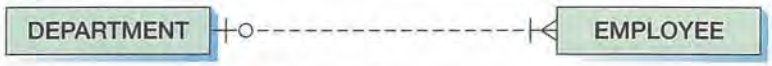
> **BY THE WAY** In addition to differences due to different versions of the E-R model, there also are differences due to software products. For example, two products that both implement the IE Crow's Foot model may do so in different ways. The result is a mess. When creating a data model diagram, you need to know not just the version of the E-R model you are using, but also the idiosyncrasies of the data modeling product you use.

[4] *Integrated Definition for Information Modeling (IDEF1X)*. Federal Information Processing Standards Publication 184, 1993.

Amazon Ex. 1008.0178



(a) Original E-R Model Version



(b) Crow's Foot Version

Figure 5-7
Minimum Cardinality Examples

E-R Diagrams Using the IE Crow's Foot Model

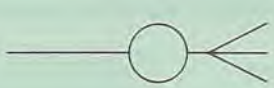
Figure 5-7 shows two versions of a one-to-many, optional-to-mandatory relationship. Figure 5-7(a) shows the original E-R model version. Figure 5-7(b) shows the crow's foot model using common crow's foot symbols. Notice that the relationship is drawn as a dashed line. The reason for this will be explained later in this chapter. For now, notice the **crow's foot symbol** used to show the many side of the relationship.

The crow's foot model uses the notation shown in Figure 5-8 to indicate the relationship cardinality. The symbol closest to the entity shows the maximum cardinality, and the other symbol shows the minimum cardinality. A hash mark indicates one (and therefore also mandatory), a circle indicates zero (and thus optional), and the crow's foot symbol indicates many. Thus, the diagram in Figure 5-7(b) means that a DEPARTMENT has one or more EMPLOYEEs (the symbol shows many and mandatory), and an EMPLOYEE belongs to zero or one DEPARTMENTs (the symbol shows one and optional).

A 1:1 relationship would be drawn in a similar manner, but the line connecting to each entity should be similar to the connection shown for the one side of the 1:N relationship in Figure 5-7(b).

Figure 5-9 shows two versions of an N:M, optional to mandatory relationship. Modeling N:M relationships presents some complications. According to the original E-R model diagram shown in Figure 5-9(a), an EMPLOYEE must have at least one SKILL and may have several. At

Figure 5-8
Crow's Foot Notation

Symbol	Meaning
	One—Mandatory
	Many—Mandatory
	One—Optional
	Many—Optional

Amazon Ex. 1008.0179



(a) Original E-R Model Version



(b) Crow's Foot Version

Figure 5-9

Two Versions of an N:M Relationship
(a) Original E-R Model Version
(b) Crow's Foot Version

the same time, although a SKILL may or may not be held by any EMPLOYEE, a SKILL may also be held by several EMPLOYEEs. The crow's foot version in Figure 5-9(b) shows the N:M maximum cardinalities using the notation in Figure 5-8. The crow's foot symbols again indicate the minimum cardinalities for the N:M relationship.

Except for Appendices B and C, for the rest of this text we will use the IE Crow's Foot model for E-R diagrams. There are no completely standard symbols for the crow's foot notation, and we explain our symbols and notation when we first use it. You can obtain various modeling products that will produce crow's foot models, and they are easily understood and related to the original E-R model. Be aware that those other products may use the oval, hash mark, crow's foot, and other symbols in slightly differently ways. Further, your instructor may have a favorite modeling tool for you to use. If that tool does not support crow's feet, you will have to adapt the data models in this text to your tool. Making these adaptations is a good learning exercise. See, for example, exercises 5.57 and 5.58.

BY THE WAY There are a number of modeling products that you can try—each will have its own idiosyncrasies. Computer Associates produces ERwin, a commercial data modeling product that handles both data modeling and database design tasks. You can download the CA ERwin Data Modeler Community Edition, which is suitable for class use, from *www.ca.com/us/software-trials.aspx*. You can use ERwin to produce either crow's foot or IDEF1X.

Although it is better at creating database designs (discussed in Chapter 5) than data models, Microsoft's Visio is a possibility. A trial version is available from the Microsoft Web site at http://office.microsoft.com/en-us/visio/default.aspx.

Finally, Sun Microsystems is continuing development of the MySQL Workbench, and a free (but somewhat limited) version is available at the MySQL Web site at http://dev.mysql.com/downloads/workbench/5.1.html. Although, like Microsoft Visio, it is better at database designs than data models, it is a very useful tool and the database designs it produces can be used with any DBMS, not just MySQL.

Strong Entities and Weak Entities

A **strong entity** is an entity that represents something that can exist on its own. For example, PERSON is a strong entity—we consider people to exist as individuals in their own right. Similarly, AUTOMOBILE is a strong entity. In addition to strong entities, the original version of the E-R model included the concept of a **weak entity**, which is defined as any entity whose existence depends on the presence of another entity.

ID-Dependent Entities

The E-R model includes a special type of entity called an **ID-dependent entity**. an ID-dependent entity is an entity whose identifier includes the identifier of another entity. Consider, for example, an entity for a student apartment in a building, as shown in Figure 5-10(a).

The identifier of such an entity is a composite (ApartmentNumber, BuildingName), where BuildingName is the identifier of the entity BUILDING. ApartmentNumber by itself is

Amazon Ex. 1008.0180

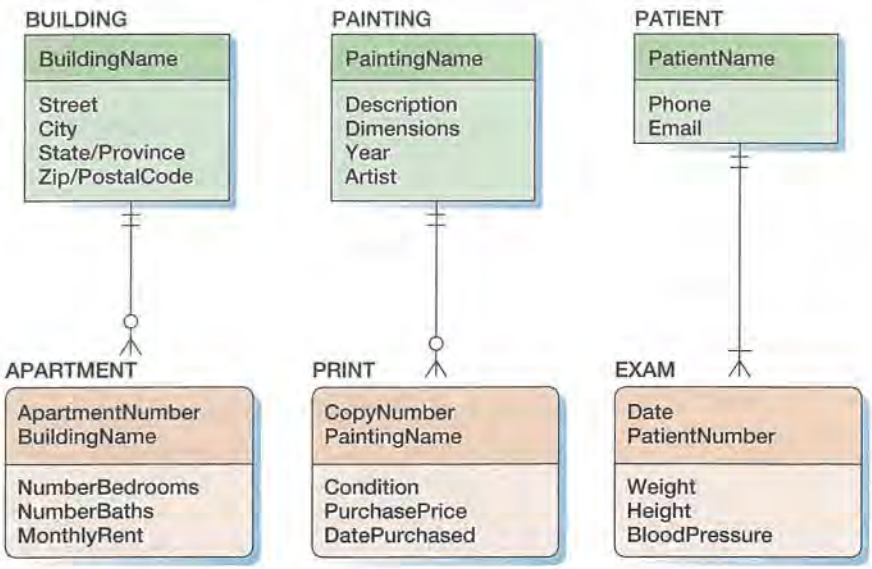


Figure 5-10
Example ID-dependent Entities

(a) APARTMENT is ID-Dependent on BUILDING

(b) PRINT is ID-Dependent on PAINTING

(c) EXAM is ID-Dependent on PATIENT

insufficient to tell someone where you live. If you say you live in apartment number 5, they must ask you, "In what building?"

Figure 5-10 shows three different ID-dependent entities. In addition to APARTMENT, the entity PRINT in Figure 5-10(b) is ID-dependent on PAINTING, and the entity EXAM in Figure 5-10(c) is ID-dependent on PAINTING, and the entity EXAM in Figure 5-10(c) is ID-dependent on PATIENT.

In each of these cases, the ID-dependent entity cannot exist unless the parent (the entity on which it depends) also exists. Thus, the minimum cardinality from the ID-dependent entity to the parent is always one.

However, whether the parent is required to have an ID-dependent entity depends on the application requirements. In Figure 5-10, both APARTMENT and PRINT are optional, but EXAM is required. These restrictions arise from the nature of the application and not from any logical requirement.

As shown in Figure 5-10, in our E-R models we use an entity with rounded corners to represent the ID-dependent entity. We also use a solid line to represent the relationship between the ID-dependent entity and its parent. This type of a relationship is called an **identifying relationship**. A relationship drawn with a dashed line (refer to Figure 5-7) is used between strong entities and is called a **nonidentifying relationship** because there are no ID-dependent entities in the relationship.

ID-dependent entities pose restrictions on the processing of the database that is constructed from them. Namely, the row that represents the parent entity must be created before any ID-dependent child row can be created. Further, when a parent row is deleted, all child rows must be deleted as well.

ID-dependent entities are common. Another example is the entity VERSION in the relationship between PRODUCT and VERSION, where PRODUCT is a software product and VERSION is a release of that software product. The identifier of PRODUCT is ProductName, and the identifier of VERSION is (ProductName, ReleaseNumber). Yet another example is EDITION in the relationship between TEXTBOOK and EDITION. The identifier of TEXTBOOK is Title, and the identifier of EDITION is (Title, EditionNumber).

Non-ID-Dependent Weak Entities

All ID-dependent entities are weak entities. But, according to the original E-R model, some entities that are weak are not ID-dependent. Consider the AUTO_MODEL and VEHICLE entity classes in the database of a car manufacturer, such as Ford or Honda, as shown in Figure 5-11.

Amazon Ex. 1008.0181

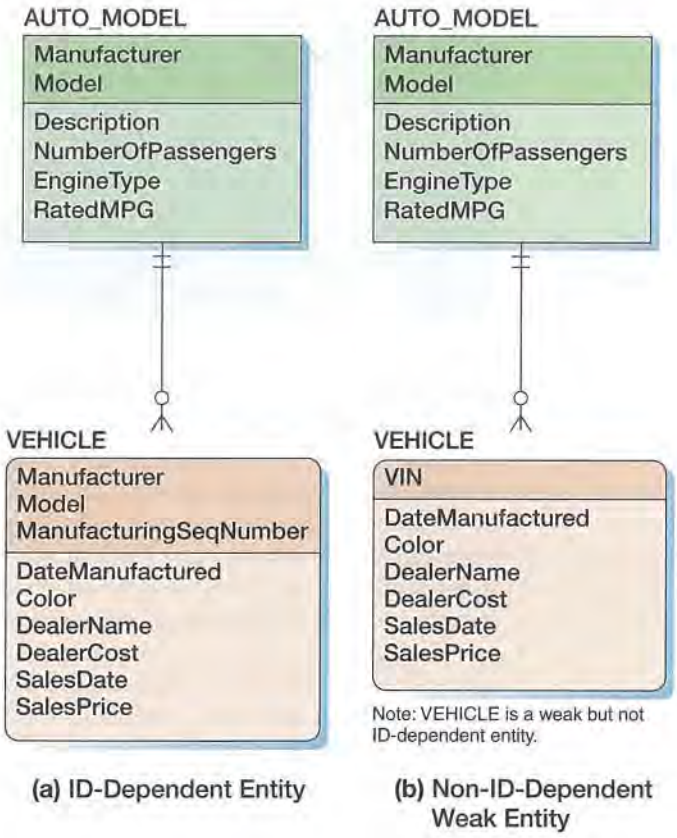


Figure 5-11
Weak Entity Examples

In Figure 5-11(a), each VEHICLE is assigned a sequential number as it is manufactured. So, for the "Super SUV" AUTO_MODEL, the first VEHICLE manufactured gets a ManufacturingSeqNumber of 1, the next gets a ManufacturingSeqNumber of 2, and so on. This is clearly an ID-dependent relationship because ManufacturingSeqNumber is based on the Manufacturer and Model.

Now let's assign VEHICLE an identifier that is independent of the Manufacturer and Model. We'll use a VIN (vehicle identification number), as shown in Figure 5-11(b). Now, the VEHICLE has a unique identifier of its own and does not need to be identified by its relation to AUTO_MODEL.

This is an interesting situation. VEHICLE has an identity of its own and therefore is not ID-dependent. Yet the VEHICLE is an AUTO_MODEL, and if that particular AUTO_MODEL did not exist, the VEHICLE itself would never have existed. Therefore, VEHICLE is now a *weak but non-ID-dependent entity*.

Consider *your* car—let's say it's a Ford Mustang just for the sake of this discussion. Your individual Mustang is a VEHICLE, and it exists as a physical object and is identified by the VIN that is required for each licensed automobile. It is *not* ID-dependent on AUTO_MODEL, which in this case is Ford Mustang, for its identity. However, if the Ford Mustang had never been created as an AUTO_MODEL—a logical concept that was first designed on paper—your car would never have been built because *no* Ford Mustangs would ever have been built! Therefore, your physical individual VEHICLE would not exist without a logical AUTO_MODEL of Ford Mustang, and in a data model (which *is* what we're talking about) a VEHICLE cannot exist without a related AUTO_MODEL. This makes VEHICLE a weak but non–ID-dependent entity. Most data modeling tools cannot model non-ID-dependent entities. So, to indicate such situations, we will use a nonidentifying relationship with a note added to the data model indicating that the entity is weak, as shown in Figure 5-11(b).

Unfortunately, an ambiguity is hidden in the definition of *weak entity*, and this ambiguity is interpreted differently by different database designers (as well as different textbook authors). The ambiguity is that in a strict sense, if a weak entity is defined as any entity whose presence

Amazon Ex. 1008.0182

- An ID-dependent entity is an entity whose identifier includes the identifier of another entity.
- Indentifying relationships are used to represent ID-dependent entities.
- A weak entity is an entity whose existence depends on another entity.
- All ID-dependent entities are weak.
- Some entities are weak, but not ID-dependent. Using data modeling tools, they are shown with nonidentifying relationships, with separate documentation indicating they are weak.

