

Second Edition

Beginning
**Database
Design
Solutions**

Understanding and Implementing Database
Design Concepts for the Cloud and Beyond

Rod Stephens

WILEY

BEGINNING DATABASE DESIGN SOLUTIONS

INTRODUCTION	xxv
▶ PART 1 INTRODUCTION TO DATABASES AND DATABASE DESIGN	
CHAPTER 1 Database Design Goals	3
CHAPTER 2 Relational Overview	29
CHAPTER 3 NoSQL Overview	47
▶ PART 2 DATABASE DESIGN PROCESS AND TECHNIQUES	
CHAPTER 4 Understanding User Needs	83
CHAPTER 5 Translating User Needs into Data Models	111
CHAPTER 6 Extracting Business Rules	145
CHAPTER 7 Normalizing Data	163
CHAPTER 8 Designing Databases to Support Software	203
CHAPTER 9 Using Common Design Patterns	215
CHAPTER 10 Avoiding Common Design Pitfalls	241
▶ PART 3 A DETAILED CASE STUDY	
CHAPTER 11 Defining User Needs and Requirements	263
CHAPTER 12 Building a Data Model	283
CHAPTER 13 Extracting Business Rules	303
CHAPTER 14 Normalizing and Refining	313
▶ PART 4 EXAMPLE PROGRAMS	
CHAPTER 15 Example Overview	327
CHAPTER 16 MariaDB in Python	339
CHAPTER 17 MariaDB in C#	355
CHAPTER 18 PostgreSQL in Python	369

CHAPTER 19	PostgreSQL in C#	389
CHAPTER 20	Neo4j AuraDB in Python	401
CHAPTER 21	Neo4j AuraDB in C#	417
CHAPTER 22	MongoDB Atlas in Python	431
CHAPTER 23	MongoDB Atlas in C#	453
CHAPTER 24	Apache Ignite in Python	467
CHAPTER 25	Apache Ignite in C#	477
► PART 5	ADVANCED TOPICS	
CHAPTER 26	Introduction to SQL	489
CHAPTER 27	Building Databases with SQL Scripts	519
CHAPTER 28	Database Maintenance	533
CHAPTER 29	Database Security	545
APPENDIX A	Exercise Solutions	557
APPENDIX B	Sample Relational Designs	649
GLOSSARY	671
INDEX	683

BEGINNING

Database Design Solutions



BEGINNING

Database Design Solutions

UNDERSTANDING AND IMPLEMENTING
DATABASE DESIGN CONCEPTS FOR THE
CLOUD AND BEYOND

Second Edition

Rod Stephens

WILEY

Copyright © 2023 by John Wiley & Sons, Inc. All rights reserved.

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.
Published simultaneously in Canada and the United Kingdom.

ISBN: 978-1-394-15572-9

ISBN: 978-1-394-15583-5 (ebk.)

ISBN: 978-1-394-15584-2 (ebk.)

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4470, or on the web at www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at www.wiley.com/go/permission.

Trademarks: WILEY and the Wiley logo are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc. is not associated with any product or vendor mentioned in this book.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Further, readers should be aware that websites listed in this work may have changed or disappeared between when this work was written and when it is read. Neither the publisher nor authors shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services or for technical support, please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

If you believe you've found a mistake in this book, please bring it to our attention by emailing our reader support team at wileysupport@wiley.com with the subject line "Possible Book Errata Submission."

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic formats. For more information about Wiley products, visit our web site at www.wiley.com.

Library of Congress Control Number: 2022946646

Cover image: © mfto/Getty Images

Cover design: Wiley

*For those who read the back cover, dedication,
introductory chapter, and glossary. The five of us
need to stick together!*

*Also to Amy, Ken, and Elissa for sanity-preserving
doses of silliness.*

ABOUT THE AUTHOR

Rod Stephens started out as a mathematician, but while studying at MIT, he discovered how much fun programming is and he's been programming professionally ever since. He's a longtime developer, instructor, and author who has written more than 250 magazine articles and 37 books that have been translated into languages around the world.

During his career, Rod has worked on an eclectic assortment of applications in such fields as telephone switching, billing, repair dispatching, tax processing, wastewater treatment, concert ticket sales, cartography, optometry, and training for professional football teams. (That's U.S. football, not one of the kinds with the round ball. Or the kind with three downs. Or the kind with an oval field. Or the indoor kind. Let's just say NFL and leave it at that.)

Rod's popular C# Helper website (www.csharpHelper.com) receives millions of hits per year and contains thousands of tips, tricks, and example programs for C# programmers. His VB Helper website (www.vb-helper.com) contains similar material for Visual Basic programmers.

ABOUT THE TECHNICAL EDITOR

John Mueller is a freelance author and technical editor. He has writing in his blood, having produced 123 books and more than 600 articles to date. The topics range from networking to artificial intelligence and from database management to heads-down programming. Some of his current books include discussions of data science, machine learning, and algorithms. He also writes about computer languages such as C++, C#, and Python. His technical editing skills have helped more than 70 authors refine the content of their manuscripts. John has provided technical editing services to a variety of magazines, performed various kinds of consulting, and he writes certification exams. Be sure to read John's blog at <http://blog.johnmuellerbooks.com>. You can reach John on the Internet at John@JohnMuellerBooks.com. John also has a website at www.johnmuellerbooks.com.

ACKNOWLEDGMENTS

Thanks to Ken Brown, Jan Lynn Neal, Archana Pragash, Melissa Burlock, and all the others who worked so hard to make this book possible.

Thanks also to technical editor and longtime friend John Mueller for giving me the benefit of his valuable experience. You can see what John's up to at www.johnmuellerbooks.com.

CONTENTS

INTRODUCTION

XXV

PART 1: INTRODUCTION TO DATABASES AND DATABASE DESIGN

CHAPTER 1: DATABASE DESIGN GOALS	3
The Importance of Design	4
Information Containers	6
Strengths and Weaknesses of Information Containers	8
Desirable Database Features	9
CRUD	10
Retrieval	10
Consistency	11
Validity	11
Easy Error Correction	12
Speed	13
Atomic Transactions	13
ACID	14
BASE	16
NewSQL	17
Persistence and Backups	17
Low Cost and Extensibility	18
Ease of Use	19
Portability	19
Security	20
Sharing	21
Ability to Perform Complex Calculations	21
CAP Theorem	22
Cloud Considerations	22
Legal and Security Considerations	23
Consequences of Good and Bad Design	24
Summary	26
CHAPTER 2: RELATIONAL OVERVIEW	29
Picking a Database	30
Relational Points of View	31

Table, Rows, and Columns	32
Relations, Attributes, and Tuples	34
Keys	34
Indexes	36
Constraints	37
Domain Constraints	37
Check Constraints	37
Primary Key Constraints	38
Unique Constraints	38
Foreign Key Constraints	38
Database Operations	40
Popular RDBs	41
Spreadsheets	43
Summary	44
CHAPTER 3: NoSQL OVERVIEW	47
<hr/>	
The Cloud	47
Picking a Database	50
NoSQL Philosophy	50
NoSQL Databases	50
Document Databases	51
Key-Value Database	52
Column-Oriented Databases	53
Graph Databases	53
Street Networks	54
Communication Networks	55
Social Media Apps	55
E-Commerce Programs	55
Algorithms	56
Hierarchical Databases	56
Less Exotic Options	59
Flat Files	59
XML Files	60
XML Basics	61
XML Practices	64
XML Summary	66
JSON Files	67
Spreadsheets	69
More Exotic Options	70
Object	70

Deductive	70
Dimensional	70
Temporal	71
Database Pros and Cons	72
Relational	72
General NoSQL	73
Quick Guidelines	74
Summary	76

PART 2: DATABASE DESIGN PROCESS AND TECHNIQUES

CHAPTER 4: UNDERSTANDING USER NEEDS 83

Make a Plan	84
Bring a List of Questions	85
Functionality	85
Data Needs	86
Data Integrity	86
Security	87
Environment	88
Meet the Customers	88
Learn Who's Who	89
Pick the Customers' Brains	93
Walk a Mile in the User's Shoes	93
Study Current Operations	94
Brainstorm	94
Look to the Future	95
Understand the Customers' Reasoning	96
Learn What the Customers Really Need	97
Prioritize	98
Verify Your Understanding	99
Create the Requirements Document	101
Make Use Cases	102
Decide Feasibility	106
Summary	106

CHAPTER 5: TRANSLATING USER NEEDS INTO DATA MODELS 111

What Are Data Models?	112
User Interface Models	114
Semantic Object Models	118

Classes and Objects	119
Cardinality	120
Identifiers	120
Putting It Together	121
Semantic Views	122
Class Types	124
Simple Objects	124
Composite Objects	124
Compound Objects	125
Hybrid Objects	125
Association Objects	126
Inherited Objects	128
Comments and Notes	129
Entity-Relationship Models	130
Entities, Attributes, and Identifiers	131
Relationships	132
Cardinality	133
Inheritance	134
Additional Conventions	136
Comments and Notes	137
Relational Models	137
Converting Semantic Object Models	138
Converting ER Diagrams	140
Summary	142
CHAPTER 6: EXTRACTING BUSINESS RULES	145
<hr/>	
What Are Business Rules?	145
Identifying Key Business Rules	147
Extracting Key Business Rules	152
Multi-Tier Applications	154
Summary	158
CHAPTER 7: NORMALIZING DATA	163
<hr/>	
What Is Normalization?	163
First Normal Form (1NF)	164
Second Normal Form (2NF)	173
Third Normal Form (3NF)	177
Stopping at Third Normal Form	181
Boyce-Codd Normal Form (BCNF)	181
Fourth Normal Form (4NF)	185

Fifth Normal Form (5NF)	190
Domain/Key Normal Form (DKNF)	193
Essential Redundancy	195
The Best Level of Normalization	197
NoSQL Normalization	197
Summary	199
CHAPTER 8: DESIGNING DATABASES TO SUPPORT SOFTWARE	203
<hr/>	
Plan Ahead	204
Document Everything	204
Consider Multi-Tier Architecture	205
Convert Domains into Tables	205
Keep Tables Focused	206
Use Three Kinds of Tables	207
Use Naming Conventions	209
Allow Some Redundant Data	210
Don't Squeeze in Everything	211
Summary	212
CHAPTER 9: USING COMMON DESIGN PATTERNS	215
<hr/>	
Associations	216
Many-to-Many Associations	216
Multiple Many-to-Many Associations	216
Multiple-Object Associations	218
Repeated Attribute Associations	221
Reflexive Associations	222
One-to-One Reflexive Associations	223
One-to-Many Reflexive Associations	224
Hierarchical Data	225
Hierarchical Data with NoSQL	228
Network Data	229
Network Data with NoSQL	231
Temporal Data	232
Effective Dates	232
Deleted Objects	233
Deciding What to Temporalize	234
Logging and Locking	236
Audit Trails	236
Turnkey Records	237
Summary	238

CHAPTER 10: AVOIDING COMMON DESIGN PITFALLS	241
Lack of Preparation	241
Poor Documentation	242
Poor Naming Standards	242
Thinking Too Small	244
Not Planning for Change	245
Too Much Normalization	248
Insufficient Normalization	248
Insufficient Testing	249
Performance Anxiety	249
Mishmash Tables	250
Not Enforcing Constraints	253
Obsession with IDs	253
Not Defining Natural Keys	256
Summary	257
PART 3: A DETAILED CASE STUDY	
CHAPTER 11: DEFINING USER NEEDS AND REQUIREMENTS	263
Meet the Customers	263
Pick the Customers' Brains	265
Determining What the System Should Do	265
Determining How the Project Should Look	267
Determining What Data Is Needed for the User Interface	268
Determining Where the Data Should Come From	269
Determining How the Pieces of Data Are Related	269
Determining Performance Needs	271
Determining Security Needs	272
Determining Data Integrity Needs	273
Write Use Cases	275
Write the Requirements Document	279
Demand Feedback	280
Summary	281
CHAPTER 12: BUILDING A DATA MODEL	283
Semantic Object Modeling	283
Building an Initial Semantic Object Model	283
Improving the Semantic Object Model	286
Entity-Relationship Modeling	289

Building an ER Diagram	289
Building a Combined ER Diagram	291
Improving the Entity-Relationship Diagram	293
Relational Modeling	294
Putting It All Together	298
Summary	299
CHAPTER 13: EXTRACTING BUSINESS RULES	303
<hr/>	
Identifying Business Rules	303
Courses	304
CustomerCourses	306
Customers	307
Pets	307
Employees	307
Orders	307
OrderItems	308
InventoryItems	308
TimeEntries	308
Shifts	309
Persons	309
Phones	309
Vendors	309
Drawing a New Relational Model	310
Summary	310
CHAPTER 14: NORMALIZING AND REFINING	313
<hr/>	
Improving Flexibility	313
Verifying First Normal Form	315
Verifying Second Normal Form	318
Pets	319
TimeEntries	320
Verifying Third Normal Form	321
Summary	323
<hr/>	
PART 4: EXAMPLE PROGRAMS	
<hr/>	
CHAPTER 15: EXAMPLE OVERVIEW	327
<hr/>	
Tool Choices	327
Jupyter Notebook	329

Visual Studio	331
Database Adapters	332
Packages in Jupyter Notebook	333
Packages in Visual Studio	334
Program Passwords	336
Summary	336
CHAPTER 16: MariaDB IN PYTHON	339
<hr/>	
Install MariaDB	340
Run HeidiSQL	340
Create the Program	343
Install pymysql	344
Create the Database	344
Define Tables	346
Create Data	348
Fetch Data	350
Summary	352
CHAPTER 17: MariaDB IN C#	355
<hr/>	
Create the Program	355
Install MySqlConnection	356
Create the Database	356
Define Tables	358
Create Data	360
Fetch Data	364
Summary	366
CHAPTER 18: PostgreSQL IN PYTHON	369
<hr/>	
Install PostgreSQL	370
Run pgAdmin	371
Design the Database	371
Create a User	371
Create the Database	373
Define the Tables	374
Define the customers Table	374
Define the orders Table	376
Define the order_items Table	377
Create the Program	378
Install Psycopy	379

Connect to the Database	379
Delete Old Data	380
Create Customer Data	380
Create Order Data	382
Create Order Item Data	383
Close the Connection	384
Perform Queries	384
Summary	386
CHAPTER 19: PostgreSQL IN C#	389
<hr/>	
Create the Program	389
Install Npgsql	389
Connect to the Database	390
Delete Old Data	391
Create Customer Data	392
Create Order Data	393
Create Order Item Data	395
Display Orders	396
Summary	399
CHAPTER 20: Neo4j AuraDB IN PYTHON	401
<hr/>	
Install Neo4j AuraDB	402
Nodes and Relationships	404
Cypher	404
Create the Program	405
Install the Neo4j Database Adapter	405
Action Methods	405
delete_all_nodes	406
make_node	407
make_link	407
execute_node_query	408
find_path	409
Org Chart Methods	410
build_org_chart	410
query_org_chart	411
Main Program	412
Summary	414

CHAPTER 21: Neo4j AuraDB IN C#	417
Create the Program	418
Install the Neo4j Driver	418
Action Methods	419
DeleteAllNodes	419
MakeNode	420
MakeLink	421
ExecuteNodeQuery	422
FindPath	422
Org Chart Methods	423
BuildOrgChart	424
QueryOrgChart	424
Main	426
Summary	428
CHAPTER 22: MongoDB ATLAS IN PYTHON	431
Not Normal but Not Abnormal	432
XML, JSON, and BSON	432
Install MongoDB Atlas	434
Find the Connection Code	436
Create the Program	439
Install the PyMongo Database Adapter	439
Helper Methods	440
person_string	440
connect_to_db	441
delete_old_data	442
create_data	442
query_data	444
Main Program	449
Summary	450
CHAPTER 23: MongoDB ATLAS IN C#	453
Create the Program	454
Install the MongoDB Database Adapter	454
Helper Methods	454
PersonString	455
DeleteOldData	456
CreateData	457
QueryData	458

Main Program	462
Summary	465
CHAPTER 24: APACHE IGNITE IN PYTHON	467
Install Apache Ignite	468
Start a Node	468
Without Persistence	469
With Persistence	470
Create the Program	470
Install the pyignite Database Adapter	471
Define the Building Class	471
Save Data	471
Read Data	473
Demonstrate Volatile Data	473
Demonstrate Persistent Data	474
Summary	474
CHAPTER 25: APACHE IGNITE IN C#	477
Create the Program	477
Install the Ignite Database Adapter	478
The Main Program	479
The Building Class	480
The <i>WriteData</i> Method	480
The <i>ReadData</i> Method	482
Demonstrate Volatile Data	483
Demonstrate Persistent Data	483
Summary	483
PART 5: ADVANCED TOPICS	
CHAPTER 26: INTRODUCTION TO SQL	489
Background	491
Finding More Information	491
Standards	492
Multistatement Commands	493
Basic Syntax	495
Command Overview	495
<i>Create Table</i>	498
<i>Create Index</i>	503

<i>Drop</i>	504
<i>Insert</i>	504
<i>Select</i>	506
<i>SELECT</i> Clause	506
<i>FROM</i> Clause	507
<i>WHERE</i> Clause	511
<i>GROUP BY</i> Clause	511
<i>ORDER BY</i> Clause	512
<i>Update</i>	513
<i>Delete</i>	514
Summary	515
CHAPTER 27: BUILDING DATABASES WITH SQL SCRIPTS	519
<hr/>	
Why Bother with Scripts?	519
Script Categories	520
Database Creation Scripts	520
Basic Initialization Scripts	520
Data Initialization Scripts	520
Cleanup Scripts	521
Saving Scripts	521
Ordering SQL Commands	522
Summary	531
CHAPTER 28: DATABASE MAINTENANCE	533
<hr/>	
Backups	533
Data Warehousing	537
Repairing the Database	538
Compacting the Database	538
Performance Tuning	538
Summary	542
CHAPTER 29: DATABASE SECURITY	545
<hr/>	
The Right Level of Security	545
Passwords	546
Single-Password Databases	546
Individual Passwords	546
Operating System Passwords	547
Good Passwords	547
Privileges	548

Initial Configuration and Privileges	553
Too Much Security	553
Physical Security	554
Summary	555
APPENDIX A: EXERCISE SOLUTIONS	557
<hr/>	
APPENDIX B: SAMPLE RELATIONAL DESIGNS	649
<hr/>	
GLOSSARY	671
<hr/>	
<i>INDEX</i>	683

INTRODUCTION

It has been estimated that more than 80 percent of all computer programming is database-related. This is certainly easy to believe. After all, a database can be a powerful tool for doing exactly what computer programs do best: store, manipulate, and display data.

Even many programs that seem at first glance to have little to do with traditional business-oriented data use databases to make processing easier. In fact, looking back on 40 some years of software development experience, I'm hard-pressed to think of a single nontrivial application that I've worked on that didn't use some kind of database.

Not only do databases play a role in many applications, but they often play a critical role. If the data is not properly stored, it may become corrupted, and the program will be unable to use it meaningfully. If the data is not properly organized, the program may be unable to find what it needs in a reasonable amount of time.

Unless the database stores its data safely and effectively, the application will be useless no matter how well-designed the rest of the system may be. The database is like the foundation of a building; without a strong foundation, even the best crafted building will fail, sometimes spectacularly (the Leaning Tower of Pisa notwithstanding).

With such a large majority of applications relying so heavily on databases, you would expect everyone involved with application development to have a solid, formal foundation in database design and construction. Everyone, including database designers, application architects, programmers, database administrators, and project managers, should ideally understand what makes a good database design. Even an application's key customers and users could benefit from understanding how databases work.

Sadly, that is usually not the case. Many IT professionals have learned what they know about databases through rumor, trial-and-error, tarot cards, and painful experience. Over the years, some develop an intuitive feel for what makes a good database design, but they may still not understand the reasons a design is good or bad, and they may leave behind a trail of rickety, poorly constructed programs built on shaky database foundations.

This book provides the tools you need to design a database. It explains how to determine what should go in a database and how a database should be organized to ensure data integrity and a reasonable level of performance. It explains techniques for designing a database that is strong enough to store data safely and consistently, flexible enough to allow the application to retrieve the data it needs quickly and reliably, and adaptable enough to accommodate a reasonable amount of change.

With the ideas and techniques described in this book, you will be able to build a strong foundation for database applications.

WHO THIS BOOK IS FOR

This book is intended for IT professionals and students who want to learn how to design, analyze, and understand databases. The material will benefit those who want a better high-level understanding of databases such as proposal managers, architects, project managers, and even customers. The material will also benefit those who will actually design, build, and work with databases such as database designers, database administrators, and programmers. In many projects, these roles overlap so the same person may be responsible for working on the proposal, managing part of the project, and designing and creating the database.

This book is aimed at readers of all experience levels. It does not assume that you have any previous experience with databases or programs that use them. It doesn't even assume that you have experience with computers. All you really need is a willingness and desire to learn.

WHAT THIS BOOK COVERS

This book explains database design. It tells how to plan a database's structure so the database will be robust, resistant to errors, and flexible enough to accommodate a reasonable amount of future change. It explains how to discover database requirements, build data models to study data needs, and refine those models to improve the database's effectiveness.

The book solidifies these concepts by working through a detailed example that designs a (sort of) realistic database. Later chapters explain how to actually build databases using a few different database products. The book finishes by describing topics you need to understand to keep a database running effectively such as database maintenance and security.

WHAT YOU NEED TO USE THIS BOOK

This book explains database design. It tells how to determine what should go in a database and how the database should be structured to give the best results.

This book does not focus on actually *creating* the database. The details of database construction are different for different database tools, so to remain as generally useful as possible, this book doesn't concentrate on any particular database system. You can apply most of the techniques described here equally to whatever database tool you use, whether it's MariaDB, PostgreSQL, SQL Server, or some other database product.

NOTE Most database products include free editions that you can use for smaller projects. For example, SQL Server Express Edition, Oracle Express Edition, and MariaDB Community Server are all free.

To remain database-neutral, most of the book does not assume you are using a particular database, so you don't need any particular software or hardware. To work through the exercises, all you need is a pencil and some paper. You are welcome to type solutions into your computer if you like, but you may actually find working with pencil and paper easier than using a graphical design tool to draw pictures, at least until you are comfortable with database design and are ready to pick a computerized design tool.

Chapters 16 through 25 build example databases using particular database offerings, so their material is tied to the databases that they demonstrate. Chapter 15, "Example Overview," introduces those chapters and lists the databases that they use.

To experiment with the SQL database language described in Chapter 26, "Introduction to SQL," and Chapter 27, "Building Databases with SQL Scripts," you need any database product that supports SQL (that includes pretty much all relational databases) running on any operating system.

HOW THIS BOOK IS STRUCTURED

The chapters in this book are divided into five parts plus appendixes. The chapters in each part are described here. If you have previous experience with databases, you can use these descriptions to decide which chapters to skim and which to read in detail.

Part I: Introduction to Databases and Database Design

The chapters in this part of the book provide background that is necessary to understand the chapters that follow. You can skim some of this material if it is familiar to you, but don't take it too lightly. If you understand the fundamental concepts underlying database design, it will be easier to understand the point behind important design concepts presented later.

Chapter 1, "Database Design Goals," explains the reasons people and organizations use databases. It explains a database's purpose and conditions that it must satisfy to be useful. This chapter also describes the basic ACID (Atomicity, Consistency, Isolation, Durability) and CRUD (Create, Read, Update, Delete) features that any good database should have. It explains in high-level general terms what makes a good database and what makes a bad database.

Chapter 2, "Relational Overview," explains basic relational database concepts such as tables, rows, and columns. It explains the common usage of relational database terms in addition to the more technical terms that are sometimes used by database theorists. It describes different kinds of constraints that databases use to guarantee that the data is stored safely and consistently.

Chapter 3, "NoSQL Overview," explains the basics of NoSQL databases, which are growing quickly in popularity. Those databases include document, key-value, column-oriented, and graph databases. Both relational and NoSQL databases can run either locally or in the cloud, but many NoSQL databases are more cloud-oriented, largely because they are newer technology so they're cloud-native.

Part II: Database Design Process and Techniques

The chapters in this part of the book discuss the main pieces of relational database design. They explain how to understand what should be in the database, develop an initial design, separate important pieces of the database to improve flexibility, and refine and tune the design to provide the most stable and useful design possible.

Chapter 4, “Understanding User Needs,” explains how to learn about the users’ needs and gather user requirements. It tells how to study the users’ current operations, existing databases (if any), and desired improvements. It describes common questions that you can ask to learn about users’ operations, desires, and needs, and how to build the results into requirements documents and specifications. This chapter explains what use cases are and shows how to use them and the requirements to guide database design and to measure success.

Chapter 5, “Translating User Needs into Data Models,” introduces data modeling. It explains how to translate the user’s conceptual model and the requirements into other, more precise models that define the database design rigorously. This chapter describes several database modeling techniques, including user-interface models, semantic object models, entity-relationship diagrams, and relational models.

Chapter 6, “Extracting Business Rules,” explains how a database can handle business rules. It explains what business rules are, how they differ from database structure requirements, and how you can identify business rules. This chapter explains the benefits of separating business rules from the database structure and tells how to achieve that separation.

Chapter 7, “Normalizing Data,” explains one of the most important tools in relational database design: normalization. Normalization techniques allow you to restructure a database to increase its flexibility and make it more robust. This chapter explains various forms of normalization, emphasizing the stages that are most common and important: first, second, and third normal forms (1NF, 2NF, and 3NF). It explains how each of these kinds of normalization helps prevent errors and tells why it is sometimes better to leave a database slightly less normalized to improve performance.

Chapter 8, “Designing Databases to Support Software,” explains how databases fit into the larger context of application design and the development life cycle. This chapter explains how later development depends on the underlying database design. It discusses multi-tier architectures that can help decouple the application and database so there can be at least some changes to either without requiring changes to both.

Chapter 9, “Using Common Design Patterns,” explains some common patterns that are useful in many applications. Some of these techniques include implementing various kinds of relationships among objects, storing hierarchical and network data, recording temporal data, and logging and locking.

Chapter 10, “Avoiding Common Design Pitfalls,” explains some common design mistakes that occur in database development. It describes problems that can arise from insufficient planning, incorrect normalization, and obsession with ID fields and performance.

Part III: A Detailed Case Study

If you follow all of the examples and exercises in the earlier chapters, by this point you will have seen all of the major steps for producing a good database design. However, it's often useful to see all the steps in a complicated process put together in a continuous sequence. The chapters in this part of the book walk through a detailed case study following all the phases of database design for the fictitious Pampered Pet database.

Chapter 11, “Defining User Needs and Requirements,” walks through the steps required to analyze the users’ problem, define requirements, and create use cases. It describes interviews with fictitious customers that are used to identify the application’s needs and translate them into database requirements.

Chapter 12, “Building a Data Model,” translates the requirements gathered in the previous chapter into a series of data models that precisely define the database’s structure. This chapter builds user interface models, entity-relationship diagrams, semantic object models, and relational models to refine the database’s initial design. The final relational models match the structure of a relational database fairly closely, so they are easy to implement.

Chapter 13, “Extracting Business Rules,” identifies the business rules embedded in the relational model constructed in the previous chapter. It shows how to extract those rules in order to separate them logically from the database’s structure. This makes the database more robust in the face of future changes to the business rules.

Chapter 14, “Normalizing and Refining,” refines the relational model developed in the previous chapter by normalizing it. It walks through several versions of the database that are in different normal forms. It then selects the degree of normalization that provides a reasonable trade-off between robust design and acceptable performance.

Part IV: Example Programs

Though this book focuses on abstract database concepts that do not depend on a particular database product, it's also worth spending at least some time on more concrete implementation issues. The chapters in this part of the book describe some of those issues and explain how to build simple example programs that demonstrate a few different database products.

Chapter 15, “Example Overview,” provides a roadmap for the chapters that follow. It tells which chapters use which databases and how to get the most out of those chapters. Chapters 16 through 25 come in pairs, with the first describing an example in Python and the second describing a similar (although not always identical) program in C#.

Chapters 16 and 17 describe examples that use the popular MariaDB column-oriented relational database running on the local machine.

Chapters 18 and 19 demonstrate the (also popular) PostgreSQL database, also running on the local machine.

Chapters 20 and 21 show how to use the Neo4j AuraDB graph database running in the cloud.

Chapters 22 and 23 describe examples that use the MongoDB Atlas document database, also running in the cloud.

Chapters 24 and 25 demonstrate the Apache Ignite key-value database running locally.

These examples are just intended to get you started. They are relatively simple examples and they do not show all of the possible combinations. For example, you can run an Apache Ignite database in the cloud if you like; there were just too many combinations to cover them all in this book.

Part V: Advanced Topics

Although this book does not assume you have previous database experience, that doesn't mean it cannot cover some more advanced subjects. The chapters in this part of the book explain some more sophisticated topics that are important but not central to database design.

Chapter 26, "Introduction to SQL," provides an introduction to SQL (Structured Query Language). It explains how to use SQL commands to add, insert, update, and delete data. By using SQL, you can help insulate a program from the idiosyncrasies of the particular database product that it uses to store data.

Chapter 27, "Building Databases with SQL Scripts," explains how to use SQL scripts to build a database. It explains the advantages of this technique, such as the ability to create scripts to initialize a database before performing tests. It also explains some of the restrictions on this method, such as the fact that the user may need to create and delete tables in a specific order to satisfy table relationships.

Chapter 28, "Database Maintenance," explains some of the database maintenance issues that are part of any database application. Though performing and restoring backups, compressing tables, rebuilding indexes, and populating data warehouses are not strictly database design tasks, they are essential to any working application.

Chapter 29, "Database Security," explains database security issues. It explains the kinds of security that some database products provide. It also explains some additional techniques that can enhance database security such as using database views to appropriately restrict the users' access to data.

Appendixes

The book's appendixes provide additional reference material to supplement the earlier chapters.

Appendix A, "Exercise Solutions," gives solutions to the exercises at the end of most of the book's chapters so that you can check your progress as you work through the book.

Appendix B, "Sample Relational Designs," shows some sample designs for a variety of common database situations. These designs store information about such topics as books, movies, documents, customer orders, employee timekeeping, rentals, students, teams, and vehicle fleets.

The Glossary provides definitions for useful database and software development terms. The Glossary includes terms defined and used in this book in addition to a few other useful terms that you may encounter while reading other database material.

HOW TO USE THIS BOOK

Because this book is aimed at readers of all experience levels, you may find some of the material familiar if you have previous experience with databases. In that case, you may want to skim chapters covering material that you already thoroughly understand.

If you are familiar with relational databases, you may want to skim Chapter 1, “Database Design Goals,” and Chapter 2, “Relational Overview.” Similarly if you have experience with NoSQL databases, you may want to skip Chapter 3, “NoSQL Overview.”

If you have previously helped write project proposals, you may understand some of the questions you need to ask users to properly understand their needs. In that case, you may want to skim Chapter 4, “Understanding User Needs.”

If you have built databases before, you may understand at least some of the data normalization concepts explained in Chapter 7, “Normalizing Data.” This is a complex topic, however, so I recommend that you not skip this chapter unless you really know what you’re doing.

If you have extensive experience with SQL, you may want to skim Chapter 26, “Introduction to SQL.” (Many developers who have used but not designed databases fall into this category.)

In any case, I strongly recommend that you at least skim the material in every chapter to see if there are any new concepts you can pick up along the way. At least look at the Exercises at the end of each chapter before you decide that you can safely skip to the next. If you don’t know how to outline the solutions to the Exercises, then you should consider looking at the chapter more closely.

Different people learn best in different ways. Some learn best by listening to lecturers, others by reading, and others by doing. Everyone learns better by combining learning styles. You will get the most from this book if you read the material and then work through the Exercises. It’s easy to think to yourself, “Yeah, that makes sense” and believe you understand the material, but working through some of the Exercises will help solidify the material in your mind. Doing so may also help you see new ways that you can apply the concepts covered in the chapter.

NOTE *Normally, when I read a new technical book, I work through every example, modifying the problems to see what happens if I try different things not covered by the author. I work through as many questions and exercises as I can until I reach the point where more examples don’t teach me anything new (or I’m tired of breaking my system and having to reinstall things). Then I move on. It’s one thing to read about a concept in the chapter; it’s another to try to apply it to data that is meaningful to you.*

After you have learned the ideas in the book, you can use it for a reference. For example, when you start a new project, you may want to refer to Chapter 4, “Understanding User Needs,” to refresh your memory about the kinds of questions you should ask users to discover their true needs.

Visit the book’s website to look for updates and addendums. If readers find typographical errors or places where a little additional explanation may help, I’ll post updates on the website.

Finally, if you get stuck on a really tricky concept and need a little help, email me at RodStephens@csharpHelper.com and I’ll try to help you out.

NOTE TO INSTRUCTORS

Database programming is boring. Maybe not to you and me, who have discovered the ecstatic joy of database design, the thrill of normalization, and the somewhat risqué elation brought by slightly denormalizing a database to achieve optimum performance. But let’s face it, to a beginner, database design and development can be a bit dull.

There’s little you can do to make the basic concepts more exciting, but you can do practically anything with the data. At some point it’s useful to explain how to design a simple inventory system, but that doesn’t mean you can’t use other examples designed to catch students’ attention. Data that relates to the students’ personal experiences or that is just plain outrageous keeps them awake and alert (and most of us know that it’s easier to teach students who are awake).

The examples in this book are intended to demonstrate the topic at hand but not all of them are strictly business-oriented. I’ve tried to make them cover a wide variety of topics from serious to silly. To keep your students interested and alert, you should add new examples from your personal experiences and from your students’ interests.

I’ve had great success in my classroom using examples that involve sports teams (particularly local rivalries), music (combining classics such as Bach, Beethoven, and Tone Loc), the students in the class (but be sure not to put anyone on the spot), television shows and stars, comedians, and anything else that interests the students.

For exercises, encourage students to design databases that they will find personally useful. I’ve had students build databases that track statistics for the players on their favorite football teams, inventory their DVD or CD collections, file and search recipe collections, store data on “Magic: The Gathering” trading cards, track role-playing game characters, record information about classic cars, and schedule athletic tournaments. (The tournament scheduler didn’t work out too well—the scheduling algorithms were too tricky.) One student even built a small but complete inventory application for his mother’s business that she actually found useful. I think he was as shocked as anyone to discover he’d learned something practical.

When students find an assignment interesting and relevant, they become emotionally invested and will apply the same level of concentration and intensity to building a database that they normally reserve for console gaming, *Star Wars*, and *World of Warcraft*. They may spend hours crafting a database to track WoW alliances just to fulfill a 5-minute assignment. They may not catch every nuance of domain/key normal form, but they’ll probably learn a lot about building a functional database.

NOTE TO STUDENTS

If you're a student and you peeked at the previous section, "Note to Instructors," shame on you! If you didn't peek, do so now.

Building a useful database can be a lot of work, but there's no reason it can't be interesting and useful to you when you're finished. Early in your reading, pick some sort of database that you would find useful (see the previous section for a few ideas) and think about it as you read through the text. When the book talks about creating an initial design, sketch out a design for your database. When the book explains how to normalize a database, normalize yours. As you work through the exercises, think about how they would apply to your dream database.

Don't be afraid to ask your instructor if you can use your database instead of one suggested by the book for a particular assignment (unless you have one of those instructors who hand out extra work to anyone who crosses their path; in that case, keep your head down). Usually an instructor's thought process is quite simple: "I don't care what database you use as long as you learn the material." Your instructor may want your database to contain several related tables so that you can create the complexity needed for a particular exercise, but it's usually not too hard to make a database complicated enough to be interesting.

When you're finished, you will hopefully know a lot more about database design than you do now, and if you're persistent, you might just have a database that's actually good for something. Hopefully you'll also know how to design other useful databases in the future. (And when you're finished, email me at RodStephens@csharpHelper.com and let me know what you built!)

CONVENTIONS

To help you get the most from the text and keep track of what's happening, we've used a number of conventions throughout the book.

NOTE *Tips, hints, tricks, and asides to the current discussion are offset and placed in italics like this.*

Activities are exercises that you should work through, following the text in the book.

1. They usually consist of a set of steps.
2. Each step has a number.
3. Follow the steps with your copy of the database.

continues

(continued)

How It Works

After most activity instruction sections, the process you've stepped through is explained in detail.

As for styles in the text:

- We *highlight* new terms and important words when we introduce them.
- We show keyboard strokes like this: Ctrl+A.
- We show filenames, URLs, and code within the text like so: `SELECT * FROM Students.`
- We present blocks of code like this:

We use a monofont type with no highlighting for code examples.

SOURCE CODE

As you work through the examples in this book, you may choose either to type in all the code manually or to use the source code files that accompany the book. All of the source code used in this book is available for download at www.wiley.com/go/beginningdbdesign2e.

CONTACTING THE AUTHOR

If you have questions, suggestions, comments, want to swap cookie recipes, or just want to say “Hi,” email me at RodStephens@csharpHelper.com. I can't promise that I'll be able to help you with every problem, but I do promise to try.

DISCLAIMER

Many of the examples in this book were chosen for interest or humorous effect. They are not intended to disparage anyone. I mean no disrespect to police officers (or anyone else who regularly carries a gun), plumbers, politicians, jewelry store owners, street luge racers (or anyone else who wears helmets and Kevlar body armor to work), or college administrators. Or anyone else for that matter.

Well, maybe politicians.

PART 1

Introduction to Databases and Database Design

- **Chapter 1:** Database Design Goals
- **Chapter 2:** Relational Overview
- **Chapter 3:** NoSQL Overview



The chapters in this part of the book provide background that is useful when studying database design.

Chapter 1 explains the reasons why database design is important. It discusses the goals that you should keep in mind while designing databases. If you keep those goals in mind, then you can stay focused on the end result and not become bogged down in the minutiae of technical details. If you understand those goals, then you will also know when it might be useful to bend the rules a bit.

Chapter 2 provides background on relational databases. It explains common relational database terms and concepts that you need to understand for the chapters that follow. You won't get as much out of the rest of the book if you don't understand the terminology.

Chapter 3 describes NoSQL databases. While this book (and most other database books) focuses on relational databases, there are other kinds of databases that are better suited to some tasks. NoSQL databases provide some alternatives that may work better for you under certain circumstances. (I once worked on a 40-developer project that failed largely because it used the wrong kind of database. Don't let that happen to you!)

Even if you're somewhat familiar with databases, give these chapters at least a quick glance to ensure that you don't miss anything important. Pay particular attention to the terms described in Chapter 2, because you'll need to know them later.

1

Database Design Goals

Using modern database tools, just about anyone can build a database. The question is, will the resulting database be useful?

A database won't do you much good if you can't get data out of it quickly, reliably, and consistently. It won't be useful if it's full of incorrect or contradictory data, nor will it be useful if it is stolen, lost, or corrupted by data that was only half written when the system crashed.

You can address all of these potential problems by using modern database tools, a good database design, and a pinch of common sense, but only if you understand what those problems are so you can avoid them.

The first step in the quest for a useful database is understanding database goals. What should a database do? What makes a database useful and what problems can it solve? Working with a powerful database tool without goals is like flying a plane through clouds without a compass—you have the tools you need but no sense of direction.

This chapter describes the goals of database design. By studying information containers, such as files that can play the role of a database, the text defines properties that good databases have and problems that they should avoid.

In this chapter, you will learn about the following:

- Why a good database design is important
- The strengths and weaknesses of various kinds of information containers that can act as databases
- How computerized databases can benefit from those strengths and avoid those weaknesses
- How good database design helps achieve database goals
- What CRUD, ACID, and BASE are, and why they are relevant to database design

THE IMPORTANCE OF DESIGN

Forget for a moment that this book is about designing databases and consider software design in general. Software design plays a critical role in software development. The design lays out the general structure and direction that future development will take. It determines which parts of the system will interact with other parts. It decides which subsystems will provide support for other pieces of the application.

If an application's underlying design is flawed, the system as a whole is at risk. Bad assumptions in the design creep into the code at the application's lowest levels, resulting in flawed subsystems. Higher-level systems built on those subsystems inherit those design flaws, and soon their code is corrupted, too.

Sometimes, a sort of decay pervades the entire system and nobody notices until relatively late in the project. The longer the project continues, the more entrenched the incorrect assumptions become, and the more reluctant developers are to scrap the whole design and start over. The longer problems remain in the system, the harder they are to remove. At some point, it might be easier to throw everything away and start over from scratch, a decision that few managers will want to present to upper management.

SPACE SPAT

An engineer friend of mine was working on a really huge satellite project. After a while, the engineers all realized that the project just wasn't feasible given the current state of technology and the design. Eventually, the project manager was forced to admit this to upper management and he was fired. The new project manager stuck it out for a while and then he, too, was forced to confess to upper management that the project was unfeasible. He, too, was fired.

For a while, this process continued—with a new manager taking over, realizing the hopelessness of the design, and being fired. That is, until eventually even upper management had to admit the project wasn't going to work out and the whole thing collapsed.

They could have saved time, money, and several careers if they had spent more time up-front on the design and either fixed the problems or realized right away that the project wasn't going to work and scrapped it at the start.

Building an application is often compared to building a house or skyscraper. You probably wouldn't start building a multibillion-dollar skyscraper without a comprehensive design that is based on well-established architectural principles. Unfortunately, software developers often rush off to start coding as soon as they possibly can because coding is more fun and interesting than design is. Coding also lets developers tell management and customers how many lines of code they have written, so it seems like they are making progress even if the lines of code are corrupted by false assumptions. Only later do they realize that the underlying design is flawed, the code they wrote is worthless, and the project is in serious trouble.

Now, let's get back to database design. Few parts of an application's design are as critical as the database's design. The database is the repository of the information that the rest of the application manages and displays to the users. If the database doesn't store the right data, doesn't keep the data safe, or doesn't let the application find the data it needs, then the application has little chance for success. Here, the *garbage-in, garbage-out (GIGO)* principle is in full effect. If the underlying data is unsound, it doesn't matter what the application does with it; the results will be suspect at best.

For example, imagine that you've built an order-tracking system that can quickly fetch information about a customer's past orders. Unfortunately, every time you ask the program to fetch a certain customer's records, it returns a slightly different result. Although the program can find data quickly, the results are not trustworthy enough to be usable.

For another example, imagine that you have built an amazing program that can track the thousands of tasks that make up a single complex job, such as building a cruise liner or passenger jet. It can track each task's state of completion, determine when you need to order new parts for them to be ready for future construction phases, and can even determine the present value of future purchases so you can decide whether it is better to buy parts now or wait until they are needed. Unfortunately, the program takes hours to recalculate the complex task schedule and pricing details. Although the calculations are correct, they are so slow that users cannot reasonably make any changes. Changing the color of the fabric of a plane's seats or the tile used in a cruise liner's hallways could delay the whole project. (I once worked on a project with a similar issue. It worked, but it was so slow that it became a serious problem.)

For a final example, suppose you have built an efficient subscription application that lets customers subscribe to your company's quarterly newsletters, data services, and sarcastic demotivational quote of the day. It lets you quickly find and update any customer's subscriptions, and it always consistently shows the same values for a particular customer. Unfortunately, when you change the price of one of your publications, you find that not all of the customers' records show the updated price. Some customers' subscriptions are at the new rate, some are at the old rate, and some seem to be at a rate you've never seen before. (This example isn't as far-fetched as it may seem. Some systems allow you to offer sale prices or special incentives to groups of customers, or they allow sales reps to offer special prices to particular customers. That kind of system requires careful design if you want to be able to do things like change standard prices without messing up customized pricing.)

Poor database design can lead to these and other annoying and potentially expensive scenarios. A good design creates a solid foundation on which you can build the rest of the application.

Experienced developers know that the longer a bug remains in a system, the harder it is to find and fix. From that it logically follows that it is extremely important to get the design right before you start building on it.

Database design is no exception. A flawed database design can doom a project to failure before it has begun as surely as ill-conceived software architecture, poor implementation, or incompetent programming can.

INFORMATION CONTAINERS

What is a database? This may seem like a trivial question, but if you take it seriously the result can be pretty enlightening. By studying the strengths and weaknesses of some physical objects that meet the definition of a database, you can learn about the features that you might like a computerized database to have.

DEFINITION *A database is a tool that stores data and lets you create, read, update, and delete the data in some manner.*

This is a pretty broad definition and includes a lot of physical objects that most people don't think of as modern databases. For example, Figure 1.1 shows a box full of business cards, a notebook, a filing cabinet full of customer records, and your brain, all of which fit this definition. Each of these physical databases has advantages and disadvantages that can give insight into the features that you might like in a computer database.

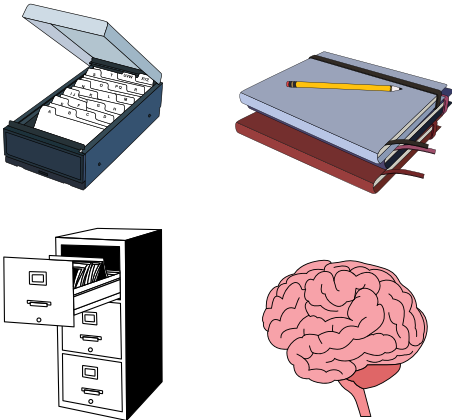


FIGURE 1.1

A box of business cards is useful as long as it doesn't contain too many cards. You can find a particular piece of data (for example, the phone number for your favorite Canadian restaurant) by looking through all the cards. You can easily expand the database by shoving more cards into the box, at least up to a point. If you have more than a dozen or so business cards, finding a particular card can be time consuming. You can even rearrange the cards a bit to improve performance for cards you use often. Each time you use a card, you can move it to the front of the box. Over time, those that are used most often will migrate to the front.