Figure 5-12
Summary of ID-dependent and Weak Entities

in the database depends on another entity, then any entity that participates in a relationship having a minimum cardinality of one to a second entity is a weak entity. Thus, in an academic database, if a STUDENT must have an ADVISER, then STUDENT is a weak entity, because a STUDENT entity cannot be stored without an ADVISER.

This interpretation seems too broad to some people. A STUDENT is not physically dependent on an ADVISER (unlike an APARTMENT to a BUILDING), and a STUDENT is not logically dependent on an ADVISER (despite how it might appear to either the student or the adviser), and, therefore, STUDENT should be considered a strong entity.

To avoid such situations, some people interpret the definition of weak entity more narrowly. They say that to be a weak entity an entity must logically depend on another entity. According to this definition, APARTMENT is a weak entity, but STUDENT is not. An APARTMENT cannot exist without a BUILDING in which it is located. However, a STUDENT can logically exist without an ADVISER, even if a business rule requires it. Characteristics of ID-dependent and weak entities are summarized in Figure 5-12.

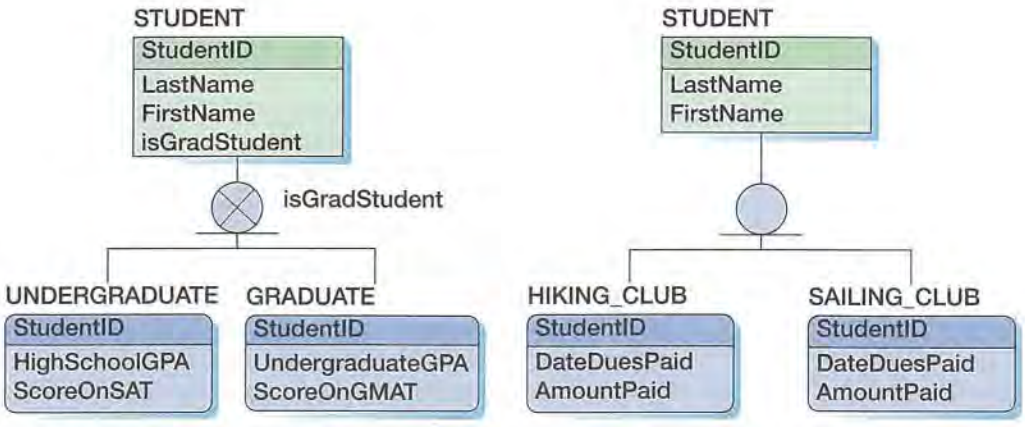
Subtype Entities

The extended E-R model introduced the concept of subtypes. A **subtype** entity is a special case of another entity called its **supertype**. Students, for example, may be classified as undergraduate or graduate students. In this case, STUDENT is the supertype, and UNDERGRADUATE and GRADUATE are the subtypes.

Alternatively, a student could be classified as a freshman, sophomore, junior, or senior. In that case, STUDENT is the supertype, and FRESHMAN, SOPHOMORE, JUNIOR, and SENIOR are the subtypes.

As illustrated in Figure 5-13, in our E-R models we use a circle with a line under it as a subtype symbol to indicate a supertype–subtype relationship. Think of this as a symbol for an optional (the circle) 1:1 (the line) relationship. In addition, we use a solid line to represent an ID-dependent subtype entity because each subtype is ID-dependent on the supertype. Also note that none of the line end symbols shown in Figure 5-8 are used on the connecting lines.

Figure 5-13
Example Subtype Entities



(a) Exclusive Subtypes with Discriminator (b) Inclusive Subtypes

Amazon Ex. 1008.0183

In some cases, an attribute of the supertype indicates which of the subtypes is appropriate for a given instance. An attribute that determines which subtype is appropriate is called a **discriminator**. In Figure 5-13(a), the attribute isGradStudent (which has only the values Yes and No) is the discriminator. In our E-R diagrams, the discriminator is shown next to the subtype symbol, as illustrated in Figure 5-13(a). Not all supertypes have a discriminator. Where a supertype does not have a discriminator, application code must be written to create the appropriate subtype.

Subtypes can be exclusive or inclusive. With **exclusive subtypes**, a supertype instance is related to at most one subtype. With **inclusive subtypes**, a supertype instance can relate to one or more subtypes. In Figure 5-13(a), the *X* in the circle means that the UNDERGRADUATE and GRADUATE subtypes are exclusive. Thus, a STUDENT can be either an UNDERGRADUATE or a GRADUATE, but not both. Figure 5-13(b) shows that a STUDENT can join either the HIKING_CLUB or the SAILING_CLUB, or both. These subtypes are inclusive (note there is no *X* in the circle). Because a supertype may relate to more than one subtype, inclusive subtypes do not have a discriminator.

The most important (some would say the only) reason for creating subtypes in a data model is to avoid value-inappropriate nulls. Undergraduate students take the SAT exam and report that score, whereas graduate students take the GMAT and report their score on that exam. Thus, the SAT score would be NULL in all STUDENT entities for graduates, and the GMAT score would be NULL for all undergraduates. Such null values can be avoided by creating subtypes.



The relationships that connect supertypes and subtypes are called **IS-A relationships** because a subtype is the same entity as the supertype. Because this is so, the identifier of a supertype and all its subtypes must be the same; they all represent different aspects of the same entity. Contrast this with HAS-A relationships, in which an entity has a relationship to another entity, but the identifiers of the two entities are different.

The elements of the entity-relationship model and their IE Crow's Foot representation are summarized in Figure 5-14. The identifier and attributes are shown only in the first example. Note that for 1:1 and 1:N nonidentifying relationships a relationship to a parent entity may be optional. For identifying relationships, the parent is always required.

Patterns in Forms, Reports, and E-R Models

A data model is a representation of how users view their world. Unfortunately, you cannot walk up to most computer users and ask questions like, "What is the maximum cardinality between the EMPLOYEE and SKILL entities?" Few users would have any idea of what you mean. Instead, you must infer the data model indirectly from user documents and from users' conversation and behavior.

One of the best ways to infer a data model is to study the users' forms and reports. From such documents, you can learn about entities and their relationships. In fact, the structure of forms and reports determines the structure of the data model, and the structure of the data model determines the structure of forms and reports. This means that you can examine a form or report and determine the entities and relationships that underlie it.

You can also use forms and reports to validate the data model. Rather than showing the data model to the users for feedback, an alternative is to construct a form or report that reflects the structure of the data model and obtain user feedback on that form or report. For example, if you want to know if an ORDER has one or many SALESPERSONs you can show the users a form that has a space for entering just one salesperson's name. If the user asks, "Where do I put the name of the second salesperson?" then you know that orders have at least two and possibly

Amazon Ex. 1008.0184

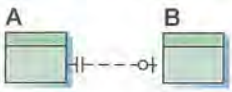
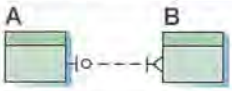
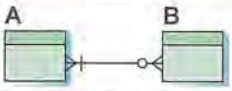
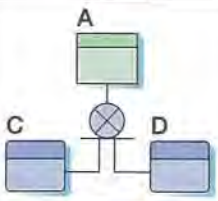
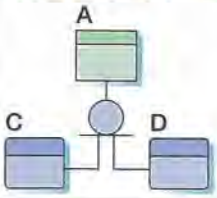
DEPARTMENT DepartmentName BudgetCode OfficeNumber	DEPARTMENT entity; DepartmentName is identifier; BudgetCode and OfficeNumber are attributes.
	1:1, nonidentifying relationship. A relates to zero or one B; B relates to exactly one A. Identifier and attributes not shown.
	1:N, nonidentifying relationship. A relates to one or many Bs; B relates to zero or one A. Identifier and attributes not shown.
	Many-to-many, nonidentifying relationship. A relates to zero or more Bs; B relates to one or more As.
	1:N identifying relationship. A relates to zero, one, or many Bs. B relates to exactly one A. Identifier and attributes not shown. For identifying relationships, the child must always relate to exactly one parent. The parent may relate to zero, one, many, or a combination of these minimum cardinalities.
	A is supertype, C and D are exclusive subtypes. Discriminator not shown. Identifier and attributes not shown.
	A is supertype, C and D are inclusive subtypes. Identifier and attributes not shown.

Figure 5-14
IE Crows Foot Symbol Summary

many salespeople. Sometimes, when no appropriate form or report exists, teams create a prototype form or report for the users to evaluate.

All of this means that you must understand how the structure of forms and reports determines the structure of the data model, and the reverse. Fortunately, many forms and reports fall into common patterns. If you learn how to analyze these patterns, you will be well on your way to understanding the logical relationship between forms and reports and the data model. Accordingly, in the next sections we will discuss the most common patterns in detail.

Strong Entity Patterns

Three relationships are possible between two strong entities: 1:1, 1:N, and N:M. When modeling such relationships, you must determine both the maximum and minimum cardinality. The maximum cardinality often can be determined from forms and reports. In most cases, to determine the minimum cardinality you will have to ask the users.

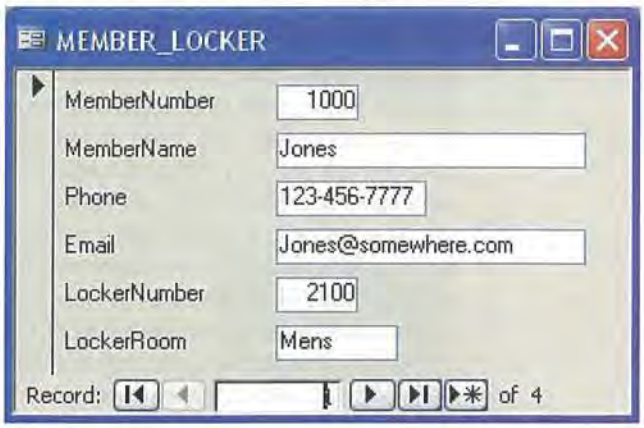
1:1 Strong Entity Relationships

Figure 5-15 shows a data entry form and a report that indicate a one-to-one relationship between the entities CLUB_MEMBER and LOCKER. The MEMBER_LOCKER form in Figure 5-15(a) shows data for an athletic club member, and it lists just one locker for that member. This form indicates that a CLUB_MEMBER has at most one locker. The report in Figure 5-15(b) shows the lockers in the club and indicates the member who has been allocated that locker. Each locker is assigned to one club member.

The form and report in Figure 5-15 thus suggest that a CLUB_MEMBER has one LOCKER, and a LOCKER is assigned to one CLUB_MEMBER. Hence, the relationship between them

Amazon Ex. 1008.0185

(a) Club Membership Data Entry Form



(b) Club Locker Report

CLUB_LOCKERS

LockerRoom	*LockerNumber*	*MemberNumber*	*MemberName*	*LockerSize*
Mens	2100	1000	Jones	Med
Mens	2115	3000	Wu	Large
Womens	2200	2000	Abernathy	Large
Womens	2217	4000	Lai	Small

Figure 5-15

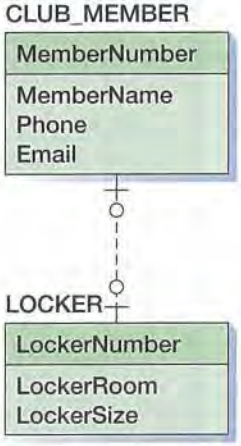
Form and Report Indicating a 1:1 Relationship

is 1:1. To model that relationship, we draw a nonidentifying relationship (meaning the relationship is strong and not ID-dependent) between the two entities, as shown in Figure 5-16. We then set the maximum cardinality to 1:1. You can tell that this is a nonidentifying relationship, because the relationship line is dashed. Also, the absence of a crow's foot indicates that the relationship is 1:1.

Regarding minimum cardinality, every club member shown in the form has a locker, and every locker shown in the report is assigned to a club member, so it appears that the relationship is mandatory to mandatory. However, this form and report are just instances; they may not show every possibility. If the club allows social, nonathletic memberships, then not every club member will have a locker. Furthermore, it is unlikely that every locker is occupied; there must be some lockers that are unused and nonallocated. Accordingly, Figure 5-16 shows this relationship as optional to optional, as indicated by the small circles on the relationship lines.

Figure 5-16

Data Model for the 1:1 Relationship in Figure 5-15



Amazon Ex. 1008.0186

BY THE WAY How do you recognize strong entities? You can use two major tests. First, does the entity have an identifier of its own? If it shares a part of its identifier with another entity, then it is an ID-dependent entity, and is therefore weak. Second, does the entity seem to be logically different from and separate from the other entities? Does it stand alone, or is it part of something else? In this case, CLUB_MEMBER and a LOCKER are two very different, separate things; they are not part of each other or of something else. Hence, they are strong.

Note also that a form or report shows only one side of a relationship. Given entities A and B, a form can show the relationship from A to B, but it cannot show the relationship from B to A at the same time. To learn the cardinality from B to A, you must examine a second form or report, ask the users, or take some other action.

Finally, it is seldom possible to infer minimum cardinality from a form or report. Generally, you must ask the users.

1:N Strong Entity Relationships

Figure 5-17 shows a form that lists the departments within a company. The company has many departments, so the maximum cardinality from COMPANY to DEPARTMENT is N. But what about in the opposite direction? To determine if a department relates to one or N companies, we need to examine a form or report that shows the relationship from a department to a company. However, assume that no such form or report exists. Assume that the users never view company data from the perspective of a department. We cannot ignore the issue, because we need to know whether the relationship is 1:N or N:M.

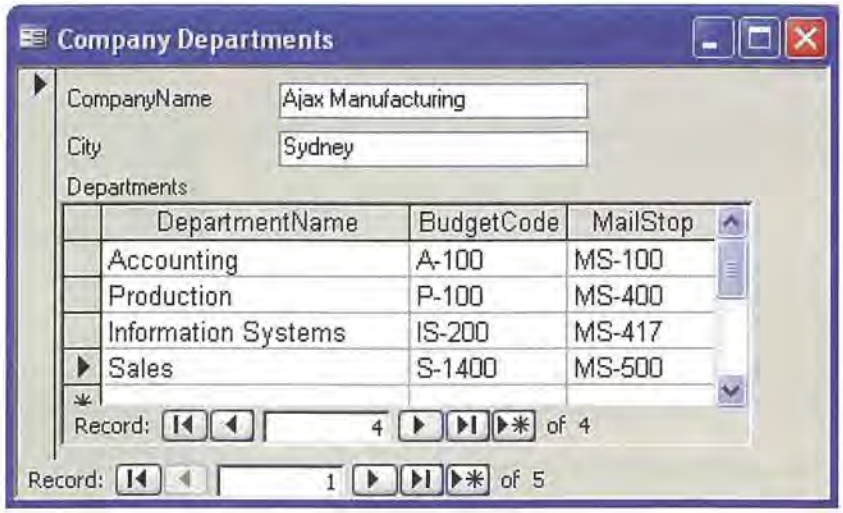
In such a case, we must ask the users or at least make a determination by thinking about the nature of the business setting. Can a department belong to more than one company? Is a department shared among companies? Because this seems unlikely, we can reasonably assume that DEPARTMENT relates to just one COMPANY. Thus, we conclude the relationship is 1:N. Figure 5-18 shows the resulting data model. Note that the many side of the relationship is indicated by the crow's foot next to DEPARTMENT.

Considering minimum cardinality, we do not know if a COMPANY must have a DEPARTMENT or if a DEPARTMENT must have a COMPANY. We will definitely need to ask the users. Figure 5-18 depicts the situation in which a DEPARTMENT must have a COMPANY, but a COMPANY need not have any DEPARTMENTs.

N:M Strong Entity Relationships

Figure 5-19(a) shows a form with data about a supplier and the parts it is prepared to supply. Figure 5-19(b) shows a report that summarizes parts and lists the companies that can supply those parts. In both cases, the relationship is many: A SUPPLIER supplies many PARTs, and a PART is supplied by many SUPPLIERs. Thus, the relationship is N:M.

Figure 5-17
Form Indicating a 1:N Relationship



Amazon Ex. 1008.0187

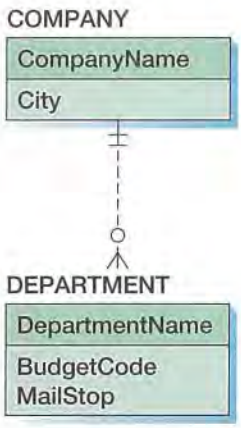


Figure 5-18

Data Model for the 1:N Relationship in Figure 5-17

Figure 5-20 shows a data model that extends the data model in Figure 5-18 to include this new relationship. A supplier is a company, so we show the supplier entity as a COMPANY.

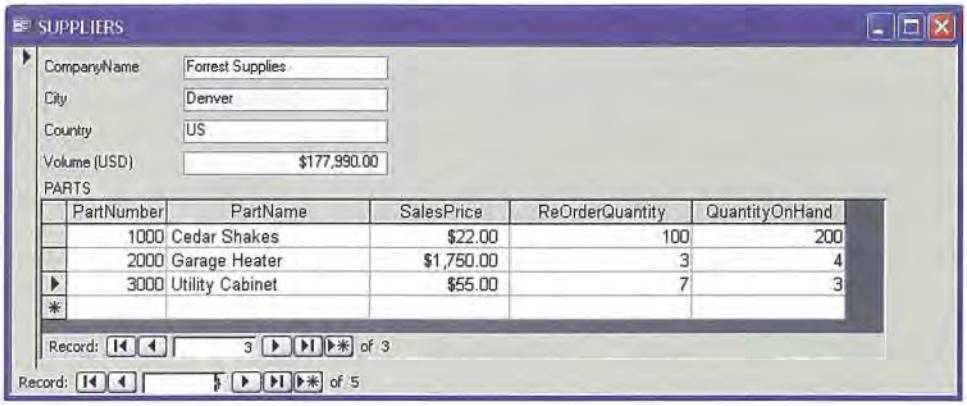
Because not all companies are suppliers, the relationship from COMPANY to PART must be optional. However, every part must be supplied from somewhere, so the relationship from PART to COMPANY is mandatory.

In summary, the three types of strong entity relationships are 1:1, 1:N, and N:M. You can infer the maximum cardinality in one direction from a form or report. You must examine a second form or report to determine the maximum cardinality in the other direction. If no form

Figure 5-19

Form and Report Indicating an N:M Relationship

(a) SUPPLIERS Form



(b) PART Report

PART

Number	PartName	SalesPrice	ROQ	QOH	CompanyName	City	Country
1000	Cedar Shakes	$22.00	100	200			
					Bristol Systems	Machester	England
					ERS Systems	Vancouver	Canada
					Forrest Supplies	Denver	US
2000	Garage Heater	$1,750.00	3	4			
					Bristol Systems	Machester	England
					ERS Systems	Vancouver	Canada
					Kyoto Importers	Kyoto	Japan
					Forrest Supplies	Denver	US
3000	Utility Cabinet	$55.00	7	3			
					Ajax Manufacturing	Sydney	Australia
					Forrest Supplies	Denver	US

Amazon Ex. 1008.0188

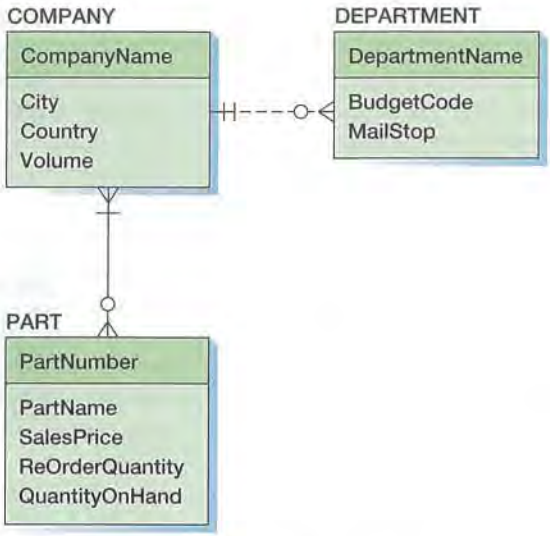


Figure 5-20

Data Model for the N:M Relationship in Figure 5-19

or report that shows the relationship is available, you must ask the users. Generally, it is not possible to determine minimum cardinality from forms and reports.

ID-Dependent Relationships

Three principle patterns use ID-dependent entities: multivalued attribute, version/instance, and association. Because the association pattern is often confused with the N:M strong entity relationships just discussed, we will look at that pattern first.

The Association Pattern

An **association pattern** is subtly and confusingly similar to an N:M strong relationship. To see why, examine the report in Figure 5-21 and compare it with the report in Figure 5-19(b).

What is the difference? If you look closely, you'll see that the only difference is that the report in Figure 5-21 contains Price, which is the price quotation for a part from a particular supplier. The first line of this report indicates that the part Cedar Shakes is supplied by Bristol Systems for $14.00.

Price is neither an attribute of COMPANY nor is it an attribute of PART. It is an attribute of the combination of a COMPANY with a PART. Figure 5-22 shows the appropriate data model for such a case.

Here, a third entity, QUOTATION, has been created to hold the Price attribute. The identifier of QUOTATION is the combination of PartNumber and CompanyName. Note that PartNumber is the identifier of PART, and CompanyName is the identifier of COMPANY. Hence, QUOTATION is ID-dependent on *both* PART and COMPANY.

Figure 5-21

Report Indicating an Association Pattern

Part Quotations

Number	*Name*	*SalesPrice*	*ROQ*	*QOH*	*Company*	*City*	*Price*
1000	*Cedar Shakes*	*$22.00*	*100*	*200*			
					Bristol Systems	Machester	$14.00
					ERS Systems	Vancouver	$12.50
					Forrest Supplies	Denver	$15.50
2000	*Garage Heater*	*$1,750.00*	*3*	*4*			
					Bristol Systems	Machester	$950.00
					ERS Systems	Vancouver	$875.00
					Kyoto Importers	Kyoto	$1,100.00
					Forrest Supplies	Denver	$915.00
3000	*Utility Cabinet*	*$55.00*	*7*	*3*			
					Ajax Manufacturing	Sydney	$37.50
					Forrest Supplies	Denver	$42.50

Amazon Ex. 1008.0189

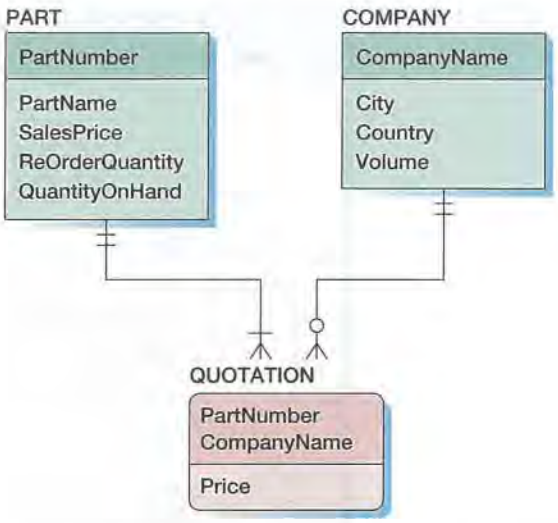


Figure 5-22
Association Pattern Data Model for the Report in Figure 5-21

In Figure 5-22, then, the relationships between PART and QUOTATION and between COMPANY and QUOTATION are both identifying. This fact is shown in Figure 5-22 by the solid, nondashed line that represents these relationships.

As with all identifying relationships, the parent entities are required. Thus, the minimum cardinality from QUOTATION to PART is one, and the minimum cardinality from QUOTATION to COMPANY also is one. The minimum cardinality in the opposite direction is determined by business requirements. Here, a PART must have a QUOTATION, but a COMPANY need not have a QUOTATION.

BY THE WAY Consider the differences between the data models in Figure 5-20 and Figure 5-22. The only difference between the two is that in the latter the relationship between COMPANY and PART has an attribute, Price. Remember this example whenever you model an N:M relationship. Is there a missing attribute that pertains to the combination and not just to one of the entities? If so, you are dealing with an association, ID-dependent pattern and not an N:M, strong entity pattern.

Associations can occur among more than two entity types. Figure 5-23, for example, shows a data model for the assignment of a particular client to a particular architect for a particular

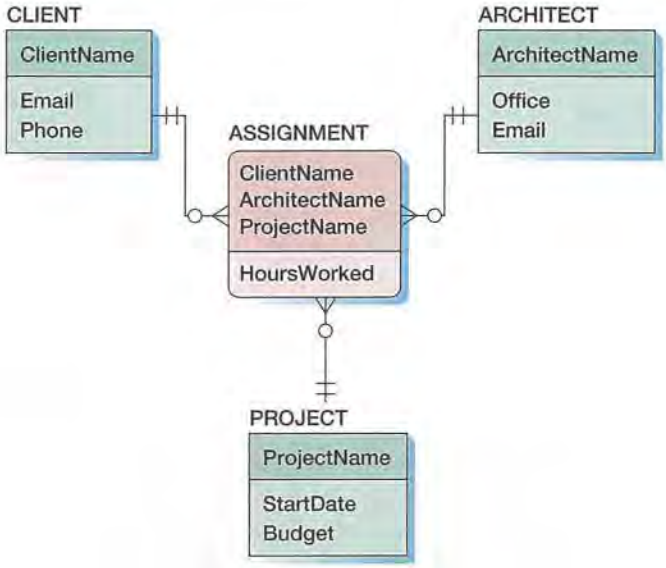


Figure 5-23
Association Pattern Data Model for the Ternary Relationship in Figure 5-4

Amazon Ex. 1008.0190

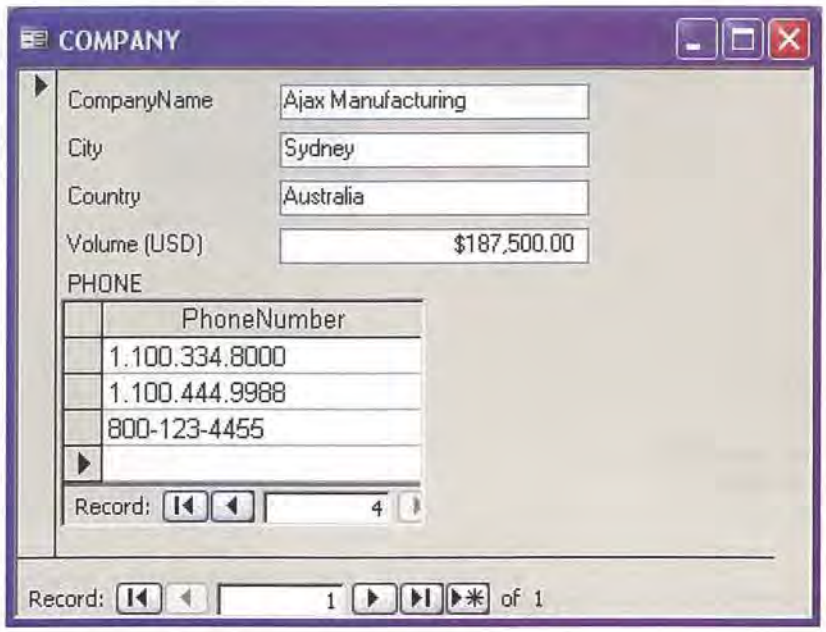


Figure 5-24
Data Entry Form with a Multivalued Attribute

project. The attribute of the assignment is HoursWorked. This data model shows how the ternary relationship in Figure 5-4(b) can be modeled as a combination of three binary relationships.