A notebook (the cardboard and paper kind, not the small laptop kind) is small, easy to use, easy to carry, doesn't require electricity, and doesn't need to boot before you can use it. A notebook database

is also easily extensible because you can buy another notebook to add to your collection when the first one is full. However, a notebook's contents are arranged sequentially. If you want to find information about a particular topic, you'll have to look through the pages one at a time until you find what you want. The more data you have, the harder this kind of search becomes.

A filing cabinet can store a lot more information than a notebook, and you can easily expand the database by adding more files or cabinets. Finding a particular piece of information in the filing cabinet can be easier than finding it in a notebook, as long as you are searching for the type of data used to arrange the records. If the filing cabinet is full of customer information sorted by customer name, and you want to find a particular customer's data, then you're in luck. If you want to find all of the customers who live in a certain city, you'll have to dig through the files one at a time.

Your brain is the most sophisticated database ever created. It can store an incredible amount of data and allows you to retrieve a particular piece of data in several different ways. For example, right now you could probably easily answer the following questions about the restaurants that you visit frequently:

- Which is closest to your current location?
- Which has the best desserts?
- Which has the best service?
- Which is least expensive?
- Which is the best for a business lunch?
- Which is your overall favorite?
- Why don't we go there tonight?

Your brain provides many different ways you can access the same restaurant information. You can search based on a variety of keys (such as location, quality of dessert, expense, and so forth). To answer these questions with a box of business cards, a notebook, or a filing cabinet would require a long and grueling search.

Still your brain has some drawbacks, at least as a database. Most notably it forgets. You may be able to remember an incredible number of things, but some become less reliable or disappear completely over time. Do you remember the names of all of your elementary school teachers? I don't. (I don't remember my own teachers' names, much less yours!)

Your brain also gets tired, and when it is tired it is less accurate.

Although your brain is good at certain tasks, such as recognizing faces or picking restaurants, it is not so good at other tasks like providing an accurate list of every item a particular customer purchased in the past year. Those items have less emotional significance than, for example, your spouse's name, so they're harder to remember.

All of these information containers (business cards, notebooks, filing cabinets, and your brain) can become contaminated with misleading, incorrect, and contradictory information. If you write different versions of the same information in a notebook, the data won't be consistent. Later when you try to look up the data, you may find either version first and may not even remember that there's

another version. (Your brain can become especially cluttered with inconsistent and contradictory information, particularly if you listen to politicians during an election year.)

The following section summarizes some of the strengths and weaknesses of these information containers.

STRENGTHS AND WEAKNESSES OF INFORMATION CONTAINERS

By understanding the strengths and weaknesses of information containers like those described in the previous section, you can learn about features that would be useful in a computerized database. So, what are some of those strengths and weaknesses?

The following list summarizes the advantages of some information containers:

- None of these databases require electricity so they are safe from power failures (although your brain requires food; as the dormouse said, feed your head).
- These databases keep data fairly safe and permanent (barring fires and memory loss). The data doesn't just disappear.
- These databases (excluding your brain) are inexpensive and easy to buy.
- These databases have simple user interfaces so that almost anyone can use them.
- Using these databases, it's fairly easy to add, edit, and remove data.
- The filing cabinet lets you quickly locate data if you search for it in the same way it is arranged (for example, by customer name).
- Your brain lets you find data by using different keys (for example, by location, cost, or quality of service).
- All of these databases allow you to find every piece of information that they contain, although it may take a while to dig through it all.
- All of these databases (except possibly your brain) provide consistent results as long as the facts they store are consistent. For example, two people using the same notebook will find the same data. Similarly if you look at the same notebook at a later time, it will show the same data you saw before (if it hasn't been modified).
- All of these databases except the filing cabinet are portable.
- Your brain can perform complex calculations, at least of a limited type and number.
- All of these databases provide atomic transactions.

The final advantage is a bit more abstract than the others so it deserves some additional explanation. An *atomic transaction* is a possibly complex series of actions that is considered to be a single operation by those who are not involved directly in performing the transaction.

The classic example is transferring money from one bank account to another. Suppose Alice writes Bob a check for \$100 and you need to transfer the money between their accounts. You pick up the

account book, subtract \$100 from Alice’s record, add \$100 to Bob’s record, and then put the notebook down. Someone else who uses the notebook might see it before the transaction (when Alice has the \$100) or after the transaction (when Bob has the \$100), but they won’t see it during the transaction where the \$100 has been subtracted from Alice but not yet given to Bob. The office bullies aren’t allowed to grab the notebook from your hands when you’re halfway through and play keep-away. It’s an all-or-nothing transaction.

In addition to their advantages, information containers like notebooks and filing cabinets have some disadvantages. It’s worth studying these disadvantages so that you can try to avoid them when you build computerized databases.

The following list summarizes some of the disadvantages that these information containers have:

- All of these databases can hold incomplete, incorrect, or contradictory data.
- Some of them are easy to lose or steal. Someone could grab your notebook while you’re eating lunch or read over your shoulder on the bus. You could even forget your notebook at the security counter as you dash to catch your flight.
- In all of these databases, correcting large errors in the data can be difficult. For example, it’s easy to use a pen to change one person’s address in an address notebook. It’s much harder to update hundreds of addresses if a new city is created in your area. (This recently happened near where I live.) Such a circumstance might require a tedious search through a set of business cards, a notebook, or a filing cabinet. It may be years before your brain makes the switch completely.
- These databases are relatively slow at creating, retrieving, updating, and deleting data. Your brain is much faster than the others at some tasks but is not good at manipulating a lot of information all at once. For example, how quickly can you list your 20 closest friends in alphabetical order? Even picking your closest friends can be difficult at times. (And don’t think you can cheat by using Facebook because you probably have hundreds of friends there and we only want your top 20.)
- Your brain can give different results at different times depending on uncontrollable factors such as your mood, how tired you are, and even whether you’re hungry.
- Each of these databases is located in a single place so it cannot be easily shared. Each also cannot be easily backed up, so if the original is lost or destroyed, you lose your data.

The following section considers how you can translate these strengths and weaknesses into features to prefer or avoid in a computerized database.

DESIRABLE DATABASE FEATURES

By looking at the advantages and disadvantages of physical databases, you can create a list of features that a computerized database should have. Some are fundamental characteristics that any database must have. (“You should be able to get data from it.” How obvious is that?)

Most of these features, however, depend at least in part on good database design. If you don’t craft a good design, you’ll miss out on some or all of the benefits of these features. For example, any decent

database provides backup features, but a good design can make backup and recovery a lot quicker and easier.

The following sections describe some of the features that a good database system should provide and explain to what degree they depend on good database design.

CRUD

CRUD stands for the four fundamental database operations that any database should provide: Create, Read, Update, and Delete. If you read database articles and discussions on the web, you will often see people tossing around the term *CRUD*. (They may be using the term just to sound edgy and cool. Now that you know the term, you can sound cool, too!)

You can imagine some specialized data-gathering devices that don't support all these methods. For example, an airplane's black-box flight data recorders record flight information and later play it back without allowing you to modify the data. In general, however, if it doesn't have *CRUD*, it's not a database.

CRUD is more a general feature of databases than it is a feature of good database design. However, a good database design provides *CRUD* efficiently. For example, suppose you design a database to track times for your canuggling league (look it up online and don't let autocorrect send you to definitions of "snuggling"), and you require that the addresses for participants include a *State* value that is present in the *States* table. When you create a new record (the *C* in *CRUD*), the database must validate the new *State* entry (so no one enters "confusion" as their state, even if it is true). Similarly, when you update a record (the *U* in *CRUD*), the database must validate the modified *State* entry. When you delete an entry in the *States* table (the *D* in *CRUD*), the database must verify that no *Participant* records use that state. Finally, when you read data (the *R* in *CRUD*), the database design determines whether you find the data you want in seconds, hours, or not at all.

Many of the concepts described in the following sections relate to *CRUD* operations.

Retrieval

Retrieval is another word for "read," the *R* in *CRUD*. (Happily "retrieval" also starts with *R*, so you don't need to memorize a new acronym.) Your database should allow you to find every piece of data. There's no point putting something in the database if there's no way to retrieve it later. (That would be a "data black hole," not a database.)

The database should allow you to structure the data so that you can find particular pieces of data in one or more specific ways. For example, you should be able to find a customer's billing record by searching for customer name or customer ID.

Ideally the database will also allow you to structure the data so that it is relatively quick and easy to fetch data in a particular manner.

For example, suppose you want to see where your customers live so you can decide whether you should start a delivery service in a new city. To obtain this information, it would be helpful to be able to find customers based on their addresses. Ideally you could optimize the database structure so that you can quickly search for customers by address.

In contrast, you probably don't need to search for customers by their middle name too frequently. (Imagine a customer calling you and saying, "Can you look up my record? I don't remember if I paid my bill last month. I also don't remember my account number or my last name but my middle name is 'Konfused.'") It would be nice if the common search by address was faster than the rare search by middle name.

Being able to find all of the data in the database quickly and reliably is an important part of database design. Finding the data that you need in a poorly designed database can take hours or days instead of mere seconds.

Consistency

Another aspect of the R in CRUD is *consistency*. (The fact that "consistency" and CRUD both start with a C is purely coincidental. Don't get too excited.) A database should provide consistent results. If you perform the same search twice in a row, you should get the same results. Another user who performs the same search should also get the same results. (Of course, this assumes that the underlying data hasn't changed in the meantime. You can't expect your net worth query to return the same results every day when stock prices fluctuate wildly.)

A well-built database product can ensure that the exact same query returns the same result, but the design also plays an important role. If the database is poorly designed, you might be able to store conflicting data in different parts of the database. For example, you might be able to store one set of contact information in a customer's order and a different set of information in the main customer record. Later, if you need to contact the customer with a question about the order, which contact information should you use?

Validity

Validity is closely related to the idea of consistency. Consistency means different parts of the database don't hold contradictory views of the same information. Validity means data is checked where possible against other pieces of data in the database. In CRUD terms, data can be validated when a record is created, updated, or deleted.

Just like physical data containers, a computerized database can hold incomplete, incorrect, or contradictory data. You can never protect a database from users who can't spell or who just plain enter the wrong information, but a good database design can help prevent some kinds of errors that a physical database cannot prevent.

For example, the database can easily verify that data has the correct type. If the user sees a `Date` field and enters "No thanks, I'm married," the database can tell that this is not a valid date format and can refuse to accept the value. Similarly, it can tell that "Old" is not a valid `Age`, "Lots" is not a valid `Quantity`, and 3 is not a valid `PhoneNumber`.

The database can also verify that a value entered by the user is present in another part of the database. For example, a poor typist trying to enter CO in a `State` field might type CP instead. The database can check a list of valid states and refuse to accept the data when it doesn't find CP listed. The user interface could also make the user pick the state from a drop-down list and avoid this problem, but the database should still protect itself against invalid data wherever possible.

The database can also check some kinds of constraints. Suppose the database contains a book-ordering system. When the customer orders 500 copies of this book (who wouldn't want that many copies?), the database can check another part of the database to see if that many copies are available (most bookstores carry only a few copies of any given book) and suggest that the customer backorder the copies if there aren't enough.

A good database design also helps protect the database against incorrect changes. Suppose you run a high-end coffee delivery service called the Brew Crew, and you've decided to drop coverage for a nearby city. When you try to remove that city from your list of valid locations, the database can tell you if you have existing customers in that city. Depending on the database's design, it could refuse to allow you to remove the city until you have apologized to those customers and removed them from the database.

All these techniques rely on a good, solid database design. They still can't protect you from a user who types first names in the last name field or who keeps accidentally bumping THE CAPS LOCK KEY, but it can prevent many types of errors that a paper and cardboard notebook can't.

Easy Error Correction

Even a perfectly designed database cannot ensure perfect validity. How can the database know that a customer's name is supposed to be spelled Pheidoux, not Fido as typed by your order entry clerk?

Correcting a single error in a notebook is fairly easy. Simply cross out the wrong value and write in the new one.

Correcting systematic errors in a notebook is a lot harder. Suppose you hire a summer intern to go door-to-door selling household products and he writes up a lot of orders for "duck tape" not realizing that the actual product is "duct tape." Fixing all the mistakes could be tedious and time consuming. (Of course tedious and time-consuming jobs are what summer interns are for, so you could make him fix them.) You could just ignore the problem and leave the orders misspelled, but then how would you tell when a customer really wants to tape a duck?

In a computerized database, this sort of correction is trivial. A simple database command can update every occurrence of the product name "duck tape" throughout the whole system. (In fact, this kind of fix is sometimes too easy to make. If you aren't careful, you may accidentally change the names of *every* product to "duct tape." You can prevent this by building a safe user interface for the database or by being really, really careful.)

Easy correction of errors is a built-in feature of computerized databases, but to get the best advantage from this feature you need a good design. If order information is contained in a free-formatted text section, the database will have trouble fixing typos. If you put the product name in a separate field, the database can make this change easily.

Although easy corrections are *almost* free, you need to do a little design work to make them as efficiently and effectively as possible.

Speed

An important aspect of all of the CRUD components is speed. A well-designed database can create, read, update, and delete records quickly.

There's no denying that a computerized database is a lot faster than a notebook or a filing cabinet. Instead of processing dozens of records per hour, a computerized database can process dozens or hundreds per second. (I once worked with a billing center that processed around 3 million accounts every three days.)

Good design plays a critical role in database efficiency. A poorly organized database may still be faster than the paper equivalent, but it will be a lot slower than a well-designed database.

CAN'T SEE THE FOREST FOR THE TREES

The billing center I mentioned two paragraphs earlier had a simple but annoying problem: they couldn't find the customers who owed them the most money. Every three days, the computer would print out a list of customers who owed money, and the list created a stack of paper almost 3 feet tall. (Sadly, I'm not exaggerating. Three feet of paper every three days! That's a small forest of trees' worth of paper every month.) Unfortunately, the list was randomly ordered (probably ordered by customer ID or shoe size or something equally unhelpful), so they couldn't figure out who owed the most money. The majority of the customers owed only a few dollars—too little to worry about—but a few customers owed tens of thousands of dollars.

We captured this printout electronically and sorted the accounts by balance. It turned out that the really problematic customers only filled two or three pages and the first five or so customers owed more than all of the others in the entire 3-foot tall stack combined.

I didn't include this story just to impress you with my mad ninja programming skillz (to be honest, it was a pretty easy project), but to illustrate how database design can make a big difference in performance. Had the database been designed properly to begin with, it would have been trivial to find the most problematic customers in a just few seconds.

Not all changes to a database's design can produce dramatic results, but design definitely plays an important role in performance.

Atomic Transactions

Recall that an atomic transaction is a possibly complex series of actions that is considered as a single operation by those not involved directly in performing the transaction. If you transfer \$100 from Alice's account to Bob's account, no one else can see the database while it is in an intermediate state where the money has been removed from Alice's account and not yet added to Bob's.

The transaction either happens completely or none of its pieces happen—it cannot happen halfway. Atomic transactions are important for maintaining consistency and validity, and are thus important for the R and U parts of CRUD.

Physical data containers like notebooks support atomic transactions because typically only one person at a time can use them. Unless you're distracted and wander off to lunch halfway through, you can finish a series of operations before you let someone else have a turn at the notebook.

Some of the most primitive kinds of electronic databases, such as flat files and XML files (which are described later in this book), don't intrinsically support atomic transactions, but the more advanced relational database products do. Those databases allow you to start a transaction and perform a series of operations. You can then either *commit* the transaction to make the changes permanent or *roll back* the transaction to undo them all and restore the database to the state it had before you started the transaction.

These databases also automatically roll back any transaction that is open when the database halts unexpectedly. For example, suppose you start a transaction, take \$100 from Alice's account, and then your company's mascot (a miniature horse) walks through the computer room, steps on a power strip, and kills the power to your main computer. When you restart the database (after sending the mascot to the HR department for a stern lecture), it automatically rolls the transaction back so that Alice gets her money back. You'll need to try the transaction again, but at least no money has been lost by the system.

Atomic transactions are more a matter of properly using database features rather than database design. If you pick a reasonably advanced database product and use transactions properly, you gain their benefits. If you decide to use flat files to store your data, you'll need to implement transactions yourself.

ACID

This section provides some more detail about the transactions described in the previous section rather than discussing a new feature of physical data containers and computerized databases.

ACID is an acronym describing four features that an effective transaction system should provide. *ACID* stands for Atomicity, Consistency, Isolation, and Durability.

Atomicity means transactions are atomic. The operations in a transaction either all happen or none of them happen.

Consistency means the transaction ensures that the database is in a consistent state before and after the transaction. In other words, if the operations within the transaction would violate the database's rules, the transaction is rolled back. For example, suppose the database's rules say an account cannot make a payment that would result in a balance less than \$0. Also, suppose that Alice's account holds only \$75. You start a transaction, add \$100 to Bob's account, and then try to remove \$100 from Alice's. That would put Alice \$25 below \$0, violating the database's rules, so the transaction is

canceled and we all try to forget that this ugly incident ever occurred. (Actually, we probably bill Alice an outrageous overdraft fee for writing a bad check.)

Isolation means the transaction isolates the details of the transaction from everyone except the person making the transaction. Suppose you start a transaction, remove \$100 from Alice's account, and add \$100 to Bob's account. Another person cannot peek at the database while you're in the middle of this process and see a state where neither Alice nor Bob has the \$100. Anyone who looks in the database will see the \$100 *somewhere*, either in Alice's account before the transaction or in Bob's account afterward.

In particular, two transactions operate in isolation and cannot interfere with each other. Suppose one transaction transfers \$100 from Alice to Bob, and then a second transaction transfers \$100 from Bob to Cindy. Logically one of these transactions occurs first and finishes before the other starts. For example, when the second transaction starts, it will not see the \$100 missing from Alice's account unless it is already in Bob's account.

OUT OF ORDER

Note that the order in which the transactions occur may make a big difference. Suppose Alice starts with \$150, Bob starts with \$50, and Cindy starts with \$50.

Now suppose the second Bob-to-Cindy transaction occurs first. If the transaction starts by removing \$100 from Bob's account, Bob is overdrawn, this transaction is rolled back, we charge Bob an overdraft fee, and we try to sell Bob overdraft protection for the very low price of only \$10 per month. After this, the Alice-to-Bob transaction occurs and we successfully move \$100 into Bob's account.

In contrast, suppose the Alice-to-Bob transaction occurs first. That transaction succeeds with no problem so, when the Bob-to-Cindy transaction starts, Bob has \$150 and the second transaction can complete successfully.

The database won't determine which transaction occurs first, just that each commits or rolls back before the other starts.

Durability means that once a transaction is committed, it will not disappear later. If the power fails, the effects of this transaction will still be there when the database restarts.

The durability requirement relies on the consistency rule. Consistency ensures that the transaction will not complete if it would leave the database in a state that violates the database's rules. Durability means that the database will not later decide that the transaction caused such a state and retroactively remove the transaction.

Once the transaction is committed, it is final.

SHADOW OF (NO) DOUBT

A high-end database might provide durability through continuous shadowing. Every time a database operation occurs, it is shadowed to another system. If the main system crashes, the shadow database can spring instantly into service. Other databases provide durability through logs. Every time the database performs an operation, it writes a record of the operation into the log. Now suppose the system crashes. When the database restarts, it reloads its last saved data and then reapplies all of the operations described by the log. This takes longer than restarting from a shadow database but requires fewer resources so it's generally less expensive.

To provide durability, the database cannot consider the transaction as committed until its changes are shadowed or recorded in the log so that the database will not lose the changes if it crashes.

BASE

Recently, some kinds of nonrelational, NoSQL databases have become increasingly popular, and they don't always satisfy the ACID rules. Their structure, plus the fact that they are often distributed across multiple servers, makes them behave differently. (We'll talk more about NoSQL databases a bit later. For now, just think of them as alternatives to relational databases.)

To better suit the asynchronous, distributed, sometimes in the cloud NoSQL lifestyle, these databases have their own feature acronym: BASE. (I'm sure this acronym was inspired partly by the fact that acids and bases are chemical opposites.)

BASE stands for Basically Available, Soft state, and Eventually consistent.

Basically Available means that the data is available. (It wouldn't be much of a database if you couldn't retrieve the data!) Relational databases spend a lot of effort requiring data to be immediately consistent (which is why you'll spend a fair amount of time studying normalization and other relational concepts later in this book). NoSQL databases spread their data across the database's clusters, so the data might not be immediately consistent. These databases do, however, guarantee that any query will return some sort of result, even if it's a failure (the electronic equivalent of a shrug).

Soft state means that the state of the data may change over time. Because these databases do not require immediate consistency, it may take a while for changes to filter through the entire system. Instead of enforcing consistency, these databases rely on the developer to provide whatever consistency is needed by the application.

Eventually consistent means that these databases do eventually become consistent, just not necessarily before the next time that you read the data (usually we're talking about seconds, not days).

We'll talk more about NoSQL databases in Chapter 3, "NoSQL Overview."

NewSQL

NewSQL is a relatively new breed of *relational database management system (RDBMS)* that attempts to combine the ACID guarantees of traditional databases with the flexibility and scalability of NoSQL databases. These are particularly useful for high-volume *online transaction processing (OLTP)* systems where distributing the data can greatly improve performance.

One way to create a NewSQL database is to split the data into disjoint subsets called *partitions* or *shards*.

Persistence and Backups

The data must be persistent—it shouldn't change or disappear by itself. If you can't trust the database to keep the data safe, then the database is pretty much worthless.

Database products do their best to keep the data safe, and in normal operation you don't need to do much to get the benefit of data persistence. When something unusual happens, however, you may need to take special action and that requires prior planning. For example, suppose the disk drives holding the database simply break, a fire reduces the computer to a smoldering puddle of slag, or a user accidentally or intentionally deletes the database. (A user tried that once on a project I was working on. We were not amused!)

In these extreme cases, the database alone cannot help you. To protect against this sort of trouble, you need to perform regular backups.

Physical data containers like notebooks are generally hard to back up, so it's hard to protect them against damage. If a fire burns up your accounts receivable notebook, you'll have to rely on your customers' honesty in paying what they owe you. Even though we like customers, I'm not sure most businesses trust them to that extent.

In theory, you could make copies of a notebook and store them in separate locations to protect against these sorts of accidents, but in practice few legit businesses do (except perhaps money laundering, smuggling, and other shady endeavors where it's handy to show law enforcement officials one set of books and the "shareholders" another).

Computerized databases, however, are relatively easy to back up. If the loss of a little data won't hurt you too badly, you can back up the database daily. If fire, a computer virus, or some other accident destroys the main database, you can reload the backup and be ready to resume operation in an hour or two.

If the database is very volatile or if losing even a little data could cause big problems (how much money do you think gets traded through the New York Stock Exchange during a busy hour?), then you need a different backup strategy. Many higher-end database products allow you to shadow every database operation as it occurs, so you always have a complete copy of everything that happens. If the main database is destroyed, you can be back in business within minutes. Some database architectures can switch to a backup database so quickly that the users don't even know it's happened.

BACKUP PLANS

It's always best to store backups away from the computer that you're backing up. Then if a big accident like a fire, tornado, or attack by space aliens occurs and destroys the whole building holding the database, the backup is still safe.

I've known of several development groups that stored their backups right next to the computer they were backing up. That guards against some kinds of stupidity but doesn't protect against big accidents. (In the teams I've worked on, about once every 10 person-years or so, someone accidentally deleted a file that we needed to recover from backups.)

I've also known of companies that had an official backup plan, but once you submitted a backup for proper storage it was shipped off-site and it took a long time to get it back if you needed it. A backup doesn't do much good if you can't access it!

In a very extreme example, I had a customer who was concerned that backups were stored only 30 miles from the database. Their thought was that the backups might not be safe in the event of a volcanic eruption or nuclear explosion. Everyone needs to pick their own level of paranoia.

Exactly how you implement database backups depends on several factors, such as how likely you think a problem will be, how quickly you need to recover from it, and how disastrous it would be to lose some data and spend time restoring from a backup. In any case, a computerized database gives you a lot more options than a notebook does.

Good database design can help make backups a bit easier. If you arrange the data so that changes occur in a fairly localized area, you can back up that area frequently and not waste time backing up data that changes only rarely.

Low Cost and Extensibility

Ideally, the database should be simple to obtain and install, inexpensive, and easily extensible. If you discover that you need to process a lot more data per day than you had expected, you should be able to somehow increase the database's capacity.

Although some database products are quite expensive, most have reasonable upgrade paths, which means you can buy the least expensive license that can handle your needs, at least in the beginning. For example, SQL Server, Oracle, and MySQL provide free editions that you can use to get started building small, single-user applications. They also provide more expensive editions that are suitable for very large applications with hundreds of users.

Similarly, many cloud databases have a free tier for smaller projects and more expensive options for larger groups. Their pricing can be fairly complicated and can involve such factors as the number of cores (the part of the computer that executes code) you get to use, the amount of memory available, the type of computer holding the data, the type of backups you want, and the number of *database transaction units* (DTUs) that you use per month. (DTUs are a weighted measure of CPU, memory, reads, and writes.)

Installing a database will never be as easy and inexpensive as buying a new notebook, but it also doesn't need to be a time-consuming financial nightmare.

Although expense and capacity are more features of the particular database product than database design, good design can help with a different kind of extensibility. Suppose you have been using a notebook database for a while and discover that you need to capture a new kind of information. Perhaps you decide that you need to track customers' dining habits so you know what restaurant gift certificate to give them on special occasions. In this case, it would be nice if you could add extra paper to the bottom of the pages in the notebook. Good database design can make that kind of extension possible.

Ease of Use

Notebooks and filing cabinets have simple user interfaces so almost anyone can use them effectively—although sometimes even they get messed up pretty badly. Should you file “United States Postal Service” under “United States?” “Postal Service?” “Snail Mail?”

A computer application's user interface determines how usable it is by average users. User interface design is not a part of database design, so you might wonder why ease of use is mentioned here.

The first-level users of a database are often programmers and relatively sophisticated database users who understand how to navigate through a database. A good database design makes the database much more accessible to those users. Just by looking at the names of the tables, fields, and other database entities that organize the data, this type of user should be able to figure out how different pieces of data go together and how to use them to retrieve the data they need. If those sophisticated users can easily understand the database, then they can build better user interfaces for the less advanced users to come.

Portability

A computerized database allows for a portability that is even more powerful than the portability of a notebook. It allows you to access the data from anywhere that you have access to the web *without actually moving the physical database*. You can access the database from just about anywhere, while the data itself remains safely at home, far from the dangers of pickpockets, being dropped in a puddle, and getting forgotten on the bus.

DEFINITION *Cloud providers may move your data around from server to server to balance their computing and network loads. That can become a problem if your data is moved from one country to another, because some countries have laws about what kinds of data you're allowed to store and where you can move it. This brings up two new terms: data residency and data sovereignty.*

Data residency is where the data is stored. Data sovereignty is the idea that data is subject to the laws of the country in which it originated. For example, Canada passed laws requiring that Canadian data be stored on Canadian servers to protect the privacy of Canadian citizens from snooping if the data was stored in another country. (Think of the U.S. National Security Agency and the USA Patriot Act.)

In a similar manner, the General Data Protection Regulation (GDPR) regulates the transfer of data outside of the European Union (EU) and the European Economic Area (EEA).

These issues mean that, in some cases, companies may require that their cloud providers keep their data physically stored in certain countries. (I'm sure this causes headaches for both the companies and the cloud providers—and keeps many lawyers employed.)

Unfortunately, the new kind of portability may be a little too easy. Although someone in the seat behind you on the airplane can't peek over your shoulder to read computerized data the way they can a notebook (well, maybe they can if you're using your laptop in airplane mode), a hacker located on the other side of the planet may try to sneak into your database and rifle through your secret cookie recipes while you're asleep.

This leads to the next topic: security.

Security

A notebook is relatively easy to lose or steal, but a highly portable database can be even easier to compromise. If you can access your database from all over the world, then so can cyber-banditos and other ne'er-do-wells.

Locking down your database is mostly a security issue that you should address by using the security tools provided by your network and database. However, there are some design techniques that you can use to make securing the database easier.

DATA DISASTERS

Many spectacular stories exist about lost or stolen laptops, hard drives, USB drives, and other media, potentially exposing confidential information to cyber villains. Hacking, improper insider access, credit-card skimming, and other attacks have resulted in even more stolen data.

The webpage <https://privacyrights.org/data-breaches> has a downloadable data file listing more than 9,000 incidents totaling more than 10 billion exposed records in the United States alone since the site began tracking incidents in 2005. The current record holder (so to speak) is a 2013 Yahoo data breach that exposed records on as many as 3 billion user accounts!

If you separate the data into categories that different types of users need to manipulate, then you can grant different levels of permission to the different kinds of users. Giving users access to only the data they absolutely need not only reduces the chance of a legitimate user doing something stupid or improper, but also decreases the chance that an attacker can pose as that user and do something malicious. Even if Clueless Carl won't mistreat your data intentionally, an online attacker might be able to guess Carl's password (which naturally is "Carl") and try to wreak havoc. If Carl doesn't have permission to trash the accounting data, then neither does the cyber-mugger.

Yet another novel aspect to database security is the fact that users can access the database remotely without holding a copy of the database locally. You can use your laptop or phablet to access a database without storing the data on your computer. That means if you do somehow lose your computer, the data may still be safe on the database server.

This is more an application architecture issue than a database design issue (don't store the data locally on laptops), but using a database design that restricts users' access to what they really need to know can help.

Sharing

It's not easy to share a notebook or a box full of business cards among many people. Two people can't really use a notebook at the same time, and there's some overhead in shipping the notebook back and forth among users. Taking time to walk across the room a dozen times a day would be annoying; express mailing a notebook across the country every day would be just plain silly.

Modern networks can let hundreds or even thousands of users access the same database at the same time from locations scattered across the globe. Although this is largely an exercise in networking and the tools provided by a particular database product, some design issues come into play. (And don't forget what I said earlier about data residency and data sovereignty.)

If you compartmentalize the data into categories that different types of users need to use as described in the previous section, this not only aids with security, but also helps reduce the amount of data that must be shipped across the network.

Breaking the data into reasonable pieces can also aid in coordination among multiple users. When your coworker in London starts editing a customer's record, that record must be locked so that other users can't sneak in and mess up the record before the edit is finished. Grouping the data appropriately lets you lock the smallest amount of data possible, which makes more data available for other users to edit.

Careful design can allow the database to perform some calculations on the server and send only the results to your boss (who's working hard on the beaches of Hawaii) instead of shipping the whole database out there and making the boss's computer do all of the work.

Good application design is also important. Even after you prepare the database for efficient use, the application still needs to use it properly. But without a good database design, these techniques aren't possible.

Ability to Perform Complex Calculations

Compared to the human brain, computers are idiots. It takes seriously powerful hardware and frighteningly sophisticated algorithms to perform tasks that you take for granted, such as recognizing faces, speaker-independent speech recognition, and handwriting recognition (although neither the human brain nor computers have yet deciphered doctors' prescriptions). The human brain is also self-programming, so it can learn new tasks flexibly and relatively quickly.

Although a computer lacks the adaptability of the human brain, it is great at performing a series of well-defined tasks quickly, repeatedly, and reliably. A computer doesn't get bored, let its attention

wander, or make simple arithmetic mistakes (unless it suffers from the Pentium FDIV bug, the f00f bug, the Cyrix coma bug, or a few others). The point is, if the underlying hardware and software work correctly, the computer can perform the same tasks again and again millions of times per second without making mistakes.

When it comes to balancing checkbooks, searching for accounts with balances less than \$0, and performing a host of other number-crunching tasks, the computer is much faster and less error-prone than a human brain.

The computer is naturally faster at these sorts of calculations, but even its blazing speed won't help you if your database is poorly designed. A good design can make the difference between finding the data you need in seconds rather than hours, days, or not at all.

CAP Theorem

The *CAP theorem* (also called *Brewer's theorem* after computer scientist Eric Brewer, who devised it) says that any distributed data store can only provide two of the following reliably:

- **Consistency**—Every read receives the most recently written data (or an error).
- **Availability**—Every read receives a non-error response, if you don't guarantee that it receives the most recently written data.
- **Partition tolerance**—The data store continues to work even if messages are dropped or delayed by the network between the store's partitions.

When a partition failure occurs, you need to decide whether to either (a) cancel the operation (and reduce availability), or (b) have the operation continue (but risk inconsistency).

Basically all of this says that network errors or tardiness can make the data temporarily unavailable, or you can use the most recent value to which you have access.

Brewer noted that you only need to sacrifice consistency or availability if the data store is partitioned. If the data is all stored in one place, then you should be able to achieve both consistency and availability.

Cloud Considerations

A *cloud database* hosts data in the cloud so it is accessible over a network. There are two common cloud deployment models. First, you can rent space on a virtual machine and run the database there. A *virtual machine* (VM) is a simulation of a physical computer running on a server somewhere.

Virtual machines have the advantage that the cloud provider can move them around, possibly hosting multiple VMs on a single physical machine. Conversely, you may also be able to use multiple computers to host a single VM. Those two capabilities together make it easier to scale an application up or down as needed.

For example, suppose you write some software to schedule appointments for nail salons. Initially you serve only a few salons, so your VM uses a small fraction of one physical server. Over time, as more and more salons sign up for your service, you need more space and faster processing, so you start using more of the server. Soon your provider moves you onto faster hardware and eventually onto a small group of servers (for a price, of course).

NOT SO VIRTUAL MACHINES

Instead of renting a virtual machine, you can rent a physical machine or even buy your own and put it on your network. Then you have full use of the machine.

That approach works and has some advantages, such as giving you *complete* control (and commensurate responsibility) and letting you know *exactly* where your data is, but it isn't really cloud computing and it doesn't give you the same easy scaling advantages.

A second way to build a cloud database is to rent a database from a cloud database provider. This has the same advantages as the virtual machine, plus the database provider will maintain the database for you. For example, the provider may perform backups, guarantee a certain level of availability, and so forth (again, for a price).

In this scenario, the database is provided *as a service (aaS)*. More precisely, it is considered a *database as a service (DBaaS)*, which is a kind of *software as a service (SaaS)*. (The phrase “as a service” is very in vogue lately.)

Note that you can create both relational and NoSQL databases in the cloud. You can also find some that are free as long as the database is relatively small. The provider hopes that your nail salon will become wildly successful and that you'll then share your success with them by renting larger virtual machines or databases.

Legal and Security Considerations

I won't talk too much about legal issues in this book, but you should determine whether you might encounter any of them. For example, I already mentioned data residency and data sovereignty earlier in this chapter. Some countries require that certain kinds of data reside physically within their borders, and you could be in big trouble if your data is stored in the cloud on foreign servers.

In addition to ensuring that your cloud servers have allowed physical locations, you need to ensure that your data is properly protected. For example, in the United States, *HIPAA* (which stands for the *Health Insurance Portability and Accountability Act* and is pronounced “hip-uh”) prohibits the disclosure of a patient's sensitive medical information without their consent or knowledge. I don't believe HIPAA requires data residency (but I'm not a lawyer, so don't take my word for it), but some states have their own special requirements. For example, all 50 U.S. states plus Washington D.C., Puerto Rico, and the U.S. Virgin Islands have some sort of law requiring you to notify residents if their personal information is compromised in a security breach.

Obviously sensitive information like credit card numbers, bank account numbers, Social Security numbers, driver's license numbers, website passwords, biometric data, business information, and other important items require top-notch security.

Certain other kinds of data are also considered personal and/or sensitive and may or may not be protected by law. *Personally identifiable information (PII)* is information that could be used to assist with identity theft and includes such items as a person's name, mother's maiden name, address and

former addresses, phone numbers, and so on. Sensitive data may include gender identity, ethnic background, political or religious affiliation, union membership, and more.

When you're shopping for cloud databases, consider the kinds of data that you will store, whether it's subject to data residency requirements, and whether it needs more than the normal amount of protection. Then mercilessly grill your cloud provider to ensure that they can handle your needs.

If you don't think these aren't important issues, think again! Governments take these things *very* seriously. For example, here are a few of the larger fines levied against companies that didn't protect their data properly.

- **Athens Orthopedic Clinic**—\$1.5 million HIPAA penalty
- **CHSPSC LLC**—\$2.3 million HIPAA penalty and \$5 million settlement for a 28-state legal action
- **Google LLC**—€60 million (\$68 million) GDPR fine
- **Facebook**—€60 million (\$68 million) GDPR fine
- **Premera Blue Cross**—\$6.85 million HIPAA penalty, plus a multistate legal settlement for \$10 million and a class action lawsuit for \$74 million, for a total of over \$90 million
- **Google Ireland**—€90 million (\$102 million) GDPR fine
- **Uber**—\$148 million for 600,000 driver and 57 million user accounts breached
- **Capital One**—\$190 million to settle a class-action lawsuit over a breach that exposed the personal data of 100 million people
- **Home Depot**—\$200 million paid to credit companies, banks, other financial institutions, and customers because attackers breached their system and compromised their point-of-sale system
- **WhatsApp**—€225 million (\$255 million) GDPR fine
- **Equifax**—\$575 million–\$700 million for “failure to take reasonable steps to secure its network”
- **Amazon**—€746 million (\$877 million) GDPR fine

Those dozen cases alone add up to more than \$2.5 billion. You can find so many more examples by searching for “biggest data breach fines” that it's not funny. In fact, it's terrifying. Many of these fines were not levied because a company was being evil; they were mostly just being lazy (not following the rules) and careless (not properly securing their data).

So find out what your exposure is and take data security very seriously.

Consequences of Good and Bad Design

Table 1.1 summarizes how good and bad design can affect the features described in the previous sections.

TABLE 1.1: Good vs. bad design

FEATURE	GOOD DESIGN	BAD DESIGN
CRUD	You can find the data you need quickly and easily. The database prevents inconsistent changes.	You find the data you need either very slowly or not at all. You can enter inconsistent data or modify and delete data to make the result inconsistent. (Your products ship to the wrong address or the wrong person.)
Retrieval	You can find the correct data quickly and easily.	You cannot find the data that you need quickly. (Your customer waits on hold for 45 minutes to get information, gets tired of waiting, and then buys from your competition.)
Consistency	All parts of the database agree on common facts.	Different pieces of information hold contradictory data. (A customer's bills are sent to one address, but late payment notices are sent to another.)
Validity	Fields contain valid data.	Fields contain gibberish. (Your company's address has the <code>state</code> value "Chaos." Although if the database does hold that value, it's probably correct on some level.)
Error Correction	It's easy to update incorrect data.	Simple and large-scale changes never happen. (Thousands of your customers' bills are returned to you because their city changed and the database didn't get updated.)
Speed	You can quickly find customers by name, account number, or phone number.	You can only find a customer's record if they know their 37-digit account number. Searching by name takes half an hour.
Atomic Transactions	Either all related transactions happen or none of them happen.	Related transactions may occur partially. (Alice loses \$100, but Bob doesn't receive it. Prepare for customer complaints!)
Persistence and Backups	You can recover from computer failure. The data is safe.	Recovering lost data is slow and painful or even impossible. (You lose all of the orders placed in the last week!)

continues

TABLE 1.1 (continued)

FEATURE	GOOD DESIGN	BAD DESIGN
Low Cost and Extensibility	You can move to a bigger database when your needs grow.	You're stuck on a small-scale database. (When your website starts receiving hundreds of orders per second, the database cannot keep up, and you lose thousands of orders per day.)
Ease of Use	The database design is clear so developers understand it and can build a great user interface.	The database design is confusing so the developers produce an "anthill" program—confusing and buggy. (I've worked on projects like that and it's no picnic!)
Portability	The design allows different users to download relevant data quickly and easily.	Users must download much more data than they need, slowing performance and giving them access to sensitive data (such as the Corporate Mission Statement, which proves that management has no clue).
Security	Users have access to the data that they need and nothing else.	Hackers and disgruntled former employees have access to everything.
Sharing	Users can manipulate the data they need.	Users lock data they don't really need and get in each other's way, slowing them down.
Complex Calculations	Users can easily perform complex analysis to support their jobs.	Poor design makes calculations take far longer than necessary. (I worked on a project where a simple change to a data model could force a 20-minute recalculation.)

SUMMARY

Database design plays an important role in application development. If the database design doesn't provide a solid foundation for the rest of the project to build upon, the application as a whole will fail.

Physical data containers like notebooks and file cabinets can behave as databases. They have strengths and weaknesses that you can use to think about corresponding strengths and weaknesses in computerized databases. With some effort, you can provide the strengths while avoiding the weaknesses.

In this chapter, you learned that a good database provides:

- CRUD
- Retrieval
- Consistency
- Validity
- Easy error correction
- Speed
- Atomic transactions
- ACID
- Persistence and backups
- Low cost and extensibility
- Ease of use
- Portability
- Security
- Sharing
- Ability to perform complex calculations

The next chapter provides an overview of the most commonly used databases today: relational databases. NoSQL databases, which are described in the chapter after that, are coming on strong, but relational databases still have some powerful advantages in many scenarios.

Before you move on to Chapter 2, however, take a look at the following exercises and test your knowledge of the database design goals described in this chapter. You can find the solutions to these exercises in Appendix A.

EXERCISES

1. Compare this book to a database (assuming you don't just use it as a notebook, scribbling in the margins). What features does it provide? What features are missing?

2. Describe two features that this book provides to help you look for particular pieces of data in different ways.

3. What does CRUD stand for? What do the terms mean?

4. How does a chalkboard implement the CRUD methods? How does a chalkboard's database features compare to those of this book?

5. Consider a recipe file that uses a single index card for each recipe with the cards stored alphabetically. How does that database's features compare to those of a book?

6. What does ACID stand for? What do the terms mean?

7. What does BASE stand for? What do the terms mean?

8. What does the CAP theorem say? Under what conditions is this not a "pick two of three" situation?

9. Suppose Alice, Bob, and Cindy all have account balances of \$100 and the database does not allow an account's balance to ever drop below \$0. Now consider three transactions: (1) Alice transfers \$125 to Bob, (2) Bob transfers \$150 to Cindy, and (3) Cindy transfers \$25 to Alice and \$50 to Bob. In what order(s) can those transactions be executed successfully?

10. Explain how a central database can protect your confidential data.

2

Relational Overview

Recall the question posed in Chapter 1: What is a database? The answer given there was:

A database is a tool that stores data and lets you create, read, update, and delete the data in some manner.

This broad definition allows you to consider all sorts of odd things as databases, including notebooks, filing cabinets, and your brain. If you're flexible about what you consider data, then this definition includes even stranger objects like a chess set (which stores board positions) or a parking lot (which stores car types and positions, although it might be hard for you to update any given car's position without the owner's consent).

This chapter begins our move into the realm of computerized databases.

Relational databases are the most commonly used computerized databases today, and most of this book (and other database books) focus on them. Relational databases are useful in a huge number of situations, but they're not the only game in town. Sometimes, a different kind of database may make more sense for your particular problem, so I'll say more about nonrelational databases in the next chapter and later in the book. For now, though, let's focus on relational databases.

DATABASE DEFINITIONS

Before we get too far, you should know a couple of basic abbreviations.

A *relational database (RDB)* includes the files that hold the data, indexes, and other information needed to store and retrieve information in a relational database.

In general, a *database management system (DBMS)* is an application that you use to manage a database. It lets you create files, define indexes (if the database supports them), and perform other administrative chores for the database.

A *relational database management system (RDBMS)* is, as you can probably imagine, a DBMS that manages a relational database.

There are also other kinds of DBMSs that manage nonrelational databases. For example, an *object-relational database (ORD)* allows you to model objects, classes, and inheritance, while also allowing you to treat objects in a relational way. You manage an ORD with (you guessed it) an *object-relational database management system (ORDBMS)*. (Don't worry if you don't know what objects, classes, and inheritance are. They're programming things that aren't important for our discussion right now.)

One last term before we really get started. A *database administrator (DBA)* is the person who manages a database. In larger projects, this may be a separate person with the title Database Administrator and who may even have a degree in database management. On smaller projects, DBA is often one of many roles played by the same person.

Before you can start learning how to properly design a relational database, you should understand some basic concepts and terms. This chapter provides an introduction to relational databases. It explains the major ideas and terms that you need to know before you can start designing and building relational databases.

In this chapter, you'll learn about relational database terms such as:

- Table and relation
- Record, row, and tuple
- Column, field, and attribute
- Constraint, key, and index

Finally, you'll learn about the operations that you can use to get data out of a relational database.

PICKING A DATABASE

There's an expression, "If all you have is a hammer, everything looks like a nail." If the only kind of database you understand is the relational database, then you'll probably try to hammer every kind of data into a relational database, and that can sometimes lead to trouble.

I once worked on a fairly large database application with 40 developers and more than 120,000 lines of code. The program loaded some fairly large relational databases and used their data to build huge, tree-like structures. Those structures allowed sales representatives to design and modify extremely complicated projects for customers involving tens of thousands of line items.

The data was naturally hierarchical but was stored in relational databases, so the program was forced to spend a long time loading each data set. Many projects took 5 to 20 minutes to load. When the user made even a simple change, the program's design required it to recalculate parts of the tree and then save the changes back into the database—a process that took another 5 to 30 minutes depending on the complexity of the model. The program was so slow that the users couldn't perform the types of experiments they needed to optimize the projects they were building. You couldn't quickly see the effects of tweaking a couple of numbers here and there.

To make matters worse, loading and saving all of that hierarchical data in a relational database required tens of thousands of lines of moderately tricky code that was hard to debug and maintain.

At one point, I performed a quick experiment to see what would happen if the data were stored in an XML database, a database that naturally stores hierarchical data. My test program was able to load and save data sets containing 20,000 items in 3 to 4 seconds.

At that point, the project was too big and the design too entrenched to make such a fundamental change. (Afterward, political pressure within the company pulled the project in too many directions and it eventually shredded like a tissue in a tug-of-war.)