The Multivalued Attribute Pattern

In the E-R model as used today,[5] attributes must have a single value. If the COMPANY entity has PhoneNumber and Contact attributes, then a company can have at most one value for phone number and at most one value for contact.

In practice, however, companies can have more than one phone number and one contact. Consider, for example, the data entry form in Figure 5-24. This particular company has three phone numbers; other companies might have one or two or four, or whatever. We need to create a data model that allows companies to have multiple phones, and placing the attribute PhoneNumber in COMPANY will not do it.

Figure 5-25 shows the solution. Instead of including PhoneNumber as an attribute of COMPANY, we create an ID-dependent entity, PHONE, that contains the attribute PhoneNumber. The relationship from COMPANY to PHONE is 1:N, so a company can have multiple phone numbers. Because PHONE is an ID-dependent entity, its identifier includes both CompanyName and PhoneNumber.

We can extend this strategy for as many multivalued attributes as necessary. The COMPANY data entry form in Figure 5-26 has multivalued Phone and multivalued Contact

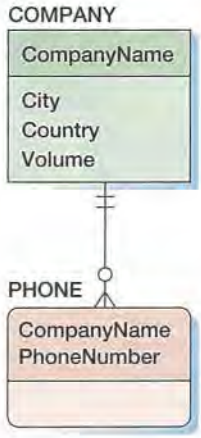


Figure 5-25
Data Model for the Form in Figure 5-24

[5] The original E-R model allowed for multivalued attributes. Over time, that feature has been ignored, and today most people assume that the E-R model requires single-valued attributes. We will do so in this text.

Amazon Ex. 1008.0191

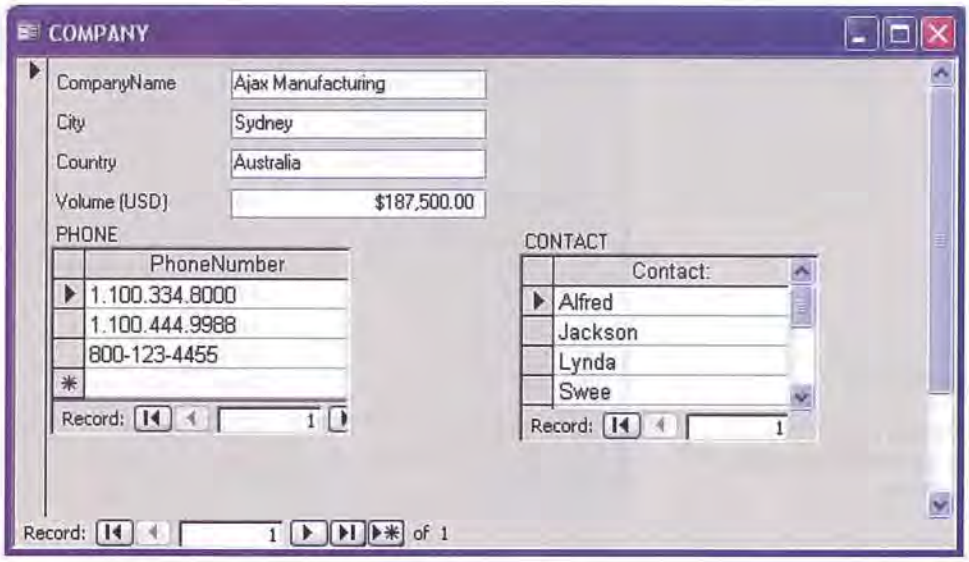


Figure 5-26

Data Entry Form with Separate Multivalued Attributes

attributes. In this case, we just create a separate ID-dependent entity for each multivalued attribute, as shown in Figure 5-27.

In Figure 5-27, PhoneNumber and Contact are independent. PhoneNumber is the phone number of the company and not necessarily the phone number of a contact. If PhoneNumber is not a general company phone number, but rather the phone number of a particular person at that company, then the data entry form would appear as in Figure 5-28. Here, for example, Alfred has one phone number and Jackson has another.

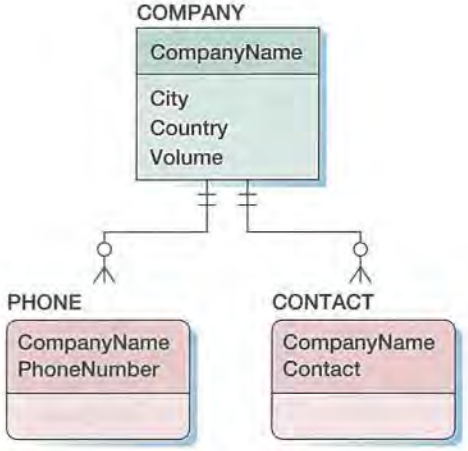
In this case, the attributes PhoneNumber and Contact belong together. Accordingly, we place them into a single ID-dependent entity, as shown in Figure 5-29. Notice that the identifier of PHONE_CONTACT is Contact and CompanyName. This arrangement means that a given Contact name can appear only once per company. Contacts can share phone numbers, however, as shown for employees Lynda and Swee. If the identifier of PHONE_CONTACT was PhoneNumber and CompanyName, then a phone number could occur only once per company, but contacts could have multiple numbers. Work through these examples to ensure that you understand them.

In all of these examples, the child requires a parent, which is always the case for ID-dependent entities. The parent may or may not require a child, depending on the application. A COMPANY may or may not require a PHONE or a CONTACT. You must ask the users to determine whether the ID-dependent entity is required.

Multivalued attributes are common, and you need to be able to model them effectively. Review the models in Figures 5-25, 5-27, and 5-29 and be certain that you understand their differences and what those differences imply.

Figure 5-27

Data Model for the Form in Figure 5-26



Amazon Ex. 1008.0192

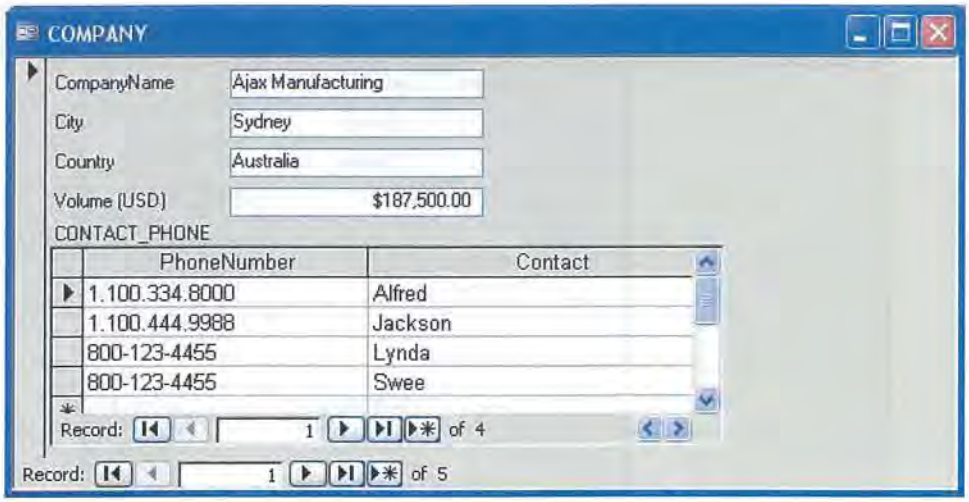


Figure 5-28
Data Entry Form with Composite Multivalued Attributes

The Archetype/Instance Pattern

The archetype/instance pattern occurs when one entity represents a manifestation or an instance of another entity. You have already seen one example of archetype/instance in the example of PAINTING and PRINT in Figure 5-9. The painting is the archetype, and the prints made from the painting are the instances of that archetype.

Other examples of archetype/instances are shown in Figure 5-30. One familiar example concerns classes and sections of classes. The class is the archetype, and the sections of the class are instances of that archetype. Other examples involve designs and instances of designs. A yacht manufacturer has various yacht designs, and each yacht is an instance of a particular design archetype. In a housing development, a contractor offers several different house models, and a particular house is an instance of that house model archetype.

As with all ID-dependent entities, the parent entity is required. The child entities (here SECTION, YACHT, and HOUSE) may or may not be required, depending on application requirements.

Logically, the child entity of every archetype/instance pattern is an ID-dependent entity. All three of the examples in Figure 5-30 are accurate representations of the logical structure of the underlying data. However, sometimes users will add additional identifiers to the instance entity and in the process change the ID-dependent entity to a weak entity that is not ID-dependent.

For example, although you can identify a SECTION by class name and section, colleges and universities often will add a unique identifier to SECTION, such as ReferenceNumber. In that

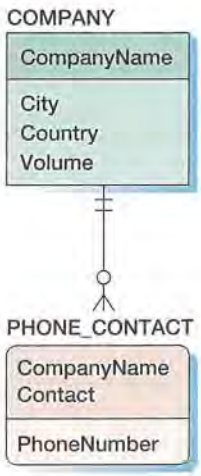


Figure 5-29
Data Model for the Form in Figure 5-28

Amazon Ex. 1008.0193

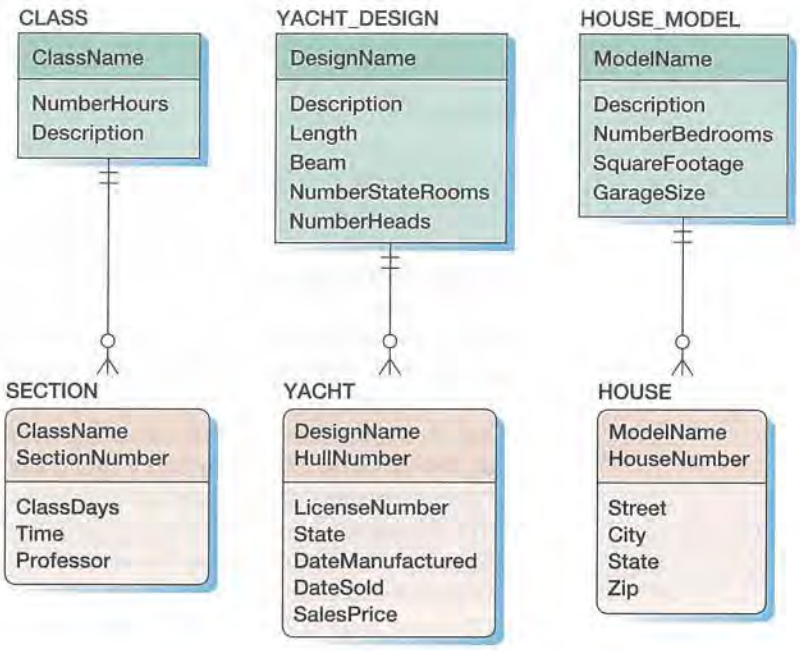


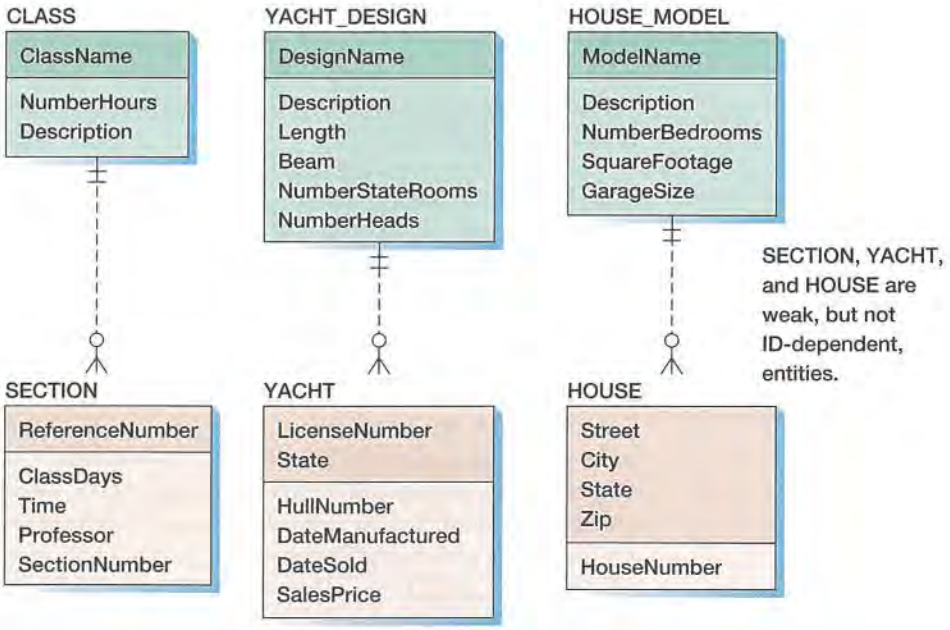
Figure 5-30
Three Archetype/Instance Pattern Examples

case, SECTION is no longer an ID-dependent entity, but it is still existence dependent on CLASS. Hence, as shown in Figure 5-31, SECTION is weak, but not ID-dependent.

A similar change may occur to the YACHT entity. Although the manufacturer of a yacht may refer to it by specifying the hull number of a given design, the local tax authority may refer to it by State and LicenseNumber. If we change the identifier of YACHT from (HullNumber, DesignName) to (LicenseNumber, State), then YACHT is no longer ID-dependent; it becomes a weak, non-ID-dependent entity.