The lesson is clear: before you spend a lot of time building the ultimate relational database and piling thousands of lines of code on top of it, make sure that's the kind of database you need. Had this project started with an XML database, it probably would have had a simpler, more natural design with much less code and would probably have lasted for many years to come.

RELATIONAL POINTS OF VIEW

Relational databases play a critical role in many important (i.e., money-related) computer applications. As is the case whenever enormous amounts of money are at stake, people have spent a huge amount of time and effort building, studying, and refining relational databases. Database researchers usually approach relational databases from one of three points of view.

The first group approaches the problem from a database-theoretical point of view. These people tend to think in terms of provability, mathematical set theory, and propositional logic. You'll see them at parties throwing around phrases like *relational algebra*, *Cartesian product*, and *tuple relational calculus*. This approach is intellectually stimulating (and looks good on a résumé) but can be a bit intimidating. These researchers focus on logical design and idealized database principles.

The second group approaches the matter from a less formal “just build the database and get it done” point of view. Their terminology tends to be less precise and rigorous but more intuitive. They tend to use terms that you may have heard before like table, row, and column. These people focus on physical database design and pay more attention to concrete bits-and-bytes issues dealing with actually building a database and getting the most out of it.

The third group tends to think in terms of flat files and the underlying disk structure used to hold data. Although these people are probably in the minority these days, their terms file, record, and field snuck into database nomenclature and stuck. Many who still use these terms are programmers and other developers who look at the database from a consumer’s “how do I get my data out of it” point of view.

These differing viewpoints have led to several different and potentially puzzling ways to consider relational databases. This can cause some confusion, particularly because the different groups have latched on to some of the same terms but used them for different meanings. In fact, they sometimes use the term *relation* in very different ways (that are described later in this chapter).

This chapter loosely groups these terms into “formal” and “informal” categories, where the formal category includes the database theoretical terms and the informal category includes everything else.

This chapter begins with informal terms. Each section initially focuses on informal terms and concepts, and then explains how they fit together with their more formal equivalents.

TABLE, ROWS, AND COLUMNS

Informally you can think of a relational database as a collection of *tables*, with each containing *rows* and *columns*. At this level, it looks a lot like a computerized workbook containing several worksheets (or spreadsheets), although a worksheet is much less constrained than a database table is. You can put just about anything in any cell in a worksheet. In contrast, every entry in a particular column of a table is expected to contain the same kind of data. For example, all of the cells in a particular column might contain phone numbers or last names.

NOTE *Actually, a poorly designed database application may allow the user to sneak some strange kinds of data into fields, too. For example, if the database and user interface aren’t designed properly, you might be able to enter a string such as “none” in a telephone number field. That’s not the field’s intent, however. In contrast, a spreadsheet’s cells don’t really care what you put in them.*

The set of the values that are allowed for a column is called the column’s *domain*. For example, a column’s domain might be telephone numbers, bank account numbers, snowshoe sizes, or hang glider colors.

Domain is closely related to data type, but it’s not quite the same. A column’s *data type* is the kind of data that the column can hold. (Yes, I know this is pretty obvious.) The data types that you can use for a column depend on the particular database you are using, but typical data types include integer, floating-point number (a number with a decimal point), string, and date.

To see the difference between domain and data type, note that street address (323 Relational Rd) and jersey color (red) are both strings. However, the domain for the street address column is valid street addresses, whereas the domain for the jersey color column is colors (and possibly not even all colors

if you only allow a few choices). You can think of the data type as the highest level or most general possible domain. For example, an address or color domain is a more restrictive subset of the domain allowing all strings.

The rows in a table correspond to column values that are related to each other according to the table's purpose. For example, suppose you have a Competitors table that contains contact information for participants in your First (and probably Last) Annual Extreme Pyramid Sports Championship (aka the Cairolympics). This table includes columns to hold competitor name, address, event, blood type, and next of kin, as shown in Figure 2.1. (Note that this is not a good database design. You'll see why in later chapters.)

Name	Address	Event	Blood Type	NextOfKin
Alice Adventure	6543 Flak Ter, Runner AZ 82018	Pyramid Boarding	A+	Art Adventure
Alice Adventure	6543 Flak Ter, Runner AZ 82018	Pyramid Luge	A+	Art Adventure
Bart Bold	6371 Jump St #27, Dove City, NV 73289	Camel Drafting	O-	Betty Bold
Bart Bold	6371 Jump St #27, Dove City, NV 73289	Pyramid Boarding	O-	Betty Bold
Bart Bold	6371 Jump St #27, Dove City, NV 73289	Sphinx Jumping	O-	Betty Bold
Cindy Copes	271 Sledding Hill, Ricky Ride CO 80281	Camel Drafting	AB-	John Finkle
Cindy Copes	271 Sledding Hill, Ricky Ride CO 80281	Sphinx Jumping	AB-	John Finkle
Dean Daring	73 Fighter Ave, New Plunge UT 78281	Pyramid Boarding	O+	Betty Dare
Dean Daring	73 Fighter Ave, New Plunge UT 78281	Pyramid Luge	O+	Betty Dare
Frank Fiercely	3872 Bother Blvd, Lost City HI 99182	Pyramid Luge	B+	Fred Farce
Frank Fiercely	3872 Bother Blvd, Lost City HI 99182	Sphinx Jumping	B+	Fred Farce
George Foreman	73 Fighter Ave, New Plunge UT 78281	Sphinx Jumping	O+	George Forman
George Foreman	73 Fighter Ave, New Plunge UT 78281	Pyramid Luge	O+	George Forman
Gina Gruff	1 Skatepark Ln, Forever KS 72071	Camel Drafting	A+	Gill Gruff
Gina Gruff	1 Skatepark Ln, Forever KS 72071	Pyramid Boarding	A+	Gill Gruff

FIGURE 2.1

A particular row in the table holds all of the values for a given competitor. For example, the values in the first row (Alice Adventure, 6543 Flak Ter, Runner AZ 82018, Pyramid Boarding, A+, Art Adventure) all apply to the competitor Alice Adventure.

Back in olden times when database developers worked with primitive tools by candlelight, everyone lived much closer to nature. In this case, it means they needed to work more closely with the underlying filesystem. It was common to store data in “flat” files without any indexes, search tools, or other fancy modern luxuries. A file would hold the related information that you might represent at a higher level as a table. The file was divided into chunks called *records*, each of which had the same size and corresponded to a row in a table. The records were divided into fixed-length *fields* that corresponded to the columns in a table.

For example, if you think of the table shown in Figure 2.1 as a flat file, the first row corresponds to a record in the file. Each record contains Name, Address, Event, and other fields to hold the data.

Though relatively few DBAs still work with flat files at this level, the terms *file*, *record*, and *field* are still with us and are often used in database documentation and discussions.

RELATIONS, ATTRIBUTES, AND TUPLES

The values in a row are *related* by the fact that they apply to a particular person. Because of this fact, the formal term for a table is a *relation*. This may cause some confusion because the word *relation* is also used informally to describe a relationship between two tables. This use is described in the section “Foreign Key Constraints,” later in this chapter.

The formal term for a column is an *attribute* or *data element*. For example, in the Competitors relation shown in Figure 2.1, Name, Address, BloodType, and NextOfKin are the attributes of each of the people represented. You can think of this as in “each person in the relation has a Name attribute.”

The formal term for a row is a *tuple* (rhymes with “scruple”). This almost makes sense if you think of a two-attribute relation as holding data pairs, a three-attribute relation as holding value triples, and a four-attribute relation as holding data quadruples. Beyond four items, mathematicians would say 5-tuple, 6-tuple, and so forth, hence the name tuple.

Don’t confuse the formal term *relation* (meaning table) with the more general and less precise use of the term that means “related to” as in “these fields form a relation between these two tables” (or “that psycho is no relation of mine”). Similarly, don’t confuse the formal term *attribute* with the less precise use that means “feature of” as in “this field has the ‘required’ attribute” (or “don’t attribute that comment to me!”). I doubt you’ll confuse the term *tuple* with anything—it’s probably confusing enough all by itself.

Theoretically a relation does not impose any ordering on the tuples that it contains, nor does it give an ordering to its attributes. Generally, the orderings don’t matter to mathematical database theory. In practice, however, database applications usually sort the records selected from a table in some manner to make it easier for the user to understand the results. It’s also a lot easier to write the program (and for the user to understand) if the order of the fields remains constant, so database products typically return fields in the order in which they were created in the table unless told otherwise.

KEYS

Relational database terminology includes an abundance of different flavors of keys. (They are key terms, so you could say that relational databases have a lot of key key terms.) In the loosest sense, a *key* is a combination of one or more columns that you use to find rows in a table. For example, a Customers table might use CustomerID to find customers. If you know a customer’s ID, then you can quickly find that customer’s record in the table. (In fact, many ID numbers, such as employee IDs, student IDs, driver’s license numbers, and so forth, were invented just to make searching in database tables easier. My library card certainly doesn’t include a 10-digit ID number for *my* convenience.)

The more formal relational vocabulary includes several other more precise definitions of keys.

In general, a *key* is a set of one or more columns in the table that have certain properties. A *compound key* or *composite key* is a key that includes more than one column. For example, you might use the combination of `FirstName` and `LastName` to look up customers.

A *superkey* is a set of one or more columns in a table for which no two rows can have the exact same values. For example, in the `Competitors` table shown in Figure 2.1, the `Name`, `Address`, and `Event` columns together form a superkey because no two rows have exactly the same `Name`, `Address`, and `Event` values. Because superkeys define fields that must be unique within a table, they are sometimes called *unique keys*.

Because no two rows in the table have the same values for a superkey, a superkey can uniquely identify a particular row in the table. In other words, a program could use a superkey to find any particular record.

A *candidate key* is a minimal superkey. That means if you remove any of the columns from the superkey, it won't be a superkey anymore.

For example, you already know that `Name/Address/Event` is a superkey for the `Competitors` table. If you remove `Event` from the superkey, you're left with `Name/Address`. This is not a superkey because everyone in the table is participating in multiple events, and therefore they have more than one record in the table with the same name and address.

If you remove `Name`, then `Address/Event` is not a superkey because Dean Daring and his roommate George Foreman share the same address and are both signed up for `Pyramid Luge`. (They also have the same blood type. They became friends and decided to become roommates when Dean donated blood for George after a particularly flamboyant skateboarding accident.)

Finally if you remove `Address`, then `Name/Event` is still a superkey. That means `Name/Address/Event` is not a candidate key because it is not minimal. However, `Name/Event` is a candidate key because no two rows have the same `Name/Event` values and you can easily see neither `Name` nor `Event` is a superkey, so the pair is minimal.

You could still have a problem if one of George's other brothers, who are all named George, moves in. If they compete in the same event, you won't be able to tell them apart. Perhaps we should add a `CompetitorId` column to the table after all.

Note that there may be more than one superkey or candidate key in a table. In Figure 2.1, `Event/NextOfKin` also forms a candidate key because no two rows have the same `Event` and `NextOfKin` values. (That would probably not be the most natural way to look up rows, however. "Yes sir, I can look up your record if you give me your event and next of kin.")

A *unique key* is a superkey that is used to uniquely identify the rows in a table. The difference between a unique key and any other candidate key is in how it is used. A candidate key *could* be used to identify rows if you wanted it to, but a unique key *is* used to constrain the data. In this example, if you make `Name/Event` be a unique key, the database will not allow you to add two rows with the same `Name` and `Event` values. A unique key is an implementation issue, not a more theoretical concept like a candidate key is.

A *primary key* is a superkey that is actually used to uniquely identify or find the rows in a table. A table can have only one primary key (hence the name “primary”). Again, this is more of an implementation issue than a theoretical concern. Database products generally take special action to make finding records based on their primary keys faster than finding records based on other keys.

Some databases allow alternate key fields to have missing values, whereas all of the fields in a primary key are required. For example, the Competitors table might have Name/Address/Event as a unique key and Name/Event as a primary key. Then it could contain a record with Name and Event but no Address value (although that would be a bit strange—we might want to require that all the fields have a value).

An *alternate key* is a candidate key that is not the primary key. Some also call this a *secondary key*, although others use the term *secondary key* to mean any set of fields used to locate records even if the fields don’t define unique values.

That’s a lot of keys to try to remember! The following list briefly summarizes the different flavors:

- **Compound key or composite key**—A key that includes more than one field.
- **Superkey**—A set of columns for which no two rows can have the exact same values.
- **Candidate key**—A minimal superkey.
- **Unique key**—A superkey used to require uniqueness by the database.
- **Primary key**—A unique key that is used to quickly locate records by the database.
- **Alternate key**—A candidate key that is not the primary key.
- **Secondary key**—A key used to look up records but that may not guarantee uniqueness.

One last kind of key is the *foreign key*. A foreign key is used as a constraint rather than to find records in a table, so it is described a bit later in the section “Constraints.”

INDEXES

An *index* is a database structure that makes it quicker and easier to find records based on the values in one or more fields. Indexes are not the same as keys, although the two are related closely enough that many developers confuse the two and use the terms interchangeably.

For example, suppose you have a Customers table that holds customer information: name, address, phone number, Swiss bank account number, and so forth. The table also contains a CustomerId field that it uses as its primary key.

Unfortunately, customers usually don’t remember their customer IDs (I know I don’t), so you need to be able to look them up by name or phone number. If you make Name and PhoneNumber two different keys, then you can quickly locate a customer’s record in three ways: by customer ID, by name, and by phone number.

NOTE Relational databases also make it easy to look up records based on non-indexed fields, although it may take a while. If the customer only remembers their address and not their customer ID or name, you can search for the address even if that field isn't part of an index. It may just take a long time. Of course, if the customer cannot remember their name, then they have bigger problems.

Building and maintaining an index takes the database some extra time, so you shouldn't make indexes gratuitously. Place indexes on the fields that you are most likely to need to search and don't bother indexing fields like apartment number or telephone extension, which you're unlikely to need to search.

CONSTRAINTS

As you might guess from the name, a *constraint* places restrictions on the data allowed in a table. In formal database theory, constraints are not considered part of the database. However, in practice, constraints play such a critical role in managing the data properly that they are informally considered part of the database. (Besides, the database product enforces them!)

The following sections describe some kinds of constraints that you can place on the fields in a table.

Domain Constraints

Relational databases let you specify some simple basic constraints on a particular field. For example, you can make a field required.

The special value *null* represents an empty value. For example, suppose you don't know a customer's income. You can place the value null in the Income field to indicate that you don't know the correct value. This is different from placing 0 in the field, which would indicate that the customer doesn't have any income.

Making a field required means it cannot hold a null value, so this is also called a *not null* constraint.

The database will also prevent a field from holding a value that does not match its data type. For example, you cannot put a 20-character string in a 10-character field. Similarly, you cannot store the value "twelve" in a field that holds integers.

These types of constraints restrict the values that you can enter into a field. They help define the field's domain, so they are called *domain constraints*. Some database products allow you to define more complex domain constraints, often by using check constraints.

Check Constraints

A *check constraint* is a more complicated type of restriction that evaluates a Boolean expression (a logical expression that evaluates to true or false) to see if certain data should be allowed. If the expression evaluates to true, then the data is allowed.

A *field-level check constraint* validates a single column. For example, in a `SalesPeople` table, you could place the constraint `Salary > 0` on the `Salary` field to mean that the field's value must be positive (even if you think that some salespeople deserve negative salaries).

A *table-level check constraint* can examine more than one of a record's fields to see if the data is valid. For example, the constraint `(Salary > 0) OR (Commission > 0)` requires that each `SalesPeople` record must have a positive salary or a positive commission (or both).

Primary Key Constraints

By definition, no two records can have identical values for the fields that define the table's primary key. That greatly constrains the data.

In more formal terms, this type of constraint is called *entity integrity*. It simply means that no two records are exact duplicates (which is true if the fields in their primary keys are not duplicates), and that all of the fields that make up the primary key have non-null values.

Unique Constraints

A *unique constraint* requires that the values in one or more fields be unique. Note that it only makes sense to place a uniqueness constraint on a superkey. Recall that a superkey is a group of one or more fields that cannot contain duplicate values. It wouldn't make sense to place a uniqueness constraint on fields that can validly contain duplicated values. For example, it would probably be silly to place a uniqueness constraint on a `Customer` table's `City` or `State` field.

Foreign Key Constraints

A *foreign key* is not quite the same kind of key defined previously. Instead of defining fields that you can use to locate records, a foreign key refers to a key in a different (foreign) table. The database uses it to locate records in the other table. Because it defines a reference from one table to another, this kind of constraint is also called a *referential integrity constraint*.

A foreign key constraint requires that a record's values in one or more fields in one table (the referencing table) must match the values in another table (the foreign or referenced table). The fields in the referenced table must form a candidate key in that table. Usually, they are that table's primary key, and most database products try to use the foreign table's primary key by default when you make a foreign key constraint.

For a simple example, suppose you want to validate the entries in the `Competitors` table's `Event` field so that the minimum wage interns who answer the phones cannot assign anyone to an event that doesn't exist.

To do this with a foreign key, create a new table named `Events` that has a single column called `Event`. Make this the new table's primary key and create records that list the possible events: `Pyramid Boarding`, `Pyramid Luge`, `Camel Drafting`, and `Sphinx Jumping`.

Next, make a foreign key that relates the `Competitors` table's `Event` field with the `Events` table's `Event` field. Now whenever someone adds a new record to the `Competitors` table, the foreign key constraint requires that the new record's `Event` value be listed in the `Events` table.

The database will also ensure that no one modifies a Competitors record to change the Event value to something that is not in the Events table.

Finally, the database will take special action if you try to delete a record in the Events table if its value is being used by a Competitors record (for example, if you decide to cancel the Pyramid Luge, but Dean Daring has already signed up for it). Depending on the type of database and how you have the relationship configured, the database will either refuse to remove the Events record or it will automatically delete all of the Competitors records that use it.

This example uses the Events table as a lookup table for the Competitors table. Another common use for foreign key constraints is to ensure that related records always go together. For example, you could build a NextOfKin table that contains information about the competitors' next of kin (including name, phone number, email address, beneficiary status, and so forth). Then you could create a foreign key constraint to ensure that every Competitors record's NextOfKin value is contained in the Name fields in some NextOfKin table record. That way you know that you can always contact the next of kin for anyone in the Competitors table.

Figure 2.2 shows the Competitors, Events, and NextOfKin tables with arrows showing the relationships among their related fields.

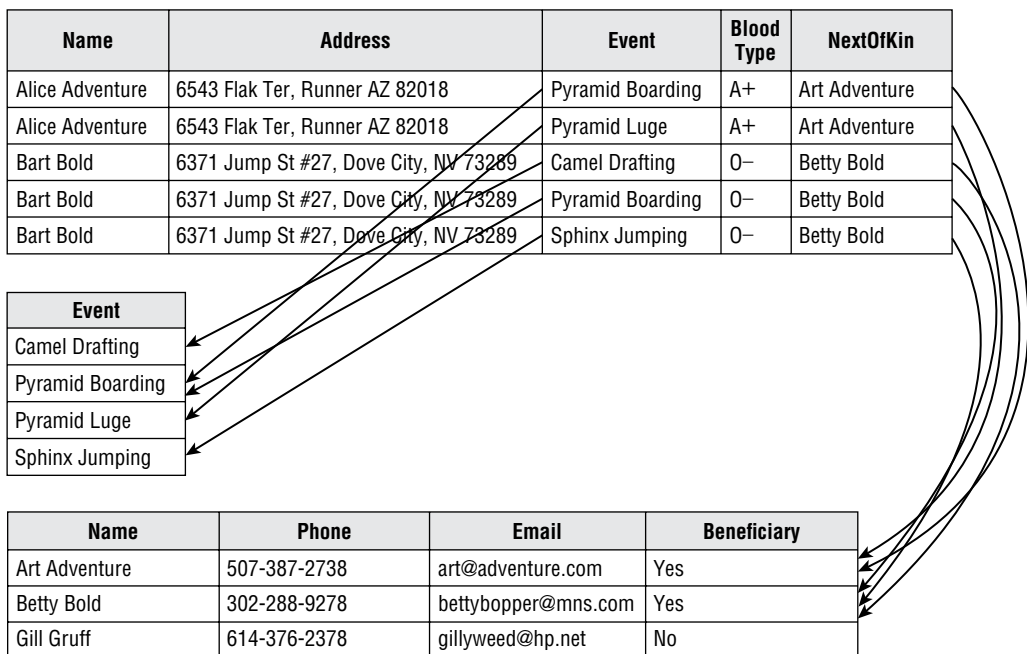


FIGURE 2.2

Foreign keys define associations between tables that are sometimes called *relations*, *relationships*, or *links* between the tables. The fact that the formal database vocabulary uses the word *relation* to mean “table” sometimes leads to confusion. Fortunately, the formal and informal database people usually get invited to different parties, so the terms usually don’t collide in the same conversation.

DATABASE OPERATIONS

Eight operations were originally defined for relational databases, and they form the core of modern database operations. The following list describes those original operations:

- **Selection**—This selects some or all of the records in a table. For example, you might want to select only the Competitors records where Event is Pyramid Luge so that you can know who to expect for that event (and how many ambulances to have standing by).
- **Projection**—This drops columns from a table or selection. For example, when you make your list of Pyramid Luge competitors, you may want to list only their names and not their addresses, blood types, events (which you know is Pyramid Luge anyway), or next of kin.
- **Union**—This combines tables with similar columns and removes duplicates. For example, suppose you have another table named FormerCompetitors that contains data for people who participated in previous years' competitions. Some of these people are competing this year and some are not. You could use the union operator to build a list of everyone in either table. (Note that the operation would remove duplicates, but for these tables you would still get the same person several times with different events.)
- **Intersection**—This finds the records that are the same in two tables. The intersection of the FormerCompetitors and Competitors tables would list those few who competed in previous years and who survived to compete again this year (i.e., the slow learners).
- **Difference**—This selects the records in one table that are *not* in a second table. For example, the difference between FormerCompetitors and Competitors would give you a list of those who competed in previous years but who are not competing this year (so you can email them and ask them what the problem is).
- **Cartesian Product**—This creates a new table containing every record in a first table combined with every record in a second table. For example, if one table contains values 1, 2, 3, and a second table contains values A, B, C, then their Cartesian product contains the values 1/A, 1/B, 1/C, 2/A, 2/B, 2/C, 3/A, 3/B, and 3/C.
- **Join**—This is similar to a Cartesian product except records in one table are paired only with those in the second table if they meet some condition. For example, you might join the Competitors records with the NextOfKin records where a Competitors record's NextOfKin value matches the NextOfKin record's Name value. In this example, that gives you a list of the competitors together with their corresponding next of kin data.
- **Divide**—This operation is the opposite of the Cartesian product. It uses one table to partition the records in another table. It finds all of the field values in one table that are associated with every value in another table. For example, if the first table contains the values 1/A, 1/B, 1/C, 2/A, 2/B, 2/C, 3/A, 3/B, and 3/C and a second table contains the values 1, 2, 3, then the first divided by the second gives A, B, C. (Don't worry, I think it's pretty weird and confusing, too, so it probably won't be on the final exam.)

The workhorse operation of the relational database is the join, which is often combined with selection and projection. For example, you could *join* Competitors records with NextOfKin records that have the correct name. Next, you could *project* to select only the competitors' names, the next of kin

names, and the next of kin phone numbers. You could then *select* only Bart Bold’s records. Finally, you could *select* for unique records so the result would contain only a single record containing the values Bart Bold, Betty Bold, 302-288-9278. That’s a result that you might actually need to use.

The following Structured Query Language (SQL) query produces this result:

```
SELECT DISTINCT Competitors.Name, NextOfKin.Name, Phone
FROM Competitors, NextOfKin
WHERE Competitors.NextOfKin = NextOfKin.Name
AND Competitors.Name = 'Bart Bold'
```

The `SELECT` clause performs selection, the `FROM` clause tells which tables to join, the first part of the `WHERE` clause (`Competitors.NextOfKin = NextOfKin.Name`) gives the join condition, the second part of the `WHERE` clause (`Competitors.Name = 'Bart Bold'`) selects only Bart’s records, and the `DISTINCT` keyword selects unique results.

DEFINITION Structured Query Language (SQL, pronounced “ess-que-ell” or sometimes “sequel”) is an English-like language that lets you perform standard operations on relational databases. It lets you define the database structure as well as perform relational CRUD and other operations on the data.

Don’t worry about the exact details for now. SQL is worth knowing, so Chapter 26, “Introduction to SQL,” provides an introduction, but for most of the book you should be able to read SQL queries intuitively without writing them.

The results of these operations are table-like objects that aren’t permanently stored in the database. They have rows and columns so they look like tables, but their values are generated on the fly when the database operations are executed. These result objects are called *views*. Because they are often generated by SQL queries, they are also called *query results*. Because they look like tables that are generated as needed, they are sometimes called *virtual tables*.

Later chapters have a lot more to say about relational database operations as they are implemented in practice.

POPULAR RDBs

There are many relational database products available for you to use. All provide the same basic features, such as the ability to build tables, perform CRUD operations, carry out the eight basic relational database operations (selection, projection, union, etc.), define indexes and keys, and so forth.

They all also provide some form of SQL. SQL is a standardized language, so many queries are the same in most RDBMSs, although there are some slight differences. For example, different systems call a 4-byte integer an `INT`, `INTEGER`, `NUMBER(4)`, or `INT4`. Many of these differences affect statements that modify the database (such as adding or deleting tables) rather than queries.

You can find a good SQL tutorial at www.w3schools.com/sql.

For a catalog of SQL differences on different RDBMSs, see https://en.wikibooks.org/wiki/SQL_Dialects_Reference.

You can find a list of around 100 RDBMSs at https://en.wikipedia.org/wiki/List_of_relational_database_management_systems and a similar list at <https://database.guide/list-of-relational-database-management-systems-rdbms>.

The following list shows the most popular RDBMSs in use as of September 2022 according to the DB-Engines page, <https://db-engines.com/en/ranking/relational+dbms>:

1. Oracle
2. MySQL
3. Microsoft SQL Server
4. PostgreSQL
5. IBM Db2
6. Microsoft Access
7. SQLite
8. MariaDB
9. Snowflake
10. Microsoft Azure SQL Database

I started writing a summary of each of the products, but found that they were so similar that it wasn't worth the effort. The following paragraphs describe some of their common features and give a few details for specific products.

Many of these products have free or community editions that have restricted features. Some restrict what you can do. For example, they place limits on table size, database size, and so on. Others restrict how long you can use the free version. The idea is that you can start with the free version, and then move to a more comprehensive (which coincidentally rhymes with “expensive”) version if you outgrow the free edition. Many of the free versions are still quite powerful, so many smaller projects can use those versions indefinitely.

Many of these can run in the cloud in some way. Some companies have separate cloud databases that are similar to their non-cloud versions. For example, Microsoft Azure SQL Database is Microsoft's cloud version of SQL Server. The cloud versions generally have their own free and non-free versions that you can try.

Although these are RDBMSs, many also include at least some NoSQL features. Some also include other features. For example, some of them have object-relational features, and PostgreSQL is really an object-relational database first.

MySQL, PostgreSQL (which is pronounced “post-gress-que-ell”), SQLite, and MariaDB are open source and freeish. The basic installations that you can obtain from the open source sites are free, but companies may provide their own versions that are not free.

For example, you can find the free community edition of MySQL at www.mysql.com/products/community, and links on that page lead to other editions that you can buy from Oracle.

You can download the free edition of PostgreSQL at www.postgresql.org. You can buy PostgreSQL in the cloud from various companies, such as Amazon, Google, Microsoft, EnterpriseDB, and others.

SQLite is a small, fast library written in C that provides a SQL database engine. It is not intended to be used as a stand-alone application, but rather as an engine embedded inside other applications. For example, it is embedded in Android, iOS, Mac OS X 10.4 and newer, and Windows 10 and newer. You can download the SQLite source code or compiled libraries at www.sqlite.org/index.html.

You can download MariaDB Community and find links to other versions at <https://mariadb.com/downloads/community>.

SPREADSHEETS

You've probably worked with spreadsheets before. Although they're not really relational databases, they have some analogous features. For example, a spreadsheet (analogous to a database) can contain multiple sheets (analogous to tables), which have rows and columns. Rows and columns are superficially similar to records and fields, but they are much less constrained, so a sheet's cells can contain just about anything.

Depending on the particular spreadsheet, you can add functions to cells to calculate new results or validate data entry in a way that is somewhat similar to the way a relational database can validate fields.

You can even use part of a sheet to validate the entries in another part of a sheet. For example, you could create a sheet listing product names, and then require that a cell on another sheet must contain one of those names.

Many spreadsheets also have some sort of programming interface or scripting capability. For example, you can write code behind a Microsoft Excel worksheet in *Visual Basic for Applications (VBA)*. It's a somewhat outdated language, but it adds to the power of Excel. Similarly, Google Sheets has a developer *application programming interface (API)* that you can use to manipulate sheets programmatically.

Spreadsheets are quite powerful and can perform some complex calculations, automatically updating when necessary, but they are not relational databases. For example, they don't perform relational queries, have indexed fields, enforce relationships among fields, or support many simultaneous users.

If you need the features of a spreadsheet, then by all means use one. However, if you find that you're writing more and more code to do things like join data from different areas, filter results, execute prepackaged queries, and check for errors, then you should consider using a relational database instead. I've worked on a couple projects that used spreadsheets that contained tens of thousands of lines of code behind the scenes and they ended in tears.

SUMMARY

Before you can start designing and building relational databases, you need to understand some of the basics. This chapter provided an introduction to relational databases and their terminology.

In this chapter you learned about:

- Formal relational database terms like relation, attribute, and tuple
- Informal terms like table, row, record, column, and field
- Several kinds of keys, including superkeys, candidate keys, and primary keys
- Different kinds of constraints that you can place on columns or tables
- Operations defined for relational databases

As I mentioned earlier, relational databases are popular, but they're not the only game in town. The next chapter describes a few nonrelational databases, some of which are growing more popular.

Before you move on to Chapter 3, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

EXERCISES

1. What does the following check constraint on the `SalesPeople` table mean?

```
((Salary > 0) AND (Commission = 0)) OR ((Salary = 0) AND (Commission > 0))
```

2. In Figure 2.3, draw lines connecting the corresponding terms.

Attribute	Row	File
Relation	Column	Relationship
Foreign Key	Table	Virtual Table
Tuple	Foreign Key	Record
View	Query Result	Field

FIGURE 2.3

For questions 3 through 6, suppose you're a numismatist and you want to track your progress in collecting the 50 state quarters created by the United States Mint. You start with the following table and plan to add more data later (after you take out the trash and finish painting your lead miniatures).

STATE	ABBR	TITLE	ENGRAVER	YEAR	GOT
Arizona	AZ	Grand Canyon State	Joseph Menna	2008	No
Idaho	ID	Esto Perpetua	Norm Nemeth	2007	No
Iowa	IA	Foundation in Education	John Mercanti	2004	Yes
Michigan	MI	Great Lakes State	Donna Weaver	2004	Yes
Montana	MT	Big Sky Country	Don Everhart	2007	No
Nebraska	NE	Chimney Rock	Charles Vickers	2006	Yes
Oklahoma	OK	Scissortail Flycatcher	Phebe Hemphill	2008	No
Oregon	OR	Crater Lake	Charles Vickers	2005	Yes

3. Is State/Abbr/Title a superkey? Why or why not?

4. Is Engraver/Year/Got a superkey? Why or why not?

5. What are all of the candidate keys for this table?

6. What are the domains of each of the table's columns?

For questions 7 through 10, suppose that you are building a dorm room database. Consider the following table. For obscure historical reasons, all of the rooms in the building have even numbers. The Phone field refers to the number of the landline phone in the room. Rooms that have no phone cost less, but students in those rooms are required to have a cell phone (so you can call and nag them if they miss too many classes).

ROOM	FIRSTNAME	LASTNAME	PHONE	CELLPHONE
100	John	Smith	Null	202-837-2897
100	Mark	Garcia	Null	504-298-0281
102	Anne	Johansson	202-237-2102	Null
102	Sally	Helper	202-237-2102	Null
104	John	Smith	202-237-1278	720-387-3928
106	Anne	Uumellmahaye	Null	504-298-0281

ROOM	FIRSTNAME	LASTNAME	PHONE	CELLPHONE
106	Wendy	Garcia	Null	202-839-3920
202	Mike	Hfuhruhurr	202-237-7364	Null
202	Jose	Johansson	202-237-7364	202-839-3920

7. If you don't allow two people with the same name to share a room (due to administrative whimsy), what are all of the possible candidate keys for this table?

8. If you do allow two people with the same name to share a room, what are all of the possible candidate keys for this table?

9. What field-level check constraints could you put on this table's fields? Don't worry about the syntax for performing the checks, just define them.

10. What table-level check constraints could you put on this table's fields? Don't worry about the syntax for performing the checks, just define them.

3

NoSQL Overview

Chapter 2, “Relational Overview,” explained some relational database concepts and the terms that go with them. This chapter explains ideas that go with nonrelational NoSQL databases.

Many of the latest NoSQL database providers offer their services in the cloud, so cloud databases and NoSQL often go hand in hand, although the pairing isn’t strictly necessary. You can run a NoSQL database on your local computer, and you can use the cloud for many things other than NoSQL databases, including relational databases.

The following section discusses cloud databases in general. The rest of the chapter talks about different NoSQL database types and provides some information that can help you pick the right database for you.

THE CLOUD

Often when people discuss the cloud, I’m reminded of this (slightly modified) dialogue from *The Matrix*:

Trinity: . . . It’s the question that brought you here. You know the question, just as I did.

Neo: What is the Cloud?

Trinity: The answer is out there, Neo, and it’s looking for you, and it will find you if you want it to.

Many people want the cloud in their business, but they don’t really know what the cloud is. They just know that they need it.

Fortunately, you don't need to take the red pill to learn about the cloud. The cloud is simply an amorphous collection of remote computer resources that you can use on demand. For example, you can use remote servers to store data, perform data analysis, or execute special-purpose applications.

In fact, unless you run your own email server, you've probably been using the cloud for years before it was even called the cloud. Your email service provides both remote storage (for emails) and a special-purpose application (email tools). Similarly, most web pages are hosted on servers owned by a web-hosting company rather than on your home computer.

In addition to services like email and web hosting, cloud services have recently expanded enormously to include all sorts of offerings that provide banking, blockchain, desktop, data management, IT, logging, payment, commerce, database, storage, and much more.

Cloud services have become popular largely because they allow you to take advantage of specialized hardware or software products without getting your hands dirty with day-to-day management. You can store your data in a cloud database without worrying about how to distribute the data across different servers, how to back up the data, or how to expand the database if your needs grow.

Cloud services are also a win from the provider's point of view because they let you "rent" software rather than buying it once and then owning it forever. For example, in the past you could buy office software such as a word processor or slideshow software outright. If the added features in later editions don't interest you, then you can use your original version for years or possibly even decades. With a cloud service model, you pay a monthly or annual fee to use the software.

This may seem like an obvious rip-off, but it does give you, the consumer, a couple of advantages. First, it allows the vendor to fix bugs and security flaws immediately without requiring you to install a new version. It may also mean the vendor can focus more on maintenance and improving the product rather than on inventing new features that no one wants just to drive new sales every two years. (That doesn't mean they actually do that, but they could.)

In general, cloud storage and services are assumed to be distributed across multiple servers. That assumption provides a couple of relatively standard benefits. For example, it is relatively easy to adjust the amount of storage space and computing power for a cloud application as your needs change. That means cloud databases can grow and shrink as necessary and you usually don't need to know the details.

NoSQL database providers often advertise easy scalability as a key benefit of their products (and that is true), but that scalability is generally due to the nature of cloud computing and it applies to relational cloud databases, too.

CONSISTENCY

One issue that widely distributed databases must address is consistency. If pieces of a database are scattered around the world, then it may take some time for a change to propagate to all the pieces.

Some databases provide *eventual consistency*. In this model, changes to the data eventually work their way throughout the data servers, but it may take a while. For example, suppose a database stores redundant copies of its data on several servers. That may improve response time by allowing users to access the different servers, but if the data changes in one location, it may take a while for the change to spread to the other locations.

In contrast, if a database provides *strong consistency*, then it always provides the latest data in response to queries. The downside is that it might take a bit longer. When someone changes a value, you cannot access that value until the update has filtered throughout the database's servers. Queries take a little longer, but everyone should have the most up-to-date data.

Eventual consistency may sound like a poor second choice, but there are situations where it works quite well. For example, consider a big sales database like the one used by Amazon. Product information such as descriptions, pictures, prices, and shipping terms makes up a huge amount of its data. That data rarely changes and when it does, it probably doesn't matter too much whether a customer sees a slightly stale description or set of reviews. From a philosophical point of view, it's as if the customer had loaded the page a few seconds earlier just before the change. For that data, eventual consistency is okay.

In contrast, the customer's shopping cart order and order history should always be up-to-date. If I click Add To Cart and then look at my shopping cart, my new copy of *The Cat in the Hat* better be there! Here, strong consistency is important to avoid confusing (or infuriating) customers.

Although you generally don't need to worry about the day-to-day management of the cloud services that you use, one detail that you do need to know about is pricing. Most cloud services use pay-as-you-go payment plans so that you are charged for the storage and services that you use. In some ways that makes sense: use more, pay more. Unfortunately, it can also make payment plans as confusing as badly translated printer setup instructions. In some plans, you might pay for session time, CPU time, disk storage, disk access speeds (slower speeds sometimes cost less), number of inbound and outbound data transactions, bandwidth, and whether you have access to a dedicated (rather than a shared) server. All of that makes it very hard to anticipate the charges that you'll incur each month.

You can go online to learn a lot more about the cloud in general and particular cloud offerings. Many vendors have free tiers with limited storage or trials that expire after a certain period of time. Both of those approaches allow you to give cloud services a try. Then, as your needs grow, you can use the widely touted cloud expandability to increase the amount of disk space and processing that you use (at increased cost, of course).

The rest of this chapter discusses different kinds of NoSQL databases, the types of operations that they support well, and considerations that can help you decide which kind of database will be most effective for you. After you pick the best option, you can decide whether you want to host your database locally or in the cloud.

PICKING A DATABASE

There’s an expression, “If all you have is a hammer, everything looks like a nail.” If the only kind of database you understand is the relational database described in Chapter 2, then you’ll probably try to hammer every kind of data into a relational database, and that can sometimes lead to trouble. (Remember the example in Chapter 2 about the program that stuffed hierarchical data into a relational database?)

The lesson is clear: before you spend a lot of time building the ultimate relational database and piling thousands of lines of code on top of it, make sure that’s really the kind of database you need. Had this project started with an XML database, it probably would have had a simpler, more natural design with much less code and would probably have lasted for many years to come.

NoSQL PHILOSOPHY

There are two NoSQL schools of thought. The first says that NoSQL stands for “not SQL” or “non-SQL.” The second says that NoSQL means “not only SQL.”

Some NoSQL providers deal only with one or more nonrelational kinds of databases, so they have a “not SQL” flavor. However, it doesn’t make sense to say that nonrelational databases are *always* better (despite what some of those providers say), so it’s a mistake to rule out relational databases completely. That would be like saying screwdrivers are better than hammers. Each has its place and it’s silly to try to turn screws with a hammer or drive nails with a screwdriver. Similarly, you should use relational databases where they fit and nonrelational ones where they make the most sense.

Some database providers work with both relational and nonrelational databases. For example, Amazon Web Services (AWS) has both relational and nonrelational offerings, so in a sense it is truly a “not only SQL” provider. In general, however, people use the term “NoSQL database” to contrast with relational databases.

Therefore, as a design strategy, I use NoSQL to mean “not only SQL” so I don’t rule out any database tools that might be useful. This chapter discusses relational and nonrelational databases and so it falls into the “not only SQL” category.

However, when I say, “NoSQL database,” I mean “non-SQL,” so I’m specifically talking about nonrelational databases. (Sometimes, I’ll even say “nonrelational database” to make it blindingly obvious.)

NoSQL DATABASES

You can declare any nonrelational database a NoSQL database, but there are four official types: document, key-value, column-oriented, and graph. The following sections describe these kinds of databases.

Document Databases

As I'm sure you can guess, *document databases*, which are also called *document stores*, hold documents. Typically, the documents use *JavaScript Object Notation (JSON)* or *Extensible Markup Language (XML)* to structure their information, so the documents aren't simply big piles of text.

This kind of database works well when your application works with reasonably self-contained objects so that you can load and store information about objects separately.

For example, suppose you want to store information about DVDs. Each document might represent a particular DVD and hold information such as its title, release date, director, cast members, and so forth. The following code shows how a JSON document might hold information for a *The Lord of the Rings: The Fellowship of the Ring* DVD:

```
{
  "title" : "The Lord of the Rings: The Fellowship of the Ring",
  "image_url" : "https://..."
  "release_date" : "2002-11-12",
  "rating" : "PG-13",
  "product_id "1276193: " ",
  "studio" : "New Line Home Video",
  "director" : "Peter Jackson",
  "run_time" : "P3H28M",
  "cast" : [
    "Elijah Wood",
    "Sean Astin",
    "Sean Bean",
    "Viggo Mortensen",
    ...
  ],
  "synopsis" : "Four hobbits take a vacation.",
  ...
}
```

Here, the ellipses aren't part of the document; they just show where I've omitted data to keep the example small.

I'll describe the rules for building a JSON file later in this chapter in the section "JSON Files," but you can probably read the example data with no trouble. For example, the entry's `title` is *The Lord of the Rings: The Fellowship of the Ring*, the `rating` is "PG-13," and the `synopsis` is "Four hobbits take a vacation."

About the only mystery (aside from "What's in the omitted pieces represented by ellipses?") is, "What does a `runtime` of P3H28M mean?" This is a standard ISO 8601 textual time format. The P stands for "period," so it represents a duration of time as opposed to a date or time of day. The 3H means 3 hours and (you can probably guess) the 28M means 28 minutes. So, that movie's runtime is 3 hours and 28 minutes.

Document databases allow you to query on fields. For example, you could search the database to fetch documents where the `director` is Peter Jackson or the `cast` array contains Sean Bean. You can also delete documents or update the values inside them.

Unfortunately, it's not as easy to modify a document. Because of the way they are stored, you can't update the title to fix a typo or change the synopsis to provide some additional detail. Instead, you must replace the document with a new version.

Different documents in the database do not necessarily all have the same fields or the same kinds of values in their fields. For example, you might add another document describing the *Zootopia* DVD. In that document, the `director` field's value would be an array to hold the two directors, Rich Moore and Byron Howard, so it wouldn't have the same structure as the document for *The Fellowship of the Ring*. Your program will need to figure out how to handle any different formats.

Key-Value Database

A *key-value database*, which is also called a *key-value store*, holds values that you can index by using a key. This is similar to the hash tables, dictionaries, and associative arrays that are provided by different programming languages. In the non-digital world, this is similar to the way you can use a dictionary to look up a word (the key) to see its pronunciation and the definition (the value). (You might have to go to a library or museum to find a printed dictionary these days.)

Key-value databases are often used to cache data in memory so that your program doesn't need to constantly work with slower storage on disk. Some key-value databases may stay completely in memory and are destroyed when the computer reboots or loses power. Others may shadow the data onto a hard disk so that it can be automatically restored if necessary, although that may slow them down.

A key-value database treats the values as blobs of data that have meaning to your program but not to the database. In contrast, a document database understands that it contains fields so it can query for values in those fields.

BLOBS AND BLOBs

You can think of the values in a key-value database as “blobs” in the sense of unstructured chunks of data, but “BLOB” also has a technical meaning. BLOB stands for “binary large object,” and it means pretty much the same thing: a bunch of 0s and 1s that a program treats as an unstructured chunk of data. Key-value databases generally store their values, not only as blobs but also as BLOBs.

If you like, you can index some of the fields in a document database to make searching on those fields faster. Because a key-value database doesn't understand the structure of whatever it is storing, it can't create indexes on the data inside the documents.

Column-Oriented Databases

Column-oriented databases are also called *wide-column databases* or *column family databases*. They allow you to treat data in a table-like format, but unlike the rows in a relational database, the rows in a column-oriented database can have different fields.

For example, suppose you own a gymnastics club and you want to build a table to store information about coaches, gymnasts, and parents. All of the records will contain basic information such as name, address, and phone number, but the different kinds of records will also contain some other types of information.

Coach records contain fields indicating their salaries, work schedule, which events they can coach (beam, floor, rings, ribbon), training they have taken, and which certifications they currently hold. Gymnast records hold fields indicating enrolled classes and practices, competition level, which skills they've mastered (somersault, handstand, switch leap, full-twisting Shaposhnikova), who their parents are, and whether they have paid their membership fees. Parent records include the kinds of "volunteer" help they can provide for the gym (cleaning, meet setup and takedown, working concessions, videography), and whether they've performed their four mandatory volunteer hours each month.

Later in the book, you'll see that it wouldn't be too hard to put this data into a relational database. You would first pull the pieces apart into separate Contacts, Coaches, Gymnasts, and Parents tables. Then you would use ID fields to connect related records.

In a column-oriented database, you can treat this as a single giant table. You wouldn't need to figure out how to pull the pieces apart or create ID fields. You also wouldn't need to use those ID fields to fetch data for a particular person. Instead, you would simply fetch the person's record from the massive table.

Even more important, if you wanted to change the kinds of data stored in the table, you wouldn't need to rearrange the database. For example, suppose you wanted to add information about judges. In a relational database, you would need to create a new table and link it to the Contacts table. In a column-oriented database, you would simply create new records, adding any new fields for the judges (such as their judging certifications, ratings, and favorite flavor of coffee for breaks).

Column-oriented databases are optimized to efficiently fetch or search the data in a column efficient manner. For example, it would be easy to search the Level column to find the level 8–10 gymnasts. Of course, a typical gym would probably only have a few hundred gymnasts and a handful of level 8–10 gymnasts, but larger column-oriented databases should be able to scan millions of column entries in a few seconds.