Similarly, although the home builder may think of a home as the third house constructed according to the Cape Codd design, everyone else will refer to it by its address. When we change the identifier of HOUSE from (HouseNumber, ModelName) to (Street, City, State, Zip), then HOUSE becomes a weak, non-ID-dependent entity. All of these changes are shown in Figure 5-31.

Figure 5-31
Three Weak But Not ID-Dependent Relationships



Amazon Ex. 1008.0194

BY THE WAY Data modelers continue to debate the importance of weak, non-ID-dependent entities. Everyone agrees that they exist, but not everyone agrees that they are important.

First, understand that existence dependence influences the way we write database applications. For the CLASS/SECTION example in Figure 5-31, we must insert a new CLASS before we can add a SECTION for that class. Additionally, when we delete a CLASS, we must delete all of the SECTIONs for that CLASS as well. This is one reason that some data modelers believe that weak, non-ID-dependent entities are important.

Skeptics say that although weak, non-ID-dependent entities may exist, they are not necessary. They say that we can obtain the same result by calling SECTION strong and making CLASS required. Because CLASS is required, the application will need to insert a CLASS before a SECTION is created and delete dependent SECTIONs when deleting a CLASS. So, according to that viewpoint, there is no practical difference between a weak, non-ID-dependent entity and a strong entity with a required relationship.

Others disagree. Their argument goes something like this: The requirement that a SECTION must have a CLASS comes from a logical necessity. It has to be that way—it comes from the nature of reality. The requirement that a strong entity must have a relationship to another strong entity arises from a business rule. Initially, we say that an ORDER must have a CUSTOMER (both strong entities), and then the application requirements change and we say that we can have cash sales, meaning that an ORDER no longer has to have a CUSTOMER. Business rules change frequently, but logical necessity never changes. We need to model weak, non-ID-dependent entities so that we know the strength of the required parent rule.

And so it goes. You, with the assistance of your instructor, can make up your own mind. Is there a difference between a weak, non-ID-dependent entity and a strong entity with a required relationship? In Figure 5-31, should we call the entities SECTION, YACHT, and HOUSE strong, as long as their relationships are required? We think not; We think there is a difference. Others think differently, however.

Mixed Identifying and Nonidentifying Patterns

Some patterns involve both identifying and nonidentifying relationships. The classic example is the line-item pattern, but there are other instances of mixed patterns as well. We begin with line items.

The Line-Item Pattern

Figure 5-32 shows a typical sales order, or invoice. Such forms usually have data about the order itself, such as the order number and order date, data about the customer, data about the salesperson, and then data about the items on the order. A data model for a typical SALES_ORDER is shown in Figure 5-33.

In Figure 5-35, CUSTOMER, SALESPERSON, and SALES_ORDER are all strong entities, and they have the nonidentifying relationships you would expect. The relationship from CUSTOMER to SALES_ORDER is 1:N, and the relationship from SALESPERSON to SALES_ORDER also is 1:N. According to this model, a SALES_ORDER must have a CUSTOMER and may or may not have a SALESPERSON. All of this is readily understood.

The interesting relationships concern the line items on the order. Examine the data grid in the form in Figure 5-32. Some of the data values belong to the order itself, but other data values belong to items in general. In particular, Quantity and ExtendedPrice belong to the SALES_ORDER, but ItemNumber, Description, and UnitPrice belong to ITEM. The lines on an order do not have their own identifier. No one ever says, "Give me the data for line 12." Instead, they say, "Give me the data for line 12 of order 12345." Hence, the identifier of a line is a composite of the identifier of a particular line and the identifier of a particular order. Thus, entries for line items are always ID-dependent on the order in which they appear. In Figure 5-33, ORDER_LINE_ITEM is ID-dependent on SALES_ORDER. The identifier of the ORDER_LINE_ITEM entity is (SalesOrderNumber, LineNumber).

Amazon Ex. 1008.0195

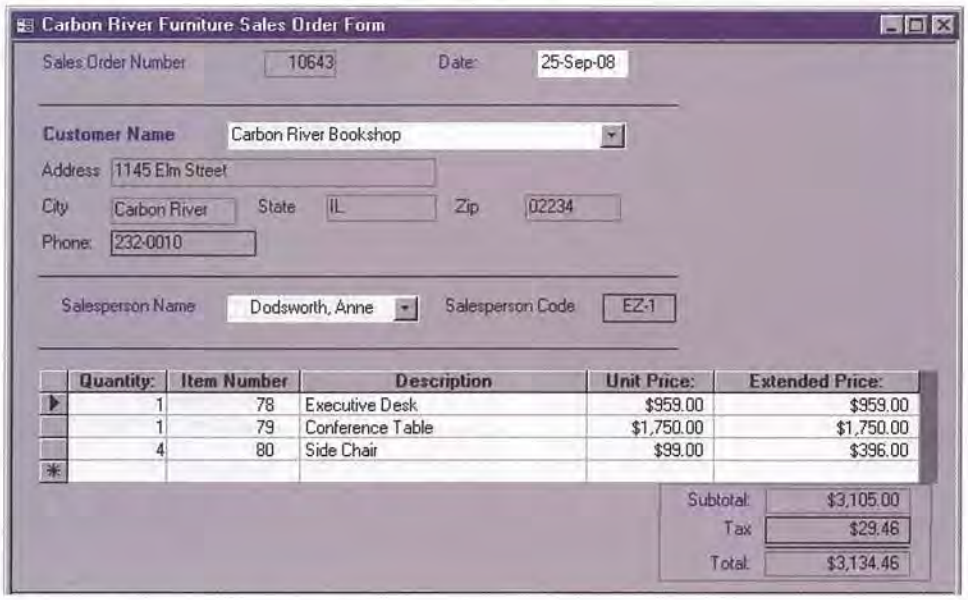


Figure 5-32

Data Entry Form for a Sales Order

Now, and this is the part that is sometimes confusing for some students, ORDER_LINE_ITEM is not existence dependent on ITEM. It can exist even if no item has yet been assigned to it. Further, if an ITEM is deleted, we do not want the line item to be deleted with it. The deletion of an ITEM may make the value of ItemNumber and other data invalid, but it should not cause the line item itself to disappear.

Now consider what happens to a line item when an order is deleted. Unlike with the deletion of an item, which only causes data items to become invalid, the deletion of the order removes the existence of the line item. Logically, a line item cannot exist if its order is deleted. Hence, line items are existence dependent on orders.

Work through each of the relationships in Figure 5-33 and ensure that you understand their type and their maximum and minimum cardinalities. Also understand the implications of this

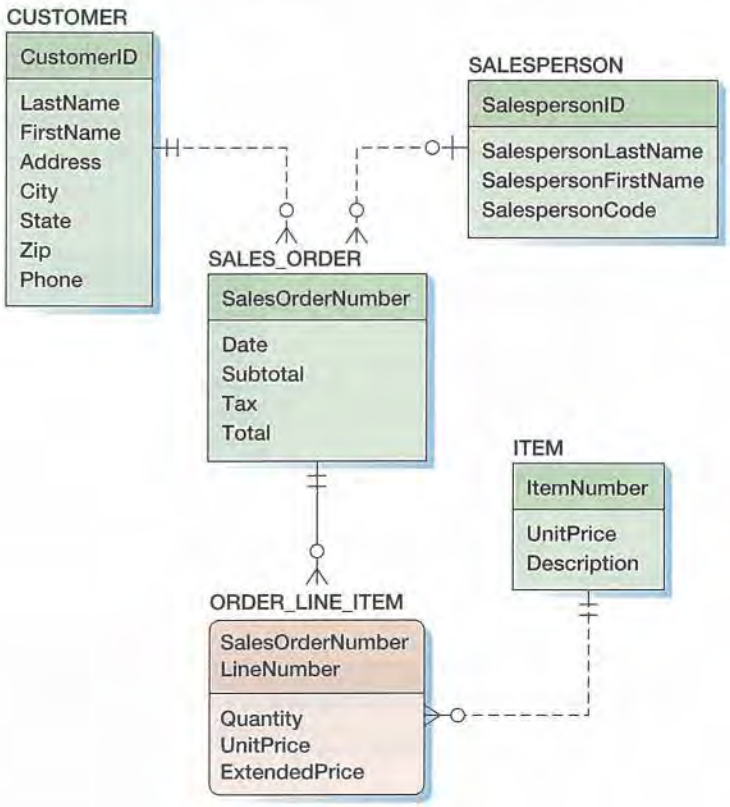


Figure 5-33

Data Model for the Sales Order in Figure 5-33

Amazon Ex. 1008.0196

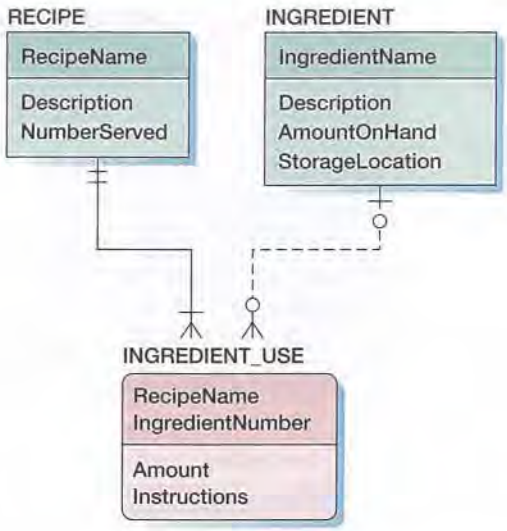


Figure 5-34
Mixed Relationship Pattern for Restaurant Recipe

model. For example, do you see why this sales order is unlikely to be used by a company in which salespeople are on commission?

Other Mixed Patterns

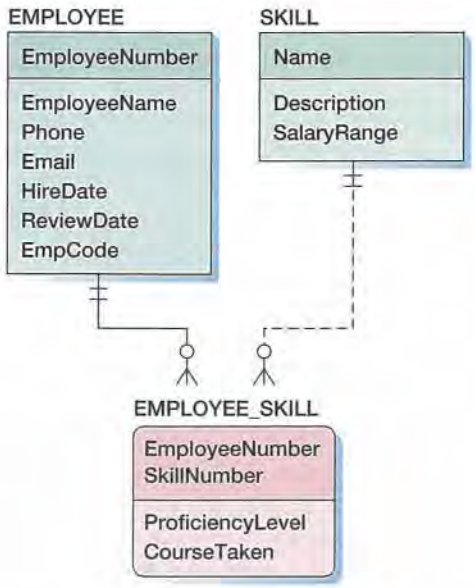
Mixed identifying and nonidentifying relationships occur frequently. Learn to look for a mixed pattern when a strong entity has a multivalued composite group and when one of the elements in the composite group is an identifier of a second strong entity.

Consider, for example, baking recipes. Each recipe calls for a certain amount of a specific ingredient, such as flour, sugar, or butter. The ingredient list is a multivalued composite group, but one of the elements of that group, the name of the ingredient, is the identifier of a strong entity. As shown in Figure 5-34, the recipe and the ingredients are strong entities, but the amount and instructions for using each ingredient are ID-dependent on RECIPE.

Or, consider employees' skill proficiencies. The name of the skill and the proficiency level the employee has are a multivalued group, but the skill itself is a strong entity, as shown in Figure 5-35. Dozens of other examples are possible.

Before continuing, compare the models in Figures 5-33, 5-34, and 5-35 with the association pattern in Figure 5-22. Make sure that you understand the differences and why the model in Figure 5-24 has two identifying relationships and the models in Figures 5-33, 5-34, and 5-35 have just one.

Figure 5-35
Mixed Relationship Pattern for Employee Skills



Amazon Ex. 1008.0197

Resident Fishing License 2009 Season *State of Washington*					License No: 03-1123432
Name:					
Street:					
City:		State:		Zip:	
For Use by Commercial Fishers Only			For Use by Sport Fishers Only		
Vessel Number:			Number Years at This Address:		
Vessel Name:			Prior Year License Number:		
Vessel Type:					
Tax ID:					

Figure 5-36

Data Entry Form Suggesting the Need for Subtypes

The For-Use-By Pattern

As stated earlier in this chapter, the major reason for using subtypes in a database design is to avoid value-inappropriate nulls. Some forms suggest the possibility of such nulls when they show blocks of data fields that are grayed out and labeled "For Use by *someone/something* Only." For example, Figure 5-36 shows two grayed-out sections, one for commercial fishers and another for sport fishers. The presence of these grayed-out sections indicates the need for subtype entities.

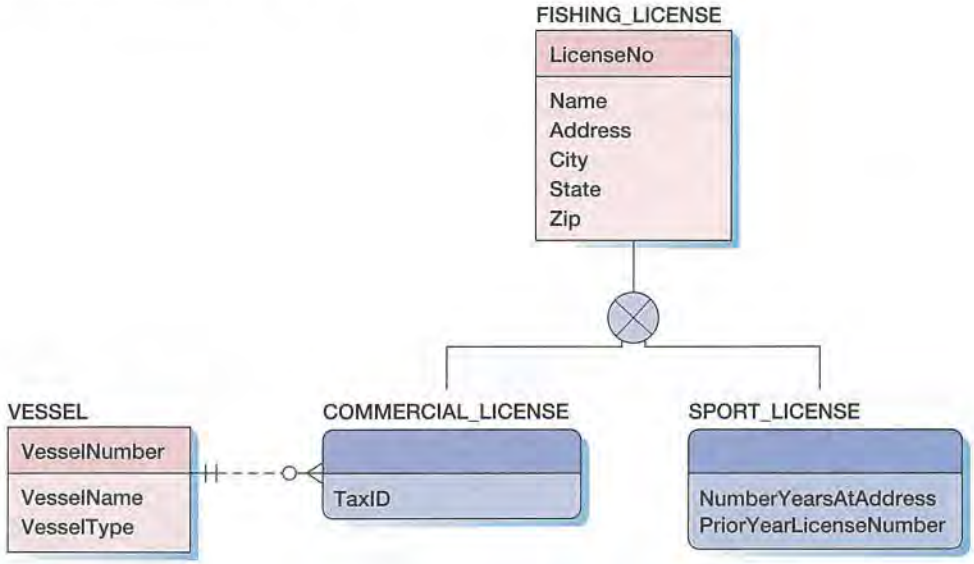
The data model for this form is shown in Figure 5-37. Observe that each grayed-out section has a subtype. Notice that the subtypes differ not only in their attributes, but that one has a relationship that the other does not have. Sometimes, the only differences between subtypes are differences in the relationships they have.