Sometimes, certain columns apply only to a few rows, so a column-oriented database may be relatively sparse and contain many "empty" entries. To avoid wasting space, these sorts of databases are usually optimized to represent sparse data without allocating huge tracts of empty disk space.

Graph Databases

A *graph database* consists of nodes, edges, and properties.

A *node* represents a piece of data such as information about a person, product, or location in a city.

An *edge* connects two nodes and represents a relationship between those two nodes. Edges are also sometimes called *links* or *relationships*.

A *property* is a fact about a node or edge.

Many other types of database focus on facts about objects such as an employee's address, an order's items, or a product's price. In contrast, graph databases focus on the relationships (represented by edges) between objects. That works well if you want to study those relationships, but it doesn't help if you want to study the objects (represented by the nodes).

For example, suppose you build a graph database showing connections among your company's employees, managers, departments, and divisions. Then finding all the employees who work in a certain department would be a graph-oriented search that would use the relationships between employees and departments.

In contrast, finding employees who entered more than 60 hours on last week's timesheet (so you can buy them cots to put in their offices so they can work even more) would only examine the employee objects' properties and not the relationships. Some graph databases can build indexes on properties so this might still be a fast search, but it wouldn't be an improvement over what relational databases do. If you only need to make this kind of query, then a relational database may do a better job.

The next few sections describe some examples that could use graph databases.

Street Networks

Figure 3.1 shows a simple ordered street network. The number on a link represents the average time in seconds it takes to cross the link. The letters on the nodes are just there for identification.

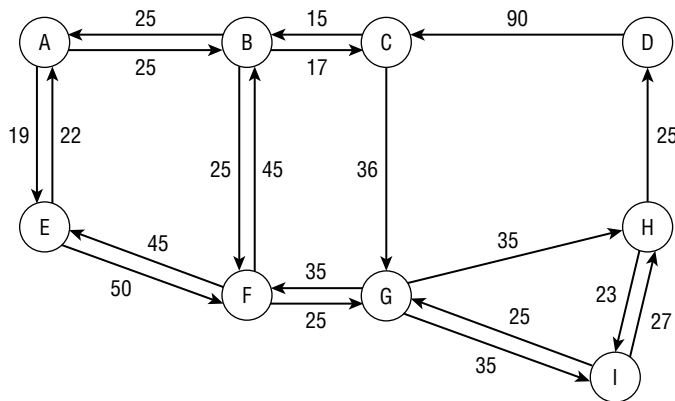


FIGURE 3.1

A typical problem for this street network might be to find the shortest route from the police station at node A to the donut store at node D. The police will be using their lights and sirens so you don't need to worry about turn penalties. (A common feature in shortest path algorithms makes it take longer to turn than to go straight at an intersection.) See if you can find the solution.

Other operations that you might perform on a street network include the following:

- Finding the shortest path from your house to your favorite bicycle store
- Finding all of the coffee shops within one mile of your house
- Finding the best route for an ice cream crawl (where you visit all the ice cream stores in town)

Communication Networks

You can use a graph database to represent a telephone network, intranet, wide area computer network, or other communication network. Just as you can in a street network, you could use the communication network to find the shortest paths between pairs of nodes.

You could also use the network to look for bottlenecks or important points of failure. For example, you might want to see if there are multiple disjoint paths between pairs of important nodes. Then if an edge in one path fails, there will still be another path between the nodes.

Social Media Apps

You could use a graph database to represent the relationships among users of a social media app. Different kinds of nodes would represent people or groups. Nodes would have properties such as Name, Photo, Birthdate, and FavoriteQuote.

Different kinds of edges might represent the relationships Follows, IsFriendsWith, and HasBlocked. Edges might have properties such as DateCreated and IsModerator (for someone who moderates a group).

Some edges might be symmetric. For example, if Alice IsFriendsWith Ben, then Ben IsFriendsWith Alice (at least as far as the app is concerned). These kinds of edges are sometimes called “undirected” because they represent the same concept in either direction.

Other kinds of edges may be asymmetric, in which case they typically imply an inverse relationship. For example, if Carter Follows BTS, then BTS IsFollowedBy Carter. (Carter is probably more excited about this than BTS is.) These edges are sometimes called “directed” because their relationship goes from one node to another but not the other way around.

After you build the graph, you can use it to study the social network. For example, you could examine other people who follow BTS to see what they have in common.

You could look also for highly connected clusters of friends and suggest missing edges. For example, suppose you find a clique of 10 people who are all mutual friends. If you find another person who is friends with 8 of the 10, then you might suggest that this person connect with the other two people.

E-Commerce Programs

Here, nodes represent customers and products. Edges represent relationships such as SearchedFor, Purchased, and Reviewed. The Reviewed edges might carry properties holding the review’s star rating and the customer’s comments. Now, when a customer searches for a particular term, you can look for previous customers who used that term and see what they searched for and what they bought. You could then provide “See what other customers searched for” and “See what other customers bought” lists.

Algorithms

Graph data structures turn out to be very useful for certain kinds of algorithms such as shortest path finding (which I've already mentioned), other route finding (such as the traveling salesperson problem and finding Hamiltonian, Eulerian, and Euclidean paths—look online for details), task scheduling, work assignment, network flow (think water flowing through pipelines), connectivity, network congestion, and more.

Hierarchical Databases

Hierarchical databases actually form a subset of graph databases, but they're important enough that I wanted to give them their own section. Hierarchical data includes values that are naturally arranged in a tree-like structure. One piece of data somehow logically contains or includes other pieces of data.

Files on a disk drive are typically arranged in a hierarchy. The disk's root directory logically contains everything in the filesystem. Inside the root are files and directories or folders that break the disk into (hopefully useful) categories. Those folders may contain files and other folders that further refine the groupings.

The following listing shows a tiny part of the folders that make up a filesystem. It doesn't list the many files that would be in each of these folders.

```
C:\
  Documents and Settings
    Administrator
    All Users
    Ben Grim
    Groo
    Rod
  Temp
    Art
    Astro
  Windows
    Config
    Cursors
    Debug
    system
    system32
      1025
      1031
      1040 short form
      1099 int
```

The disk's root directory is called `C:\`. It contains the `Documents and Settings`, `Temp`, and `Windows` directories. `Documents and Settings` contains folders for the administrator and all users in general, in addition to folders for the system's other users.

The `Temp` directory contains temporary files. It contains `Art` and `Astro` folders that hold temporary files used for specific purposes.

The `Windows` directory contains various operating systems files that you should generally not mess with.

Many other kinds of data can also be arranged hierarchically. Figure 3.2 shows an org chart that is arranged hierarchically. The lines indicate which people report to which others.

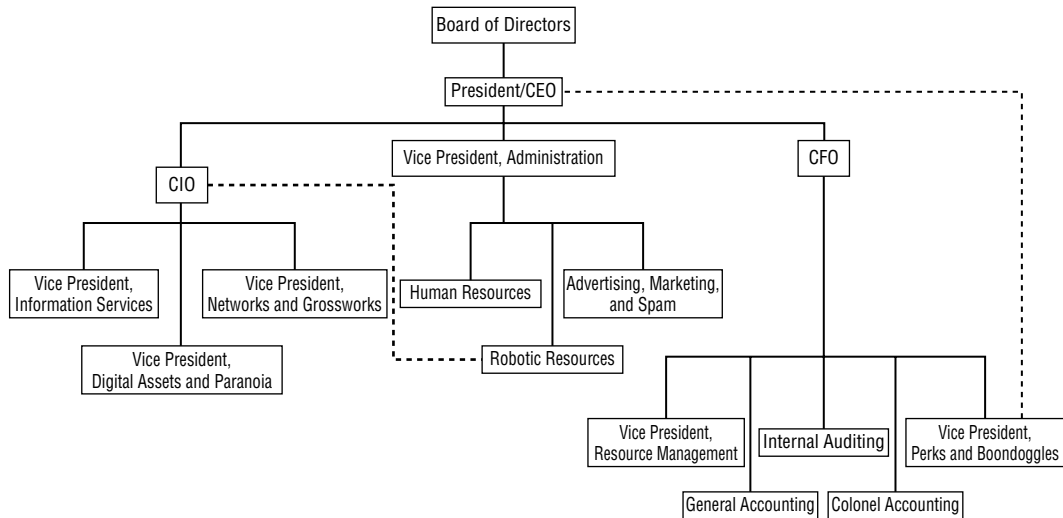


FIGURE 3.2

Figure 3.3 shows the same information in a slightly different format. This version is arranged more vertically in a way similar to that used by File Explorer (formerly Windows Explorer) to show a disk's filesystem.

In a pipe system, typically big pipes feed into smaller ones to form a hierarchy. Water flows from a treatment plant to large distribution pipes that break into smaller and smaller pipes that eventually feed into houses, coffee shops, and drinking fountains.

Similarly, electricity flows from a power plant across high-voltage, long-distance transmission lines at a few hundred thousand volts (there's less power loss at higher voltages). Next a transformer (the electrical kind, not a robot that turns into a car) lowers the voltage to 13,800 or so volts for more local transport. Some of it is used by factories and large businesses. The rest moves through more transformers that reduce the voltage to 110 or 220 volts (in the United States anyway) for use by your latte machine and desktop computer. (It doesn't even stop there. Your computer again reduces the voltage to 5 volts or so to power your USB plasma ball, missile launcher, and web camera.)

Other examples of data that you can arrange hierarchically include the following:

- A family tree tracing your ancestors back in time (you have parents, who have their own parents, who also have parents, and so forth).
- A family tree tracing someone's descendants forward in time (that person has some children, who each have more children, and so on).

- The parts of any complicated object (a computer has a keyboard, mouse, screen, and system box; the system box includes a fan, power supply, peripherals, and a motherboard; the motherboard includes a chip, heat sink, and memory; and so forth).
- Order tracking information (customers have orders; orders have basic information such as dates and addresses, in addition to order items and possibly sub-orders; order items have an inventory item and quantity; an inventory item has description, part number, price, and in some applications sub-items).
- Even the information in this book forms a hierarchy (it has chapters that contain sections; sections contain sub-sections and paragraphs; paragraphs contain sentences, which contain words, which contain characters).

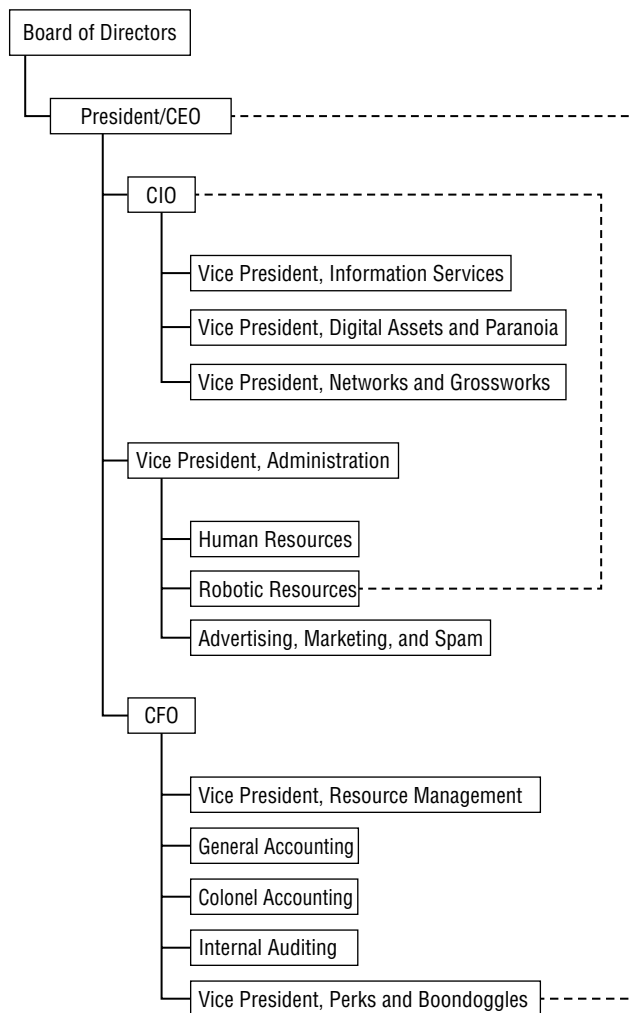


FIGURE 3.3

A hierarchical, tree-like structure is a special kind of graph where the edges are directed (for example, from parent to child in a family tree) and there are no loops (for example, no one is their own grandparent, except in science fiction stories).

ALMOST HIERARCHICAL

Unfortunately, some data sets are almost, but not quite, hierarchical. For example, an org chart is hierarchical if every employee reports to exactly one person, but if you add in matrix management where employees may work in more than one department and have several managers for different purposes, then the result is no longer a hierarchy.

Similarly, some operating systems allow you to create links from one part of the filesystem to another. This lets you create a folder that seems to be inside one directory but entering it warps you to a completely different part of the filesystem. In that case, the filesystem isn't really a hierarchy anymore. Instead of a tree, it's more like a bush, with grafts and intertwined branches that make strange backward connections like some sort of space alien hybrid plant or a plate of spaghetti.

If you store hierarchical data in a graph database, then you can perform the usual graph database operations, although they may have slightly different meanings. For example, you could find the shortest path between two nodes, but you might need to follow some of the links backward. In a family tree, you would need to climb up the tree and then back down to find your cousins.

Other graph algorithms may not make too much sense in a hierarchical data set. For example, it probably doesn't make sense to search for routes that visit many nodes or for groups of highly connected nodes.

LESS EXOTIC OPTIONS

Relational and NoSQL databases represent the state of the art for storing, manipulating, and retrieving data, but there are times when the best tech is low tech. If you don't need to execute a bunch of queries and rearrange the data, then there are many simple ways that you can store a pile of data and then read it back later. The following sections describe some of these low-tech approaches.

Flat Files

Flat files are simply files containing text. Nothing fancy. No **bold**, *italic*, *different font faces*, or other special font tricks. Just text.

You can add structure to these files—for example, by separating values with commas or using indentation to hierarchy—but the basic file is just a pile of characters. Some structured variations such as JSON files and XML files are described later in this chapter.

Text files provide no special features. They don't help you search for data and don't provide aggregate functions such as calculating totals, averages, and standard deviations. Writing code to perform any

one of those kinds of operations is fairly easy, but it's extra work and providing flexible ad hoc search capabilities is hard. If you really need those, consider a relational or NoSQL database.

Programs cannot modify flat files in general ways. For example, you may be able to truncate a file, add data to the end, or change specific characters within the file, but in general you cannot insert or delete data in the middle of the file. Instead, you must rewrite the entire file to make those sorts of changes.

Although flat files provide few services, don't scoff at them. They are extremely simple, easy to understand, and easy to read and write in most programming languages, so they are a good choice for some kinds of data. You can also open a flat file in any text editor and make changes without needing to write a complex user interface. Try that with a relational database or a graph database hosted in the cloud!

If a piece of data is relatively simple and seldom changes, a flat file may be an effective, simple way to store the data. For example, a flat file is a fine place to store a message of the day. Each day you can type in one of your favorite obscure quotes for a program to display when it starts. (To quote Ralph Waldo Emerson, "The next thing to saying a good thing yourself, is to quote one.")

Flat files are also good places to store configuration settings. A configuration file lists a series of named values that a program can read when it needs them. Often, a program loads its configuration information when it starts and doesn't look at the configuration file again.

Flat files don't provide many fancy features, but if you don't need any, they may be the easiest solution.

XML Files

XML is a language for storing hierarchical data. XML itself doesn't provide any tools for building, searching, updating, validating, or otherwise manipulating data, and anyone who tells you otherwise is trying to sell you something.

However, XML is a fairly useful format for storing, transferring, and retrieving hierarchical data, and there are several common tools that can make working with XML files easy. Some of those other tools let you build, search, and otherwise work with XML files, but those features aren't built into XML itself.

XML GRAPH GOTCHAS

XML is mostly used to store hierarchical data, but some tools can use it to store more general graphs that are not hierarchical. Unfortunately, only some tools can do that. For example, the normal .NET serializer cannot do that, but the binary serializer can. The downside, of course, is that the result is in binary, so you cannot easily read and edit it with a text editor.

You can also store a graph in XML by giving each node an ID and a list of the IDs of its neighbor nodes. Then you can load all the nodes and reconstruct the links in your program. Of course, that means more work for you.

XML files (and the tools that manipulate them) are moderately complicated, so this book doesn't explain everything there is to know about them. The following sections provide a quick overview of XML to help you recognize when XML files might be a better choice than a fancier database. You can find many other books and online resources that cover XML in excruciating detail.

The following sections provide some more details about XML files.

XML Basics

An XML file is a relatively simple text file that uses special tokens to define a structure for the data that it contains. People often compare XML to the Hypertext Markup Language (HTML) used to build web pages because they both use tokens surrounded by *<pointy brackets>*, but the two languages have several big differences.

One major difference between XML and HTML is that XML is extensible. (The X isn't part of the name just to sound edgy and cool—that's just an added bonus.) HTML commands are predefined by the language specification, and if you try to invent new ones, it's unlikely that a typical browser will know what to do with them.

In contrast, XML defines general syntax and options, but you get to make up the tokens that contain the data as you go along. All you need to do is start using a token surrounded by *<pointy brackets>*. You follow the token by whatever data it should contain and finish with a closing token that is the same as the opening token except that it starts with a slash.

For example, the following text shows a single XML token called `Name` that contains the value Rod Stephens:

```
<Name>Rod Stephens</Name>
```

Programs that read XML ignore whitespace (nonprinting characters such as spaces, tabs, and carriage returns), so you can use those characters to make the data more readable. For example, you can use carriage returns and tabs to indent the data and show its hierarchical structure.

You can invent new tokens at any time. For example, the following code shows a `Person` element that includes three fields called `FirstName`, `LastName`, and `NetWorth`. The text uses carriage returns and indentation to make the data easy to read:

```
<Person>
  <FirstName>Rod</FirstName>
  <LastName>Stephens</LastName>
  <NetWorth>$16.32</NetWorth>
</Person>
```

A second important way in which XML and HTML differ is that XML is much stricter about properly nesting and closing opened tokens. For example, the HTML `<p>` command tells a browser to start a new paragraph. A paragraph can't contain another paragraph, so if the browser sees another `<p>` tag, it knows to end the current paragraph and start a new one, so you don't need to explicitly include a closing `</p>` tag.

Similarly, if the browser sees a list item tag ``, it automatically closes the preceding one if one is open.

For another example, the horizontal rule `<hr>` element cannot contain text (or anything else, for that matter), so the browser assumes a closing tag immediately after `<hr>` just as if you had typed `<hr></hr>` or the shorthand version `<hr/>`. All in all, HTML is relatively forgiving.

In contrast, XML requires every opening token to have a corresponding closing token. (However, it does allow you to use the shorthand syntax for tokens that immediately open and then close, as in `<Closed />`.)

XML requires that elements be properly nested. One element can completely contain another, but they cannot not overlap so one contains only part of another.

For example, the following text includes a `FirstName` element. While that element is open, the text starts a `LastName` element but the `FirstName` element closes before the `LastName` element does. (The weird indentation makes the overlap easier to see.) This violates XML's nesting rules, so this is not a properly formed piece of XML.

```
<Person>
  <FirstName>Rod
    <LastName>Stephens
  </FirstName>
  </LastName>
  <NetWorth>$16.32</NetWorth>
</Person>
```

An XML file can also define *attributes* for an element. For example, in the following XML code, the `Person` element has an attribute named `profession` with the value `Dilettante`:

```
<Person Profession="Dilettante">
  <FirstName>Rod</FirstName>
  <LastName>Stephens</LastName>
  <NetWorth>$16.32</NetWorth>
</Person>
```

Note that attributes must be enclosed in quotes even if they are numeric values.

You can enclose a comment in an XML file by starting it with the characters `<!--` and ending it with the characters `-->`. For example, the following XML code adds a comment to the previous code:

```
<!-- The book's author -->
<Person Profession="Dilettante">
  <FirstName>Rod</FirstName>
  <LastName>Stephens</LastName>
  <NetWorth>$16.32</NetWorth>
</Person>
```

The final XML rule covered here is that the file must have a single root element that contains all other elements. This makes the file an absolutely pure, true hierarchy of data. Well, almost. The file can also begin with an optional XML declaration that gives the XML version.

The following text shows a slightly more elaborate XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<ClassSchedule>
  <Class Name="Wall Climbing for Beginners" Room="Atrium">
    <!-- Note: Requires Falling 101. -->
    <Instructor>Peter Parker</Instructor>
    <Students>
      <Student>
        <FirstName>Ben</FirstName>
        <LastName>Breaker</LastName>
      </Student>
      <Student>
        <FirstName>Carla</FirstName>
        <LastName>Crash</LastName>
      </Student>
      <Student>
        <FirstName>Dirk</FirstName>
        <LastName>Drop</LastName>
      </Student>
    </Students>
  </Class>

  <Class Name="Advanced Pyrotechnics" Room="Field 3">
    <!-- Note: Requires fire-retardant suit. -->
    <Instructor>Johnny Storm</Instructor>
    <Fees Materials="$45" />
    <Students>
      <Student>
        <FirstName>Erica</FirstName>
        <LastName>Enflame</LastName>
      </Student>
      <Student>
        <FirstName>Frank</FirstName>
        <LastName>Flammable</LastName>
        <NickName>Flamb&#xE9;</NickName>
      </Student>
    </Students>
  </Class>
</ClassSchedule>
```

This file begins with an XML declaration indicating that it uses XML version 1.0 and the UTF-8 character encoding. It then starts a `ClassSchedule` element that holds all of the document's other content.

The `ClassSchedule` element contains two `Class` elements. Each `Class` element has `Name` and `Room` attributes that give the class's name and location.

The `Class` elements contain `Instructor` and `Fees` elements that hold basic information about the classes. Each also includes a `Students` element that contains information about all the students enrolled in the class. The detailed student information is contained in `Student` elements that hold `FirstName` and `LastName` elements.

Note that the elements need not contain exactly the same kinds of content. For example, the second class contains a `Fees` element, but the first does not. Similarly, the final `Student` element contains a `NickName` element, but none of the other `Student` elements do. (The text `é` in that value makes the `NickName` data include the character with Unicode hexadecimal value E9. That's the é character, so the nickname is Flambé.)

Because you can make up XML elements as you go along, they allow more flexibility than some other kinds of databases. A relational database, for example, defines exactly what fields are contained in every record in a table. In an XML file, you can add new elements at any point in the file. The XML file's elements provide self-documenting names (if you give your elements reasonable names and not just “e1” and “N32”). This kind of flexible, self-describing database is called *semi-structured*.

XML *schema files* allow you to provide some validation. For example, they let you indicate that a particular element must contain certain other elements, that an element must contain a date or number, or that an element is required.

XML Practices

I see XML files used most frequently in one of three ways.

First, XML files are hierarchical, so it's natural to use them to hold hierarchical data. It's straightforward to map purely hierarchical data such as a simple family tree or an org chart into an XML file.

Second, XML files are often used to hold table-like data. The basic structure closely follows the structure of a relational database. The root element holds several “table” elements. Each of those elements holds “records” that hold “fields.” For example, the following XML document holds data about a simple company's customers and their orders:

```
<AllData>
  <Customers>
    <Customer ID="1">
      <FirstName>Alfred</FirstName>
      <LastName>Gusenbauer</LastName>
    </Customer>
    <Customer ID="2">
      <FirstName>David</FirstName>
      <LastName>Thompson</LastName>
    </Customer>
    <Customer ID="3">
      <FirstName>Alberto</FirstName>
      <LastName>Selva</LastName>
    </Customer>
  </Customers>
  <Products>
    <Product ID="273645" Description="Toothbrush" Price="$1.95" />
    <Product ID="78463" Description="Pencil" Price="$0.15" />
    <Product ID="48937" Description="Notepad" Price="$0.75" />
  </Products>
  <CustomerOrders>
    <CustomerOrder Date="12/27/2008" CustomerId="2">
      <Item ID="1" ProductId="78463" Quantity="12" />
    </CustomerOrder>
  </CustomerOrders>
</AllData>
```

```

        <Item ID="2" ProductId="48937" Quantity="2" />
    </CustomerOrder>
</CustomerOrders>
</AllData>

```

The file starts with an `AllData` root element. That element contains three more elements that define table-like structures holding customer, product, and customer order information.

Each of these “tables” defines “records.” For example, the `Customers` element includes `Customer` “records” that hold `FirstName` and `LastName` values.

This XML document uses ID numbers to link records in different “tables” together. In this example, the single `CustomerOrder` element represents an order placed by customer 2 (David Thompson) who ordered 12 items with ID 78463 (pencils) and two items with ID 48937 (notepads).

The third XML file structure I see regularly is a simple list of values. The following XML document uses this structure to hold configuration settings for an application:

```

<Settings>
  <NormalColor>Black</NormalColor>
  <WarningColor>Green</WarningColor>
  <ErrorColor>Yellow</ErrorColor>
  <PanicSound>panic.wav</PanicSound>
  <BugEmail>bugs@panic.com</BugEmail>
</Settings>

```

This kind of XML file gives a little more structure than a flat text file used to hold settings, and it lets a program use XML tools to easily load and read setting values.

This flat structure is also useful when each XML document corresponds directly to some sort of object that a program will use. For example, the following XML file defines a letter. A program could load this data and use its fields to print and mail the letter.

```

<Letter>
  <ToName>Hulk Hogan</ToName>
  <ToStreet>2615 Grappler St, #12</ToStreet>
  <ToCity>Gripper</ToCity>
  <ToState>CA</ToState>
  <FromName>Hakuh&#333; Sh&#333;</FromName>
  <Body>
Respected Sir,

Regarding your challenge: Bring it! Your dohy&#333; or mine?

Sincerely,
  </Body>
</Letter>

```

The following code shows the same data but in a more structured format:

```

<Letter>
  <To>

```

```
<Name>Hulk Hogan</Name>
<Address>
  <Street>2615 Grappler St, #12</Street>
  <City>Gripper</City>
  <State>CA</State>
</Address>
</To>
<From>HakuH&#333; Sh&#333;</From>
<Body>
Respected Sir,

Regarding your challenge: Bring it! Your dohy&#333; or mine?

Sincerely,
  </Body>
</Letter>
```

This version creates a `To` element that includes all the information about the letter’s recipient. The `To` element contains the recipient’s name and an `Address` element that holds address information. You could add similar information for the sender. (The special character codes make the `FromName` value Hakuhō Shō and the body contains the word dohyō. Look them up online if you’re interested.)

XML Summary

XML files are hierarchical, which makes them a natural choice for storing hierarchical data. Aside from graph databases, other kinds of databases are not likely to be able to re-create the hierarchical object structure as quickly as XML tools can. (See the section “Picking a Database” in Chapter 2, “Relational Overview”.)

XML files allow you to create elements within other elements just about anywhere you like, so they are semi-structured. This can be convenient if you’re not sure of the data’s exact format ahead of time, or if the data design changes during development. For example, you could easily add extra `To`, `Cc`, or `Bcc` elements to the previous letter example even if you didn’t realize you would need them when you wrote the original letter. (Of course, the program that prints the letter may need some modifications to use the new fields, but at least you can store valid data.) This is similar to the way NoSQL databases let you change the data’s structure as needed.

Because XML files are plain-old text files, they have some of the limitations of text files. In particular, you cannot add, delete, or modify data in the middle of an XML file. To update an XML file, a program typically reads the file into memory, makes its changes, and then writes the result back into the file.

This read-modify-write nature means XML documents are not great multiuser databases. An XML document works fine if many users need to read it, but it’s harder to allow them to update the file without interfering with each other.

Note that many versions of some other kinds of databases provide XML support. For example, Microsoft Excel workbooks can save their data in XML files. SQL Server and Oracle can execute queries to extract data and then return the result in an XML format for the program to manipulate or save into a file.

TIP You can search online to find many tutorials covering XML and its related technologies such as XSL, XSLT, XPath, XQuery, and others. For example, you can get started at the W3Schools website, www.w3schools.com/xml.

JSON Files

JavaScript Object Notation (JSON) is a standard for textual storage and interchange of information, much as XML is. Before you roll your eyes and ask if we really need another language to do what XML does, consider how verbose XML is. Even relatively simple object hierarchies can take up a considerable amount of space when represented by XML. JSON is a more compact format that stores more or less the same kinds of information in less space.

When XML first came out, I immediately thought, “This is a really verbose language. It could be so much more concise, but I guess people are willing to spend the extra space to get a more readable format. And after all, storage space is cheaper and network speed is faster than ever before.”

People soon decided that, yes, a simpler, more concise format would be nice, so *now* you can roll your eyes.

Like XML, JSON is a language for storing hierarchical data. The rules for building a JSON document are fairly simple:

- Fields hold key:value pairs, and fields are separated by commas.
- Keys must be strings surrounded by double quotes. (Sorry, Python users, you can’t use single quotes.)
- Values can be (double) quoted strings, numbers, objects, arrays, Boolean values, or null.
- An object is a group of fields enclosed in curly braces—{ }—and separated by commas. (Remember that fields are key:value pairs.)
- An array is a group of values enclosed in square brackets—[]—and separated by commas.

JSON GRAPH GOTCHAS

Like XML, JSON is mostly used to store hierarchical data. As is also the case when you use XML, some JSON tools can store more general graphs that are not hierarchical.

If you really must, you can also store node IDs and write your own code to rebuild graphs. (And as is the case when you do that in XML, it’s more work for you.)

Here’s a JSON version of the example shown earlier in this chapter:

```
{
  "title" : "The Lord of the Rings: The Fellowship of the Ring",
  "image_url" : "https://..."
}
```