The nonidentifying relationship from VESSEL to COMMERCIAL_LICENSE is shown as 1:N, mandatory to mandatory. In fact, this form does not have sufficient data for us to conclude that the maximum cardinality from VESSEL to COMMERCIAL_LICENSE is N. This fact was determined by interviewing users and learning that one boat is sometimes used by more than one commercial fisher. The minimum cardinalities indicate a commercial fisher must have a vessel, and that only vessels that are used for licenses are to be stored in this database.

The point of this example is to illustrate how forms often suggest the need for subtypes. Whenever you see a grayed out or otherwise distinguished section of a form with the words "For use by . . . ," think "subtype."

Figure 5-37

Data Model for Form in Figure 5-36



Amazon Ex. 1008.0198

Recursive Patterns

A recursive relationship occurs when an entity type has a relationship to itself. The classic examples of recursive relationships occur in manufacturing applications, but there are many other examples as well. As with strong entities, three types of recursive relationships are possible: 1:1, 1:N, and N:M. Let's consider each.

1:1 Recursive Relationships

Suppose you are asked to construct a database for a railroad, and you need to make a model of a freight train. You know that one of the entities is BOXCAR, but how are BOXCARs related? To answer that question, envision a train. Except for the first boxcar, each has one boxcar in front and, except for the last boxcar, each boxcar has one boxcar in back. Thus, the relationship is 1:1 between boxcars, with an optional relationship for the first and last cars.

Figure 5-38 shows a data model in which each BOXCAR has a 1:1 relationship to the BOXCAR ahead. The BOXCAR entity at the head of the train has a 1:1 relationship to ENGINE. (This model assumes a train has just one engine. To model trains with multiple engines, create a second recursive relationship among engines. Construct that relationship just like the Boxcar Ahead relationship.)

Figure 5-39 shows example entity instances that illustrate this data model. Not surprisingly, this set of entity instances looks just like a train.

An alternative model is to use the relationship to represent the BOXCAR behind. Either model works. Other examples of 1:1 recursive relationships are the succession of U.S. presidents, the succession of deans in a college of business, and the order of passengers on a waiting list.

1:N Recursive Relationships

The classic example of a 1:N recursive relationship occurs in organizational charts, in which an employee has a manager who may, in turn, manage several other employees. Figure 5-40 shows an example managerial chart. Note that the relationship between employees is 1:N.

Figure 5-38

Data Model for a 1:1 Recursive Relationship

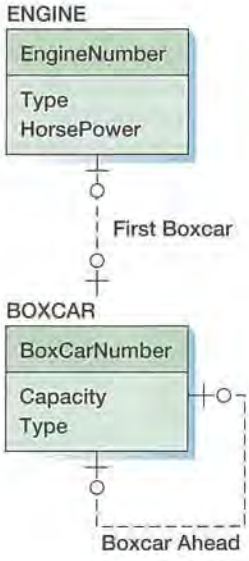
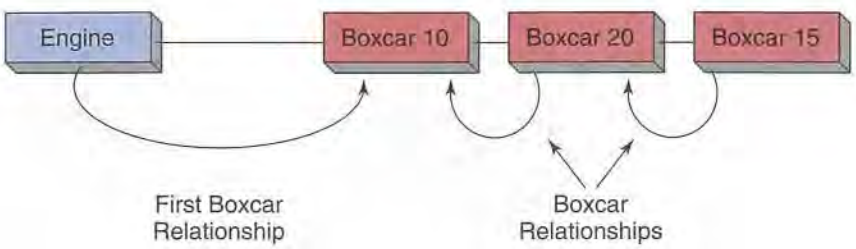


Figure 5-39

Sample Entities for the Data Model in Figure 5-38



Amazon Ex. 1008.0199

Figure 5-40
Managerial Relationships

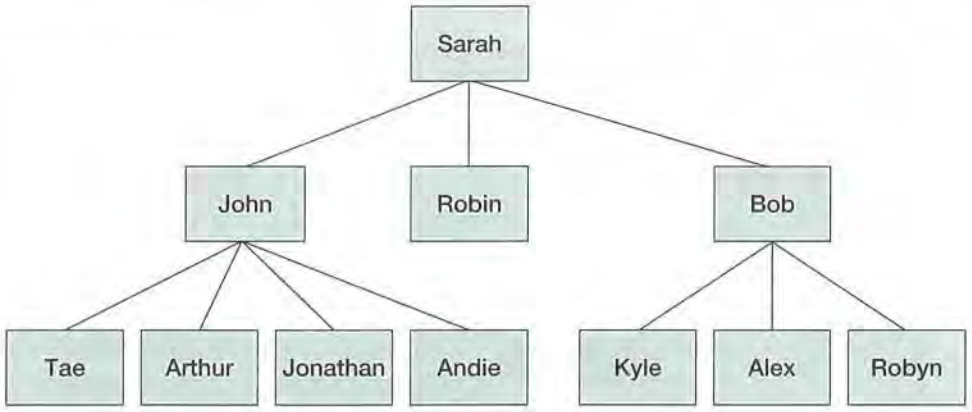


Figure 5-41
Data Model for the Management Structure in Figure 5-40 as a 1:N Recursive Relationship

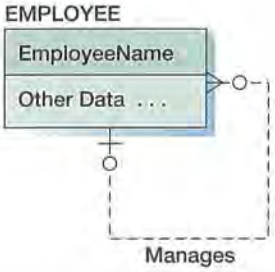


Figure 5-41 shows a data model for the managerial relationship. The crow's foot indicates that a manager may manage more than one employee. The relationship is optional to optional because one manager (the president) has no manager and because some employees manage no one.

Another example of a 1:N recursive relationship concerns maps. For example, a world map has a relationship to many continent maps, each continent map has a relationship to many nation maps, and so forth. A third example concerns biological parents where the relationship from PERSON to PERSON is shown by tracing either mother or father (but not both).

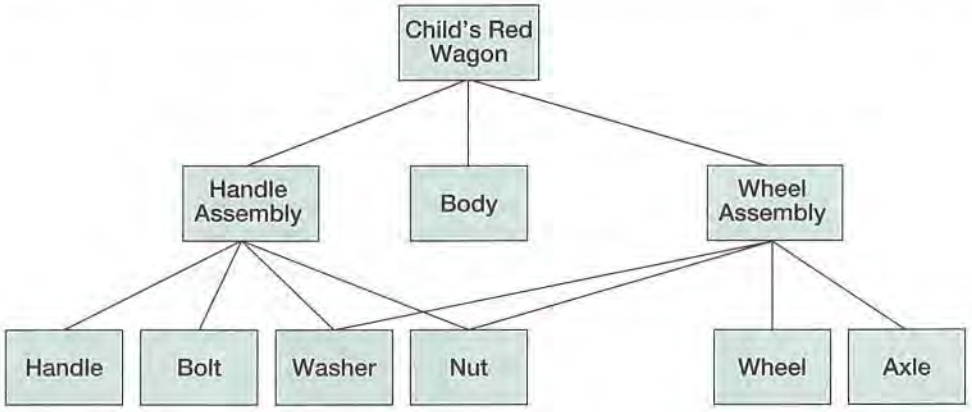
N:M Recursive Relationships

N:M recursive relationships occur frequently in manufacturing applications, where they are used to represent bills of materials. Figure 5-42 shows an example.

The key idea of a bill of materials is that one part is composed of other parts. A child's red wagon, for example, consists of a handle assembly, a body, and a wheel assembly, each of which is a part. The handle assembly, in turn, consists of a handle, a bolt, a washer, and a nut. The wheel assembly consists of wheels, axles, washers, and nuts. The relationship among the parts is N:M, because a part can be made up of many parts and because a part (such as washers and nuts) can be used in many parts.

The data model for a bill of materials is shown in Figure 5-43. Notice that each part has an N:M relationship to other parts. Because a part need not have any component parts, and because a part need not have any parts that contain it, the minimum cardinality is optional to optional.

Figure 5-42
Bill of Materials



Amazon Ex. 1008.0200

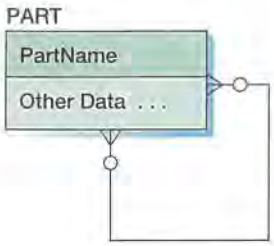


Figure 5-43

Data Model for the Bill of Materials in Figure 5-42 as an N:M Recursive Relationship

BY THE WAY What would happen to the data model if the diagram showed how many of each part are used? Suppose, for example, that the wheel assembly requires four washers and the handle assembly requires just one. The data model in Figure 5-43 will not be correct for this circumstance. In fact, adding Quantity to this N:M relationship is analogous to adding Price to the N:M relationship in Figure 5-22. See question 5.67.

N:M recursive relationships can be used to model directed networks, such as the flow of documents through organizational departments or the flow of gas through a pipeline. They also can be used to model the succession of parents, in which mothers, fathers, and stepparents are included.

If recursive structures seem hard to comprehend, don't fret. They may seem strange at first, but they are not difficult. Work through some data examples to gain confidence. Make up a train and see how the model in Figure 5-38 applies or change the example in Figure 5-40 from employees to departments and see how the model in Figure 5-41 needs to be adjusted. Once you have learned to identify recursive patterns, you'll find it easy to create models for them.

The Data Modeling Process

During the data modeling process, the development team analyzes user requirements and constructs a data model from forms, reports, data sources, and user interviews. The process is always iterative; a model is constructed from one form or report and then supplemented and adjusted as more forms and reports are analyzed. Periodically, users are asked for additional information, such as that needed to assess minimum cardinality. Users also review and validate the data model. During that review, prototypes evidencing data model constructs may need to be constructed, as explained earlier.

To give you an idea of the iterative nature of data modeling, we will consider the development of a simple data model for a university. As you read this example, strive to appreciate how the model evolves as more and more requirements are analyzed.

BY THE WAY One of the authors worked on a large data model for the U.S. Army's logistical system. The model contained over 500 different entity types, and it took a team of seven people more than a year to develop, document, and validate. On some occasions, the analysis of a new requirement indicated that the model had been conceived incorrectly, and days of work had to be redone. The most difficult aspect of the project was managing complexity. Knowing which entities related to which; whether an entity had already been defined; and whether a new entity was strong, weak, a supertype, or a subtype required a global understanding of the model. Memory was of poor help because an entity created in July could be a subtype of an entity created hundreds of entities earlier in February. To manage the model, the team used many different administrative tools. Keep this example in mind as you read through the development of the Highline University data model.

Amazon Ex. 1008.0201

Figure 5-44

Highline University Sample College Report

College of Business
Mary B. Jefferson, Dean

Phone: 232-1187 Campus Address: Business Building, Room 100

Department	Chairperson	Phone	Total Majors
Accounting	Jackson, Seymour P.	232-1841	318
Finance	HeuTeng, Susan	232-1414	211
Info Systems	Brammer, Nathaniel D.	236-0011	247
Management	Tuttle, Christine A.	236-9988	184
Production	Barnes, Jack T.	236-1184	212

Suppose the administration at a hypothetical university named Highline University wants to create a database to track colleges, departments, faculty, and students. To do this, a data modeling team has collected a series of reports as part of its requirements determination. In the next sections, we will analyze these reports to produce a data model.

The College Report

Figure 5-44 shows an example report from Highline University about one college within the university, specifically, the College of Business. This example is one instance of this report; Highline University has similar reports about other colleges, such as the College of Engineering and the College of Social Sciences. The data modeling team needs to gather enough examples to form a representative sample of all the college reports. Here, assume that the report in Figure 5-44 is representative.

Examining the report, we find data specific to the college—such as the name, dean, telephone number, and campus address—and also facts about each department within the college. These data suggest that the data model should have COLLEGE and DEPARTMENT entities with a relationship between them, as shown in Figure 5-45.

The relationship in Figure 5-45 is nonidentifying. This relationship is used because DEPARTMENT is not ID-dependent, and, logically, a DEPARTMENT is independent of a COLLEGE. We cannot tell from the report in Figure 5-44 whether a department can belong to many colleges. To answer this question, we need to ask the users or look at other forms and reports.

Assume we know from the users that a department belongs to just one college, and the relationship is thus 1:N from COLLEGE to DEPARTMENT. The report in Figure 5-44 does not show us the minimum cardinalities. Again, we must ask the users. Assume we learn from the users that a college must have at least one department, and a department must be assigned to exactly one college.

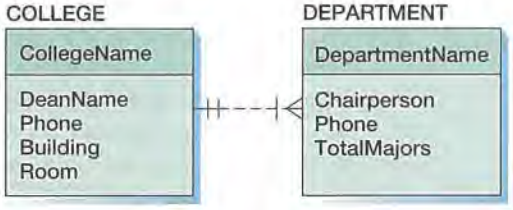
The Department Report

The Department Report shown in Figure 5-46 contains departmental data along with a list of the professors who are assigned to that department.

This report contains data concerning the department's campus address. Because these data do not appear in the DEPARTMENT entity in Figure 5-45, we need to add them, as shown

Figure 5-45

Data Model for the College Report in Figure 5-44



Amazon Ex. 1008.0202

Information Systems Department
College of Business

Chairperson: Brammer, Nathaniel D
Phone: 236-0011
Campus Address: Social Science Building, Room 213

Professor	Office	Phone
Jones, Paul D.	Social Science, 219	232-7713
Parks, Mary B	Social Science, 308	232-5791
Wu, Elizabeth	Social Science, 207	232-9112

Figure 5-46
Highline University Sample Department Report

in Figure 5-47(a). This is typical of the data modeling process. That is, entities and relationships are adjusted as additional forms, reports, and other requirements are analyzed.

Figure 5-47(a) also adds the relationship between DEPARTMENT and PROFESSOR. We initially model this as an N:M relationship, because a professor might have a joint appointment. The data modeling team must further investigate the requirements to determine whether joint appointments are allowed. If not, the relationship can be redefined as a nonidentifying 1:N, as shown in Figure 5-47(b).

Another possibility regarding the N:M relationship is that some attribute about the combination of a professor and a department is missing. If so, then an association pattern is more appropriate. At Highline, suppose the team finds a report that describes the title and employment terms for each professor in each department. Figure 5-47(c) shows an entity for such a report, named APPOINTMENT. As you would expect from the association pattern, APPOINTMENT is ID-dependent on both DEPARTMENT and PROFESSOR.

A chairperson is a professor, so another improvement on the model is to remove the Chairperson data from DEPARTMENT and replace it with a chairperson relationship. This has been done in Figure 5-47(d). In the Chairs/Chaired By relationship, the PROFESSOR is the parent entity. A professor can be a chair of zero or one departments, and a department must have exactly one professor as chair.

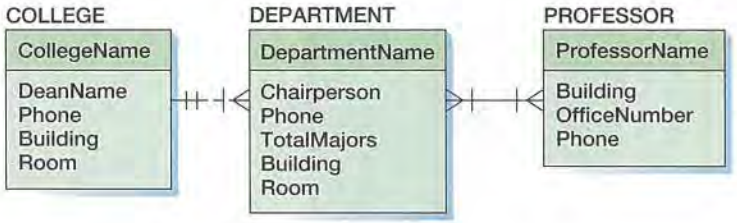
With the Chairs/Chaired By relationship, the attribute Chairperson is no longer needed in DEPARTMENT, so it is removed. Normally, a chairperson has his or her office in the department office; if this is the case, Phone, Building, and Room in DEPARTMENT duplicate Phone, Building, and OfficeNumber in PROFESSOR. Consequently, it might be possible to remove Phone, Building, and Room from DEPARTMENT. However, a professor may have a different phone from the official department phone, and the professor may also have an office outside of the department's office. Because of this possibility, we will leave Phone, Building, and Room in DEPARTMENT.

The Department/Major Report

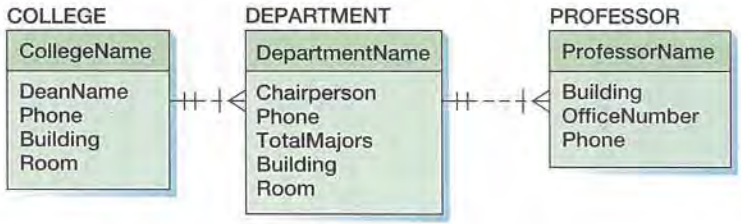
Figure 5-48 shows a report of a department and the students who major in that department. This report indicates the need for a new entity called STUDENT. Because students are not ID-dependent on departments, the relationship between DEPARTMENT and STUDENT is nonidentifying, as shown in Figure 5-49. We cannot determine the minimum cardinality from Figure 5-48, but assume that interviews with users indicate that a STUDENT must have MAJOR, but no MAJOR need have any students. Also, using the contents of this report as a guide, attributes StudentNumber, StudentName, and Phone are placed in STUDENT.

There are two subtleties in this interpretation of the report in Figure 5-48. First, observe that Major's Name was changed to StudentName when the attribute was placed in STUDENT. This was done because StudentName is more generic. Major's Name has no meaning outside the context of the Major relationship. Additionally, the report heading in Figure 5-48 has an ambiguity. Is the phone number for the department a value of DEPARTMENT.Phone or a value

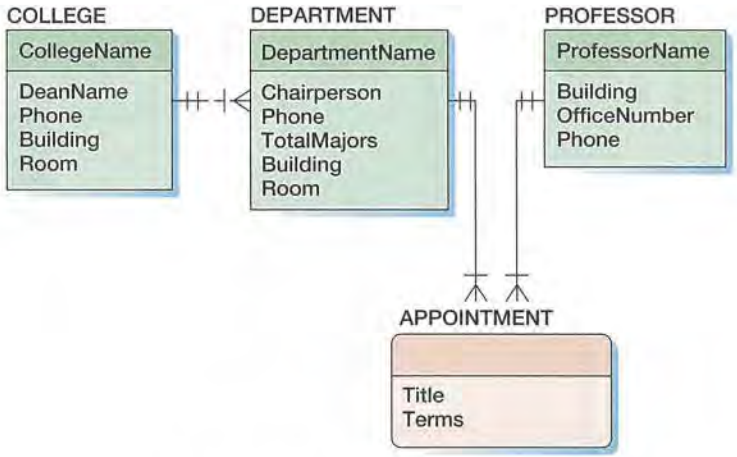
Amazon Ex. 1008.0203



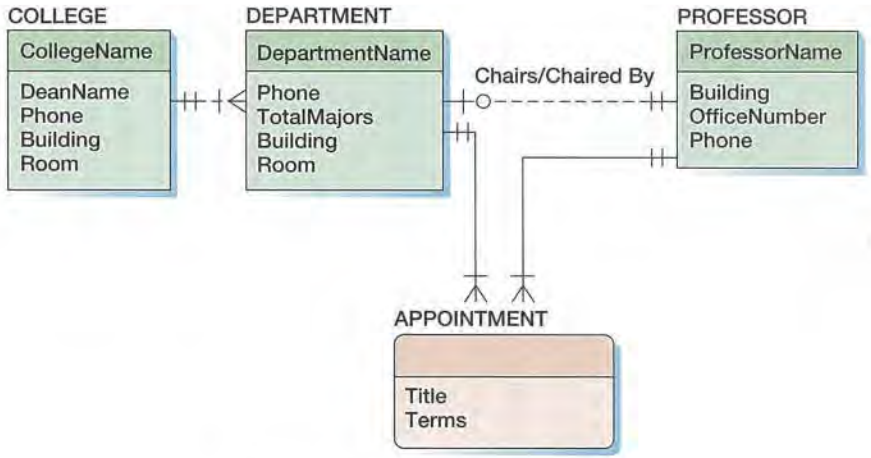
(a) Data Model Using an N:M Relationship



(b) Data Model Using 1:N Relationship



(c) Data Model Using Association Pattern



(d) Data Model Using Association Pattern and 1:1 Relationship

Figure 5-47

Alternate Data Models for the DEPARTMENT-to-PROFESSOR Relationship

Amazon Ex. 1008.0204

Student Major List
Information Systems Department

Chairperson: Brammer, Nathaniel D Phone: 236-0011

Major's Name	Student Number	Phone
Jackson, Robin R.	12345	237-8713
Lincoln, Fred J.	48127	237-8713
Madison, Janice A.	37512	237-8713

Figure 5-48
Highline University Sample Department Student Report

of PROFESSOR.Phone? The team needs to investigate this further with the users. Most likely, it is a value of DEPARTMENT.Phone.

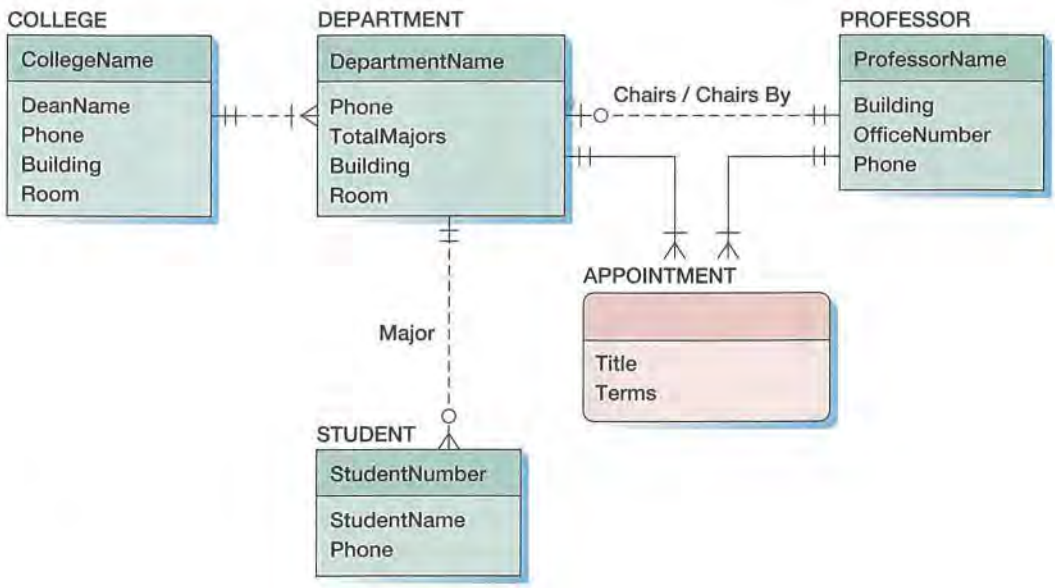
The Student Acceptance Letter

Figure 5-50 shows the acceptance letter that Highline sends to its incoming students. The data items in this letter that need to be represented in the data model are shown in boldface. In addition to data concerning the student, this letter also contains data regarding the student's major department as well as data about the student's adviser.

We can use this letter to add an Advises/Advised By relationship to the data model. However, which entity should be the parent of this relationship? Because an adviser is a professor, it is tempting to make PROFESSOR the parent. However, a professor acts as an adviser within the context of a particular department. Therefore, Figure 5-51 shows APPOINTMENT as the parent of STUDENT. To produce the report in Figure 5-50, the professor's data can be retrieved by accessing the related APPOINTMENT entity and then accessing that entity's PROFESSOR parent. This decision is not cut-and-dried, however. One can make a strong argument that the parent of the relationship should be PROFESSOR.

According to this data model, a student has at most one adviser. Also, a student must have an adviser, but no professor (via APPOINTMENT) need advise any students. These constraints cannot be determined from any of the reports shown and will need to be verified with the users. The acceptance letter uses the title *Mr.* in the salutation. Therefore, a new attribute called Title is added to STUDENT. Observe that this Title is different from the one in APPOINTMENT. This difference will need to be documented in the data model to avoid confusion. The acceptance letter also shows the need to add new home address attributes to STUDENT.

Figure 5-49
Data Model with STUDENT Entity



Amazon Ex. 1008.0205

Mr. Fred Parks
123 Elm Street
Los Angeles, CA 98002

Dear **Mr. Parks:**

You have been admitted as a major in the **Accounting** Department at Highline University, starting in the Fall Semester, 2009. The office of the Accounting Department is located in the **Business** Building, Room **210.**

Your adviser is professor **Elizabeth Johnson**, whose telephone number is **232-8740** and whose office is located in the **Business** Building, Room **227.** Please schedule an appointment with your adviser as soon as you arrive on campus.

Congratulations and welcome to Highline University!

Sincerely,

Jan P. Smathers
President

JPS/rkp

Figure 5-50
Highline University Sample Student Acceptance Letter

The acceptance letter reveals a problem. The name of the student is Fred Parks, but we have allocated only one attribute, StudentName, in STUDENT. It is difficult to reliably disentangle first and last names from a single attribute, so a better model is to have two attributes: StudentFirstName and StudentLastName. Similarly, note that the adviser in this letter is Elizabeth Johnson. So far, all professor names have been in the format Johnson, Elizabeth. To accommodate both forms of name, ProfessorName in PROFESSOR must be changed to the two attributes ProfessorFirstName and ProfessorLastName. A similar change is necessary for DeanName. These changes are shown in Figure 5-52, which is the final form of this data model.