```

    "release_date" : "2002-11-12",
    "rating" : "PG-13",
    "product_id "1276193: " ",
    "studio" : "New Line Home Video",
    "director" : "Peter Jackson",
    "run_time" : "P3H28M",
    "cast" : [
        "Elijah Wood",
        "Sean Astin",
        "Sean Bean",
        "Viggo Mortensen",
        ...
    ],
    "synopsis" : "Four hobbits take a vacation.",
    ...
}

```

This document’s text is surrounded by curly braces, so it contains an object. When you read this data into a program, you will probably use it to create an object in the object-oriented programming sense.

The object contains fields that hold information about the DVD. For example, the `director` is Peter Jackson and the `release_date` is 2002-11-12. The date is stored in Coordinated Universal Time (UTC) format. If your program is going to parse the data, you can use whatever format you like. I picked UTC to make the result more globally consistent (and to avoid the whole “Is 11/12/2022 November 12th or December 11th?” issue). There are also many tools that can read and write JSON files, and they generally use UTC time formats.

UTC

Coordinated Universal Time is basically the time at Greenwich England, which lies at the prime meridian (or zero meridian) at 0° longitude. It was formerly called Greenwich Mean Time (GMT) and is sometimes still called Zulu Time (which makes you sound like a submarine commander or fighter pilot).

Coordinated Universal Time would be abbreviated CUT in English and Temps Universel Coordonné would be abbreviated TUC in French, so UTC was chosen as a compromise. (This way, everyone can be annoyed, particularly those whose languages weren’t even considered like Swedish, German, and Japanese.)

Notice that the `cast` field contains an array holding cast member names. The names are values not fields, so they don’t have separate keys. They are all part of the `cast` array.

That’s the gist of JSON. (Not only are JSON files shorter than XML files, but the rules are shorter.) There are a few additional details, such as how to include line breaks inside strings (you can’t, although you can use `\n` to indicate where a new line should be) and how to represent Unicode characters (use `\uXXXX`), but the basics are pretty straightforward. For more information, visit www.json.org.

Spreadsheets

Spreadsheets display rows and columns of data. They allow the user to create formulas that depend on other data in the spreadsheet, make charts and graphs to visualize the data, print the data, and import and export the data in text and other formats such as XML and JSON. A spreadsheet may also support relatively sophisticated analysis tools such as statistical functions and iterated solution finding (basically making a bunch of guesses to see which ones work best).

Spreadsheets allow you to easily update some or all of the data, and they automatically recalculate values that depend on the data you changed.

Because many users understand spreadsheets and are comfortable with them, they can perform some of their own analysis, so you may be able to avoid some work generating a zillion different kinds of reports.

AD HOC HELP

In most of the larger projects I've worked on, we tried to build in ad hoc query tools so the users could define their own reports. That not only lets you save all the time you would have spent building dozens of reports yourself (one application had more than 100 reports), but it also keeps the users busy so they have less time to dream up gratuitous feature change requests while you're trying to implement the basic functionality.

If these are the sorts of things you need to do with your data, then using a spreadsheet may save you a lot of time and trouble building a more complicated database.

However, spreadsheets don't support complex queries. They also don't automatically check the data's integrity, so it's easy for you to enter incorrect or inconsistent values.

Some spreadsheets allow you to write scripting code that can add a lot of features such as integrity checks and complex analysis that aren't provided by the spreadsheet itself. If you're going to go to all that trouble, however, you may as well admit that you need more than the spreadsheet was intended to do and consider using a more powerful database such as a relational database.

SAVE AS SPREADSHEET

Many applications provide spreadsheet data as a form of output. They store their data in a relational or other kind of database and then dump results into a spreadsheet format for users to manipulate. Even if you don't use a spreadsheet, you could consider providing output in a comma-separated file so that users can import it into a spreadsheet.

MORE EXOTIC OPTIONS

These kinds of databases are more unusual than those described earlier. They tend to be very specialized and may only work well for a specific subset of database problems. Some are variations on other, less unusual kinds of databases.

Object

Many modern programming languages are object-oriented. They use programming abstractions called *objects* to represent items such as customers, orders, penny stocks, and betting slips.

An *object database* manages objects. It provides some sort of query syntax for retrieving objects from the database. It also provides tools for saving changes to an object back into the database.

Object databases can also provide some useful concurrency features. For example, if two users of a program need to work with an object representing a particular episode of *The Mandalorian* television show, the database gives them the same logical object. If the two users access the object simultaneously, the database referees so the users don't interfere with each other.

An object database is also sometimes called an *object-oriented database*, *object database management system (ODBMS)*, and *object store*. Some developers make a distinction among these different terms, but at this level they're close enough to the same thing, so you don't need to worry about it.

Object databases aren't really all that unusual. Many relational and NoSQL database products provide methods that a program can use to move objects in and out of the database.

Deductive

A *deductive database* is one that can make deductions based on rules and facts contained within the database. They are a sort of cross between logic programming and relational databases. Some of these databases allow the programmer to guide an evaluation performed by a program.

Dimensional

A *dimensional database* (sometimes called a *multidimensional database*) represents different aspects of data as dimensions rather than as separate tables in a relational database.

You can think of dimensional data as forming a multidimensional hyper-rectangular box (also called a hypercube or multidimensional array) where each dimension represents some important facet of the data. For example, Figure 3.4 shows a three-dimensional picture with Year, Sales Rep, and Product Line as dimensions. Each little cube or cell in the larger box contains information relating to a particular selection of the dimensions. In other words, a particular cell would contain information about a selected sales rep's sales for a selected product line in a particular year—for example, Crazy Bob's yo-yo sales for 2025.

Dimensional databases are particularly useful for scrounging through old data, looking for patterns, and they make useful data warehouses. However, if the data is sparse (a lot of the cells in Figure 3.4 are empty), they can waste a lot of space, so they are not usually appropriate for day-to-day use for new data entry.

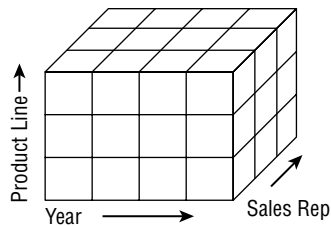


FIGURE 3.4

Much as column-oriented databases are optimized to handle sparse data, dimensional databases may also be designed to represent missing data without wasting huge amounts of space.

Temporal

A *temporal database* has built-in time information. One of the simplest pieces of temporal data that this kind of database stores is the data's *valid time*: the time during which it is valid.

For example, suppose you build an inventory and sales database for your jewelry store. To make mall visitors think they are getting a good deal, you constantly raise and lower prices. When you're ready for your bi-monthly vacation, you raise prices so that the reduced sales load won't overwhelm your brother-in-law Joey (who's *otnay ootay ightbray*) while you're gone. When you get back, you have your bi-monthly "Once-In-A-Lifetime Blockbuster Overstock Sale!" to clear inventory. To be able to later track sales at various prices over time, you need to know the times during which different prices were in effect.

If you were on vacation from April 1 until April 14 and prices were "normal," then those prices have a valid time of those two weeks. If you return and cut prices by 40 percent from April 15 until your next vacation on May 22, the new prices have a valid time of April 15 through May 22.

For other examples, imagine tracking mileage and fuel use in your fleet of rental scooters, prices of lobster sold at "market price" in a restaurant, or daily coffee prices. You could store only the latest information in each of these cases, but then you lose the ability to look back in time to search for important trends.

THE ULTIMATE HOARDER'S DATABASE

I've heard reasonably plausible arguments that a database should never delete or overwrite any information. Instead, it should just mark the old data as deleted and then create a new record for the new data. That lets you go back and comb through the data looking for information.

If you have a database that grows very quickly, you may want to occasionally purge older data or move it into a data warehouse, but with disk space costing only a few pennies per gigabyte these days, it's easy to imagine saving every piece of data for years, at least for small and medium-sized databases.

It's not too hard to add time fields to other kinds of databases such as relational or document databases. You'll need a little extra programming to keep track of which values you should use at different times, but this technique can be very useful for data that changes frequently over time.

DATABASE PROS AND CONS

It can be hard to decide whether you should use a relational or NoSQL database. In fact, with enough determination, you can probably force most data into any kind of database, just not necessarily efficiently. For example, in the “Comparing Database Types” case study in Chapter 2, “Relational Overview,” I mentioned a program that stored hierarchical data in a relational database. It worked, just not very well.

Note that vendors may address database issues differently. For example, a document-oriented database vendor could include indexing capabilities to let you search for values inside the documents much as a relational database lets you search for values in records. To understand your options, you'll need to pick some vendors and dig deeper to see what they offer. Meanwhile, the following sections summarize some of the advantages and disadvantages of different kinds of databases.

Relational

Relational databases have many handy features that make using their data easy and safe. A high-quality RDBMS has the following features:

- It provides a relatively simple and intuitive model for you to understand the data.
- Uniqueness constraints and referential integrity protect the data.
- You can join multiple tables to build complex reports.
- Privileges on tables, fields, and views allow users to access only the data that they need.
- Backup and restore features are relatively easy.
- Optimized indexes make searching and joins extremely fast.
- Stored procedures let you move some code out of compiled programs and into the database.
- A normalized database automatically prevents an assortment of data anomalies (aka errors) that you'll learn about in later chapters.
- Atomic transactions let you commit or roll back data operations.
- Placing constraints inside the database means you can use many programs on the same data without worrying that one of them will corrupt the data.

Relational databases provide some wonderful features, as long as the data fits naturally into a table-like format.

Sometimes, however, those features are unnecessary or even a hindrance. For example, suppose you want to store survey results where every item is optional and respondents can enter free-formatted comments about various issues. In that case, a relational database won't provide any help beyond

what's available in a flat file. For example, it can't link survey fields to other tables, can't validate entries, and can't enforce uniqueness constraints.

In fact, if the survey is anonymous so there are no identifying fields, then two responses might have exactly the same values. In that case, the response "table" has no superkeys, so technically it's not even a relation in the theoretical sense. You could fix that by adding an artificial ID to each record, but that should be a hint that perhaps this data doesn't fit the relational model very well.

Because relational databases are designed to hold data that fits more or less naturally into tables, they also sometimes have problems with data that has a different shape. They don't naturally store hierarchical data or graph data. You can certainly cram those kinds of data into a relational database, but you won't get much benefit from a relational approach.

Relational databases also have a few more general problems such as the following:

- Maintenance time and cost tend to increase over time if the database grows extremely large.
- Storage size can also increase over time, making it hard to fit enough disk space on a single server in extreme cases.
- Changing the database's structure (and then transforming the data to fit the new structure) can be difficult.
- Scaling up can be hard if demands increase too much too quickly.
- Performance can decrease if the user load increases greatly.

Relational databases have been around for a long time, so there are ways to address each of those problems. But they complicate database management, particularly if the database gets too big or has too many simultaneous users.

If your data fits naturally into tables, then by all means use a relational database. It is a relatively mature technology with many benefits. However, if your data doesn't fit well in tables, you don't need relational features, and you don't want to enforce strict data formats, then you should consider the NoSQL options described in the next section.

General NoSQL

Different types of NoSQL databases have different strengths and weaknesses, but the most common types have the following advantages in common:

- They can scale relatively easily and can handle huge amounts of data.
- Distributing data can improve performance for certain kinds of operations.
- They can handle unstructured, semi-structured, or structured data.
- They allow you to easily change the database's structure, and you may not even need to translate the data into the new format.
- They're generally easy for developers to use. Being able to easily change the database's structure makes it easier to modify during agile projects.
- They are easy to distribute across a network, which can decrease downtime.

SCHEMAS

A database *schema* defines the required structure of the data. In a relational database, the schema is very well-defined and specifies data types, field-level constraints, relational constraints, and other properties on every table and field.

Nonrelational databases may specify little or none of the data's structure. That makes it easier to change the database design as a project progresses, although it also means the database cannot validate the data as thoroughly. It can't enforce rules that are not defined.

Disadvantages of nonrelational databases include all the things that make relational databases useful, such as the features in the following list:

- Not requiring a particular data format gives you more freedom but prevents the database from validating the data.
- Distributing data across the network can decrease downtime but makes it harder to synchronize the different pieces of the database so they are consistent.
- Distributing data across the network makes backups more complicated.
- Many NoSQL databases provide no transactions or very simple ones.
- NoSQL databases are newer and less standardized. That means different databases may have different query syntaxes. In contrast, relational databases have been around since the 1970s, so SQL is fairly standard for all RDBMSs.

If you have data that naturally fits a particular kind of NoSQL database such as a graph database, use it. If your data can be easily partitioned, a NoSQL database may be just what you need, particularly if you expect an enormous amount of data or a huge user load.

Quick Guidelines

The following list summarizes features that work well with the various database types. In general, if something isn't listed, then that database type isn't good at it. For example, relational databases aren't good at storing hierarchical data, so that isn't listed for them.

Relational databases work well for:

- Data validation
- Protections against data anomalies
- Complicated queries and joins among different tables
- Flexible ad hoc queries

Document databases work well for:

- Objects that are self-contained so you can load and save them separately
- Documents that have different formats
- Cloud storage
- Massive scaling (terabytes)

Key-value databases work well for:

- Fast in-memory caching shared between applications
- Cloud storage

Column-oriented databases work well for:

- Data that acts like a single enormous table with records that might have different columns
- Selecting data by columns
- Operations that use only one or a few columns
- Cloud storage
- Massive scaling (terabytes)

Graph databases work well for:

- Graph, network, or hierarchical data
- Applications that are more interested in relationships among objects than in the properties of the objects
- Network algorithms such as shortest path finding or maximal flow calculations

Flat files work well for:

- Small and simple values
- Values that don't change often
- Data that you might want to edit with a text editor
- Data that you want to distribute by copying files to new locations
- Situations where you want to use copies of old files as a simple historical list of previous values

XML files work well for:

- Hierarchical data
- Applications that can use existing XML tools such as XPath and XSLT
- Data where schema validation is sufficient

- Applications that import from and export to products that understand XML
- Data that you might want to edit with a text editor
- Data that you want to distribute by copying files to new locations

JSON files work well for:

- Hierarchical data
- Applications that can use existing JSON tools such as JSON Tree Viewer and JSON Minify
- Data where schema validation is sufficient
- Applications that import from and export to products that understand JSON
- Data that you might want to edit with a text editor
- Data that you want to distribute by copying files to new locations

Spreadsheets work well for:

- Data that doesn't require much validation
- Table-like data
- Tables and graphs
- End users who are comfortable with spreadsheets
- End users who want to experiment with the data on their own

Object databases work well for:

- Object-oriented programs

SUMMARY

Before you start building a database, you should take at least a few minutes to decide what kind of database would best fit your needs. Though you can probably use any kind of database for most purposes, some lend themselves more naturally to certain problems than others. You can store a data hierarchy or network in a relational database, but it may be a lot faster to use a graph database or a simple XML file. You certainly can store simple configuration settings in an object-oriented database, but a flat text file (possibly in XML or JSON) will do just as well with a lot less hassle and possibly less expense.

In this chapter, you learned how to pick the database type that will work best with your data. You learned that

- “NoSQL” can mean “not SQL” or “not only SQL.” Often, it means “not only SQL” when you're talking about general strategies and “not SQL” when you're talking about a specific type of NoSQL database.
- You can use all sorts of databases in the cloud.

- Cloud databases may either support strong or eventual consistency.
- Relational databases allow complex data relations, sophisticated data validation, integrity constraints, cascading updates and deletes, ad hoc queries, and many more useful features.
- NoSQL databases may support only simple transactions.
- Different kinds of NoSQL databases store specialized kinds of data:
 - Document databases store documents, usually in XML or JSON format.
 - Key-value databases associate keys with values. They are sometimes used as in-memory caches.
 - Column-oriented databases store records that may have many columns. Different records may use different columns.
 - Graph databases store nodes connected with relationships.
 - Hierarchical databases are good for storing and manipulating hierarchical data such as org charts and family trees. Hierarchical data is a subset of graph data, so you can store it in a graph database.
- Flat files are good for storing simple values or complete documents, although they lack features for concurrency and easy updating.
- Spreadsheets are good for drawing charts and graphs, and are convenient for users who already know how to use them.
- XML files are good for storing hierarchical data, although they lack features for concurrency and easy updating.
- JSON files are good for storing hierarchical data, although they lack features for concurrency and easy updating.
- Object databases are good for integrating programming objects into the database.
- Deductive databases can make logical deductions based on rules and data stored in the database.
- Dimensional databases consider data in hypercubes and make it easy to study data based on dimensional selections (such as sales by a particular representative or during a particular year).
- Temporal databases include time with the data so they can record and work with information that changes over time.

The following chapters change the book's focus from general database concepts and terminology to design techniques. They describe the tasks you must perform to design a database from scratch. Chapter 4 starts the process by explaining how to gather user requirements so the database you design has a good chance of actually satisfying the users' needs.

Before you move on to Chapter 4, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to selected exercises in Appendix A.

EXERCISES

For the following scenarios, list the type(s) of databases that might make good choices for storing the data.

1. A dog breeding database that records the ancestors of a single dog for five generations.
2. A similar dog breeding database that records the ancestors and descendants of a single dog for five generations each way.
3. Application settings that record which windows a user had open and where they were positioned the last time the application was used.
4. Total sales figures by month, arranged to make it easy to see trends graphically.
5. The same as Exercise 4, but the users want to be able to draw similar data for several product lines on the same graph.
6. The users also want to be able to print it and tweak the numbers to see what the graph would look like if they exceed expectations next quarter. (We'll call this the WTR — Wishful Thinking Report.)
7. A map showing the main vessels and arteries leaving the human heart.
8. A very large amount of sales data, including information about customers, orders, inventory items, and sales representatives. You need to be able to perform ad hoc queries.
9. The same as Exercise 8, except your manager just returned from a technical seminar where they learned the phrase “object-oriented” and now they're determined to use object-oriented techniques in everything.
10. A simple recipe book. You should be able to find recipes by name, part of meal (entrée, aperitif, dessert, and so forth), or main ingredient.
11. A *Magic: The Gathering* game and trading card tracking system. You need to be able to sort cards by their monetary value, number of duplicates, and power in the game. You also want to be able to define “power decks” for competition.
12. A movie, television, book, and musical search tool. You want to be able to search all three types of media by title, author, star rating (how good you thought it was), Motion Picture Association of America (MPAA) rating (G, PG, PG-13, and so forth), actor, or director. And perhaps studio. And genre. And group titles by studio or publisher. And anything else you might think of later. For example, if you enter **John Lithgow**, the program should return a (very long) list of movies, television shows, books, and musicals that feature John Lithgow.

13. A database to hold statistics for your favorite sports teams: football, baseball, polo, hurling, or whatever. You need to be able to find all the players on a particular team, find players with the best stats (most yards rushing, highest hitting percentage, most chuckers chucked, most . . . uh . . . hurls hurled?).

14. An inspirational message of the day database. Every day when the application starts, it should display a random message selected from the database.

PART 2

Database Design Process and Techniques

- ▶ **Chapter 4:** Understanding User Needs
- ▶ **Chapter 5:** Translating User Needs into Data Models
- ▶ **Chapter 6:** Extracting Business Rules
- ▶ **Chapter 7:** Normalizing Data
- ▶ **Chapter 8:** Designing Databases to Support Software
- ▶ **Chapter 9:** Using Common Design Patterns
- ▶ **Chapter 10:** Avoiding Common Design Pitfalls

The chapters in this part of the book contain the bulk of the information about the database design process. They discuss the major steps in database design, starting from the beginning of a project and working through various design and refinement stages to provide a fully functional database.

Chapter 4 explains how you can learn about the customers' needs. If you don't understand your customers' needs, how can you possibly build a database that satisfies them? This chapter also explains how to ensure that the customers agree with you on what the database should do so that everyone is happy with the final result.

Chapter 5 shows how to translate the customer needs (defined in Chapter 4) into several different kinds of data models. These data models allow you to formally represent the database's needs. They let you study the data and rearrange the pieces in order to build toward a flexible design.

Chapter 6 explains how to identify business rules. It tells how to modify the data models that you developed in Chapter 5 to make it easier to manage business rules, which may change relatively frequently.

Chapter 7 describes one of the best-known steps in designing a relational database: normalization. A properly normalized database is robust and more resistant to certain kinds of potential data errors than a non-normalized database.

Chapter 8 discusses some database design issues that affect the database's use and maintenance in a larger application. The techniques described in this chapter make it easier for application developers to build an effective, flexible user interface for the database.

Chapter 9 describes some patterns that occur in many applications. It then explains solutions that you can use to make handling those patterns simple.

Chapter 10 describes some pitfalls that often hinder database designers. Avoiding these problems can make the database design process easy, fast, and effective.

After you finish working through these chapters, you will have a good understanding of database design. You will know how to decide what data belongs in a database, how to build data models, and how to convert those models into powerful, flexible database designs.

4

Understanding User Needs

The previous chapters discussed databases in general terms. Chapter 1, “Database Design Goals,” explained the goals of database design and described some of the types of databases that are available. Chapter 2, “Relational Overview,” and Chapter 3, “NoSQL Overview,” described some specific kinds of databases in slightly greater detail. With this basic understanding, you’re ready to take the first step in designing an actual database to solve a particular problem: understanding the user’s needs.

Designing any custom product, whether it’s a database, beach house, or case mod (you can search online for images of case mods), is largely a translation process. You need to translate the customers’ needs, wants, and desires from the sometimes fuzzy ideas floating around in their heads into a product that meets their needs.

The first step in the translation process is understanding the user’s requirements. Unless you know what the user needs, you cannot build it. Designing the best order-processing database imaginable won’t do you a bit of good if the customer really wants a circuit design database or an ostrich race handicapping system.

Just as the database design forms the foundation on which the rest of the application’s development stands, your understanding of the user’s needs forms the foundation of the database design. If you don’t know what the user needs, how can you possibly design it?

If you don’t understand the customer’s needs thoroughly and completely, you may as well pack it in now. There’s little satisfaction in wasting months of your life and a pile of your company’s money to build something unusable. Make sure you’re on the right road before you stomp on the accelerator and burn rubber down a dead-end alley.

This chapter explains techniques that you can use to learn about the customer’s needs. It describes methods that you can use to record those needs in a concrete and verifiable way.

The sections that follow describe some of the steps that you can take to better understand your customer’s needs. In some projects, you may not need to follow all of these steps. For example, if your customer is a single person with very concrete ideas about what needs to be done, then

you may not have to spend much time learning who's who or brainstorming. If your customer works with government classified data, you may not be allowed to "walk a mile in the user's shoes," and you may have access to only some of the business's documentation, so there's no point in trying.

WORKING BLINDFOLDED

I once knew a developer who was working on a classified project. He had clearance to see the source code but not the data, so every week his customer brought him a giant printout of the latest run with all the data carefully clipped out with scissors. He would try to guess what was going on and make some suggestions so that the customer's developers could try to fix the code. The cycle then repeated the next week. What an odd way to work!

In other projects, the steps may work best in a different order. You might find it better to brainstorm before visiting the customers' site and watching them work.

These are just some steps that I've found useful in trying to understand a customer's situation. You'll have to adjust them as necessary to fit each of your projects.

In this chapter, you learn how to:

- Understand the customers' needs and motivations.
- Gather and document user requirements.
- Cull requirements from existing practices and information.
- Build use cases to understand the user's needs and to measure success or failure.
- Anticipate changes and future needs to build the most flexible database possible.

After you master these techniques, you'll be ready to move on to the next step and actually start designing the database.

MAKE A PLAN

Although the steps described in this chapter sometimes occur in different order, the following list summarizes the order that's most typical. Feel free to add, remove, and rearrange them as necessary.

1. Bring a list of questions.
2. Meet the customers.
3. Learn who's who.
4. Pick the customers' brains.
5. Walk a mile in the user's shoes.
6. Study current operations.
7. Brainstorm.

8. Look to the future.
9. Understand the customers' reasoning.
10. Learn what the customers really need.
11. Prioritize.
12. Verify your understanding.
13. Create the requirements document.
14. Make use cases.
15. Decide feasibility.

This list isn't perfect but it makes a good meta-plan—a plan for making the project's plan. (Hopefully, it won't be as useless as the traditional pre-meeting agenda planning meeting.)

BRING A LIST OF QUESTIONS

From the very first day, you should start thinking of questions to ask the customers to get a better idea of the project's goals and scope.

The following sections list some questions that you can ask your customers to help understand their needs. You'll see many of them described in greater detail later in this chapter.

This list is by no means complete—the questions that you need to ask will depend to a large extent on the type of project. Use them only as a starting point. It's helpful to have something to work from when you start, however. Then you can wander off in promising directions as the discussions continue.

Functionality

These questions deal with what the system is supposed to accomplish and, to a lesser extent, how. It is usually best to avoid deciding how the system should do anything until you thoroughly understand what it should do so you don't become locked into one idea too early, but it's still useful to record any impressions the customers have of how the system should work.

- Who are the players?
- What should the system do?
- Why are you building this system? What do you hope it will accomplish?
- What should it look like? (Sketch out the user interface.)
- What response times do you need for different parts of the system? (Typically, interactive response times should be under 5 seconds, whereas reports and other offline activities may take longer.)
- What reports are needed?
- Do the end users need to be able to define new reports?
- Do power users and administrators need to be able to define new reports?

Data Needs

These questions help clarify the project's data needs. Knowing what data is required will help you start defining the database's tables.

- What data is needed for the user interface?
- Where should that data come from?
- How are those pieces of data related?
- How are these tasks handled today? Where does the data come from?

Data Integrity

These questions deal with data integrity. They help you define some of the integrity constraints that you will build into the database.

- What values are allowed in which fields?
- Which fields are required? (For example, does a customer record need an email address? A phone number? A fax number? A pager number? At least one of those but not all of them?)
- What are the valid domains (i.e., allowed values) for various fields? What phone number formats are allowed? What are allowed birth years? How long can customer names be? Addresses? Do addresses need extra lines for suite or apartment number? Do addresses need to handle U.S. ZIP Codes such as 12345? ZIP+4 Codes such as 12345-6789? Canadian postal codes such as T1A 6G9? Or other countries' postal codes?
- Which fields should refer to foreign keys? (For example, an address's State field might need to be in the States table and a CustomerID field might need to be in the Customers table. I've even seen customers with a big list of standard comments, and a Comments field can only take those values.)
- Should the system validate cities against postal codes? For example, should it verify that the 10005 ZIP Code is in New York City, New York? (You may be able to use the web-based tools described at www.usps.com/business/web-tools-apis/address-information-api.htm. Then you can avoid a lot of work, let someone else maintain the ZIP Code data, and generally look smart.)
- Do you need a customer record before you can place orders?
- If a customer cancels an account, do you want to delete the corresponding records or just flag them as inactive?
- What level of system reliability is needed?
 - Does the system need 24/7 access?
 - How volatile is the data? How often does it need to be backed up?
 - How disastrous will it be if the system crashes?

- How quickly do you have to be back up and running?
- How painful will it be if you lose some data during a crash?

Security

These questions focus on the application's security. The answers to these questions will help you decide which database product will work best (different products provide different forms of security) and what architecture to use.

- Does each user need a separate password? (Generally a good idea.)
- Do different users need access to different pieces of data? (For example, sales clerks might need to access customer credit card numbers, but order fulfillment technicians probably don't.)
- Does the data need to be encrypted within the database?
- Do you need to provide audit trails recording every action taken and by whom? (For example, you can see which clerk increased the priority of a customer who was ordering the latest iPod, and then ask that clerk why that happened.)
- What different classes of users will there be?

NOTE *I often use three classes of users. First, clerks do most of the regular work. They enter orders, print invoices, discuss the latest Wait Wait ... Don't Tell Me! episode around the water cooler, and so forth.*

Second, supervisors can do anything that clerks can and they also perform managerial tasks. They can view reports, logs, and audit trails; assign clerks to tasks; grant bonuses; and the like.

Third, superusers or key users can do everything. They can reset user passwords; go directly into database tables to fix problems; change system parameters, such as the values that users can pick from drop-downs; and so forth. There should only be a couple of superusers and, for day-to-day tasks, they should log in as supervisors rather than superusers, to prevent accidental catastrophes.

- How many of each class of user will there be? Will only one person need access to the data at a time? Will there be hundreds or even thousands (as is the case with some web applications)?
- Is there existing documentation describing the users' tasks and responsibilities?
- What kinds of security features does the database need? Do you have to work with a data security consulting firm?
- Do you need to provide multifactor authentication (MFA) so the application sends you a one-time code via text, voice message, or some other method whenever you log in?

- Does the data include sensitive information, such as medical, financial, or other personal information?
- Is the data subject to data sovereignty laws?
- Are there other legal considerations? Does your customer have agreements or guarantees (express or implied) with *their* customers regarding the data?

Environment

These questions deal with the project's surrounding environment. They gather information about other systems and processes that the project will replace or with which it will interact.

- Does this system enhance or replace an existing system?
 - Is there documentation describing the existing system?
 - Does the existing system have paper forms that you can study?
 - What features in the existing system are required? Which are not?
 - What features in the existing system do the users like? Which do they dislike?
 - What kinds of data does the existing system use? How is it stored? How are different pieces of data related?
 - Is there documentation for the existing system's data?
- Are there other systems with which this one must interact?
 - Exactly how will it interact with them?
 - Will the new project send data to existing systems? How?
 - Will the new project receive data from existing systems? How?
 - Is there documentation for those systems?
- How does your business work? (Try to understand how this project fits into the bigger picture.)

MEET THE CUSTOMERS

Before you can start any project, you need to know what it is about. Are you building an inventory system, a supply chain model, or a stock price tracker and predictor (also called a random number generator)?

The best way to understand the system that you need to design and build is to interrogate the customers. I use the rather unfriendly word “interrogate” because, to do the job right, you need much more than a simple chat over tea and crumpets. Learning about the customers' requirements can be a long and grueling process. It can take days or even weeks of cross-examination; studying existing practices; poring over dusty scrolls, tomes, and other corporate documentation; and spying on the customers while they do their daily jobs.

When it's over, the customers shouldn't hate you outright, but they might wish you would go away and leave them alone for a while. A good question and answer session should leave everyone feeling exhausted but with the warm glow of satisfaction that comes from moving a lot of information from their brains to yours.

Customers who are truly dedicated to the company are usually willing to field even the most obtuse questions as long as you're willing to dish them out. Former Prime Minister of the United Kingdom Benjamin Disraeli once said, "Talk to a man about himself, and he will listen for hours." Most customers are more than happy to share the ins and outs of their corner of the business universe with you for as long as you can stand it.

NOTE *It may sound boring listening to customers drone on about their supply chains, but I've found that once you dig deeply enough, almost any business can be pretty interesting. I've worked on projects spanning such topics as fuel tax collection, wastewater treatment, ticket sales, and school enrollment. Every time, after I learned enough, I discovered hidden complexity that I would never have imagined.*

The goal isn't to torture the customers (although it may sometimes seem like it to them) but to give you an absolute and complete understanding of the problem that you're attempting to solve. You want as few surprises as possible after you're done researching the problem. Unexpected difficulties and feature requests are the biggest reasons why software projects finish late, come in over budget, or fail completely.

The sooner you identify a system's features and the more completely you anticipate potential problems, the easier it will be for you to plan for them and the less they will mess up your meticulously crafted plan. Your initial encounters with the customer give you your first chance to address these issues so that they don't bite you later.

So when you first start a project, meet the customers. Get to know them and what they do. Even if the problem you are trying to solve is only a small part of their business, get a feel for the overall picture. Sometimes, you'll find unexpected connections that may make your job easier or that may lead to surprising benefits in a completely unrelated area.

When you first meet the customers, it usually doesn't hurt to warn them that you're going to be a major pest for a while. This can also help you figure out who's who. Those who are committed to the project and are eager to succeed will take your warning well. Those who are less than dedicated may tip their hand at this point. This idea leads naturally to the next section.

LEARN WHO'S WHO

Ideally, a project team works well together, everyone does the best possible job without conflict, and the project moves along smoothly to create a finished product that meets the customers' needs. In practice, however, it doesn't always work out that way. Like the bickering superheroes in an X-Men

movie, everyone has their own personal abilities, agenda, and motivation that don't always coincide with those of the other team members.

As you get to know the customers (and your team members), it's important to realize that not everyone shares the same vision of the product. You need to figure out what customer is the leader, who are team players, who have little or no say in specifying the project, and who will be supervillains.

WARNING *No one wants a supervillain on their project, but you should be aware that they exist. I've worked on projects where customers ostracized members of the project team, tried to delete all of the project files, spread dark rumors among senior management, and even slashed tires. Hopefully, you won't encounter any of these types, but it's best that you identify these malicious people as early as possible if they are present.*

The following list describes some of the roles that customers (and developers) often play in a project. Naturally, these cannot categorize everyone, but they define some characteristics that you should look for.

Executive Champion—This is the highest-ranking customer driving the project. Often, this person doesn't participate in the project's day-to-day trials and tribulations. The Executive Champion will fight for truth, justice, and getting you that extra laptop you need. In the end, the Executive Champion must be able to take on any supervillains, or you might be in trouble.

Customer Champion—This person has a thorough understanding of the customers' needs. Lesser champions may help define pieces of the project, but this is the person you run to when the others are unclear. For the purposes of this chapter, this is the most important person on the project. This person must have enough time and resources (also known as "people") to help you define the project and answer your questions. Ideally, they also have enough clout to make decisions when the heroes start bickering over who has to fight Magneto and who gets to fly the invisible plane.

Customer Representative—A Customer Representative is someone assigned to answer your questions and help define the project. Often, they are people who do the day-to-day work of your customers' business. Sometimes, they are experts in only parts of the business, so you need more than one around to cover all of the business's issues.

Stakeholder—This is anyone who has an interest in the project. Some of these folks fall into other categories like Customer Champion or Customer Representative. Others are affected by the outcome but have no direct say in the system's design. For example, frontline clerks rarely get to toss in their two cents when you design a point-of-sales system, and sales and marketing people may eventually need to sell your system (although sales and marketing people are usually prohibited from knowing anything about the systems they sell—I think there's a law or something). Stakeholders are like the civilians whose fate is determined by the battling superheroes and who are easily crushed by falling debris and exploding robot monsters. Although many of them have no direct power over the outcome, you should keep them in mind and try to minimize collateral

damage. (In a really well-run company, these people have their own representatives on the project to watch out for them.)

Sidekick/Gopher—This is someone who can help you get the more mundane resources that you need, such as conference rooms, airline tickets, donuts, and kryptonite. Although this isn't a glamorous job, an effective Sidekick can make everything run more smoothly. (Sometimes, they also provide comic relief. On one project, the Sidekick invited everyone out to a huge celebratory lunch, his treat, only to find that the restaurant didn't take credit cards, so we all had to chip in. In all fairness, though, that could have happened to any of us.)

Short-Timer—This is someone who is only going to be around for a short while. This may be someone who is about to be promoted to a new division, who will retire soon, or who is just plain fed up and about to walk out. A dedicated short-timer can be a huge asset, particularly those who are about to retire and take a lifetime's worth of experience with them. Others don't care all that much whether the project succeeds or fails after they're gone. (These are like the red-shirts on *Star Trek* who don't contribute much. When Kirk says, "Spock, Bones, and Smith, meet me in the transporter room," guess who isn't coming back?)

Devil's Advocate—This is an important role for avoiding groupthink. Left unchecked, some groups become irrationally optimistic and can make extremely poor decisions. A Devil's Advocate can help bring the hopeless dreamers back to Earth and keep the project realistic . . . as long as the Devil's Advocate doesn't get out of hand. The purpose of the Devil's Advocate is to maintain a reality check, not to defeat the entire project. If this person shoots down every idea anyone comes up with, you might gently mention that eventually you need to decide on an approach and get something done or you may all need to update your résumés.

Convert—This is someone who originally is against the project but who you convert to your cause. Strangely, both finding and converting this person is usually surprisingly easy, at least for bigger projects. If you talk to the disenfranchised stakeholders (the frontline users who have no say in the matter), you can usually find some who are dead set against the project, if for no other reason than it represents change. Take one or two of these people who have a fair amount of experience and make them Stakeholder Representatives. Get them involved early in the process and take their suggestions seriously. If you act on some of their suggestions, you'll show that you have the Stakeholders' interests in mind and you'll win their loyalty. They'll tell the rest of the Stakeholders, and if all goes well, you'll have more support than you can imagine. And who knows, you may build a better product with their input.

Generic Villain—These range from simple defeatists and layabouts to Arch Supervillains actively trying to sabotage the project. Try to identify these people early so you know what you're up against. (On one project, we had a supervillain at the vice president level. We also had an Executive Champion at the same level, so we were able to hold our own, but it was pretty tough going. It's easy to get squashed when such heavy-hitters collide.)

Don't feel constrained by this list. These are just some of the characters that I've encountered, and you may meet others.

I don't mean to imply that every project is subject to continual harassment, interference, and sabotage. I've worked on plenty of projects where everyone really was pulling for the common good

and we achieved impressive results. Just keep your eyes open. Identify the main players as quickly as possible so you know who to ask questions and where to run when the fighting erupts.

Know The Players

If you're familiar with the Dilbert comic strip, then you should think about the main characters Dilbert, Alice, Wally, Asok the Intern, and the Pointy-Haired Boss. (If you're not familiar with Dilbert, I suggest that you look it up online. This should be required reading.) Assume they are your customers and you need to design them a database.

Who will play which customer roles? In particular, who will be:

- Executive Champion
- Customer Representative
- Sidekick
- Villain

What are your chances for success?

How it works

Unfortunately, in Dilbert, the only candidate for Executive Champion is the Pointy-Haired Boss. He is incompetent (but doesn't know it), is a bad manager, and is unable to defend against any attacks from villains, so you're in trouble from the start.

Alice and Dilbert generally know what's going on and try to do the right thing. They will be your best bets for Customer Representatives.

Asok means well and is competent but he's new to the company and doesn't know how everything works, so he won't be the best Customer Representative. He might make a good Sidekick, however.

Wally is a serious layabout. He actively seeks to avoid work even if doing the work would be easier. He's a villain, although on a minor scale. He won't destroy the project single-handedly, but he may waste other people's time.

Your overall chances depend entirely on whether the project will face outside attack. If any serious villain appears, the Pointy-Haired Boss will collapse like papier-mâché armor in a joust and the project will fail.

If no one else is interested in taking over or ruining the project, you might have a chance to finish before the Pointy-Haired Boss plays too active a role and messes everything up. (But then again, how long do things run without interference in a Dilbert cartoon?)

PICK THE CUSTOMERS' BRAINS

Once you figure out more or less who the movers and shakers are, you can start picking their brains. Sit down with the Customer Champion and Customer Representatives and find out what the customers think they need. Find out what they think the solution should look like. Find out what data they think it should contain, how that data will be presented, and how different parts of the data are related.

Get input from as many Stakeholders as you can. Always keep in mind, however, that the Customer Champion is the one who understands the customers' needs thoroughly and has the authority to make the final decisions. While you should consider everyone's opinions, the Customer Champion has the final word.

Depending on the scope of the project, this can take a while. I've been on projects where the initial brain-picking sessions took only a few hours, and I've been on others where we spent more than a week talking to the customers. One project was so complex that part of the project was still defining requirements after other parts of the project had been underway for months.

Take your time and make sure the customers have finished telling you what they think they need.

WALK A MILE IN THE USER'S SHOES

Often following the customers' day-to-day operations can give you some extremely helpful perspective. Ideally, you could do the customers' jobs for them for a while to thoroughly learn what's involved. Unless you are in your customers' industry (and if you are, why are they hiring you?), however, you probably aren't qualified to do their jobs.

While you may not be able to actually do the customers' jobs, you may be able to sit next to them while they do it. Warn them that you will probably reduce productivity slightly by asking stupid and annoying questions. Then ask away. Take notes and learn as much as you can. Sometimes, your outsider's point of view can lead to ideas that the customers would never have discovered.

ANOTHER POINT OF VIEW

Remember the billing center I mentioned in Chapter 1? The one that printed out a 3-foot-tall pile of paper every three days listing all the accounts that owed money?

Because of our outsider computer nerd viewpoint, we knew there was a better approach. We installed a printer emulator (a program that looks like a printer to the system but actually captures the data instead of killing trees with it) and dumped the data into a file. We then sorted the file by account balance and displayed the result to the user.

We were actually there looking at a completely different problem, but when we saw this one we jumped all over it and in about a week we were heroes. (The other project turned out well, too, but was more complicated and took much longer.)

Take notes while you're watching the customers do their jobs. Draw pictures and diagrams if that helps you visualize what they're doing. Pictures can also be very helpful in asking the customers if you have the right idea. If the customers will let you, print screenshots and even take photographs. (However, keep in mind that many businesses are required to safeguard the privacy of their clients' data, so don't expect them to let you walk out with screenshots or photographs showing credit card information, medical histories, or records of political contributions. Be sure you ask before you try to take any material away and ask before you even bring a cell-phone camera into the building.)

STUDY CURRENT OPERATIONS

After you've walked a mile or two in the customers' shoes, see if there are other ways that you can study the current operation. Often, companies have procedure manuals and documentation that describes the customers' roles and responsibilities. In fact, that kind of documentation is required for certain kinds of International Organization for Standardization (ISO) certifications. Some bigger companies like to display huge banners that say things like "ISO-9000 Certified." These may just be there to cover holes in the wall, but if they have such a banner then they probably have more documentation than you can stomach.

Make sure the documentation is up-to-date and that the customers' practices actually match the documentation. If they differ, find out which version of reality your database should support.

Look around for any existing databases that the customers use. Don't forget the lesson of the earlier chapters that there are many different kinds of databases. Don't just look for relational databases. Look also for note files, filing cabinets, boxes of index cards, tickler files (cubbies where customers place items that should be examined on a certain date), and so forth. Generally, snoop around and find out what information is kept where.

Figure out how that information is used and how it relates to other pieces of information. Different physical databases often contain redundant information and that forms a relationship. For example, a filing cabinet holding information about customers includes all of the customers' data. A pile of invoices also includes the customers' names, addresses, ID numbers, and other information that is duplicated in the customer files. Paper orders probably contain the same information. These are the sorts of pieces of data that tie the whole process together.

BRAINSTORM

At this point, you should have a decent understanding of the customers' business and needs. To make sure that the customer hasn't left anything out, you can hold brainstorming sessions. Bring in as many Stakeholders as you can and let them run wild. Don't rule out anything just yet. If a Stakeholder says the database should record the color of customers' shoes when they make a purchase, write it down. If someone else says they need to track the number of kumquats eaten by assembly line workers, write it down.

Continue brainstorming until everyone has had their say and it's clear that no new ideas are appearing.

Occasionally, extra creative people (sometimes known to management as “troublemakers”) look like they’re going to go on forever. Let them speak for a while, but if it’s clear that they really can’t stem the flood of ideas, split up. Have everyone go off separately and write down anything else relevant that they can think of, then come back and dump all of the ideas in a big pile.

Try not to let the Customer Champion suppress the group’s creativity too early. Although the Customer Champion has the final say, the goal right now is to gather ideas, not to decide which ones are the best.

The goal at this point isn’t to accept or eliminate anything as much as it is to write everything down. You want to be sure that everything relevant is considered before you start designing. Later, when you’ve started laying out tables and indexes and changes are more difficult to make, you don’t want someone to step in and say, “Owl voltages! Why didn’t someone think of owl voltages?” Hopefully you have owl voltages written down somewhere and crossed out so you can say they were considered and everyone agreed they were not a priority.

NOTE *Different development shops take different approaches if the earth-shatteringly important owl voltage requirement somehow got missed during brainstorming. I prefer to grudgingly add it to the requirements, while ensuring that the customers understand that this sort of last-minute change might affect the schedule. If you grumble a little, they usually take the hint and only insist on changes that really are important.*

Other shops simply say, “Sorry, that wasn’t in the original requirements and we’re not doing it, so there!” Although this is technically correct, it increases the chances that the final product won’t meet the customers’ needs.

LOOK TO THE FUTURE

During the brainstorming process, think about future needs. Explicitly ask the customers what they might like to have in future releases. You may be able to include some of those ideas in the current project, but even if you can’t it’s nice to know where things are headed. It will help you design your database flexibly so that you can more easily incorporate changes in the future. (It also shows that you care.)

For example, suppose your customer Paula Marble runs a plumbing supply shop but thinks that someday it might be nice to add a little café and call the whole thing “Paula’s Plumbing and Pastries.” After you hide your snickers behind a cough, think about how this might affect the database and the rest of the project.

Plumbing supplies are generally nonperishable, but pastries must be baked fresh daily and some of the ingredients that go into pastries are extremely perishable. You may want to think about using separate inventory tables to hold information about nonperishable plumbing items that clients can purchase (for example, gaskets, thread tape, pipe wrenches, and so forth) and perishable cooking items that the clients won’t buy directly (like flour, eggs, raspberries, and so on).

You might not even track quantity in stock for finished pastries (the clients either see them in the case or not), but you probably want to be able to record prices for them nonetheless. In that case, you will have entries in an inventory table that will contain prices but that will never hold quantities.

You don't necessarily need to start planning the future database just yet (after all, Paula may decide to name the business "Paula's Plumbing and Tattoo Palace" instead), but you can keep these future changes in mind as you study the rest of the problem.

UNDERSTAND THE CUSTOMERS' REASONING

Occasionally, you'll come across customers who think they know something about database design. They may say that you should use a particular table structure, an object-relational hierarchical data model, or an acute polar space modulator.

Sometimes, these suggestions make perfect sense. Other times, you'll think the customer clicked the Google "I'm Feeling Lucky" link and stumbled into an endless morass of techno-babble and database conspiracy theories.

Even if the suggestions seem to make no sense whatsoever, don't dismiss them out of hand. Remember that customers have a different perspective than you do. They know a lot more than you do about their particular business. They may or may not know anything about database design, but it's entirely possible that they have a reason for their obscure requests.

For example, suppose you're trying to design a sales and inventory system for Thor's Thimbles. The president and CEO, Thor, says he thinks you need to use a temporal database, although the way he pronounces it makes you think he probably doesn't understand what that means (or perhaps it's just his Scandinavian accent). You think, "How hard can it be to sell thimbles?" and ignore him.

After you spend a month building a really slick relational database, you discover that old Thor isn't so naive after all. It turns out that the company sells hundreds of different models of thimbles made from such materials as stainless steel, anodized aluminum, gold, and platinum. The value of the more exotic models changes daily with precious metal prices. Almost as volatile are the collectors' models, such as the Great Scientists of History series and the Sports Immortals (the Pete Rose Hall of Fame model can bring up to \$200 at auction).

Suddenly what you thought was a simple problem really does have hundreds of variables changing rapidly over time, and you realize that you probably should have built a temporal database. You have egg on your face, and Thor decides that his brother-in-law, who originally suggested the temporal database to Thor, might be able to do a better job than you.

Even if a customer's suggestion seems odd, take it seriously. Dig deeper to find out why the customer thinks that will be useful. Take the approach my doctor takes when I tell him that I think I have scurvy, the plague, or some other nonsense. He keeps an absolutely straight face and asks, "Why do you think that?" I won't be right, but the symptoms I used in my incorrect diagnosis may help him decide that I really have a cold. (I envision him with the other doctors sitting in the break room later, laughing and saying, "You'll never guess what my patient thought he had today! Ha, ha, ha!")

THE CUSTOMER IS ALWAYS RIGHT, RIGHT?

Suppose you have a customer who says you should use an XML-enabled object-relation database. You look into the problem and don't think that makes any sense. You ask the customer and he gives you a bunch of half-justifications that don't really add up. In the end he says, "Just do it."

How should you respond?

This is a tricky situation. Everyone dreads the customer who tells you point-blank to do something that you know doesn't make sense. Do you waste the customer's time and money to pursue the wrong course of action? Or do you tell the customer that you won't do it and risk getting fired?

Everyone has to make this call for themselves. You're the one who has to sleep at night after making the decision.

My personal philosophy is that I put the customer's needs first. If I think the customer is telling me to do something incorrect, I'll say so. But if the customer insists and I think I can do what he wants, I'll go ahead and do my best. In the end, it's the customer's money after all. The customer is paying for my time and experience. If I make a big deal out of it and get fired, he'll probably just go out and find someone less experienced who blunders in without seeing the consequences of following the misguided advice and will make matters worse than I would.

However, I've rarely come to this point with a customer. Usually if you can explain your concerns in terms that customers can understand, they'll either convince you that there's reason to their madness or they'll realize that the issues you've raised make sense.

LEARN WHAT THE CUSTOMERS REALLY NEED

Sometimes, the customers don't completely understand what they need. They think they do and they almost certainly understand the symptoms of their problems, but they don't always make the right cause-and-effect connections.

They may think a database or a new computer program will magically increase their sales, reduce their costs, walk their dogs, and wash their cars. In fact, a well-designed database will increase consistency, reduce data entry errors, provide reports, and otherwise help the customers manage their data, but that won't necessarily translate into higher profits.

As you look over the customers' operation, keep in mind that their real goals may not be exactly what they think they are. Their real goals probably include things like making bigger profits, making fewer mistakes so they don't get yelled at as much by managers and clients, and finishing their daily work in time to go watch their kids' soccer practice.

Look for the real causes of the customers' problems and think about ways that you can address them. If you can see a way to improve operations, suggest it (always keeping in mind that they probably know a whole lot more about their business than you do, so there's a good chance that your idea won't fly).

NOTE *By the way, never ever tell a customer, "What you really need is a slap in the head and a better product." That sort of nonconstructive criticism may be gratifying, but it usually generates an unfavorable response.*

PRIORITIZE

At this point, you should have a fair understanding of the customers' business, at least the pieces that are relevant to your project. You should understand at least roughly which customers will be playing which roles during the upcoming drama. At a minimum, you should know who the Customer Champion and Customer Representatives are so you know who to pester with questions.

You should also have a big list of desired features. This list will probably include a lot of unicorns and pixie dust—things that would be nice to have but that are obviously unrealistic. It may also include things that are reasonable but that would take too much time for your current project.

To narrow the wish list to a manageable scope, sit down with the customers and help them prioritize. You'll need the Customer Representatives who understand what is needed so they can make the decisions. Sometimes, you may need the Customer Champion either in the meeting or available for consultation to make the tough calls.

Group the features into three categories. Priority 1 (or release 1) features are things that absolutely must be in the version of the project that you're about to start building. This should be the bare-bones essentials without which the project will be a failure.

Priority 2 (or release 2) features are those that the customers can live without until the first version is in use and you have time to start working on the next version. If development goes well, you might be able to pull some of these features into the first release, but the customers should not count on it.

Priority 3 (or release 3) features are those that the customers think would be nice but that are less important than the Priority 1 and 2 features. This is where you place the unicorns and pixie dust so you can ignore them for now.

MOSCOW

Another prioritization scheme uses the acronym MOSCOW, which stands for Must, Should, Could, Won't. Must requirements must go in release 1, Should requirements go into release 1 if at all possible, Could requirements are usually deferred to release 2 unless they are really easy to implement, and Won't requirements are the lowest priority and may not make it into any release.

This is another place where different development shops take different approaches. In the more flexible approach that I prefer, these categories are somewhat flexible. If, during development, you discover that some Priority 2 feature would be really easy to implement, you can pull it into the current release. In contrast, if some Priority 1 feature turns out to be unexpectedly hard, you might ask the customers how important it *really* is and suggest that it be bumped to the Priority 2 list to avoid endangering the schedule.

NOTE *You don't need to tell the customers this, but the Priority 3 features are unlikely to ever make it into production. By the time release 1 is finished, the customers will have thought of a plethora of other Priority 1 and 2 features that they want in release 2, so the release 3 features will remain unimplemented in the next version, and so on forever by induction.*

To make this sort of shuffling easier, it can be helpful to further prioritize the items within each category. If an item is high up on the list of Priority 1 items, then it is not a good candidate for deferral to the next release. Similarly, if an item is high up in the Priority 2 list, you might be willing to spend a little extra effort to bring it into the first release.

DEVELOPMENT METHODOLOGIES

These approaches assume a relatively “big-plan-up-front” style of development where releases are thought out well in advance. That approach often fits well with complicated relational databases that are hard to build halfway.

Some development methodologies use more agile approaches where releases happen quickly and incrementally. Those projects often favor NoSQL databases that are less rigid so they can be modified as development progresses.

For more information on different development methodologies, see my book *Beginning Software Engineering, Second Edition* (Wiley, 2022).

In a hardline development approach, the categories are fixed after the requirements phase ends and items never move from one category to another. This prevents the customers from promoting items from Priority 2 to Priority 1, so it can save you some trouble. However, this approach also makes it hard for you to downgrade a feature that turns out to be a real project albatross.

VERIFY YOUR UNDERSTANDING

With your notebook (and brain) bursting at the seams with all of this information, it's *almost* time to move on to the next chapter and start building a data model. Before you do that, however, you should verify one last time that you really understand the customers' needs. This may be your last chance to avoid a future filled with tar and feathers, so be sure you've gotten it right.

Walk through your understanding of the system and explain it to the customers as if they were building the system for you and not the other way around. They should make comforting grunts and noises like “yup” and “uh huh.”

Watch out for words such as “but,” “except,” and “sort of.” When they use those words, make sure that they’re only emphasizing something that you already know and not adding a new twist to things. Often at this stage, the customers think of new situations and exceptions that they didn’t think of before.

Pay particular attention to exceptions—cases where things mostly work one way but occasionally work in another. Exceptions are the bane of database designers and, as you’ll see in later chapters, you need to handle them in a special way.

For example, suppose you need to allow for returns. (A client might decide that the Kathryn Janeway sculpture he ordered is too short or clashes with his Predator statue.) While reviewing your understanding of the project, you say, “The receiving clerk enters the RMA (return merchandise authorization number) and clicks Done, right?” Your Customer Representatives look sheepishly at each other and say, “Well . . . usually but sometimes they don’t have an RMA. Then they just write in ‘None.’” This is an important exception that the customers didn’t tell you about before and you need to write it down.

For another example, suppose your customers currently use paper order forms that have shipping and billing address sections. You say, “The form needs to hold one shipping address and one billing address?” Your customer replies, “Well, sometimes we need two shipping addresses because different parts of the order go to different addresses.” Someone pulls out an order form where a second address and additional instructions have been scribbled in the margin.

This is a huge exception. It’s easy enough to add little notations to a paper form, but it’s impossible to add more than one address value to a single set of fields in a database. You can work around the issue if you plan for it, but it can be a major headache if you don’t learn about it ahead of time. If you don’t plan for this, the users will probably put extra addresses in a comment field where the database can’t verify it.

For a final example, suppose a customer record needs a billing address. While you’re reviewing your understanding the customer says, “Oh yeah, and a shipping address because sometimes they buy one as a gift.” Now you have to wonder if sometime later someone will decide that you also need a contact address in case you have questions about the order, or a corporate address where you can send legal correspondence. Or perhaps a whole slew of branch-office addresses, or an executive address where you can send golf clubs to bribe the client’s CEO.

When your customer expands a single field (or a group of fields such as an address), you should ask seriously whether it’s going to happen again. If the record needs to hold many copies of the same field, you can easily pull them into a separate table if you plan ahead of time, but it can be hard to add new copies of fields to a table after you build it and its user interface. A single customer record can hold one or two addresses but not an ill-defined, ever-expanding number. It’s better to know ahead of time and plan for an arbitrary number of related addresses.

Sometimes in database design, it’s better to only allow one item or many items. There’s no such thing as two.

CREATE THE REQUIREMENTS DOCUMENT

The *requirements document* describes the system that you are going to build. This document is sometimes called the *product requirements document (PRD)*, the *requirements specification*, *specification*, or *spec*. As all of these names imply, this document specifies the project's requirements.

Note that the requirements document is usually not a document. It's often a collection of electronic documents, prototypes, memos, diagrams on sticky notes, emails, captured video conference calls, user stories, diagrams scribbled on napkins, and other artifacts that collectively define the system.

At a minimum, the requirements document needs to spell out what you're planning to build and what it will do. It needs to explain how the customers will use the final application to solve their problems. It can also include any design or architecture that you've already done, and it can include (possibly as attachments or appendixes) summaries of the discussions you've had while deciding on the project's features.

The requirements document keeps everyone on track during later design and development. It can also prevent finger-pointing when someone starts yelling about how you forgot to include the telepathic user interface. You can simply point to the requirements document and say, "Sorry, but the telepathic interface isn't in here." In fact, if you considered this issue during brainstorming and dumped the telepathic interface into the Priority 3 "unicorns and pixie dust" category, having it listed there will probably allow you to skip the whole argument. The potential wave-maker can see that the issue has been shelved for now and will probably not bother stirring up trouble on a dead issue.

(I've worked on some projects that had enormous requirements documents, sometimes running to 500 or more pages. In that case, it's hard for anyone to remember everything that's in there and you may end up revisiting some issues occasionally. Hopefully you thought ahead and created an index.)

The requirements document should define *deliverables* (also called *milestones*, not to be confused with millstones) that the customers can use to gauge the project's progress. These should be tasks that you complete along the way, that you can show to the customer, and *that can be verified in some meaningful way*. It's important that they be verifiable. Saying that you're 25 percent done thinking about the design doesn't do the user any good. Saying that you have built a database and filled it with test data drawn from a legacy system is much more useful and verifies that the database can hold that kind of data.

If you make the database design a deliverable (usually a good idea), then you need to be able to somehow verify that the design meets the customers' needs. Usually that means an extensive review where a lot of people put their heads together and try to poke holes in your carefully crafted design.

Prototypes also make excellent deliverables. Customers can experiment with a prototype to better understand what the system will do, and they can give you feedback if you're not heading in the right direction. If you're building a full-blown user interface for the database, then you could mock up some prototype screens (probably with no error checking and possibly with just a little concocted data) to give the customers a feel for the completed application.

Some of the deliverables defined by the requirements document should be *final deliverables*. These are deliverables that determine whether the project is finished. Like all of the other deliverables, they must be measurable to be useful.

A particularly useful technique for deciding when a project has met its goals is to create use cases, which are described in the following section.

MAKE USE CASES

A *use case* is a script that the users can follow to practice solving a particular problem that they will face while using your finished product. These can range in complexity from the very simple (such as logging in or closing the application) to the extremely complex (such as scheduling a fleet of trucks to perform in-home dog grooming).

Depending on how complete the user interface design is when you are writing the use cases, they may be sketchy or extremely detailed. They may spell out every keystroke and mouse movement that the user must make, or they may provide vague instructions such as, “The user will use the Order Entry form to place a new order.”

When the project is finished, the customers should review all the use cases and verify that the finished project can handle them all. (In self-defense, you should run through the use cases before you tell the customers that you’re finished. That way, you don’t look silly when the product cannot handle simple chores during an executive dog-and-pony show.)

Some of the things that you might specify when writing up a use case include:

- **Goals**—A summary of what the use case should achieve.
- **Summary**—An executive overview that even your Executive Champion can understand.
- **Actors**—Who will do what? This includes people, your finished system, other systems, and so forth—anyone or anything that will do something.
- **Pre- and post-conditions**—The conditions that should be true *before* and *after* the use case is finished. For example, a pre-condition to placing a new order might be that the client placing the order already exists.
- **Normal Flow**—The normal steps that occur during the use case.
- **Alternative Flow**—Other ways that the use case might proceed. For example, when a user tries to look up a customer, what happens if the customer isn’t there?
- **Notes**—Just in case there are special considerations that the person following the use case needs to know.

Many developers like to draw use case diagrams to show what actors perform what tasks. These seem to usually work at one of two levels.

A higher-level use case shows which actors perform which tasks. For example, the Student actor enrolls in a class and takes the class, the Instructor actor teaches the class and assigns grades, and so forth. (If the student is taking an acting class, then things can get really confusing.) This type of use case diagram provides little detail about how the actors accomplish their tasks. It’s useful early on when you know what you want to do but don’t yet know how the system will do it.

Figure 4.1 shows a high-level use case diagram. Actors are shown as stick figures, tasks are shown in ellipses, and lines connect actors to tasks. More elaborate use case diagrams use other kinds of arrows, lines, and annotations to provide more detail.

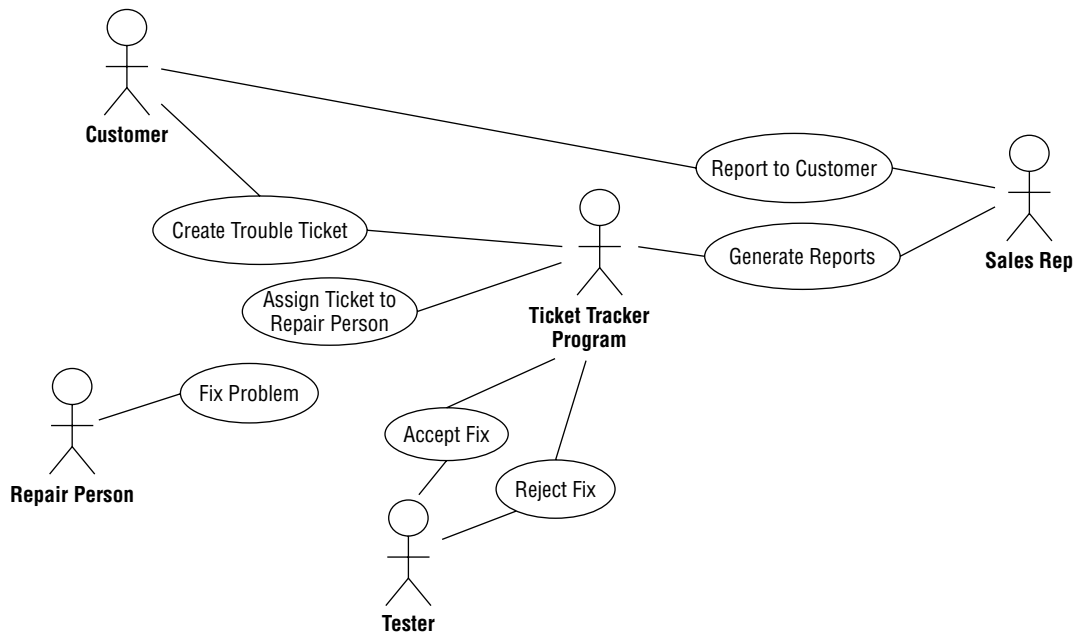


FIGURE 4.1

The second kind of use case lists more specific steps that actors take to perform a task, although the steps are still listed at a fairly high level.

Neither of these kinds of use case diagram provides enough detail to use as a script for testing, although they do list the cases that you must test. Because they are shown at such a high level, they are great for executive presentations. For more information on use case diagrams, look for books about Unified Modeling Language (UML), which includes use case diagrams, or search the web for “use case diagram.”

Typical use cases might include:

- The user logging in
- The user logging out
- Switching users (if the program allows that)
- Creating a new customer record
- Editing a customer record
- Marking a customer record as inactive
- Creating a new order for an existing customer
- Creating a new order for a new customer

- Creating an invoice for an order
- Sending out late payment notices
- Creating a replacement invoice in case the customer lost one
- Receiving a payment and recording the invoice as paid
- Defining a new inventory item (for example, when the CEO decides that you should start selling Rogaine for Dogs)
- Adding new items to inventory (for example, when you restock your fuzzy dice supply)

The list can go on practically forever. A large project can include hundreds of use cases, and it might take quite a while to write them all down and even longer to later verify that the finished project handles them all.

In addition to being measurable (you want to be able to tell whether the program can pull its weight), use cases should be as realistic as possible. There's no point in verifying that the program can handle a situation that will never occur in real life.

SILLY SYNCHRONICITY

In one project, the program we were writing needed to be able to handle 20 simultaneous users. One customer performed a test where 20 people all sitting in the same room walked step by step through the same use case at the same time. They all typed in the same text and clicked the Find button at the same time. The program gave terrible performance because every user's computer tried to access the same database records at exactly the same time. In a more realistic test, every user tried to access a different record and everything was fine.

ENROLLMENT USE CASES

Suppose you are building a program to let students log on over the Internet and enroll in classes. All enrollments are tentative until a specific date when they are all processed. (That gives the school a chance to juggle schedules; for example, if a graduating student really needs a class, another student might get bumped for now.) To accommodate this flexibility, students should enter alternate choices.

For this exercise, make a list of database use cases that you could use to look for data that you have not built into the design and to later test to ensure that all of the data is present. You don't need to explain how a user will perform a certain task; just briefly describe the task and list the kinds of data that must be stored or accessed during that task. Add any questions that need further study or feedback from the customers.

You should perform use cases covering every task that the final users of the system would perform. Here's the list that I came up with.

continues

(continued)

TASK	DATA NEEDS
Log on successfully or unsuccessfully.	Verify UserName and Password in Students table. (How do we generate these? How do we guarantee security?)
Enter desired schedule.	Let students pick from drop-down lists so we don't need to verify that they typed meaningful choices. Refer to course schedule tables to give students choices. Save student selections in student selections tables. (Allow students to prioritize their selections?)
Generate final schedules.	Refer to course schedule tables to obtain Capacity. Refer to global tables to learn minimum enrollment to not cancel a class. (Does this vary by class? By department?) Process student selection tables, adding students to desired classes in the course tables. If a class fills, bump lower-priority students, consult their selections, and assign a replacement course. If a class has too few students, notify the administrator to cancel the class. Consult the selections of any students in the canceled class and assign replacement courses. When finished, review the course tables and copy student course assignments into student data tables. Check global tables (vary by department?) to learn minimum and maximum normal course load. If a student falls outside of those bounds, look up the student's counselor in the student tables and notify that counselor via email.
Send schedules to students.	Obtain student schedules and email addresses from student tables. Email them the schedules.
Email course rosters.	Obtain course roster data from course tables. Get the name of each course's instructor from the course tables. Get the instructor's email address from the instructor tables. Email the class's roster to the instructor.
Manually adjust schedules.	Allow administrators to manually adjust schedules to handle special circumstances. This will require free access to course tables, student data tables, and student course assignment tables.

DECIDE FEASIBILITY

At some point, you should step back, take a deep breath, and decide whether the project is feasible. Is it even possible to design a database to do everything that the customer wants it to do?

Can you really build a database to hold records for 17 million customers, provide simultaneous access for 80 service representatives, log every transaction with timestamps and user IDs, give interactive responses to queries in less than 2 seconds 90 percent of the time, and still fit it all on a 16 MB flash drive?

Okay, the last condition is pretty unrealistic, but seriously someone must think about the project's viability at some point. No one will be happy to hear that you can't solve all the customers' problems, but everyone will be a lot happier if the project is canceled early instead of after you've wasted a year of everyone's time and a king's ransom in funding.

If it really looks like you can't complete the project, make the tough call and ask everyone to rethink. Perhaps the customers can give up some features to make the project possible. Or perhaps everyone should just walk away and move on to a more realistic project.

SUMMARY

Building any custom product is largely a translation process whether you're building a small database, a gigantic Internet sales system similar to the one used by Amazon, or a really tricked-out snowboard. You need to translate the half-formed ideas floating around in the minds of your customers into reality.

The first step in the translation process is understanding the customers' needs. This chapter explained ways that you can gather information about the customers' problems, wishes, and desires so you can take the next step in the process.

In this chapter, you learned how to:

- Try to figure out which customers will play which roles.
- Pick the customers' brains for information.
- Look for documentation about user roles and responsibilities, existing procedures, and existing data.
- Watch customers at work and study their current operations directly.
- Brainstorm and categorize the results into Priority 1, 2, and 3 items.
- Verify your understanding of the customers' needs.
- Write requirements documents with verifiable deliverables, including use cases.

After you've achieved a good understanding of the customers' needs and expectations, you can start turning that understanding into data models. The following chapter explains how to convert the customers' needs into informal data models that help you better understand the database, and then how to convert the informal models into more formal ones that you can actually use to build a database.

Before you move on to Chapter 5, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions in Appendix A.

EXERCISES

1. In Figure 4.2, draw lines connecting the customer roles with their corresponding descriptions.

Customer Role	Description
Convert	Someone who won't be around for long. May be helpful or may not care all that much.
Customer Champion	Answers your questions about the project.
Customer Representative	Anyone who has an interest in the project.
Devil's Advocate	Makes things generally run smoothly. Not glamorous but very useful.
Executive Champion	Provides a reality check and prevents groupthink.
Generic Bad Guy	Ranges from annoying naysayer to malicious saboteur/super villain.
Short-Timer	A user who originally was against your project that you include in the development process to bring them onto your side.
Sidekick/Gopher	The highest ranking customer driving the project. Willing to fight super villains.
Stakeholder	Thoroughly understands the customers' needs. Has the authority to make decisions that stick.

FIGURE 4.2

2. Which of the following is *not* true of a use case?
 - A. It's a script for performing some task.
 - B. It should describe a realistic operation.
 - C. It should cover the customer's entire operation from start to finish.
 - D. It should be verifiable.
3. Brainstorming sessions should ideally include:
 - A. Customer Representatives
 - B. A Devil's Advocate
 - C. All interested Stakeholders
 - D. All of the above

4. If a customer says you should use a hierarchical XML database, you should:
-
- A. Politely say, “Thank you,” and ignore this nugget of wisdom.
 - B. Ask the customer why they think that.
 - C. Do as the customer says. (It’s their money.)
 - D. Study the problem to see if that kind of database makes sense.
5. During a visit to view the customers’ operation, you see someone repeatedly stamping the front of an order with the current date, turning the order over, turning it over again, and stamping the front with the date again. You should:
-
- A. Ask someone what that’s all about.
 - B. Suggest that the manager fire this crazy and possibly dangerous employee.
 - C. Ignore the whole issue and stay focused on your own tasks.
 - D. Avoid eye contact with this employee at all costs.
6. Look at the ZIP Code lookup form at <https://tools.usps.com/zip-code-lookup.htm?byaddress>. What are this form’s data needs? Which fields are required? (How does the user know those fields are required?) What are the domains for the fields? Which *could* involve a foreign key validation?
-
7. Which of the following is *not* a security issue that you should consider when studying the project?
-
- A. The number of classes of users the database must support
 - B. Whether you need to provide audit trails to record changes to the data
 - C. The frequency with which you need to perform backups
 - D. Whether the users should have individual passwords
8. You are called upon to design a database for a florist shop named “Frank’s Floral Fantasies.” Frank thinks that he might want to track the medicinal and homeopathic properties of his plants because he thinks that might improve his sales of echinacea, St. John’s Wort, and other plants. What priority should this requirement get?
-
- A. Priority 1, definitely in this release
 - B. Priority 2, probably in the next release
 - C. Priority 3, with the unicorns and pixie dust
 - D. It depends (you need more information)

9. What does MOSCOW stand for?

10. Write a use case for logging in to your computer's operating system. Ignore biometric login options and MFA. Just focus on the login screen.

11. You're halfway finished designing your database when a Vice Presidential Supervillain says that your project is doomed to failure because you didn't include a sufficient allowance for farbulistic granilation. You need to cancel the whole thing and start over with them in control. How should you handle this attack?

5

Translating User Needs into Data Models

Chapter 4, “Understanding User Needs,” discussed ways that you can work with customers to gain a full understanding of the problem at hand. The result should be a big pile of facts, goals, needs, and requirements that should be part of the new database and its surrounding ecosystem. You might already have made some connections among various parts of this information, but mostly it should be a big heap of requirements that doesn’t say too much about the database’s design and construction.

This kind of pile of information is sometimes called a *contextual list*. It’s basically just a list of important stuff (although it might be fairly elaborate and include requirements documents, diagrams, charts, and all sorts of other supporting documentation).

The next step in turning the conceptual list into a database is converting it into a more formal model. You can compare the formal model to the contextual list and ensure that the model can handle all of your requirements.

You can also use the model to verify that you’re on track. You can explain the model to the customers and see if they think it will handle all of their needs or if they forgot to mention something important while you were following the procedures described in Chapter 4.

CONSTANT CONFIRMATION

Constantly verifying that you’re on track is an important part of any project. It’s much easier to hit a target if you’re constantly checking the map and making any necessary adjustments. You wouldn’t aim your car at a parking space, close your eyes, and step on the gas pedal, would you? It’s much easier to park if you keep an eye on your progress, the other cars, the skateboarders slamming nosegrinds off the curb, kids riding on shopping carts, and everything else that’s going on in the parking lot.

After you build a data model (or possibly more than one), you can use it to build a relational model. The *relational model* is a specific kind of formal model that is structurally very similar to the one used by relational databases. That makes it relatively easy to convert the relational model into an actual database in MySQL, Postgres, or some other relational database product.

In this chapter, you learn how to:

- Create user interface models.
- Create semantic object models.
- Create entity-relationship models.
- Convert those types of models into relational models.

After you master these techniques, you'll be ready to start pulling the models apart and rearranging the pieces to improve the design by making it robust and flexible.

WHAT ARE DATA MODELS?

Despite what some managers seem to believe, a model isn't a silver bullet or enchanted wand that will magically make a project succeed. A model by itself doesn't do anything. It doesn't build a database, it isn't a piece of software (although there are software tools that can help you build a model), and the final user of your database never sees a model.

A model is a plan. It's a blueprint for building something, in this case a database. The purpose of the model isn't to do anything by itself. Instead it gives you a concrete way to think about the database that you are going to build. By studying the model's pieces, you can decide whether it represents all the data that you need to use to meet your customers' needs.

A model is also useful for ensuring that everyone on the project has the same understanding of what needs to be done. If everyone understands the model, then everyone should have the same ideas about what data should be stored, which tables should contain it, and how the tables are related. They should also agree on the business rules that determine how the data is used and constrained.

It's very important that everyone actually understands the model. I've seen developers build remarkably complicated models and then dump them on hapless end users, expecting those users to understand the models' every subtle nuance. The developers ended up walking the users through the models until the users' heads were spinning, and the developers could have convinced them that up was down and red was blue if they would just stop the meeting. Models are for those who know how to understand them, not necessarily for everyone.

After you build a model, you should look at it and ask questions such as:

- Where do we store customer information?
- How many contact names can we store for a customer?
- Where do we store the contacts' favorite colors?
- What if we need to store multiple price points for the same product?

- How do we store the 17 kinds of addresses we need for customers?
- Where do we store supplier information?
- If someone asks about an order they placed but haven't received, how can we figure out where it is?
- Where can we enter special instructions for an order?
- How do we know when we need to restock left-handed cable stretchers?

You should also work through any use cases to see if the model can handle them. You can't actually fill out insurance claim forms and look in the warehouse for missing orders yet, but you should be able to say, "This table contains the data that we would need to do that."

The end users can help a lot with this part. Although they may not understand the models, they do understand their business and can ask these sorts of questions while you and the other developers try to figure out if the model can handle them.

If the model cannot handle all of your (and the users') questions, then you need to adjust the model. You might need to add fields or tables, change a field's data type, make new connections between tables, or make other changes to satisfy the requirements. In extreme cases, it may be easiest to start a new model from scratch.

This chapter discusses four kinds of models that grow successively closer to the final database implementation.

First, a *user interface model* views the database at a very high level as seen from the end user's point of view. Depending on how you are going to use the database, this might be as the user will view the database through forms on a computer screen. This model is very far from the final database implementation, and it doesn't tell much about the database design. This model is useful for understanding what data is needed by the project and how you might use it to navigate through the user interface.

The second and third types of models described in this chapter are *semantic object models* and *entity-relationship models*. These are roughly at the same distance from the final database. They are at a slightly lower level than the user interface model and show relationships among data entities more explicitly. They are still at a moderately high logical level, however, and do not provide quite enough detail to build the final database.

The fourth type of model described in this chapter is the *relational model*. This model mimics the structure of a relational database closely enough that you can actually sit down and start building the database.

In a typical database design project, you might start with a user interface model. I like to start there because I figure if the user is going to see something, we better have a place for it in the database. Conversely, if the user isn't going to see it in some manner, do we really need it in the database? (However, that's just me. I like designing user interfaces. Some people prefer to skip that and let someone else worry about the user interface.)

Next, you use what you learned from the user interface model to build either a semantic object model or an entity-relationship model. These models serve the same purpose, so you generally don't need to build them both. Work through this chapter and the exercises at its end and decide which one you prefer.

Finally, you convert the semantic object model or entity-relationship model into a relational model. Now, you have something that could be turned into a database. There are still some steps to go through as you refine the relational model to improve the final database's reliability and performance, but those are subjects for later chapters.

NoSQL MODELS

If you're using a NoSQL database, these types of models can still be useful. A user interface model still helps you understand the data you will need and where you will store it. Semantic and entity-relationship models will still help you understand how pieces of data are related.

Even the relational model may help, at least for some part of your database, although you may not want to build it in as much detail. For example, you don't need to define relational tables to hold data that will be stored in a graph database.

Remember, these models are intended to improve your understanding of the data and the ways in which different bits are related—so with that in mind, anything that increases understanding is beneficial. Don't be afraid to add notes that clarify confusing issues. Feel free to modify the basic modeling techniques described here. There's some benefit to sticking close to standard notations because it lets others who have studied the same notation understand what you are doing. But if adding a number in a box by each link or a colored triangle helps you and your team gain a better handle on the design, do it. Just be sure to make a note of your additions and changes so that everyone knows what your additions mean.

USER INTERFACE MODELS

A *user interface model* examines the database at a very high level from the point of view of end users and other systems that manipulate the data. In most database applications, a user will eventually see the data in some form. For example, an order entry and tracking application might use a series of screens where the user can perform such chores as entering orders, tracking orders, marking an order as paid, looking up available inventory, and so forth. Those screens form the database's user interface.

Some databases don't have their own user interfaces, at least not that a human will see. Some databases are designed to store data for other applications to manipulate. In that case, it is the interfaces that those other applications provide that the human user sees. If possible, you should consider what those applications will need to display and plan accordingly. Sometimes, it is useful to build throwaway interfaces to view the data on forms, in spreadsheets such as Google Sheets or Microsoft Excel, or in text files.

You should consider how those other applications will obtain the data from your database. The way in which those applications interact with your database forms a nonhuman interface, and you should plan for that one, too. For example, suppose you know that a dispatch system will need to fetch information about employees from your database and information about pending repair jobs from

another system. You should think about the kinds of employee data that the dispatch system will need, such as a repairperson's skills, equipment, assigned vehicle, and so forth. Then you can design your database to make fetching this data easy and efficient.

To build the user interface model, start by making rough sketches of the screens that the user will see. Often these first sketches can be modified versions of paper forms if any exist.

Fill the fields with sample data to make it easier to understand what belongs on each screen. These sketches can be anything from crayon scribbles on bar napkins, to forms drawn with your favorite computerized drawing tool, to full user interface prototypes. Figure 5.1 shows a mocked-up Find Orders screen built with Visual Basic. This form holds only controls and doesn't include any code to do anything more than just sit there and look pretty.

The screenshot shows a window titled 'Find Orders' with a search form and a results table. The search form has fields for Customer Name, Customer ID, Contact Name, Order Date (with a date range from 1/1/27 to 10/1/27), and Order ID, along with a Search button. The results table lists several orders with columns for Customer Name, Contact Name, Date, and Status. The row for 'No Nonsense Toys' with contact 'Cindy Traz' and date '5/12/2027' is highlighted.

Customer Name	Contact Name	Date	Status
Bob's Burgers	Bob Alfraz	4/1/2027	Closed
Bob's Burgers	Betty Alfraz	6/8/2027	Pending
No Nonsense Toys	Cindy Traz	3/21/2027	Closed
No Nonsense Toys	Cindy Traz	5/12/2027	Returned
No Nonsense Toys	Mike Traz	5/16/2027	Closed
Passing Wind Kites	Benjamin Hill	6/16/2027	Open

FIGURE 5.1

In addition to the image in Figure 5.1, you should include text explaining what the various parts of the form do. In this case, that text might say:

- The user enters selection criteria in the upper part of the form and clicks the Search button.
- The program displays a list of matching order records in the bottom of the form.
- The user can select an order from the list and click Open to open that order's detail form.

At this level, the user probably thinks of each order as containing all of the information on this form. If you were to fill out an order on a piece of paper, that paper would include blanks for you to fill in customer name, customer ID, contact name, order date, and so forth. The order would also contain a status, although you might represent that by putting the order in boxes on your desk labeled Pending, Open, Closed, Ignored, and so forth, rather than by having a status box on the paper form.

The form and its description also raise some important questions:

- What fields should be allowed as selection criteria?
- Should we index the selection criteria fields in the database to make searching faster? Some or all of those fields?
- When the user selects an order and clicks Open, how does the program open the Orders record? (Searching for the exact combination of fields shown in the list would be slow, and there might even be two entries with the same values if someone placed two orders on the same day. It might be wise to add an order ID field to make finding the record again easier.)

When you select an order from the form shown in Figure 5.1 and click Open, the program displays the form shown in Figure 5.2.

The screenshot shows a window titled "Order Detail" with the following sections:

- Order Summary:**
 - Date Placed: 4/1/27
 - Date Shipped: 4/4/27
 - Date Paid: 5/14/27
 - Status: Closed
 - Ship Method: Priority
 - Billing Method: Invoice/Net 30
- Shipping Information:**
 - Customer Name: No Nonsense Toys
 - Customer ID: 261786
 - Street: 1562 Prank Pl, Suite 27
 - City: Conundrum
 - State: MA
 - ZIP: 02162
- Billing Information:**
 - Customer Name: No Nonsense Toys
 - Customer ID: XXXXXXXXXXXXXXXX
 - Street: 37281 Clever Ct
 - City: Conundrum
 - State: MA
 - ZIP: 02162
- Contact Information:**
 - Name: Cindy Trax
 - Phone: 404-555-0137
 - FAX: (empty)
 - Email: Cinder@NoNonsense.com
- Order Items:**

SKU	Description	Price	Quantity	Total
3728-209	Deluxe Whoopie Cushion	\$6.95	10	\$69.50
1029-302	Assorted Nose Glasses	\$0.75	20	\$15.00
3762-102	Beginning Database Design Book	\$39.99	1	\$39.99
4762-398	Exploding Shoelaces	\$1.95	6	\$11.70
2821-201	X-Large Chocolate Cockroaches (50)	\$4.95	12	\$59.40
- Totals:**
 - Subtotal: \$195.59
 - Tax: \$9.78
 - Shipping: \$35.00
 - Grand Total: \$240.37

FIGURE 5.2

This form shows the fields that should be associated with an order. These include:

- Various dates such as the date the order was placed, the date the products were shipped, and the date the customer paid
- The order's current status
- The shipping method (for example, Priority, Overnight, Armored Courier, and so forth)

- The billing method (credit card, invoice net 30)
- Various addresses such as the shipping and billing addresses
- Contact information for when we get confused (or want to send spam to the unsuspecting contact)
- The order's line items
- Subtotal, taxes, shipping, and grand total

Both of these forms involve orders and both provide some information about the order data. The Order Detail form includes a lot of the fields that must be stored to represent an order. The Find Orders screen tells which order fields should be allowed as search criteria (and thus may make good keys) and which order fields should be displayed in the result list.

Each of these forms tells a little bit more about the order data. Other mocked-up forms would provide even more information. For example, the application would need an order entry form and a form to update order information (such as changing the addresses or setting order status to Closed). Depending on how the work was divided among employees, there might be special forms for performing a single specific task. For example, an order fulfillment clerk (who puts things in a box and ships them) would need to be able to change an order's status to Shipped but probably doesn't need to be able to change a customer's credit card numbers. In fact, going through the screens and deciding which employees should be able to do which tasks gives you an initial indication of the application's security requirements.

Still other forms would give hints about other parts of the database. A full-fledged database for this application would need to include forms for managing inventory (how many whoopee cushions do we have in stock and when do we order more?), supplier information (who sells us our nose glasses?), employee information (who is assigned to pester delinquent customers this week?), advertising data (which spam campaigns gave us the most new contacts?), and so forth.

A large application might include dozens or even hundreds of forms, each of which gives only a partial glimpse of the information contained in the database. Together, these mocked-up screens form a user interface model that shines spotlights into the data needed to support the application.

With the user interface model in hand, you are now ready to build a more formal model that shows the entities used by the application in greater detail. The first of those models discussed in this chapter is the semantic object model.

User Interface Models

Sketch out a form where the user could enter shift information for employees. What data must be displayed on the form?

How It Works

Figure 5.3 shows a mocked-up employee shift form.

FIGURE 5.3

This form includes the following data:

- Employee name (selected from a combo box).
- The starting day of the week the user is viewing and editing for this employee (selected from a combo box). (Which weeks will we allow the user to pick? How far in the future?)
- The user should also be able to select past weeks (from a combo box) from which to copy.
- The hours that the employee is scheduled to work. These records (in the `EmployeeShifts` table?) will include employee, date, start time, and stop time.
- Total hours scheduled. This can be calculated from the shift data.

The form will also need to look up minimum and maximum normal hours so that we can warn the user if something is unusual. For example, if the user is scheduled to work 70 hours in a week, the form can ask the user to verify before accepting the changes.

SEMANTIC OBJECT MODELS

A *semantic object model (SOM)* is intended to represent a system at a fairly high level. Although the ideas are somewhat technical, they still relate fairly closely to the way people think about things, so semantic object models are relatively understandable to users.

Classes and Objects

Intuitively, a *semantic class* is a type of thing that you might want to represent in your system. It can include physical objects such as people, furniture, inventory items, and invoices. It can also include logical abstractions such as report generators, tax years, and work queues.

Technically, a semantic class is a named collection of attributes that are sufficient to identify a particular entity. For example, a `PERSON` class might have `FirstName` and `LastName` attributes. If you can identify members of the `PERSON` class by using their `FirstName` and `LastName` attribute values, then that's good enough. (In real life, many people have the same first and last names, so that's probably not enough, but we'll let it slide for now.)

By convention, the names of semantic classes are written in ALL CAPS as in `EMPLOYEE`, `WORK_ORDER`, or `PHISHING_ATTACK`. Some people prefer to use hyphens instead of underscores, so the last two would be `WORK-ORDER` and `PHISHING-ATTACK`.

A *semantic object (SO)* is an instance of a semantic class. It is an entity instance that has all of the attributes defined by the class filled in. For example, an instance of the `PERSON` class might have `FirstName` "Stephen" and `LastName` "Colbert."

Traditionally, the attributes that define a semantic class and that distinguish semantic objects are written in mixed case as in `LastName`, `InvoiceDate`, and `DaysOfConfusion`.

Attributes come in three flavors: simple, group, and object.

A *simple attribute* holds a simple value such as a string, number, or date. For example, `LastName` holds a string and `EmployeeId` holds a number.

A *group attribute* holds a composite value—a value that is composed of other values. For example, an `Address` attribute might contain `Street`, `Suite`, `City`, `State`, and `ZipCode`. You could think of these as separate attributes, but that would ignore the structure built into an address. These values really go together, so, to represent them together, you use a group attribute.

ADDRESS EXCESS

You might need other fields if you work with non-U.S. addresses. For example, if you ship products to several countries, you'll need to add a `Country` attribute, you might want to use `Region` instead of `State`, and it may be better to use `PostalCode` instead of `ZipCode`. You might also want to add an extra line after the street address to hold things like business name, urbanization, or building name.