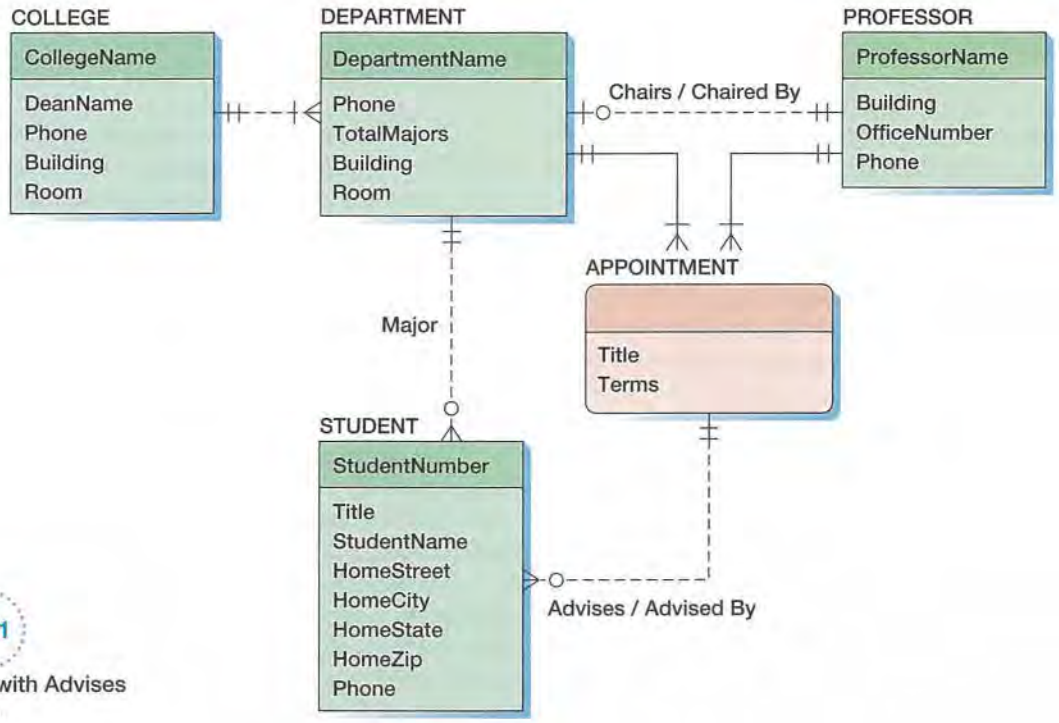


Figure 5-51
Data Model with Advises Relationship

Amazon Ex. 1008.0206

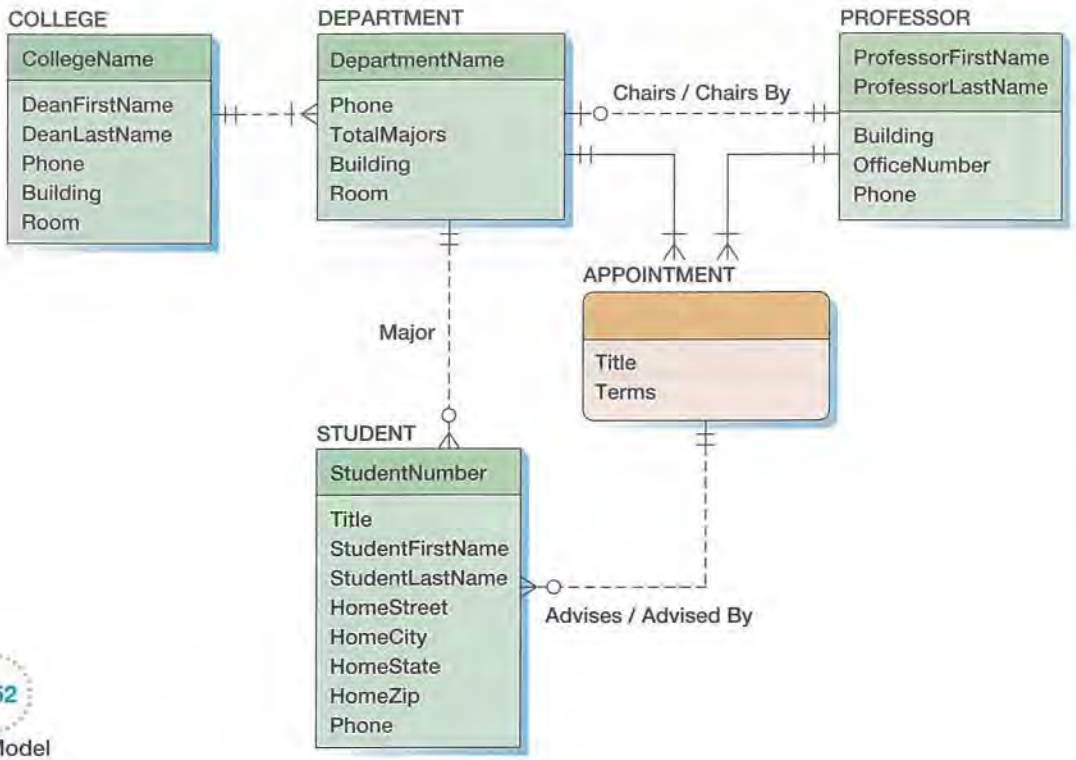


Figure 5-52
Final Data Model

This section should give you a feel for the nature of a data modeling project. Forms and reports are examined in sequence, and the data model is adjusted as necessary to accommodate the knowledge gained from each new form or report. It is very typical to revise the data model many, many times throughout the data modeling process. See question 5.68 for yet another possible revision.

Summary

When databases are developed as part of a new information systems project, the database design is accomplished in two phases. First, a data model is constructed from forms, reports, data sources, and other requirements. The data model is then transformed into a database design. A data model is a blueprint for a database design. Like blueprints for buildings, data models can be altered as necessary, with little effort. Once the database is constructed, however, such alterations are time consuming and very expensive.

The most prominent data model in use today is the entity-relationship, or E-R, data model. It was invented by Peter Chen and extended by others to include subtypes. An entity is something that users want to track. An entity class is a collection of entities of the same type and is described by the structure of the entities in the class. An entity instance is one entity of a given class. Entities have attributes that describe their characteristics. Identifiers are attributes that name entity instances. Composite identifiers consist of two or more attributes.

The E-R model includes relationships, which are associations among entities. Relationship classes are associations among entity classes, and relationship instances are associations among entity instances. Today, relationships are not allowed to have attributes. Relationships can be given names so that they can be identified.

The degree of a relationship is the number of entity types that participate in the relationship. Binary relationships have only two entity types. In practice, relationships of degrees greater than two are decomposed into multiple binary relationships.

The difference between an entity and a table is that you can express an entity relationship without specifying foreign keys. Working with entities reduces complexity and makes it easier to revise the data model as work progresses.

Relationships are classified according to their cardinality. Maximum cardinality is the maximum number of instances that can participate in a relationship instance. Minimum cardinality is the least number of entities that must participate in a relationship.

Amazon Ex. 1008.0207

Relationships commonly have one of three maximum cardinalities: 1:1, 1:N, or N:M. In rare instances, a maximum cardinality might be a specific number, such as 1:15. Relationships commonly have one of four basic minimum cardinalities: optional to optional, mandatory to optional, optional to mandatory, or mandatory to mandatory. In rare cases, the minimum cardinality is a specific number.

Unfortunately, many variations of the E-R model are in use. The original version represented relationships with diamonds. The Information Engineering version uses a line with a crow's foot, the IDEF1X version uses another set of symbols, and UML uses yet another set. To add further complication, many data modeling products have added their own symbols. In this text, we will use the IE Crow's Foot model with symbols, as summarized in Figure 5-14. Other models and techniques are summarized in Appendices B, C, and E.

An ID-dependent entity is an entity whose identifier includes the identifier of another entity. Such entities use an identifying relationship. In such relationships, the parent is always required, but the child (the ID-dependent entity) may or may not be required, depending on application requirements. Identifying relationships are shown with solid lines in E-R diagrams.

A weak entity is an entity whose existence depends on the presence of another entity. All ID-dependent entities are weak. Additionally, some entities are weak, but not ID-dependent. Some people believe such entities are not important; others believe they are.

A subtype entity is a special case of another entity called its supertype. Subtypes may be exclusive or inclusive. Exclusive subtypes sometimes have discriminators, which are attributes that specify a supertype's subtype. The most important (and perhaps only) reason for creating subtypes in a data model is to avoid value-inappropriate nulls.

Relationships among nonsubtype entities are called HAS-A relationships. Relationships among supertype/subtype entities are called IS-A relationships.

The elements of a data model are constructed by analyzing forms, reports, and data sources. Many forms and reports fall into common patterns. In this text, we discussed the 1:1, 1:N, and N:M strong entity patterns. We also discussed three patterns that use ID-dependent relationships: association, multivalue attribute, and version/instance. Some forms involve mixed identifying and nonidentifying patterns. Line items are the classic example of mixed forms, but there are other examples as well.

The for-use-by pattern indicates the need for subtypes. In some cases, subtypes differ because they have different attributes, but they also can differ because they have different relationships. A recursive relationship occurs when an entity has a relationship to itself. The three types of recursive relationship are 1:1, 1:N, and N:M.

The data modeling process is iterative. Forms and reports are analyzed, and the data model is created, modified, and adjusted as necessary. Sometimes, the analysis of a form or report will require that earlier work be redone. *C'est la vie!*

Key Terms

association pattern
attribute
binary relationship
cardinality
child
composite identifier
crow's foot symbol
data model
degree
discriminator
entity
entity class
entity instance
entity-relationship (E-R) diagrams
entity-relationship (E-R) model
exclusive subtype
extended E-R model
HAS-A relationship
ID-dependent entity
identifier
identifying relationship
IE Crow's foot model
inclusive subtype
Information Engineering (IE) model
Integrated Definition 1, Extended (IDEF1X)
IS-A relationship
mandatory-to-mandatory (M-M) relationship
mandatory-to-optional (M-O) relationship
many-to-many (N:M) relationship
maximum cardinality
minimum cardinality
nonidentifying relationship
one-to-many (1:N) relationship
one-to-one (1:1) relationship
optional-to-mandatory (O-M) relationship
optional-to-optional (O-O) relationship
parent
relationship
relationship class
relationship instance
strong entity
subtype
supertype
ternary relationship
Unified Modeling Language (UML)
weak entity

Amazon Ex. 1008.0208

Review Questions

5.1 Describe the two phases in designing databases that arise from the development of new information systems.

5.2 In general terms, explain how a data model could be used to design a database for a small video rental store.

5.3 Explain how a data model is like a building blueprint. What is the advantage of making changes during the data modeling stage?

5.4 Who is the author of the entity-relationship data model?

5.5 Define *entity*. Give an example of an entity (other than one presented in this chapter).

5.6 Explain the difference between an entity class and an entity instance.

5.7 Define *attribute*. Give an example attribute for the entity in your answer to Review Question 5.5.

5.8 Define *identifier*. Give an example identifier for the entity in your answer to Review Question 5.5.

5.9 Give an example of a composite identifier.

5.10 Define *relationship*. Give an example of a relationship (other than one presented in this chapter). Name your relationship.

5.11 Explain the difference between a relationship class and a relationship instance.

5.12 What is the degree of relationship? Give an example of a relationship of degree three (other than one presented in this chapter).

5.13 What is a binary relationship?

5.14 Explain the difference between an entity and a table. Why is this difference important?

5.15 What does cardinality mean?

5.16 Define the terms *maximum cardinality* and *minimum cardinality*.

5.17 Give examples of 1:1, 1:N, and N:M relationships (other than those presented in this chapter). Use the traditional diamond notation to diagram your examples.

5.18 Give an example for which the maximum cardinality must be an exact number.

5.19 Give examples of M-M, M-O, O-M, and O-O relationships (other than those presented in this chapter). Use the circle and hash mark notation on the diamond portrayal of relationships.

5.20 Explain, in general terms, how the traditional E-R model, the IE Crow's Foot version, the IDEF1X version, and the UML version differ. Which version is used primarily in this text?

5.21 Explain how the notations shown in Figure 5-7 differ.

5.22 Explain how the notations shown in Figure 5-9 differ.

5.23 What is an ID-dependent entity? Give an example of an ID-dependent entity (other than one presented in this chapter).

5.24 Explain how to determine the minimum cardinality of both sides of an ID-dependent relationship.

5.25 What rules exist when creating an instance of an ID-dependent entity? What rules exist when deleting the parent of an ID-dependent entity?

Amazon Ex. 1008.0209

5.26 What is an identifying relationship? How is it used?

5.27 Explain why the relationship between PRODUCT and VERSION discussed on page 154 is an identifying relationship.

5.28 What is a weak entity? How do weak entities relate to ID-dependent entities?

5.29 What distinguishes a weak entity from a strong entity that has a required relationship to another entity?

5.30 Define *subtype* and *supertype*. Give an example of a subtype–supertype relationship (other than one presented in this chapter).

5.31 Explain the difference between exclusive subtypes and inclusive subtypes. Give an example of each.

5.32 What is a discriminator?

5.33 Explain the difference between IS-A and HAS-A relationships.

5.34 What is the most important reason for using subtypes in a data model?

5.35 Describe the relationship between the structure of forms and reports and the data model.

5.36 Explain two ways forms and reports are used for data modeling.

5.37 Explain why the form and report in Figure 5-15 indicate that the underlying relationship is 1:1.

5.38 Why is it not possible to infer minimum cardinality from the form and report in Figure 5-15?

5.39 Describe two tests for determining if an entity is a strong entity.

5.40 Why does the form in Figure 5-17 not indicate that the underlying relationship is 1:N? What additional information is required to make that assertion?

5.41 Explain why two forms or reports are usually needed to infer maximum cardinality.

5.42 How can you assess minimum cardinality for the entities in the form in Figure 5-17?

5.43 Explain why the form and report in Figure 5-19 indicate that the underlying relationship is N:M.

5.44 Name three patterns that use ID-dependent relationships.

5.45 Explain how the association pattern differs from the N:M strong entity pattern. What characteristic of the report in Figure 5-21 indicates that an association pattern is needed?

5.46 In general terms, explain how to differentiate an N:M strong entity pattern from an association pattern.

5.47 Explain why two entities are needed to model multivalued attributes.

5.48 How do the forms in Figures 5-26 and 5-28 differ? How does this difference affect the data model?

5.49 Describe, in general terms, the archetype/instance pattern. Why is an ID-dependent relationship needed for this pattern? Use the CLASS/SECTION example shown in Figure 5-30 in your answer.

5.50 Explain what caused the entities in Figure 5-31 to change from ID-dependent entities.

5.51 Summarize the two sides in the argument about the importance of weak, but not ID-dependent, entities.

5.52 Give an example of the line-item pattern as it could be used to describe the contents of a shipment. Assume that the shipment includes the names and quantities of various items as well as each item's insured value. Place the insurance value per item in an ITEM entity.

Amazon Ex. 1008.0210