Sometimes, I think it would be easier to just include six or seven blank lines and ask the user to type the address exactly as it should appear on an envelope or package. Of course, then you couldn't search based on `City`, `ZipCode`, or other fields. This is a data normalization issue that we'll talk about in later chapters.

An *object attribute* represents a relationship with some other semantic object. For example, a relationship may represent logical containment. A `COURSE` class would have a `STUDENT` object attribute to represent the students taking the course. Similarly the `STUDENT` class would have a `COURSE` object attribute representing the courses that a student was taking. Each of these classes is related to the other, so they are called *paired classes*. Similarly, their related attributes are called *paired attributes*.

Cardinality

An attribute's *cardinality* tells how many values of that attribute an object might have. For example, at the start of some volleyball tournaments each team's roster must contain between 6 and 12 players.

You write the lower and upper bounds beside the attribute to which they apply separated by a period. The volleyball team roster's `PLAYERS` attribute would have cardinality 6.12. (I have no idea why it's a single period and not a dash or ellipsis. At this point it's just a historical quirk. If you decide to use a dash in your diagrams, I doubt the Semantic Object Model Police will break down your door and haul you away.)

Usually the minimum cardinality is 0 if the value is optional or 1 if it is required.

The maximum cardinality is usually 1 if at most one value is allowed or N if any number of values is allowed.

Probably the most common cardinalities are:

- **1.1**—Exactly one value required. For example, suppose you are building a database to track restaurant orders. In the `ORDER` class, the `ServerName` attribute would have cardinality 1.1 because every order must have exactly one server.
- **1.N**—Any number of values but at least one required. For example, the `ORDER` class's `Item` attribute would hold the items requested by the diners and would have cardinality 1.N. It wouldn't make sense to send an order to the kitchen if it didn't contain any items, but it could contain a potentially infinite number of items. (In practice, though, I might double-check with the server if the kitchen received an order for 13,000 hamburgers.)
- **0.1**—An optional single value. For example, the server might want to record a `Comment` to go with the order. ("Extra cheese on the milkshake.")
- **0.N**—Any number of optional values. For example, a collection of `Comment` attributes. ("Dressing on the side for salad 1." "No mayo on burger 2." "Recognize poor tipper, use day-old breadsticks.")

Those are the most common cardinalities, but you may need to use others for your application. For example, suppose an entrée comes with two side dishes included, and you can add up to two more for an extra charge. In that case, the `SideDishes` field would have cardinality of 2.4.

Identifiers

An *object identifier* is a group of one or more attributes that the users will typically use to identify an object in the class.

An object identifier can include a single attribute such as `CustomerId` or a group of attributes such as `FirstName`, `MiddleName`, and `LastName`.

You indicate an identifier by writing the text **ID** to the left of its attributes. Often identifiers contain unique values so every item in the class will have different values for the identifier. For example, `CustomerId`, `SocialSecurityNumber`, and `ISBN` are unique identifiers for customers, employees, and books, respectively. You can indicate a unique identifier by underlining the ID notation to its left.

Sometimes, non-unique identifiers are used to find groups of objects. For example, many systems might search for customers by `LastName` and `FirstName`, but sometimes those values are not unique, so you shouldn't underline them. That way, your database won't implode if you have two customers named Zaphod Beeblebrox.

Putting It Together

Figure 5.4 shows a simple representation of a `CUSTOMER` class that demonstrates these notational features.

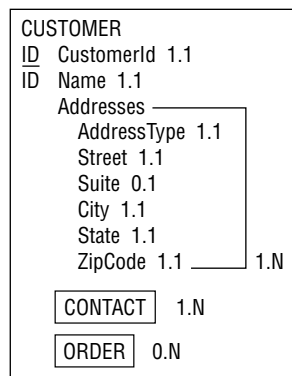


FIGURE 5.4

A big box surrounds the whole class definition. The class name, `CUSTOMER`, goes at the top.

`CustomerId` is a simple attribute that is used to identify customers, so it gets the ID notation. `CustomerId` values are unique so the ID is underlined. This value is required and a customer can have only one ID, so its cardinality is 1.1.

Users sometimes want to search for customers by name, so the `Name` attribute is also an identifier. It is possible that two customers could have the same name, however, so here ID isn't underlined.

The `CUSTOMER` class includes address information stored in the `Addresses` group attribute. Each address has the attributes `AddressType` (this will be something like `Shipping` or `Billing`), `Street`, `Suite`, `City`, `State`, and `ZipCode`. All of these except `Suite` are required and can hold only one value, so they have cardinality 1.1. The `Suite` attribute is optional, so its cardinality is 0.1. Lines show the attributes contained inside the `Addresses` value. The 1.N to the lower right of the group indicates that a `CUSTOMER` object must have one or more `Addresses` values (each containing a `Street`, maybe `Suite`, `City`, `State`, and `ZipCode`).

Finally, the class has two object attributes named `CONTACT` and `ORDER`. The `CONTACT` attribute represents one or more contact people for the customer. The box around the attribute tells you that this is an object attribute. Its cardinality 1.N indicates that the `CUSTOMER` must have at least one contact.

The `ORDER` attribute represents the orders placed by this customer. You might think that this should have cardinality 1.N. After all, why would you need a customer who doesn't place any orders? However, when you first create a customer record it will have no associated orders, so we should allow that. You might also want to be able to make a customer record in anticipation of future orders. For both of those reasons, this design sets the cardinality of `ORDER` to 0.N.

DESIGN DECISIONS

This is a design decision, and in your application, you could take the other route and not allow customers without orders. You can look at the user interface model to see which would be more natural. Do you want to provide a screen where a user can create a customer record without an order, or do you want to make the order entry screen allow for creating a new customer?

Building a Semantic Object Model

Make a semantic object model for an `EMPLOYEE_WEEK` class that holds information about employees scheduled for a week. This class should have object identifier fields `EmployeeId` and `StartDate`. It should also have a group attribute named `Shift` that includes `StartTime` and `StopTime`, and it should hold one `Shift` for each of the seven days of the week.

How It Works

Figure 5.5 shows the semantic object model for the `EMPLOYEE_WEEK` class.

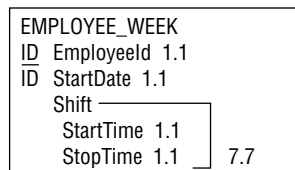


FIGURE 5.5

Semantic Views

Sometimes, it is useful to define different views into the same data. For example, consider the kinds of information a company typically tracks for its employees. That information might include:

- Normal contact information such as name, address, phone number, and next of kin
- Work-related contact information such as title, office number, extension, pager number, and locker number at the country club (if you're an executive)

- Confidential salary information, including your complete salary and annual bonus history
- Other confidential information such as your stock plan and 401K program participation, insurance selections, annual performance reviews, and golf handicap (with and without witnesses)

Some of this information, such as your name and title, is freely available to anyone who wants it.

Other semi-public information is available to anyone within the company but not outside the company. (Many companies worry that executive recruiters with the company phonebook could steal employees away with all of their valuable skills and the proprietary information locked inside their heads.) This information includes your office number, extension, project history, and birth date (excluding the year). It does not include your home address, annual performance reviews, salary history, or other financial data.

Other more sensitive information should be available to your manager and other superiors but not to the general population of coworkers. This information includes such things as your annual performance reviews and work history. However, your manager does not need to know how much you are having deducted for retirement contributions, whether you participate in the company stock plan, whether you are deducting the extra \$750 a month for the dental plan, and the amount garnished from your wages to pay off your outstanding Pokémon debt. Those sorts of information should be hidden from your manager. (Depending on the way your company is structured, your manager might not even need to know your salary.)

The people in the Human Resources department are the ones who arrange to siphon money out of your paycheck for such perks as the stock plan and dental insurance, so they obviously need to know that information. However, they probably don't need access to your annual performance reviews.

Figure 5.6 shows an `EMPLOYEE` class and four views that give access to different parts of the employee data. For simplicity, I've shown each attribute as if it were a simple attribute when actually most of these are group or object attributes. For example, the `OfficeData` attribute is really a compound attribute including `Title`, `Office`, `Extension`, `BirthDate`, and so forth.

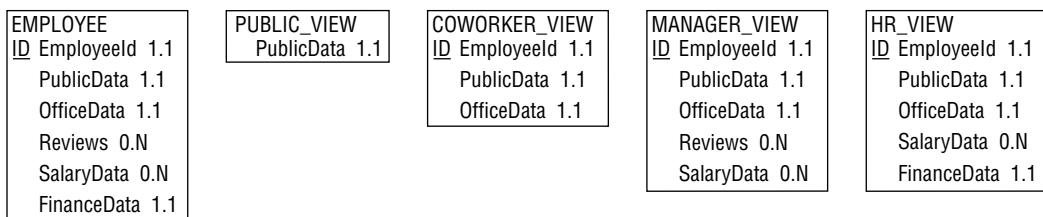


FIGURE 5.6

Defining these different views allows you to make data available only to those who need it. (This notion of *view* maps directly to the relational database concept of *view*, so defining views now will help you later.)

After you finish building a complete semantic object model, you should check each of the views to ensure that they contain all of the information needed for each class of user and nothing else. For example, you should run through all the use cases for managers to see if the `EMPLOYEE` class's

`MANAGER_VIEW` provides enough information to handle those use cases. You should also check that every piece of data included in the `MANAGER_VIEW` is actually used. If something isn't used in at least one use case, then managers might not need it and it might not belong in the `MANAGER_VIEW`.

Class Types

The following sections describe some of the types of classes that you might need to use while building semantic object models. Some of these are little more than names for simple cases. Others, such as association classes and derived classes, introduce new concepts that are useful for building models.

Simple Objects

A *simple object* or *atomic object* is one that contains only single-valued simple attributes. For example, an inventory item class might include the attributes `Sku`, `Description`, `UnitPrice`, and `QuantityInStock`. Each inventory item's data must include exactly one value for each of these attributes.

Figure 5.7 shows a simple `INVENTORY_ITEM` class.



FIGURE 5.7

Composite Objects

A *composite object* contains at least one multivalued, non-object attribute. For example, suppose you allow online customers to provide product reviews for inventory items. Then you could add a multivalued `Reviews` attribute to the class shown in Figure 5.7 to get the composite object shown in Figure 5.8.

COMPLEXITY COMPROMISED

There's some disagreement among developers about these terms. Some call an object with a multivalued, non-object attribute a "complex object" or "complex type" and use "composite" to mean an object that contains more than one data element. I think the terms defined here are more common, but if there's any doubt in your discussion with other developers, you should agree on common definitions.

Note that the multivalued attribute need not be a simple attribute. For example, suppose you decide not to use a simple attribute to hold customer comments. Instead, for each comment you store the customer's user name, a numeric rating, and comments. Figure 5.9 shows the revised `INVENTORY_ITEM` class.

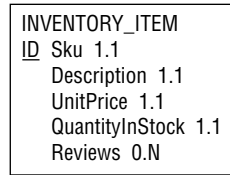


FIGURE 5.8

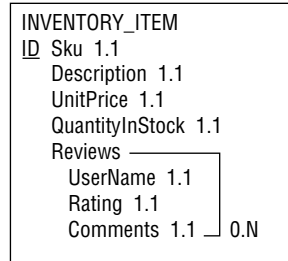


FIGURE 5.9

Compound Objects

A *compound object* contains at least one object attribute. For example, consider the `CUSTOMER` class shown in Figure 5.10. This class contains basic information such as a customer name and shipping and billing addresses. Its `CONTACT` object attribute stores information about the person we should contact if we have a question about this customer. (This is also the person who gets our junk mail.) The `SALES_REPRESENTATIVE` object attribute refers to another object representing the sales representative who is charged with keeping this customer happy. (Okay, not too much junk mail.)

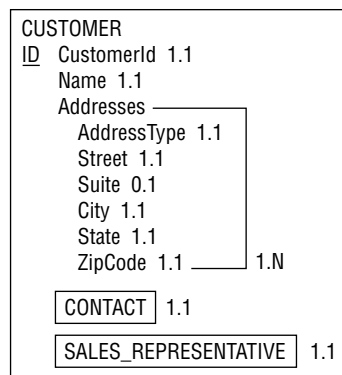


FIGURE 5.10

Hybrid Objects

A *hybrid object* contains a combination of the other kinds of attributes. For example, it might contain a multivalued group that contains an object attribute. The `ORDER` class shown in Figure 5.11 contains

a `LineItems` group attribute to represent the items in the order. Each `LineItems` entry contains an `INVENTORY_ITEM` object attribute that refers to an object of the type shown in Figure 5.9.

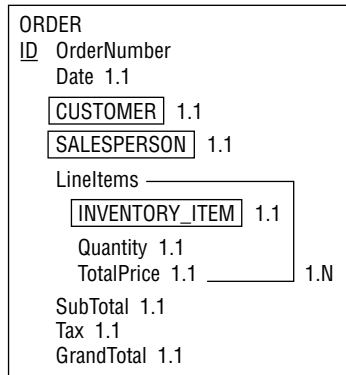


FIGURE 5.11

Association Objects

An *association object* represents a relationship between two other objects and stores extra information about the relationship.

Association objects are particularly useful for many-to-many relationships, where an object of one class can be associated with many objects of a second class, and an object of the second class can be associated with many objects of the first class.

For example, consider the `PROJECT` and `DEVELOPER` classes. A `PROJECT` may include many `DEVELOPERS` and a `DEVELOPER` may work on many `PROJECTS`, so the two classes have a many-to-many relationship. Figure 5.12 shows this relationship modeled with straightforward object attributes.

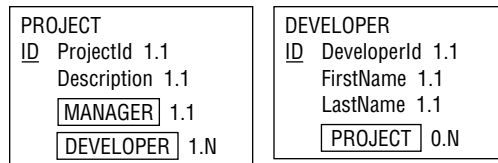


FIGURE 5.12

If this is all there is to the relationship, then this model is fine. However, if there is extra information that should be stored with the relationship, this model has no place to store that information.

For example, suppose developers play different roles in a project. A developer might be a technical lead, toolsmith, tester, writer, meeting snack supplier, or even the project's manager. In that case, there's no place to store this information in Figure 5.12. You can't place it in the `PROJECT` class because data in that class applies to the project as a whole and not to a specific developer on the project. You can't place the information in the `DEVELOPER` class because a developer might play different roles on different projects.

The solution is to create an association class to connect these classes and store the extra information. Figure 5.13 shows the new design. A `PROJECT_ROLE` object connects the `PROJECT` and `DEVELOPER` classes to represent the relationship that a particular developer has with a particular project. The `roleName` attribute stores the information about the type of role that a particular developer plays in the project (technical lead, tester, snack supplier, and so forth).

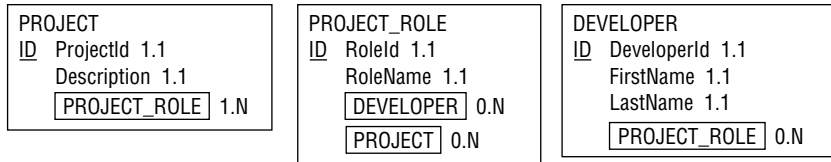


FIGURE 5.13

For a concrete example, consider Dr. Frankenstein’s famous Build-a-Friend project. The following table shows this `PROJECT` object’s attribute values:

PROJECTID	DESCRIPTION	PROJECT_ROLE
Build-a-Friend	Make a friend out of spare parts.	Role1
		Role2

The following table shows the attribute values for the two `DEVELOPER` objects:

DEVELOPERID	FIRSTNAME	LASTNAME	PROJECT_ROLE
Dr. Frankenstein	Ted	Frankenstein	Role1
Igor	Igor	Johnson	Role2

Finally, the following table shows the values for `PROJECT_ROLE` objects:

ROLEID	ROLENAME	DEVELOPER	PROJECT
Role1	Mad Scientist	Dr. Frankenstein	Make-a-Friend
Role2	Flunky	Igor	Make-a-Friend

From this data, you can figure out which developers play which roles on what projects.

Artful Associations

Suppose you're putting together a database to record *World of Warcraft* adventures. You want to remember which player participated in which adventure. You also want to know what character they played during the adventure.

Make a semantic object model to record this information.

1. Create `PLAYER` and `ADVENTURE` classes.
2. Make a `PLAYER_CHARACTER` association class to fit between `PLAYER` and `ADVENTURE`. This class should store the character's name in addition to data linking the other two classes.

How It Works

1. Create `PLAYER` and `ADVENTURE` classes.

The `PLAYER` class stores player information (`PlayerId`, `FirstName`, `LastName`, and so forth), plus an object attribute pointing to one or more `PLAYER_CHARACTER` objects. Those objects represent this player's characters in various adventures.

The `ADVENTURE` class stores adventure information (`AdventureId`, `Description`), plus another object attribute pointing to one or more `PLAYER_CHARACTER` objects. Those objects represent all of the characters in the adventure.

2. Make a `PLAYER_CHARACTER` association class to fit between `PLAYER` and `ADVENTURE`. This class should store the character's name in addition to data linking the other two classes.

The `PLAYER_CHARACTER` class stores the name of the character that the player used in this adventure. An object attribute points to the single `PLAYER` who played this character. Another object attribute points to the single `ADVENTURE` in which the player used this character.

Figure 5.14 shows the classes.

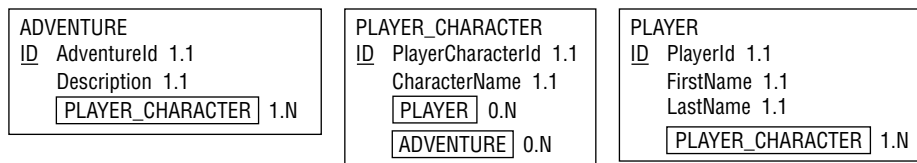


FIGURE 5.14

Inherited Objects

Sometimes, one class might share most of the characteristics of another class but with a few differences.

For example, say you've built a `CAR` class that has typical automobile attributes: `Make`, `Model`, `Year`, `NumberOfCupholders`, and so forth.

Now suppose you decide you need a `RACECAR` class. A racecar is a type of car, so it has all of the same attributes that a car has. In addition, it has some racecar-specific attributes such as `ZeroTo60Time`, `ZeroTo100Time`, `TopSpeed`, and `QuarterMileTime`. You could build a whole new class that duplicates all of the `CAR` attributes and adds the new ones. That would not only be extra work (something any self-respecting database designer should avoid), but it also wouldn't acknowledge the relationship between the two classes.

Instead you can make `RACECAR` a *subclass* or *subtype* of the `CAR` class. To denote a subclass in a semantic object model, create a `RACECAR` class that contains only the new attributes not included in `CAR`. Include an object attribute in `CAR` linking to the `RACECAR` class and using the notation `0.ST` in place of the cardinality to indicate that `RACECAR` forms an optional subtype for `CAR`. Then place an object attribute in the `RACECAR` class, linking it back to the `CAR` class and using the notation `p` in place of the cardinality to indicate that the link refers to the parent class.

Figure 5.15 shows a `CAR` class and a `RACECAR` subclass. In this case, the `RACECAR` class is said to *inherit* from the `CAR` class. `CAR` is called the *parent class*, *superclass*, or *supertype*.

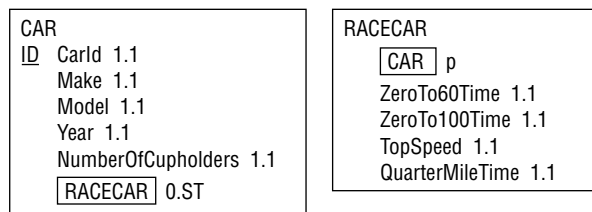


FIGURE 5.15

In more complicated models, a class can have multiple subclasses, nested subclasses, or multiple parent classes.

For example, suppose you decide that you also want to store information about motorcycles. Motorcycles and cars share some information but one isn't really a special type of the other, so you create a new `VEHICLE` class to hold the common features. You then pull the common attributes from the `CAR` class into `VEHICLE` and make both `CAR` and `MOTORCYCLE` subclasses of `VEHICLE`. In this example, you have multiple classes (`CAR` and `MOTORCYCLE`) inheriting from a common parent class (`VEHICLE`). You also have a nested class `RACECAR` inheriting from the `CAR` subclass.

Comments and Notes

Semantic object models are fairly good at capturing the basic classes involved in a project, and through object attributes they do a decent job of showing which classes are related to other classes. However, they don't capture every possible scrap of information about a project.

For example, semantic object models don't indicate an attribute's domain. There's nothing in Figure 5.15 that shows that the `CAR` class's `Make` attribute must take values from an enumerated list (Ford, GM, Yugo, DeLorean, and so forth), that `Model` must come from a list that depends on `Make`, and that `NumberOfCupholders` should be an integer between 0 and 99. (Some of the bigger SUVs may need three-digit numbers.)

For an even stranger example, suppose you build a `VOLLEYBALL_TEAM` class to represent volleyball teams. Depending on the tournament, a volleyball team might have 2, 4, or 6 players, but other values are not allowed. (Although I've seen some really weird formats including the "executive retreat" event where as many 12 people wearing slacks and dress shirts but no shoes squeeze onto the court.) A semantic object model lets you specify a minimum and maximum for the `PLAYER` object attribute, but it cannot handle allowing the specific values 2, 4, or 6.

A semantic object model also doesn't necessarily capture all of the meaning of the relationships between classes. For example, suppose you build `BAND` and `ARTIST` classes to store information about your favorite heavy metal bands. You would like to make separate fields in the `BAND` class to represent lead vocal, lead guitar, lead trombone, and other key band members, but because these are all object attributes, you need to represent them in the model as `ARTIST`. You'd really like to make `LeadVocal`, `LeadGuitar`, and `LeadTrombone` attributes that have as their domain `ARTIST` objects.

Although you cannot make those kinds of attributes, you can jot down notes saying what each of the `ARTIST` objects in the `BAND` class represents. You can add them as a footnote to the class, in a separate document, or in any other way that will make it easy for you to remember the meanings of these associations.

A NOTE ABOUT NOTES

You can also work around this problem by making an association class `BAND_MEMBER` that has a `Role` attribute in addition to `BAND` and `ARTIST` object attributes. Then, for example, you could use a `BAND_MEMBER` object to associate the `BAND` `Spiñal Tap` with the `ARTIST` `David St. Hubbins` with `Role` set to `Lead Vocal`.

Remember, the point of a semantic model (or any model for that matter) is to help you understand the problem. If the model alone doesn't capture the full scope of the problem, then add comments, notes, attachments, video clips, dioramas, and other extras to clarify. A model can only do so much, and if it's missing something, write it down. You may not need this information now to build the initial model, but you'll need it later to build the database.

ENTITY-RELATIONSHIP MODELS

An *entity-relationship diagram* (*ER diagram* or *ERD*) is another form of object model that in many ways is similar to a semantic object model. It also allows you to represent objects and their relationships, although it uses different symbols. ER diagrams also have a different focus, providing a bit more emphasis on relationships and a bit less on class structure (hence, the R in "ER diagram").

The following sections explain how to build basic ER diagrams to study the entities and relationships that define a project.

Entities, Attributes, and Identifiers

An *entity* is similar to a semantic object. It represents a specific instance of something that you want to track in the object model. Like semantic objects, an entity can be a physical thing (such as an employee, work order, or espresso maker) or a logical abstraction (such as an appointment, discussion, or excuse for being late to work).

Similar entities are grouped into *entity classes* or *entity sets*. For example, the employee entities Bowb, Phrieda, and Gnick belong to the `Employee` entity set.

Like semantic objects, entities include attributes that describe the objects that they represent.

There are a couple of different methods for drawing entity sets. In the first method, a set is contained within a rectangle. Its attributes are drawn within ellipses and attached to the set with lines. If one of the attributes is an identifier (also called a *key* or *primary key*), then its name is underlined.

Figure 5.16 shows a simple `Employee` entity set with three attributes. (Some developers write entity set names in ALL CAPS, whereas others use Mixed Case.)

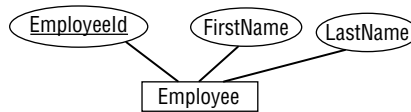


FIGURE 5.16

CURIOUS CASING Database developers (and programmers in general) use several styles for joining words together.

In Pascal case, the words are joined together with each word capitalized as in *ThisIsPascalCase*. It's called Pascal case because it was popularized by standard usage in the Pascal programming language.

Camel case is similar to Pascal case except the first word is not capitalized as in *thisIsCamelCase*. It's called camel case because it looks sort of like a camel that's low at both ends (I guess the camel is grazing or looking down) and has one or more humps in the middle. (Unlike real camels, you're not restricted to only one or two humps.)

In snake case, words are joined with underscores as in *this_is_snake_case* or *THIS_IS_SNAKE_CASE* depending on whether you want your variables to seem like they're shouting. (It's called snake case because the underscores sort of slither along the ground.)

Finally, in kebab base, words are joined with dashes as in *this-is-kebab-case* or *THIS-IS-KEBAB-CASE*, again depending on whether your variables are shouting. (As you can probably guess, it's called kebab case because the words look like they've been skewered like a kebab.)

One problem with this notation is that it takes up a lot of room. If you add all of the attributes to the `Employee` class (`EmployeeId`, `FirstName`, `LastName`, `SocialSecurityNumber`, `Street`, `Suite`, `City`, `State`, `ZipCode`, `HomePhone`, `CellPhone`, `Fax`, `Email`, and so forth), you'll get a pretty cluttered picture. If you then try to add `Department`, `Project`, `Manager`, and other classes to the picture with all of their attributes, you can quickly build an incomprehensible mess.

A second approach is to draw entity sets in a manner similar to the one used by semantic object models, and then place only the set's name in the ER diagram. Lines and other symbols, which are described shortly, connect the entity sets to show their relationships. This approach allows you greater room for listing attributes while removing them from the main ER diagram so that it can focus on relationships.

Relationships

An ER diagram indicates a relationship with a diamond containing the relationship's name. The name is usually something very descriptive such as `Contains`, `Works For`, or `RulesWithAnIronFist`, so often the relationship is perfectly understandable on its own. If the name isn't enough, you can add attributes to a relationship just as you can add them to entities: by placing the attribute in an ellipse and attaching it to the relationship with a line.

Normally entity names are nouns such as `Voter`, `Person`, `Forklift`, and `Politician`. Relationships are verbs such as `Elects`, `Drives`, and `Deceives`. When you see entities and relationships connected in an ER diagram, they appear as easy-to-read caveman or Tarzan phrases such as `Voter Elects Politician`, `Person Drives Forklift`, and `Politician Deceives Voter`.

Figure 5.17 shows the `Person Drives Forklift` relationship.



FIGURE 5.17

Note that every relation implicitly defines a reverse relation. The phrase `Person Drives Forklift` implicitly defines the relation `Forklift IsDrivenBy Person`. Usually you can figure out the relation's direction from the context. You can help by drawing the relationships from left-to-right and top-to-bottom whenever possible. (If you use a right-to-left language like Aramaic or Urdu, or a top-to-bottom language like Japanese or Korean, use your best judgment.)

I've also seen ER diagrams that include arrows above or beside a relationship to show its direction. For example, Figure 5.18 shows an ER diagram that includes three objects and two relationships. The arrows make it easier to see that `Customer Places Order` and `Shipper Ships Order`.

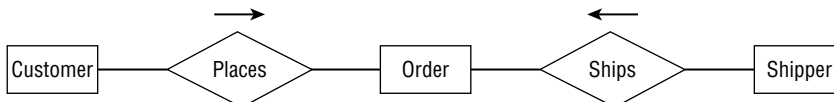


FIGURE 5.18

Cardinality

To add cardinality information, ER diagrams add one or more of three symbols to the lines leading in and out of entity sets. The three symbols are:

- **Ring**—A ring (or circle or ellipse, but strangely not a 0) means zero.
- **Line**—A short line (or dash or bar, but not a 1) means one.
- **Crow's foot**—A crow's foot (or teepee or whatever you call it) means many.

These aren't too hard to remember because the number 0 looks like a circle, the number 1 looks a line, and the crow's foot looks like several 1s.

If two of these symbols are present, they give the minimum and maximum number of entities that can be associated with the relation. For example, if the line entering an entity includes a circle and line, then zero or one of those items is associated with the relation.

For a concrete example, consider Figure 5.19. The relationship *Swallows* connects the classes *SwordSwallower* and *Sword*. The two lines beside *SwordSwallower* mean that the relationship involves between 1 and 1 *SwordSwallower*. In other words, the relationship requires exactly one *SwordSwallower*. The circle and crow's foot beside *Sword* mean that the relationship involves between 0 and many swords. That means this is a one-to-many relationship. One sword swallower swallows many swords.

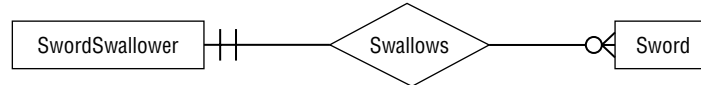


FIGURE 5.19

ER diagrams only have three symbols for representing three cardinalities: 0, 1, and many. (It reminds me of those primitive tribes that only have words for the numbers 1, 2, and many. I wonder if they played a role in developing ER diagrams?) This means you cannot specify cardinality as precisely as you can with semantic object models, which let you explicitly give upper and lower bounds.

For example, suppose you want to represent 2 to 4 jugglers juggling 5 or more flaming torches. (It's hardly juggling if two people just stand there holding four torches. Even I could do that, if they're not too heavy.) In a semantic object model, you would give the jugglers the cardinality 2..4 and the torches 5..N. Because ER diagrams don't have symbols for 2, 4, or 5, you're out of luck if you're building an ER diagram.

But wait! The point of these models is to gain an understanding of the system, not to rigidly follow the rules to their ridiculous conclusions, so I see no reason why you shouldn't merge the best of both systems and use ER diagrams that specify cardinality in the semantic object model style. (Yes, I know I may be cast out for such sacrilege.)

Figure 5.20 shows how I would model the jugglers. You won't find many people who use this combined notation on the Internet, so you should understand the normal ER symbols, too, but this version seems easy enough to understand.



FIGURE 5.20

Inheritance

Like a semantic object model, an ER diagram can represent inheritance. An ER diagram represents inheritance as a special relationship named `ISA` (read as “is a”) that’s drawn inside a triangle. One point of the triangle points toward the parent class. Other lines leading into the triangle attach on the triangle’s sides.

For example, a space shuttle crew contains several different kinds of astronauts, including Commander, Pilot, Mission Specialist, Payload Specialist, and Bartender. (I’m guessing about that last one.) All of these have the common crew member attributes plus additional attributes that relate to their more specialized roles. For example, a Commander, Pilot, and Mission Specialist have special NASA space training (I’ll call them “space trained”).

A Payload Specialist is a doctor, physicist, database design book author (hint, hint), or other professional who comes along for the ride to perform some specific mission such as watching spiders spin webs in microgravity.

Figure 5.21 shows one way that you might model this inheritance hierarchy in an ER diagram. The `PayloadSpecialist` inherits directly from `Astronaut`. `SpaceTrained` also inherits from `Astronaut`, although the relationship diagram probably will include only subclasses of `SpaceTrained` and not any `SpaceTrained` entities. `Commander`, `Pilot`, and `MissionSpecialist` inherit from `SpaceTrained`.

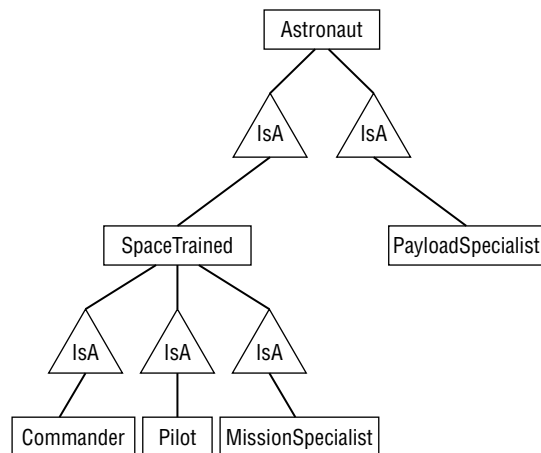


FIGURE 5.21

Sometimes, you might see the `IsA` symbol shared by more than one inherited entity. The result implies a sibling relationship that probably doesn't mean much (for example, `SpaceTrained` and `PayloadSpecialist` are related only by the fact that they inherit from a common parent entity) but it does make the diagram less cluttered.

Figure 5.22 shows the same inheritance diagram shown in Figure 5.21 but with this new notation.

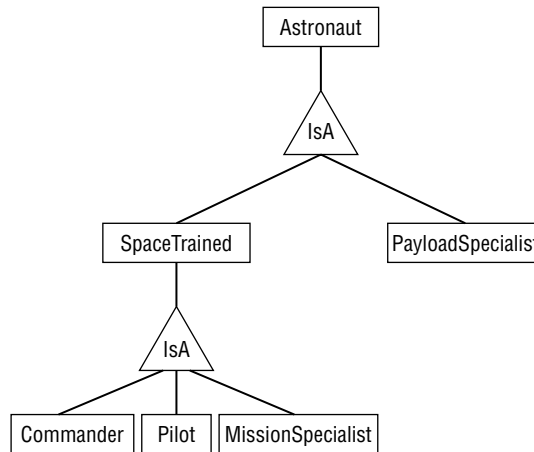


FIGURE 5.22

ER Diagrams

Create an ER diagram to represent the `Passenger`, `Driver`, and `Car` entities.

1. Make a `Person` class with `PersonId`, `FirstName`, and `LastName` fields.
2. Show `Passenger` and `Driver` inheriting from `Person`.
3. Display the relationships between the `Driver` and `Passenger` classes and the `Car` class.

How It Works

1. Make a `Person` class with `PersonId`, `FirstName`, and `LastName` fields.

Draw `Person` in a rectangle. Attach ellipses holding `PersonId` (underlined because it's the key), `FirstName`, and `LastName`.

2. Show `Passenger` and `Driver` inheriting from `Person`.
3. Display the relationships between the `Driver` and `Passenger` classes and the `Car` class.

Connect `Driver` with `Car` via a `Drives` relationship. This relationship must involve exactly one `Driver` and one `Car`. (This model doesn't allow backseat drivers.)

Connect `Passenger` to `Car` via a `Rides In` relationship. This relationship must involve exactly one `Car` but may involve any number of `Passengers` (even none).

Figure 5.23 shows the finished diagram.

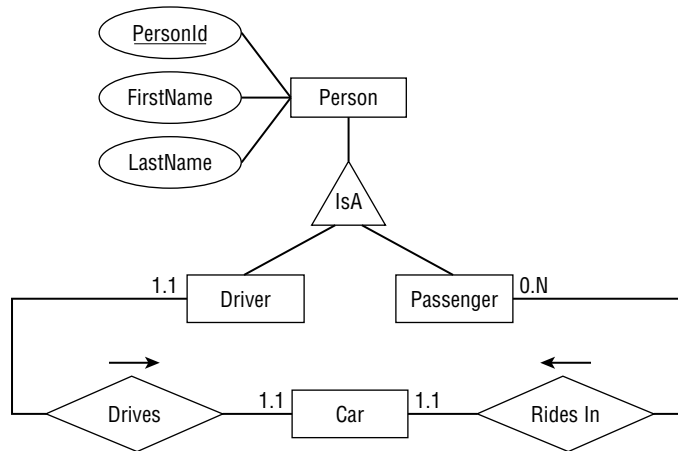


FIGURE 5.23

Additional Conventions

ER diagrams use a few other conventions to add fine shades of meaning to a model.

If every entity in an entity set *must* participate in the relationship, then the diagram includes a thick or double line. This is called a *participation constraint* because each entity must participate.

For example, consider the `Pilot Flies Airplane` relationship. During flight, every airplane must have a pilot (otherwise it’s called a “smoking pile of metal” instead of an “airplane”). This is a participation constraint on the `Airplane` entity set because all entities in that set must participate in the relationship (that is, have a pilot).

If an entity can participate in *at most one* instance of the relationship set, the diagram uses an arrow to connect the entity to the relationship. This is called a *key constraint*. For example, during flight a pilot can fly at most one airplane, so the `Pilot` entity set has a key constraint on the `Flies` relationship. (I suppose, however, a pilot could fly a drone while in the cockpit, and thus, technically fly two planes at the same time.)

If an entity must be involved in exactly one instance of a relationship set, it gets a thick or double arrow to indicate both participation and key constraints. For example, during flight an airplane must have one and only one pilot, so it would get the thick or double arrow.

Figure 5.24 shows the `Pilot Flies Airplane` relationship. Each `Pilot` can fly at most one airplane, so `Pilot` is connected to the relationship with an arrow (key constraint). A `Pilot` might sometimes be a passenger who’s not flying the airplane, so there’s no participation constraint on `Pilot` for this relationship. On the other side of the relationship, the `Airplane` must have one and

only one `Pilot`, so it gets the double arrow to indicate both key and participation constraints. The cardinalities are between 1 and 1 for both entities because there's a one-to-one relationship between `Pilot` and `Airplane` (ignoring copilots) in this relationship.



FIGURE 5.24

A *weak entity* is one that cannot be identified by its attributes alone. For example, consider a database to store submarine race results. A `Race` entity holds information about a particular race. A `Result` entity holds information about how a submarine performed in a race. The `Result` entity has attributes to store a reference to the `Race` entity, a reference to a `Sub` entity, and result information such as `Time`, `FinishPosition`, and `TorpedoesFired`.

Alone, there's no reasonable way to find a specific `Result` entity. There is no combination of `Result` attributes that really makes sense as a search key. You could search for a combination of `Time` and `FinishPosition` but that doesn't identify a particular `Result`.

Instead, you would either search for a particular `Race` and use it to find its associated `Results`, or search for a particular `Sub` and use it to find its associated `Results`.

In an ER diagram, you draw a weak entity with a thick rectangle and connect it to its identifying relationship with a thick arrow. Figure 5.25 shows the `Race`, `Sub`, and `Result` entity sets and their relationships.

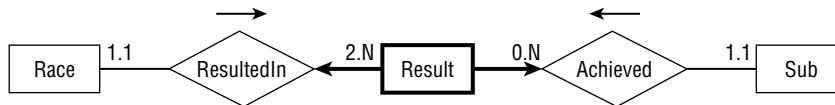


FIGURE 5.25

Comments and Notes

As is the case with semantic object models, you shouldn't be afraid to add notes, comments, scribbles, and anything else to make an ER diagram easier to understand. Annotate entity set definitions to show the domain and cardinality of an entity's attributes. Add notes to further explain confusing entities and relationships.

The purpose of an ER diagram is to help you understand a project, not to become a technically correct but uninformative doodle.

RELATIONAL MODELS

Chapter 2, "Relational Overview," explained basic concepts of relational databases such as tables, tuples, rows, and columns. (If you don't remember Chapter 2, go back and skim through it quickly to refresh your memory.)

Converting semantic object models and ER diagrams into a relational version isn't too difficult once you know how the concepts described in Chapter 2 map to those described so far in this chapter. The following table shows how key terms from Chapter 2 map to the terms used in semantic object models and ER diagrams.

THEORY	DATABASE	FILE	SOM	ER
Relation	Table	File	Class	Entity Set
Tuple	Row	Record	Object	Entity
Attribute	Column	Field	Attribute	Attribute

To convert semantic object models and ER diagrams into relational models, you simply map the classes or entity sets to tables. You then figure out which columns in the tables form the foreign key relationships among the tables.

The following sections work through examples of converting SOM and ER models into relational ones.

Converting Semantic Object Models

Consider the simple semantic object model shown in Figure 5.26. A **CUSTOMER** object has one or more **Addresses**, one or more **CONTACTS**, and one or more **ORDERS**. The **CONTACT** class contains only simple attributes. The **ORDER** class contains a simple **Date** and a group attribute to hold information about **Items** ordered.

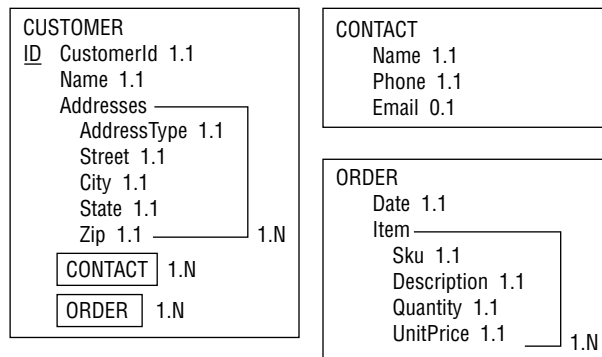


FIGURE 5.26

This model leads immediately to three relational tables: Customers, Contacts, and Orders.

If the semantic object model includes inheritance relationships, build a table for each of the object sets. Use the parent class's primary key as a foreign key in the child class to connect the two in a one-to-one relationship. For example, if **CUSTOMER** inherits from **PERSON**, then add a **PersonId** field in the Customers table to associate the corresponding records in the two tables.

The CUSTOMER class's CONTACT and ORDER attributes indicate that there should be a link from the Customers table to the Contacts and Orders tables. To do this, you can place foreign key fields in the Contacts and Orders tables to hold the CustomerId values of their corresponding Customer records. To make understanding the relational model easier, call those fields CustomerId so they match the name in the Customers table.

At this point, the relational model is practically finished. Only one little problem remains: a relational record cannot hold a potentially unlimited number of columns. In this example, a row in the Customers table cannot have an unlimited number of columns to hold multiple address values for every row. Similarly, the Orders table cannot have an unlimited number of columns to hold item data.

The solution is similar to the one used to allow a Customers record to correspond to multiple Contacts and Orders records. Create new tables to hold the repeated items. Then use foreign key fields to link those records back to their owning Customers and Orders records.

Figure 5.27 shows the resulting relational model.

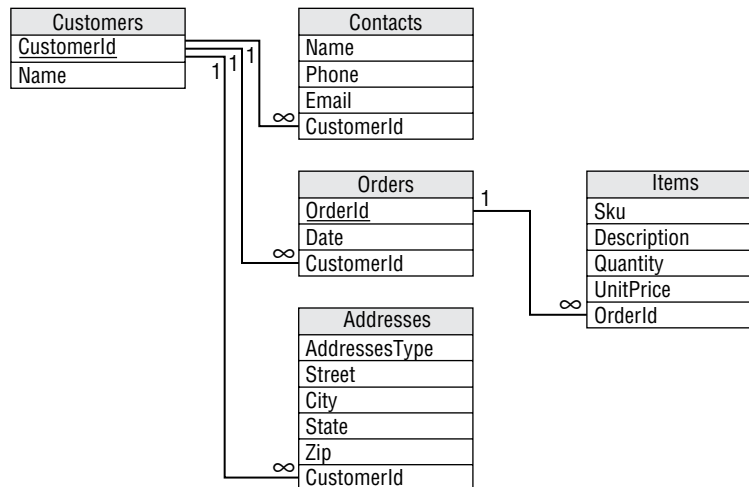


FIGURE 5.27

Each table's primary key is underlined. (Only the Customers and Orders tables have primary keys.)

Lines connect the fields that form foreign key relationships. The numbers at the ends of these lines give the numbers of items participating in the relationship (the infinity symbol ∞ means "many"). In this example, all of the relationships are one-to-many relationships.

This diagram shows relationships among tables but doesn't show much other detail. In particular, it doesn't show the fields' data types or whether they are required. If you expand each table's representation, you can add some of this information. Figure 5.28 shows the same model with columns to show the fields' data types and whether each is required.

There's only so much information you can add to one of these diagrams, however. Even this relatively simple diagram is pretty big if you add data type and required data. Often, it's better to stick to the simpler version and place additional information in separate documents.

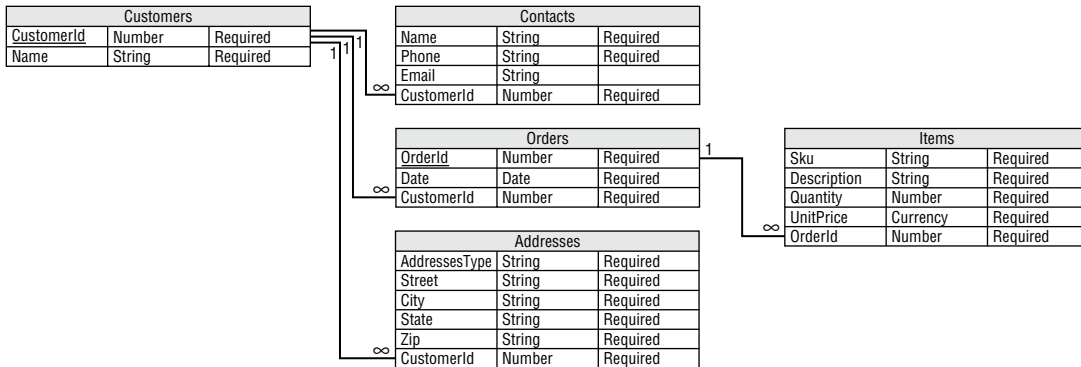


FIGURE 5.28

As is the case with all models, you should write down notes to record any information that is not fully captured by the diagram alone. For example, Figure 5.28 does not show which fields are required, their meanings (what does Sku mean, anyway?), more precise cardinalities (what if “one-to-many” should really be “one-to-four”), and so forth.

Though the figure gives data types for each of the tables’ fields, that does not necessarily completely specify the fields’ domains. For example, the Zip field should contain a 5-digit ZIP Code or a ZIP+4 Code similar to 12345-5678, UnitPrice should be a positive monetary value, and the Email field should hold a properly formatted email address such as `comments@whitehouse.gov`.

You should write down all of these and any other constraints that are not obvious from the diagram. (In case you’re curious, Sku stands for “stock keeping unit” and is pronounced “skew.” It’s like a serial number that you can use to identify products.)

Converting ER Diagrams

Figure 5.29 shows an ER diagram that covers a situation similar to the one modeled by the semantic object model shown in Figure 5.26.

Each `Customer` entity has at least one `Address`, `Contact`, and `Order`. Those are all participation constraints, so they are drawn with double lines.

The `Address`, `Contact`, and `Order` entities are accessed through their corresponding `Customer` entities. That makes them weak entities, so they are drawn with thick rectangles and they have thick arrows pointing to their identifying relationships. (If you want to allow the users to search for orders directly, perhaps by an `OrderId`, then `Order` would not be a weak entity.)

The `Order` entity must be associated with at least one `Item`, so it has a participation constraint drawn with a double line. The `Item` entity is weak, so it is drawn with a thick rectangle and uses a thick arrow to connect to its identifying relationship.

The entities in the ER diagram lead directly to the relational tables `Customers`, `Addresses`, `Contacts`, `Orders`, and `Items`.

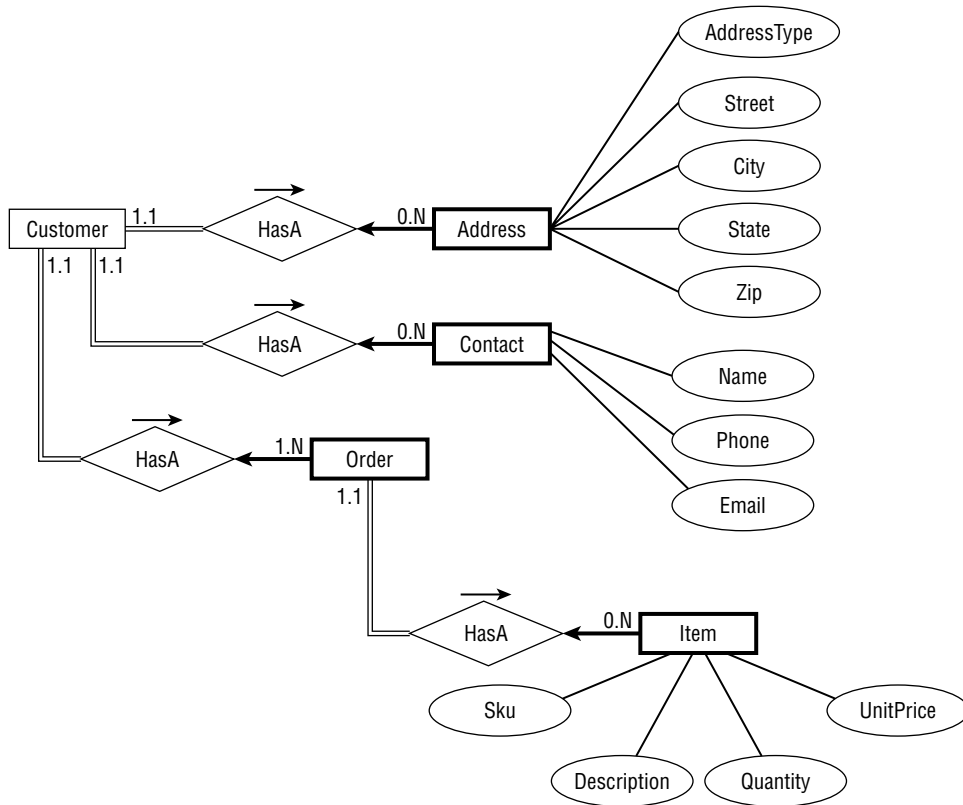


FIGURE 5.29

To connect a weak entity with its owner, make sure the owner's table has a primary key. Then add a foreign key field to the weak entity's table that refers back to the owner's primary key.

The resulting relational model is the same as the one generated by the semantic object model and is shown in Figure 5.27.

You can handle inheritance the same way you did for semantic object models. Build a table for each of the entities. Use the parent class's primary key as a foreign key in the child class to connect the two in a one-to-one relationship. For example, if `Politician` inherits from `Weasel`, then add a `WeaselId` field in the `Politicians` table to link the corresponding records in the two tables.

As is the case when translating a semantic object model into a relational model, you will need to write down any extra conditions, constraints, or other information that is not completely captured by the model. See the end of the previous section for some examples of things you might want to write down.

SUMMARY

Different kinds of models help define a problem. They identify the entities that are significant to the problem, and they clarify the relationships among those entities. You can then use the models to test your understanding of the problem and to verify that the models provide the data you need to satisfy the problem's use cases and other requirements.

This chapter explained how to build different kinds of models. In this chapter, you learned how to:

- Build user interface models to learn what kind of data the database will need to store.
- Build semantic object models to study the objects that will interact while solving the problem.
- Build entity-relationship diagrams to study the entities that are involved in the problem and to examine their interactions.
- Convert semantic object models and entity-relationship diagrams into relational models.

After you've built a relational model, you can use it to start building a database. Before you begin, however, there are several techniques that you can use to make the model more robust and efficient. The first of these techniques, extracting business rules, is described in the next chapter.

Before you move on to Chapter 6, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

EXERCISES

1. Draw a semantic object model for a small college with the classes `STUDENT`, `INSTRUCTOR`, `COURSE`, and `PROJECT`. The rules are:
 - a. All students must be enrolled in at least one course or one project (or they're dropped).
 - b. Similarly, an instructor must teach at least one course or supervise at least one project (or their funding is cut off).
 - c. A student cannot work on more than one project at a time (they're too time-consuming).
 - d. An instructor can teach any number of courses and supervise any number of projects.
 - e. A project or course must have an instructor.
 - f. A course must have at least 5 students (or it's canceled).
 - g. A project must have between 1 and 5 students.
 - h. `STUDENT` and `INSTRUCTOR` should be subclasses of a `PERSON` class that contains common elements such as name, address, and phone number.
 - i. Student data must include past courses and projects and their grades for them.

Write down any special conditions and features that the semantic object model cannot handle with its normal notation.

2. Draw two ER diagrams for the situation described in Exercise 1: one to show the inheritance relationships and one to show the main entity relationships. Write down descriptions of any constraints and any special conditions that are not represented by the diagram alone.

3. Convert either the semantic object model that you built for Exercise 1 or the ER diagram you built for Exercise 2 into a relational model.
-
4. Mike's Trikes sells tricycles. Not the little kiddie models, the giant motorized half-ton behemoths you occasionally see on the road that are somewhere between a motorcycle with an extra wheel and a car with one missing.

Draw a semantic object model for Mike with the classes `CUSTOMER`, `SALESPERSON`, `MANAGER`, `CONTRACT`, `PAYMENT`, and `SHIFT`. Use the following assumptions:

- a. `CUSTOMER` and `SALESPERSON` are subclasses of the `PERSON` class that holds contact information (name, address, phone). `MANAGER` is a subclass of `SALESPERSON`.
- b. A salesperson sells a payment contract to a customer. The salesperson gets a commission so you need to keep track of who sold the contract.
- c. A customer doesn't have a record until that customer buys a contract.
- d. `SHIFT` objects track dates and times that a salesperson works.
- e. Customers make payments that should be subtracted from the customer's balance. A `PAYMENT` object should record the payment's date and amount, and the customer who made it.
- f. You should be able to find all of the contracts that a particular salesperson sold.
- g. You should be able to find all of the contracts that a particular customer purchased. You should also be able to check the customer's current balance.

Write down any special conditions and features that the semantic object model cannot handle with its normal notation.

5. Draw two ER diagrams for the situation described in Exercise 4: one to show the inheritance relationships and one to show the main entity relationships. Write down descriptions of any constraints and any special conditions that are not represented by the diagram alone.
-
6. Convert either the semantic object model that you built for Exercise 4 or the ER diagram you built for Exercise 5 into a relational model.
-
7. Suppose you want to make a database to represent your most expensive purchases. These include your house and vehicles so you make `HOUSE` and `VEHICLE` classes. You decide to expand the model to include `CAR` and `TRUCK` classes. Then you buy a camper. Because it shares attributes with both `HOUSE` and `TRUCK`, you decide that it should inherit from both of those classes.

Draw a semantic object model showing these inheritance relations. Add a few additional non-object attributes of your choosing to each class.

8. Draw an ER diagram representing the inheritance hierarchy described in Exercise 7.

6

Extracting Business Rules

Chapter 5, “Translating User Needs into Data Models,” explained how to build models to represent the entities involved in a database project and to study the interactions among those entities. The final kind of model described in that chapter, the relational model, has a structure that closely mimics the organization of a relational database. You can easily convert a relational model into a working relational database.

Before you do, however, you should optimize the relational model to make the final database as flexible, robust, and efficient as possible. Optimizing the model now is easier than reorganizing the database later, so it’s worth taking some time to make sure you get the database design right the first time.

The first step in optimizing the database is extracting business rules. Keeping business rules separate from other database constraints and relations, at least logically, makes later changes to the database easier.

In this chapter, you will learn:

- Why business rules are important
- How to identify business rules
- How to modify a relational model to isolate business rules

After you understand business rules, you’ll be able to use them to make your databases more flexible and easier to maintain.

WHAT ARE BUSINESS RULES?

Business rules describe the objects, relationships, and actions that a business finds important and worth writing down. They include rules and policies that define how a business operates and handles its day-to-day operations. They generally help a business satisfy its goals and meet its obligations.

For example, some general business rules might be:

- The nearest clerk greets customers by saying “Welcome to Cloud Nine” when they enter the store.
- Clerks ask to see a customer’s ID when writing a check for more than \$20 or charging more than \$50. No signature is required when charging less than \$25.
- Whoever unlocks the door in the morning makes the first pot of coffee (or risks mutiny).
- Save the good scotch for the landlord.

Because this is a database design book, this chapter is only concerned with the database-related business rules. Some examples of those are:

- Don’t create a Customer record until the customer buys something and has an associated order.
- Customer records must include first and last names. (If Bono, Everlast, or Madonna buys something, get an autograph and use “Star” for the last name.)
- If a student doesn’t enroll in at least one class, change the Status field to Inactive.
- If a salesperson sells more than 10 hot tubs in one month, award a \$200 bonus.
- All Contact records must include at least one phone number or email address.
- If an order totals more than \$100 before taxes and shipping, give the customer a 10 percent discount.
- If an order totals more than \$50 before taxes and shipping, give the customer free shipping.
- Employees get a 1 percent discount. (Because we’re a people company and we value our employees!)
- If the in-stock quantity of an inventory item drops below the number of items sold during the previous month, order more.

From a database viewpoint, business rules are constraints. Some are simple constraints such as:

- All orders must include a ContactPhoneNumber.

Simple rules such as this one map easily to the features provided by a relational database. It’s easy to indicate that a field has a certain data type or that it is required (as in this case).

Other business rules may represent quite complex constraints such as:

- A student’s number of course hours plus number of project hours must be between 1 and 14.

You can implement some of these more complex rules with check constraints or foreign key constraints. Recall from Chapter 2, “Relational Overview,” that check constraints include field-level constraints that apply to a single field in a table, and table-level constraints can examine more than one field in the same record.

Still other business rules are even more complex:

- An instructor must have a combination of classes, labs, and office hours totaling at least 30 contact hours with up to one-half office hour per hour of class, 1 office hour per hour of lab, and thesis supervision counts as 2 hours.

This constraint may require you to gather data from several different tables. This kind of very complex check is probably best performed by code either in a stored procedure inside the database or in external software.

All of these rules are implemented as constraints in one form or another, whether as easy database features (requiring a field), as harder database features (check constraints and foreign keys), or in code (inside or outside of the database).

IDENTIFYING KEY BUSINESS RULES

Writing down all of the business rules is worthwhile in its own right so that you can ensure that they all are implemented somehow in the database. It's also worth categorizing the business rules so that you can build them properly.

How you implement a business rule depends not only on how tricky it is to build, but also on how likely it will be to change later. If a rule is likely to change later, then you may be better off building it by using a more complicated but more flexible method.

For example, suppose you only ship orders to states where you have a warehouse and those include Wyoming, Nebraska, Colorado, and Kansas. A business rule requires that the State field in an order's shipping address must be WY, NE, CO, or KS. You can implement this as a simple field-level check constraint in the Orders table. Three minutes' work and you're a hero! No big deal.

But now suppose you open a new warehouse in Utah. To allow for this change, you'll need to edit this check constraint. This isn't the end of the world, but this change requires that you modify the structure of the database.

Now suppose the company policy changes so some warehouses are allowed to ship to certain other states. You'll need to change the database's check constraints again to allow for the change. This still isn't the end of the world, but once more you're required to change the structure of the database to accommodate a change to a business rule.

Now consider an alternative approach. Suppose instead of making this business rule a field-level check constraint on the State field, you make it a foreign key constraint. You create a ShippingStates table and fill it with the values WY, NE, CO, and KS. Then you make the Orders table's State field a foreign key referring to the new ShippingStates table. Now the Orders table will accept only records that have a State value that is listed in the ShippingStates table.

If you need to change the states that are allowed, you only need to add or remove records from the ShippingStates table. Admittedly, the difference in difficulty between this approach and the previous one is small. The previous approach required changing the database's structure, but the new approach only requires changing the data.

Not only does changing the data take a bit less effort, but it also requires less skill. This rule implemented as a check constraint might look like this:

```
State = 'WY' Or State = 'NE' Or State = 'CO' Or State = 'KS'
```

This isn't terribly difficult code, but it is code and only someone familiar with database programming will be able to make changes to it.

Data in the States table, however, is relatively easy to understand. Even your customers can add entries to this table (possibly with a few hints from you).

Placing the validation data in a separate table also allows the users to understand it more easily. Most users would be intimidated by the previous check constraint (even if they can find it inside the database), but they can easily understand a list of allowed values in a table.

To identify these key business rules, ask yourself two questions. First, how easy is it to change a rule? If a rule is very complex, it will be difficult to change without messing it up. If implementing the rule is as simple as making a field required or not in a table, then you won't lose a huge amount of time if the customer later decides that the Lumberjacks table's PreferredAxe field isn't required after all.

Second, how likely is the rule to change? If a rule is likely to change frequently, then it's probably worth some extra planning to make changing the rule easier.

Types of rules that make good candidates for extra attention include:

- **Enumerated Values**—For example, allowed shipping states, order statuses (Pending, Approved, Shipped), and service names (Installation, Repair, Dog Washing).
- **Calculation Parameters**—For example, suppose you give free shipping on orders over \$50. Will you later change that to \$75? \$100? (Based on experience, I would say that these values are almost certain to change eventually.)
- **Validity Parameters**—For example, suppose full-time students must take 8 to 16 credits. Will we ever make this 12 to 16 credits? 8 to 20 credits? Suppose you require that all projects include between 2 and 5 students. Will you ever want to allow a single student to have a project? Or will you allow a bigger team if a group of friends wants to work together badly enough to bribe you with donuts and latte?
- **Cross-Record and Cross-Table Checks**—These kinds of checks are more complicated. For example, you might require that the date and time of a poker game be after the date the tournament started. (Although the Olympics schedules competitions before the opening ceremony. They probably use some sort of time-warp effect at international levels.)
- **Generalizable Constraints**—If you think you might need to apply a similar constraint later, you should think about generalizing the constraint and moving it out of the database proper. For example, suppose your buyer skipped a decimal point and ordered 100 sets of crampons (those spiky things that ice climbers wear on their boots) instead of 10. To move the excess inventory, you offer a \$50 bonus to any salesperson who can sell 10 pairs in a week. That's fine, but next month you might end up with an excess inventory of ice axes. After you fire your buyer, you might want to change the incentive to give a \$30 bonus to any salesperson who sells 5 ice axes. You can make these changes easier if you pull the product name or ID, number of sales, bonus amount, and duration (weekly) out into another table and then use those parameters to calculate bonuses. (And, yes, it can be hard to anticipate this sort of thing when you build the initial database. This is something more to consider when you're adding the bonuses to an existing database.)
- **Very Complicated Checks**—Some checks are so complex that it's just easier to move them into code, either as stored procedures within the database or in external code modules. For example, suppose you can only register for the course Predicate Calculus (in the Mathematics

department) if you have already taken (and passed) Propositional Calculus (in the Mathematics department) or Logic I and Logic II (in the Philosophy department). Or if you have the instructor's permission or your advisor's. You can probably implement this as a table-level check constraint, but it might be worth thinking about moving this rule somewhere else, particularly because you might be able to generalize it to handle prerequisites for other courses.

Types of rules that are usually not worth special attention and can be just implemented directly in the database include:

- **Enumerated Types with Fixed Values**—Although it might make sense to move allowed values for a State field into a new table, it probably doesn't make sense to do the same for a Handedness field. Unless you're planning to start marketing to octopi and squids, Left Handed and Right Handed (and possibly Ambidextrous) are probably the only values you'll ever need for this field.
- **Data Type Requirements**—Requiring specific data types for a field is one of the bigger advantages to using a relational database. It hardly ever makes sense to use a very generic data type such as string because you're not sure whether the field will need to be a currency amount or a date. If you are that unsure, you probably need to study this field some more or split it into multiple fields.
- **Required Values**—If a GolfRound record really needs a Caddie entry (so the golfer knows who to blame for using the 3-wood on that 124-yard par 3 hole), just make it required and worry about something more complicated.
- **Sanity Checks**—For example, all inventory items should have a price of at least 0. You might want to allow 0 cost for loss leaders (or perhaps not), but if I ever find a store that sells a product for less than nothing (i.e., pays me), I'm going down there with a dump truck and cleaning them out. (Now that I think about it, I've bought a few products that would have been overpriced at negative amounts. I might have to think a bit harder depending on what the product is.) If the sanity checks are so broad that they'll never need to be changed, just wire them in and don't worry about them.

Somewhere in the middle ground are business rules that have never changed in the past but that you cannot swear won't change in the future. They might be easy to implement as checks within the database, but there still might be some advantage to extracting them to accommodate changes.

For example, suppose you require that Resident Advisors (RAs) have passed all of their general education requirements. It's been that way for five years, but before that the rule was different. Chances are the rule won't change again, at least not for a long time, but you never know. There has been talk about exempting RAs from the writing requirement ('cause riting ain't emportunt enuff).

In cases such as this one, you need to rely on the judgment of those who make the rules. (Then when the unexpected happens, you can blame them.)

So, write down all of the business rules you can discover. Include the domains of every field and any simple bounds checks such as $\text{Price} > 0$ in addition to more complicated rules.

Group the rules by how likely they are to change and how hard they would be to change. Then take a closer look at the ones that are likely to change and that will be hard to change and see if you shouldn't pull them out of the database's structure.

Find the Business Rules

Consider this partial list of business rules for a custom woodworking shop:

- Accept no new orders if the customer has an unpaid balance on completed work.
- All customer records must include daytime and evening phone numbers even if they are the same.
- Always wear proper eye protection.
- Clean the shop thoroughly at the end of the day.
- Create a customer record when the customer places their first order.
- Don't use nonportable power tools alone.
- Give a 10 percent discount if the customer pays in full in advance.
- If there is less than 1 pound of standard size screws, add them to the reorder list.
- If we have fewer than 3 bottles of glue, add glue to the reorder list.
- Leave no power tool plugged in when not in use, even for a minute.
- No customer can ever have an outstanding balance greater than \$10,000.
- No customers allowed in the painting area.
- Order 25 percent extra material for stock items.
- Replace a tool if you won't need it in the next hour. Replace all tools at the end of the day.
- Require half of an order's payment up front (so we can buy materials) if the order is more than \$1,000.
- Walt is not allowed to use the nail gun. Ever! We all know why.
- When we have fewer than 2 pounds of standard size nails, add them to the reorder list.

For this exercise, do the following:

1. Identify the database-related rules and indicate when they would apply.
2. Identify the rules that are simple or that seem unlikely to change so they can be built into the database.
3. Identify the rules that might change or that are complicated enough to deserve special attention.

How It Works

1. The following are the database-related rules:
 - *Accept no new orders if the customer has an unpaid balance on completed work.* When the customer places a new order, see if that customer has an unpaid balance on a completed order and require payment if so.
 - *All customer records must include daytime and evening phone numbers* even if they are the same. When a customer places a new order, be sure to get these numbers.

- *Create a customer record when the customer places their first order.* When the customer places a new order, try to look up the customer's record. If there is no record, create one.
- *Give a 10 percent discount if the customer pays in full in advance.* When the customer places a new order, give this discount if they pay in advance. (You should also mention the discount at this time.)
- *If there is less than 1 pound of standard size screws, add them to the reorder list.* When we use screws, check this. (In practice, we'll probably check weekly or whenever we notice we're running low.)
- *If we have fewer than 3 bottles of glue, add glue to the reorder list.* When we use up a bottle of glue, check this.
- *No customer can ever have an outstanding balance greater than \$10,000.* When the user places a new order, check the outstanding balance and place the order on hold until the balance is under \$10,000. Also when we receive a payment, check for orders on hold so we can release them if the customer's balance is low enough.
- *Order 25 percent extra material for stock items.* When ordering supplies, see if an item is a stock item (one that we use a lot) and if so add 25 percent to the order.
- *Require half of an order's payment up front (so we can buy materials) if the order is more than \$1,000.* When the user places a new order, check the cost and require this payment if necessary.
- *When we have fewer than 2 pounds of standard size nails, add them to the reorder list.* When we use nails, check this. (In practice, we'll probably check weekly or whenever we notice we're running low.)

All of the other business rules (such as not using nonportable tools alone and keeping Walt away from the nail gun) may be important rules, but the program can't do much to enforce them so they're not database-related.

2. The following rules are simple or seem unlikely to change, so they can be built into the database:

- *Accept no new orders if the customer has an unpaid balance on completed work.* This seems unambiguous and unlikely to change, although if we need an exception mechanism (for brother-in-law Frank), then this rule cannot be built into the database's structure.
- *All customer records must include daytime and evening phone numbers* even if they are the same. This seems unambiguous and unlikely to change. (Will we ever need more than two phone numbers?)
- *Create a customer record when the customer places their first order.* Again, this seems unambiguous and unlikely to change.

3. The following rules seem likely to change or are complicated enough to deserve special attention:

- *Give a 10 percent discount if the customer pays in full in advance.* The parameter "10 percent" might change.
- *If there is less than 1 pound of standard size screws, add them to the reorder list.* The parameter "1 pound" might change.

- *If we have fewer than 3 bottles of glue, add glue to the reorder list.* The parameter “3 bottles” might change.
- *No customer can ever have an outstanding balance greater than \$10,000.* The parameter “\$10,000” might change. (Do we need an exception mechanism?)
- *Order 25 percent extra material for stock items.* The parameter “25 percent” might change.
- *Require half of an order’s payment up front (so we can buy materials) if the order is more than \$1,000.* The parameters “half” and “\$1,000” might change.
- *When we have fewer than 2 pounds of standard size nails, add them to the reorder list.* The parameter “2 pounds” might change.

A few of these rules follow the pattern, “If we have less than X, then reorder.” It might be worthwhile to generalize this rule and apply it to all inventory items. The item’s record would have fields `ReorderWhen` (to indicate the quantity on hand that triggers a supply order) and `ReorderQuantity` (to indicate how much to order).

KAN DO!

This idea of using a value to trigger reorders is part of the Kanban manufacturing process. See [https://en.wikipedia.org/wiki/Kanban#Kanban_\(cards\)](https://en.wikipedia.org/wiki/Kanban#Kanban_(cards)) for details.

EXTRACTING KEY BUSINESS RULES

Now that you’ve identified the business rules that will be tricky to implement within the database or that might change frequently, pull them out of the database. There are a couple of standard approaches for doing that.

First, if the rule is a validation list, convert it into a foreign key constraint. Shipping to a set of specific states is the perfect example. Simply create a `States` table, enter the allowed states, and then make the `Orders` table’s `State` field be a foreign key referring to the `States` table.

Second, if the rule is a fairly straightforward calculation with parameters that may change, pull the parameters out and place them in a table. For example, if you want to give salespeople who sell at least \$250,000 worth of cars in a month a \$5 bonus, pull the parameters \$250,000 and \$5 out and put them in a table. In some businesses, you might even want to pull out the duration one month.

I’ve written several applications that had a special `Parameters` table containing all sorts of oddball parameters that were used to perform calculations, check constraints, and otherwise determine the system’s behavior. The table had two fields: `Name` and `Value`. To see if a salesperson should get the bonus, you would look up the record with the `Name` `BonusSales`, get the corresponding `Value`, and see if their sales totaled at least that much. If so, you would look up the `BonusAward` parameter, get its `Value`, and give the salesperson that big a bonus. (This approach works particularly well when program code performs the checks. When the program starts, it can load all of the parameters into a collection. Later, it can look up values in the collection without hitting the database.)

Third, if a calculation is complicated, extract it into code. That doesn't necessarily mean you need to write the code in C++, Python, C#, Ada, or the latest programming language flavor-of-the-month. Many database products can store and execute stored procedures. A *stored procedure* can select and iterate through records, perform calculations, make comparisons, and do just about anything that a full-fledged programming language can.

So, what's the point of moving checks into a stored procedure? Partly it's a matter of perception. Pulling the check out of the database's table structure and making it a stored procedure separates it logically from the tables. That makes it easier to divide up maintenance work on the database into structural work and programming work.

STORED EFFICIENCIES

Stored procedures have some other advantages as well. For example, a stored procedure might need to examine many records to produce a simple result. In that case, it can perform its calculations on the database server and return just its result so that the user's computer doesn't need to examine all of the records. That not only reduces the load on that user's computer, but it also reduces network traffic.

Stored procedures can also be precompiled, so they may be faster than executing a query.

You can also give access to a stored procedure that you wouldn't give to a user. For example, the user can call the procedure and get the result without viewing all of the records, which might contain sensitive data.

Of course, you can also build the check into code written in a traditional programming language. You might be able to invoke that code from the database, or you might use it in the project's user interface.

Finally, if you have a rule that you might want to generalize, well, you're going to have to use your judgment and imagination. For example, suppose an author of a database design book earns a 5 percent royalty on the first 5,000 copies sold, 7 percent on the next 5,000, and 10 percent on any copies after that. You could code those numbers into a stored procedure to calculate royalties but then later, when Steven Spielberg turns the book into a blockbuster movie, you better believe the author will want better terms for the sequel!

Rather than writing these values into the code, place them in a table. In this case, those values are associated with a particular record in the Books table. You might want more or less than three percentage values for different royalty points, so you'll need to pull the values into their own table (in ER diagram terms, the new table will be a weak entity with an identifying relationship to the Books table).

Figure 6.1 shows a tiny part of the relational model for this database. To find the royalty rates for a particular book, you would look up the RoyaltyRates records for that book's BookId.

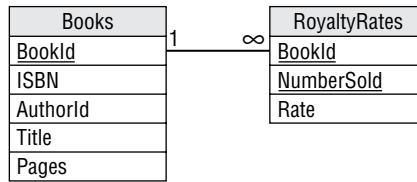


FIGURE 6.1

Now, it will be a little more work calculating royalty payments than before (although you can still hide the details in a stored procedure if you like), but it is easy to create new royalty schedules for future books.

MULTI-TIER APPLICATIONS

A *multi-tier application* uses several different layers to handle different data-related tasks. The most common form of multi-tier application uses three tiers. (The tiers are also often called layers, so you'll hear talk of three-layer systems.)

The first tier (often called the *user interface tier* or *user interface layer*) is the user interface. It displays data and lets the user manipulate it. It might perform some basic data validation such as ensuring that required fields are filled in and that numeric values are actually numbers, but it doesn't implement complicated business rules.

The third tier (often called the *data tier* or *database tier* or *layer*) is the database. It stores the data with as few restrictions as possible. Normally, it provides basic validation (NumberOfRockets is required, MaximumTwistyness must be between 0.0 and 1.0) but it doesn't implement complicated business rules, either.

The middle tier (often called the *business tier* or *business layer*) is a service layer that moves data between the first and third tiers. This is the tier that implements all of the business rules. When the user interface tier tries to send data back to the database, the middle tier verifies that the data satisfies the business rules and either sends the data to the data tier or complains to the user interface tier. When it fetches data from the database, the middle tier may also perform calculations on the data to create derived values to forward to the user interface tier. For example, the database might store temperature in degrees Celsius and the middle tier might convert that into degrees Fahrenheit or degrees Kelvin for the user interface to display.

Figure 6.2 shows the three-tier architecture graphically.

The main goal of a multi-tier architecture is to increase flexibility. The user interface and database tiers can work relatively independently while the middle tier provides any necessary translation. For example, if the user interface changes so a particular value must be displayed differently (perhaps in a dropdown instead of in a text box), it can make that change without requiring any changes to the database. If the database must change how a value is stored (perhaps as a string Small/Medium/Large instead of as a numeric size code), the user interface doesn't need to know about it. The middle tier might need to be adjusted to handle any differences, but the first and third tiers are isolated from each other.

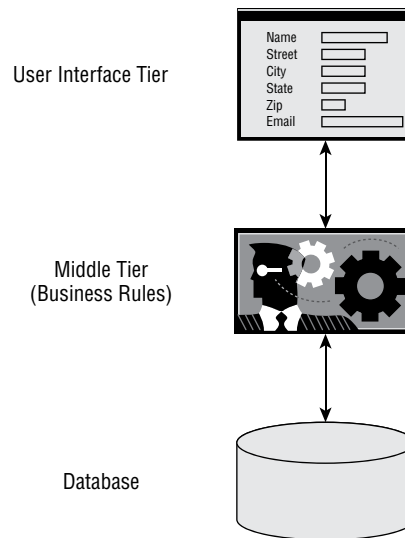


FIGURE 6.2

The middle tier also concentrates most of the business logic. The user interface and database perform basic validations, but the middle tier does all of the heavy lifting.

Another advantage of multi-tier systems is that the tiers can run on different computers. The database might run on a computer at corporate headquarters, the middle-tier libraries might run on a second computer (or even be split across two other computers), and the user interface can run on many users' computers. Or all three tiers might run on the same computer. Separating the tiers lets you shuffle them around to fit your computing environment. In a modern cloud environment, some of the pieces may run on a cloud service provider's computers and you might not even know where they are physically.

In practice, there's some benefit to placing at least some checks in the database tier so, if there's a problem in the rest of the application, the database has the final say. For example, if the user interface contains an obscure bug so customers who order more than 999 pencils on leap year day are charged \$-32,767, then the database can save the day by refusing that obviously harebrained price.

There's also some value to placing basic checks in the user interface so the application doesn't need to perform unnecessary round-trips to the database. For example, it doesn't make a lot of sense to ship an entire order's data across the network to the corporate database only to have it rejected because the order is for -10 buggy whips. The user interface should be smart enough to know that customers cannot order less than zero of something.

Adding validations in both the user interface and the database requires some redundancy, but it's worth it. (Also notice that it allows the user interface developers and database programmers to do their work separately so they can work in parallel.)

Although multi-tier architecture is far outside the scope of this book, it's worth knowing a little about it so you understand that there's another benefit to extracting complex business rules from the

database's table structure. Even if you implement those rules in stored procedures within the database, you still get some of the benefits of a logical separation and flexibility almost as if you had a hidden extra tier.

Multi-Tier Applications

Consider the following database-related business rules:

- All Customers records must include at least one associated Orders record.
- All Orders records must include at least one associated OrderItems record.
- In a Customers record, Name, Street, City, State, and Zip are required.
- In an Orders record, Customer, Date, and DueDate are required.
- In an OrderItems record, Item and Quantity are required.
- In an OrderItems record, Quantity must be at least 1.
- In an OrderItems record, Quantity must normally (99 percent of the time) be no greater than 100. In very rare circumstances, it might be greater.
- Only one order can be assigned a particular DueDate. (We have special products that take an entire day to install and only enough staff to handle one installation per day.)

Decide where each of these rules should be implemented in a three-tier application.

1. Identify the rules that should be implemented in the database's structure.
2. Identify the rules that should be implemented in the middle tier.
3. Identify the rules that should be implemented in the user interface.

Note that there will be some overlap. For example, the user interface may validate a required field to avoid a round-trip to the database if the field is missing.

How It Works

1. Identify the rules that should be implemented in the database's structure.

Whether a rule should be implemented in the database's structure depends on whether it will change. For example, if the users decide that it might be useful to create a Customers record with no associated Orders records after all, then that rule should not be implemented in the database layer. The following list shows the rules that seem extremely unlikely to change, so they can be implemented in the database's structure:

- In Customers record, Name, Street, City, State, and Zip are required.
- In an Orders record, Customer, Date, and DueDate are required.
- In an OrderItems record, Item and Quantity are required.
- In an OrderItems record, Quantity must be at least 1.
- Only one order can be assigned a particular DueDate.

The last one might be a bit iffy if the customer decides to add new staff so they can perform more than one installation per day. I'd check with the customer on this, but this rule seems to belong here.

2. Identify the rules that should be implemented in the middle tier.

The rules in the middle tier are the most complicated and the most subject to change. The following list shows the rules that should probably be implemented in the middle tier:

- All Customers records must include at least one associated Orders record.
- All Orders records must include at least one associated OrderItems record.
- In an OrderItems record, Quantity must normally (99 percent of the time) be no greater than 100. In very rare circumstances, it might be greater.

3. Identify the rules that should be implemented in the user interface.

Whether a rule should be implemented in the user interface depends mostly on the rule's simplicity and the likelihood that it will change. These rules can prevent unnecessary trips to the middle and database tiers so they can save time. However, the user interface shouldn't be unduly constrained so that we have to make changes to it when business rules change. (Hint: If it's implemented in the middle tier, there's probably a reason, so you might not want to implement it here, too.) The following list shows the rules that probably should be enforced in the user interface:

- In Customers record, Name, Street, City, State, and Zip are required.
- In an Orders record, Customer, Date, and DueDate are required.
- In an OrderItems record, Item and Quantity are required.
- In an OrderItems record, Quantity must be at least 1.

The following table summarizes the places where these rules are implemented.

RULE	DATABASE	MIDDLE TIER	UI
All Customers records must include at least one associated Orders record.		X	
All Orders records must include at least one associated OrderItems record.		X	
In Customers record, Name, Street, City, State, and Zip are required.	X		X
In an Orders record, Customer, Date, and DueDate are required.	X		X
In an OrderItems record, Item and Quantity are required.	X		X
In an OrderItems record, Quantity must be at least 1.	X		X

RULE	DATABASE	MIDDLE TIER	UI
In an OrderItems record, Quantity must normally (99% of the time) be no greater than 100. In very rare circumstances, it might be greater.		X	
Only one order can be assigned a particular DueDate.	X		

The final rule demonstrates an unusual combination: a rule that is easy to implement in the database but hard to implement in the middle tier or user interface. Only the database has ready access to every order at all times so it can see if a new order's DueDate will conflict with a different order's DueDate.

If you want to move this rule to another tier, then you could create a stored procedure to perform the check and return true or false to indicate whether a new order's due date is acceptable. Then the UI or middle tier could call that procedure.

Inserting a middle tier between the user interface and the database means that you will need to write some extra code to pass messages back and forth. For example, suppose the user tries to create some data that violates a middle tier business rule. In that case, the middle tier sends an error message to the user interface and the user interface alerts the user.

Similarly, suppose the user tries to violate the database's constraints. The database sends an error code back to the middle tier, the middle tier forwards that information to the user interface, and the user interface alerts the user.

Adding a middle tier increases flexibility, but allowing the user interface and database tiers to communicate adds some overhead.

SUMMARY

Business rules are, quite simply, the rules that govern how a business runs. They cover everything from a list of acceptable attire for Casual Fridays to the schedule of performance bonuses.

As far as databases are concerned, business rules help define the data model. They define every field's domain (what values and ranges of values they can contain), whether fields are required, the fields' data types, and any special conditions on the fields.

Some rules are simple and unlikely to change, so they can be easily implemented by using the database's features. Other rules are complex or subject to occasional change. You can make changing those rules easier by separating them from the database's structure either physically or logically.

In this chapter, you learned how to:

- Understand business rules.
- Identify key business rules that may deserve special attention.
- Isolate key business rules physically or logically by extracting their data into tables, moving them into stored procedures, and moving them into a middle tier.

Separating business rules from the database's table structure is one way to make a database more efficient and flexible. The next chapter describes another important way to improve the database's behavior: normalization.

Before you move on to Chapter 7, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

EXERCISES

For Exercises 1 through 3, consider Happy Sherpa Trekking, a company that sells and rents trekking equipment (boots, backpacks, llamas, yaks). They also organize guided adventure treks. Figure 6.3 shows a relational model for that part of the business.

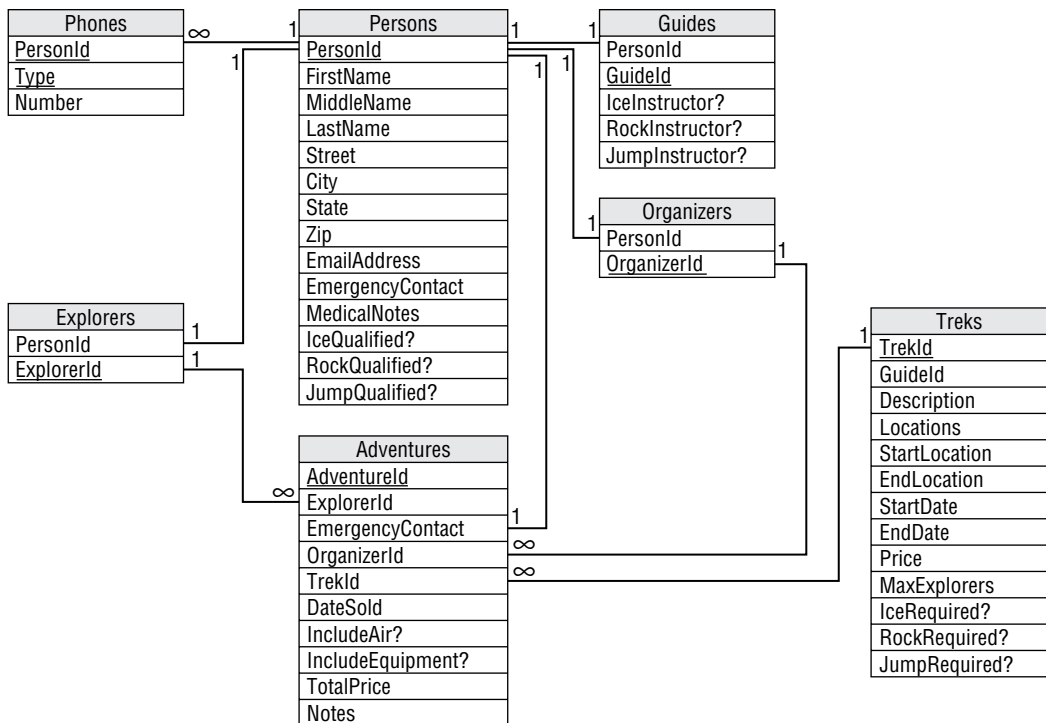


FIGURE 6.3

The company uses the following terminology to try to get everyone in an Indiana Jones mood:

- **Explorer**—A customer. Must be qualified to go on a trek.
- **Adventure**—A particular explorer's trek. This is basically the contract with the customer.
- **Guide**—The person who will lead the trek. Must be qualified and an instructor for any needed skills for a particular trek.

- **Organizer**—A salesperson who sells adventures to explorers.
- **Ice/Rock/Jump**—These refer to ice climbing, rock climbing, and parachute jumping skills.
- **Qualified?**—These indicate whether an explorer or guide has training in a particular skill. For example, if IceQualified? is Yes, then this person has ice-climbing training.
- **Instructor?**—These indicate whether a guide is qualified to teach a particular skill. For example, if RockInstructor? is Yes, then this guide can teach rock climbing.
- **Required?**—These indicate whether a trek requires a particular skill. For example, if JumpRequired? is Yes, then this trek requires parachute jumping skills (for example, the popular “Parachute into the Andes and Hike Out” trek).

The company requires an emergency contact for all explorers and that contact cannot be going on the same trek (in case an avalanche takes out the whole group).

The company gives a 10 percent discount if the explorer purchases airline flights with the adventure. Similarly, the company gives a 5 percent discount if the explorer rents equipment with the adventure. (During requirements gathering, one of the company’s owners asks which promotion gives the customer the biggest discount: applying a 10 percent discount followed by a 5 percent discount, applying a 5 percent discount followed by a 10 percent discount, or adding the two and applying a 15 percent discount. What do you think?)

If the explorer purchases airfare with the adventure, the organizer uses the Notes field to record flight information such as the explorer’s starting airport and meal preferences.

1. For each of the database’s tables, make a chart to describe the table’s fields. Fill in the following columns for each field:
 - **Field**—The field’s name.
 - **Required**—Enter Yes if the field is always required, No if it is not, or ? if it is sometimes required and sometimes not.
 - **Data Type**—The field’s data type as in Integer, String, or Yes/No.
 - **Domain**—List or describe the allowed values. If the allowed values are a list that might need to change, write “List:” before the list. You will extract these values into a new table. If the value must be in another table, list the foreign field as in Persons.PersonId.
 - **Sanity Checks**—List any basic sanity checks such as MaxExplorers > 0. Remember, these just verify extremely basic information such as a price is greater than \$0.00. Don’t enter more complex checks such as looking up a value in a list. Also don’t enter data type validations such as the fact that an ID really is an ID or that a date is a date. (Note that this kind of sanity check has nothing to do with the explorers’ sanity. If it did, then we’d never get any customers for the “Tubing Over Niagara Falls” or “August in Tampa” packages.)
 2. For each of the database’s tables, list the related business rules that are unlikely to change (and that are not too horribly complicated) so they should be implemented in the table’s field or table checks. (For example, before the database creates a new Adventures record, it should verify that the corresponding trek has space available.)
-

Include any fields that can be validated against a list of values that will never change (such as State) and more complex format validations (such as phone numbers).

Do not include fields that are foreign key constraints because that validates them completely. For example, a Phones record's PersonId value must be in the Persons table so it needs no further validation.

3. List any business rules that are likely to change, that fit better in a lookup table, that are really complicated, or that are just too weird to build irrevocably into the database. Next to each, describe how you might extract that business rule from the database's structure.

4. List the new tables (and their fields) that you would create to implement these changes.

7

Normalizing Data

Chapter 6, “Extracting Business Rules,” explained how you can make a database more flexible and robust by extracting certain business rules from the database’s structure. By removing some of the more complex and changeable rules from the database’s check constraints, you make it easier to change those rules later.

Another way to make a database more flexible and robust is to “normalize” it. Normalization makes the database more able to accommodate changes in the structure of the data. It also protects the database against certain kinds of errors.

This chapter explains what normalization is and tells how you can use it to improve your database design.

In this chapter, you will learn:

- What normalization is
- What problems different types or levels of normalization address
- How to normalize a database
- How to know what level of normalization is best for your database

After you normalize your relational model, you’ll finally be ready to build the database.

WHAT IS NORMALIZATION?

Depending on how you design a relational database, it may be susceptible to all sorts of problems. For example:

- It may contain lots of duplicated data. This not only wastes space, but it also makes updating all of those duplicated values a time-consuming chore.
- It may incorrectly associate two unrelated pieces of data so you cannot delete one without deleting the other.

- It may require you to create a piece of data that shouldn't exist in order to represent another piece of data that should exist.
- It may limit the number of values that you can enter for what should be a multivalued piece of data.

In database terminology, these issues are called *anomalies*. (Anomaly is a euphemism for “problem.” I’m not sure why this needs a euphemism—I doubt the database’s feelings would be hurt by the word “problem.”)

Normalization is a process of rearranging the database to put it into a standard (normal) form that prevents these kinds of anomalies.

There are seven different levels of normalization. Each level includes those before it. For example, a database is in Third Normal Form if it is in Second Normal Form plus it satisfies some extra properties. That means if a database is at one level of normalization, then by definition it gets the advantages of the “lower” levels.

The different levels of normalization in order from weakest to strongest are:

- First normal form (1NF)
- Second normal form (2NF)
- Third normal form (3NF)
- Boyce-Codd normal form (BCNF)
- Fourth normal form (4NF)
- Fifth normal form (5NF)
- Domain/key normal form (DKNF)

A database in DKNF has amazing powers of protection against anomalies, can leap tall buildings, and has all sorts of other super-database powers.

The following sections explain the properties that a database must satisfy to officially earn one of these coveted uber-database titles. They also explain the data anomalies that each level of normalization prevents.

FIRST NORMAL FORM (1NF)

First normal form (1NF) basically says that the data is in a relational database. It's sort of the price to play the game if you want be a relational database.

Most of the properties needed to be in 1NF are enforced automatically by any reasonable relational database engine. There are a couple of extra properties added on to make the database more useful, but mostly these rules are pretty basic. The official qualifications for 1NF are:

1. Each column must have a unique name.
2. The order of the rows and columns doesn't matter.

3. Each column must have a single data type.
4. No two rows can contain identical values.
5. Each column must contain a single value.
6. Columns cannot contain repeating groups.

The first two rules basically come for free when you use a relational database product such as Postgres, MySQL, or SQL Server. All these require that you give columns different names. They also don't really care about the order of rows and columns, although when you select data you will probably want to specify the order in which it is returned for consistency's sake. For example, you might want to sort a list of returned Student records by name.

Rule 3 means that two rows cannot store different types of data in the same column. For example, the Value field in a table cannot hold a string in one row, a date in another, and a currency value in a third. This is almost a freebie because database products won't let you say, "This field should hold numbers or dates."

One way to run afoul of Rule 3 is to store values with different types converted into a common form. For example, you could store a date written as a string (such as "3/14/2027") and a number written as a string (such as "17") in a column designed to hold strings. Although this is an impressive display of your cleverness, it violates the spirit of the rule. It makes it much harder to perform queries using the field in any meaningful way. If you really need to store different kinds of data, split them apart into different columns that each holds a single kind of data. (In practice, many databases end up with just this sort of mishmash field. In particular, users often enter important data in comment or notes fields and a program must later search for values in those fields. Not the best practice but it does happen.)

Rule 4 makes sense because, if two rows *did* contain identical values, how would you tell them apart? The only reason you might be tempted to violate this rule is if you don't need to tell the rows apart. For example, suppose you fill out an order form for a pencil, some paper, and a tarantula. Oh, yeah, you also need another pencil so you add it at the end.

NOTE *This reminds me of the joke where someone wants to buy two new residents for their aquarium by mail order (this was before Internet shopping), but they don't know whether the plural of octopus is octopi or octopuses. So they write, "Dear Octopus Barn, please send me an octopus. Oh and please send me another one."*

Now the form's list of items contains two identical rows listing a pencil. You don't care that the rows are identical because the pencils are identical. In fact, all you really know is that you want two pencils. That observation leads to the solution. Instead of using two identical rows, use one row with a new Quantity field and set Quantity to 2.

Note that Rule 4 is equivalent to saying that the table can have a primary key. Recall from Chapter 2, "Relational Overview," that a primary key is a set of columns that you can use to uniquely identify

rows. If no two rows can have exactly the same values, then you must be able to pick a set of columns to uniquely identify the rows, even if it includes every column.

In fact, let's make that a new rule:

4a. Every table has a primary key.

Rule 5 is the one you might be most tempted to violate. Sometimes, a data entity includes a concept that needs multiple values. The semantic object models described in Chapter 5, “Translating User Needs into Data Models,” even let you explicitly set an attribute's cardinality so that you can make one attribute that holds multiple values.

For example, suppose you are building a recipe table and you give it the fields Name, Ingredients, Instructions, and Notes (which contains things such as “Sherri loves this recipe” and “For extra flavor, increase ants to 3 tbs”). This gives you enough information to print out a recipe, and you can easily follow it (assuming you have some talent for cooking and a garden full of ants).

However, the Ingredients, Instructions, and Notes fields contain multiple values. Two hints that this might be the case are the fact that the column names are plural and that the column values are probably broken up into subvalues by commas, periods, carriage returns, or some other delimiter.

Storing multiple values in a single field limits the usefulness of that field. For example, suppose you decide that you want to find all of your recipes that use ants as an ingredient. Because the Ingredients field contains a bunch of different values all glommed together, you cannot easily search for a particular ingredient. You might be able to search for the word “ants” within the string, but you're likely to get extraneous matches such as “currants.” You also won't be able to use indexes to make these searches in the middle of a string faster.

The solution is to break the multiple values apart, move them into a new table, and link those records back to this one with this record's primary key. For the recipe example, you would create a RecipeIngredients table with fields RecipeId, Ingredient, and Amount. Now, you can search for RecipeIngredients records where Ingredient is “ants.”

Similarly, you could make a RecipeInstructions table with fields RecipeId, StepNumber, and Instruction. The StepNumber field is necessary because you want to perform the steps in the correct order. (I've tried rearranging the steps and it just doesn't work! Baking bread before you mix the ingredients gives you a strange little brick-like puddle.) Now, you can search for Recipes records and matching RecipeInstructions records that contain the word “preheat” to see how hot the oven must be.

Note, you only need to separate a field's values if they are logically distinct for whatever purposes you will use them. For example, you might want to search for individual ingredients. It's a bit less clear that you'll need to search for particular instructions. It's even less sure that you'll want to search for specific values within a Notes field. Notes is more or less a free-format field, and it's not clear that any internal structure is useful there.

For an even more obvious example, consider an Authors table's Biography field. This field contains an author's brief biography. You could break it into sentences (or words, or even letters), but the individual sentences don't have any real context so there's little point. You will display the Biography as a whole anyway, which means there's not much benefit in chopping it up arbitrarily.

Rule 6 means you cannot have multiple columns that contain values that are not distinguishable. For example, suppose you decide that each Exterminators record should be able to hold the animals that an exterminator is qualified to remove (muskrat, ostrich, platypus, and so forth). (Don't worry, this is a humane pest control service, so the exterminators catch the critters and release them far away so they can bother someone else.)

You already know from Rule 5 that you can't just cram all of the animals into a single Critters field. Rule 6 says you also cannot create columns named Critter1, Critter2, and Critter3 to hold different animals. Instead, you need to split the values out into a new table and use the Exterminators table's primary key to link back to the main records.

Figure 7.1 shows a relational model for the recipe data. The ingredients and instructions have been moved into new tables but the Notes field remains as it was originally.

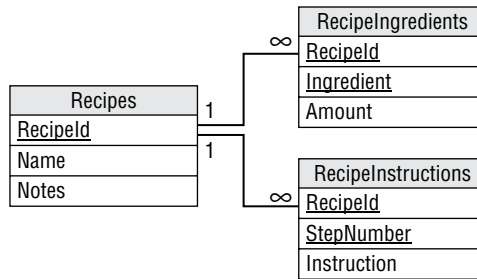


FIGURE 7.1

Arranging Data in First Normal Form

The following table contains information about airline flights. It contains data about a party of two flying from Denver to Phoenix and a party of three flying from San Diego to Los Angeles. The first two columns give the start and destination cities. The final column gives the connection cities (if any) or the number of connections. The rows are ordered so that the frequent flyer passengers are at the top, in this case in the first three rows.

CITY	CITY	CONNECTIONS
DEN	PHX	1
SAN	LAX	JFK, SEA, TPA
SAN	LAX	JFK, SEA, TPA
DEN	PHX	1
SAN	LAX	JFK, SEA, TPA

Your mission, should you decide to accept it, is to put this atrocity into first normal form:

1. Make sure every column has a unique name. If the table has this problem (sorry, I mean “anomaly”), rename those columns so they have different names.
2. Make sure that the order of the rows and columns doesn’t matter. If the order of the rows matters, add a column to record the information implied by their positions.
3. Make sure that each column holds a single data type. If a column holds more than one type of data, split it into multiple columns, one for each data type.
4. Make sure that no two rows can contain identical values. If two rows contain identical values, add a field to differentiate them.
5. Make sure that each column contains a single value. If a column holds multiple data values, split them out into a new table.
6. Make sure that multiple columns don’t contain repeating groups. In this example, you need to think about the two city fields and decide whether they contain distinguishable values.

How It Works

1. Make sure that every column has a unique name.

Rule 1 says each column must have a unique name, but this table has two columns named City. This also sort of violates the rule that the order of the columns cannot matter because we’re using the ordering to know which is the start city and which is the destination city.

To fix this problem, rename those columns to StartCity and DestinationCity. After you rename the columns, the table looks like this:

STARTCITY	DESTINATIONCITY	CONNECTIONS
DEN	PHX	1
SAN	LAX	JFK, SEA, TPA
SAN	LAX	JFK, SEA, TPA
DEN	PHX	1
SAN	LAX	JFK, SEA, TPA

2. Make sure that the order of the rows and columns doesn’t matter. If the order of the rows matters, add a column to record the information implied by their positions.

Rule 2 says the order of the rows and columns doesn’t matter. After making the first change, the order of the columns doesn’t matter anymore because you can use the column names rather than their order to tell which city is which. However, you’re using the order of the rows to determine which passengers have the highest priority (frequent flyers get the caviar and pheasant while the others get Twinkies and Spam).

To fix this, take whatever concept the row ordering represents and move it into a new column. After you make a new Priority column and explicitly list the passengers' priorities, the ordering of the rows doesn't matter because you can retrieve the original idea of who has higher priority from the new column. Now the table looks like this:

STARTCITY	DESTINATIONCITY	CONNECTIONS	PRIORITY
DEN	PHX	1	1
SAN	LAX	JFK, SEA, TPA	1
SAN	LAX	JFK, SEA, TPA	1
DEN	PHX	1	2
SAN	LAX	JFK, SEA, TPA	2

3. Make sure that each column holds a single data type. If a column holds more than one type of data, split it into multiple columns, one for each data type.

Rule 3 says that each column must have a single data type. Here the Connections column holds either a list of connecting cities or the number of connections, two different kinds of data. There are at least two reasonable solutions for this problem (at least for right now).

First, you could make two columns, ConnectingCities and NumberOfConnections, and split these values into their proper columns. This would be the better solution if you really needed both of these types of values.

In this case, however, the number of connections is just a count of the number of connecting cities. If you knew the cities, you could just count them to get the number of cities. The better solution in this case is to list the connecting cities and calculate the number of those cities when necessary. Here's the new table:

STARTCITY	DESTINATIONCITY	CONNECTIONS	PRIORITY
DEN	PHX	LON	1
SAN	LAX	JFK, SEA, TPA	1
SAN	LAX	JFK, SEA, TPA	1
DEN	PHX	LON	2
SAN	LAX	JFK, SEA, TPA	2

4. Make sure that no two rows can contain identical values. If two rows contain identical values, add a field to differentiate them.

Rule 4 says that no two rows can contain identical values. Unfortunately, this table’s second and third rows are identical. The question now becomes, “Do you care that you cannot tell these records apart?”

If you are a cold, heartless, big corporate airline behemoth and you don’t care who is flying, just that *someone* is flying, then you don’t care. In that case, add a Count field to the table and use it to track the number of identical rows. This would be the new design:

STARTCITY	DESTINATIONCITY	CONNECTIONS	PRIORITY	COUNT
DEN	PHX	LON	1	1
SAN	LAX	JFK, SEA, TPA	1	2
DEN	PHX	LON	2	1
SAN	LAX	JFK, SEA, TPA	2	1

However, if you’re a warm, friendly, mom-and-pop airline, then you do care who has which flight. In that case, what is the difference between the two identical rows? The answer is that they represent different customers, so the solution is to add a column to differentiate between the customers. If you add a CustomerId column, then you don’t need the Count column and the table becomes:

STARTCITY	DESTINATIONCITY	CONNECTIONS	PRIORITY	CUSTOMERID
DEN	PHX	LON	1	4637
SAN	LAX	JFK, SEA, TPA	1	12878
SAN	LAX	JFK, SEA, TPA	1	2871
DEN	PHX	LON	2	28718
SAN	LAX	JFK, SEA, TPA	2	9287

This works for now, but what if one of these customers wants to make the same trip more than once on different dates? In that case, the table will hold two identical records again. Again, you can ask yourself, what is the difference between the two identical rows? The answer this time is that the trips take place on different dates, so you can fix it by adding a Date column.

STARTCITY	DESTINATIONCITY	CONNECTIONS	PRIORITY	CUSTOMERID	DATE
DEN	PHX	LON	1	4637	4/1/27
SAN	LAX	JFK, SEA, TPA	1	12878	6/21/27

STARTCITY	DESTINATIONCITY	CONNECTIONS	PRIORITY	CUSTOMERID	DATE
SAN	LAX	JFK, SEA, TPA	1	2871	6/21/27
DEN	PHX	LON	2	28718	4/1/27
SAN	LAX	JFK, SEA, TPA	2	9287	6/21/27

Rule 4a says the table must have a primary key. The combination of CustomerId and Date can uniquely identify the rows. This seems like a safe combination because one customer cannot take two flights at the same time. (Customers, however, have been known to book multiple flights at the same time and then cancel all but one. It's also barely possible that someone would fly between the same two cities twice on the same day. We should probably add a flight number field, but let's just ignore this weird possibility for now.)

5. Make sure that each column contains a single value. If a column holds multiple data values, split them out into a new table.

Rule 5 says each column must contain a single value. This table's Connections column clearly violates the rule. To solve this problem, make a new Connections table and move the connection data there. To tie those records back to their original rows in this table, add columns that correspond to the primary key columns in the original table (CustomerId and Date).

That single change leads to a big problem, however. The values in the combined Connections column implicitly defined the connections' order. When you use the new table, you cannot tell which connections come before which. (Unless you use the ordering of the rows to decide which comes first, and you know that's not allowed!)

The solution is to add a ConnectionNumber field to the new table so that you can figure out how to order the connections.

Figure 7.2 shows the tables together with lines connecting the corresponding records.

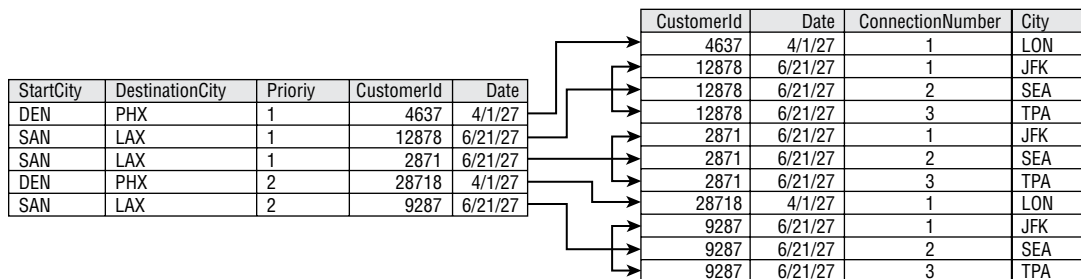


FIGURE 7.2

Figure 7.3 shows a relational model for these tables. (In the context of airline connections, that infinity symbol is kind of depressing.)

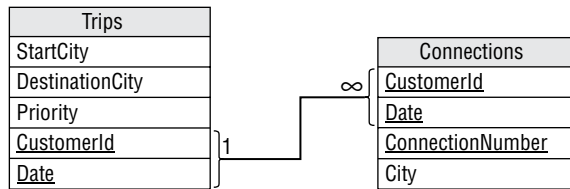


FIGURE 7.3

6. Make sure that multiple columns don't contain repeating groups. In this example, you need to think about the two city fields and decide whether they contain distinguishable values.

Rule 6 says that multiple columns don't contain repeating groups. In this example, the two city fields contain very similar types of values: cities. Unlike the exterminator example described earlier, however, these values are distinguishable. Starting and ending destination are not the same thing (unless you just fly around in a circle), and they are not the same as connecting cities. (Although I had a travel agent once who may not have fully understood the difference, judging by the fact that my rental car was reserved in the connecting city.)

Database purists would say that having two fields containing the same kind of data is a bad thing and that you should move the values into a new table.

My take on the issue is that it depends on how you are going to use the values. Will you ever want to ask, "Which customers ever visit San Jose?" If so, then having these cities in two separate fields is cumbersome because you'll have to ask the same question about each field. In fact, you'll also have to ask about the connecting cities. In this case, it might be better to move the start and destination cities out of this table.

In contrast, suppose you never ask what cities customers visit and instead ask, "Which customers start from Berlin?" or "Which customers finish in Madrid?" In that case, keeping these values in separate fields does no harm.

For now I'll leave them alone and take up this issue again in the next section.

As a quick check, you should also verify that the new table is in 1NF. If you run through the rules, you'll find that most of them are satisfied trivially, but a few are worth a moment of reflection.

Rule 4 says that no two rows can contain identical values and the table must have a primary key. Assuming no customer takes more than one trip per day, then the Trips table's CustomerId/Date fields and the Connections table's CustomerId/Date/ConnectionNumber fields make reasonable primary keys. If you need to allow customers to take more than one trip per day (which happens in the real world), then you probably need to add another TripNumber field (or perhaps a StartTime field) to both tables.

Rule 5 says that every column must contain a single value. If we split apart the values in the original table's Connections column, then the new table should be okay.

Of course we'll also need a separate table to track where the luggage goes.

SECOND NORMAL FORM (2NF)

A table is in *second normal form (2NF)* if:

1. It is in 1NF.
2. All of the non-key fields depend on all of the key fields.

To see what this means, consider the alligator wrestling schedule shown in the following table. It lists the name, class (amateur or professional), and ranking for each wrestler, together with the time when this wrestler will perform. The Time/Wrestler combination forms the table's primary key.

TIME	WRESTLER	CLASS	RANK
1:30	Annette Cart	Pro	3
1:30	Ben Jones	Pro	2
2:00	Sydney Dart	Amateur	1
2:15	Ben Jones	Pro	2
2:30	Annette Cart	Pro	3
3:30	Sydney Dart	Amateur	1
3:30	Mike Acosta	Amateur	6
3:45	Annette Cart	Pro	3

Although this table is in 1NF (don't take my word for it—verify it yourself), it is trying to do too much work all by itself, and that leads to several problems.

NOTE Notice that the *Wrestler* field contains both first and last names. This would violate 1NF if you consider those as two separate pieces of information. For this example, let's assume that you only need to display first and last name together and will never need to perform searches on last name only, for example. This is confusing enough without adding extra columns.

First, this table is vulnerable to update anomalies. An *update anomaly* occurs when a change to a row leads to inconsistent data. In this case, update anomalies are caused by the fact that this table holds a lot of repeated data. For example, suppose Sydney Dart decides to turn pro, so you update the Class

entry in the third row. Now that row is inconsistent with the Class entry in row 6 that still shows Sydney as an amateur. You'll need to update every row in the table that mentions Sydney to fix this problem.

Second, this table is susceptible to deletion anomalies. A *deletion anomaly* occurs when deleting a record can destroy information that you might need later. In this example, suppose you cancel the 3:30 match featuring Mike Acosta (because he strained a muscle while flexing in front of the mirror this morning). In that case, you lose the entire 7th record in the table, so you lose the fact that Mike is an amateur, that he's ranked 6th, and that he even exists. (Presumably, he disappears in a puff of smoke.)

Third, this table is subject to insertion anomalies. An *insertion anomaly* occurs when you cannot store certain kinds of information because it would violate the table's primary key constraints. Suppose you want to add a *new* wrestler, Nate Waffle, to the roster but you have not yet scheduled any matches for him. (Nate's actually the contest organizer's nephew so he doesn't really wrestle alligators; he just wants to be listed in the program to impress his friends.) To add Nate to this table, you would have to assign him a wrestling match, and Nate would probably have a heart attack. Similarly, you cannot create a new time for a match without assigning a wrestler to it.

Okay, I confess I pulled a fast one here. You *could* create a record for Nate that had Time set to null. That would be really bad form, however, because all of the fields that make up a primary key should have non-null values. Many databases *require* that all primary key fields not allow nulls. Because Time/Wrestler is the table's primary key, you cannot give Nate a record without assigning a Time and you're stuck.

The underlying problem is that some of the table's columns do not depend on *all* of the primary key fields. The Class and Rank fields depend on Wrestler but not on Time. Annette Cart is a professional whether she wrestles at 1:30, 2:30, or 3:45.

The solution is to pull the columns that do not depend on the *entire* primary key out of the table and put them in a new table. In this case, you could create a new Wrestlers table and move the Class and Rank fields into it. You would then add a WrestlerName field to link back to the original table.

Figure 7.4 shows a relational model for the new tables.

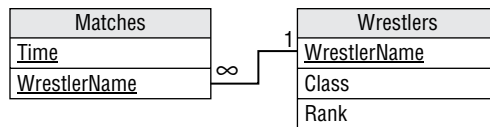


FIGURE 7.4

Figure 7.5 shows the new tables holding the original data. Here I've sorted the matches by wrestler name to make it easier to see the relationship between the two tables. (It's a mess if you sort the matches by time.)

Matches	
Time	WrestlerName
1:30	Annette Cart
2:30	Annette Cart
3:45	Annette Cart
1:30	Ben Jones
2:15	Ben Jones
3:30	Mike Acosta
2:00	Sydney Dart
3:30	Sydney Dart

Wrestlers		
WrestlerName	Class	Rank
Annette Cart	Pro	3
Ben Jones	Pro	2
Mike Acosta	Amateur	6
Sydney Dart	Amateur	1

FIGURE 7.5

The new arrangement is immune to the three anomalies described earlier. To make Sydney Dart a professional, you only need to change her `Wrestlers` record. You can cancel the 3:30 match between Mike Acosta and Hungry Bob without losing Mike’s information in the `Wrestlers` table. Finally, you can make a row for Nate in the `Wrestlers` table without making one in the `Matches` table.

You should also verify that all of the new tables satisfy the 2NF rule, “All of the non-key fields depend on all of the key fields.” The `Matches` table contains no fields that are not part of the primary key, so it satisfies this requirement trivially.

The primary key for the `Wrestlers` table is the `WrestlerName` field, and the `Class` and `Rank` fields depend directly on the value of `WrestlerName`. If you move to a different `WrestlerName`, you get different values for `Class` and `Rank`. Note that the second wrestler might have the same `Class` and `Rank`, but that would be mere coincidence. The new values belong to the new wrestler.

Intuitively, the original table had problems because it was trying to hold two kinds of information: information about matches and information about wrestlers. To fix the problem, we broke the table into two tables to hold those two kinds of information separately.

If you ensure that every table represents one single, unified concept such as wrestler or match, then the tables will be in 2NF. It’s when a table tries to play multiple roles, such as storing wrestler and match information at the same time, that it is open to data anomalies.

Arranging Data in Second Normal Form

Suppose you just graduated from the East Los Angeles Space Academy and you rush to the posting board to find your ship assignments:

CADET	POSITION	SHIP
Ash, Joshua	Fuse Tender	Frieda’s Glory
Barker, Sally	Pilot	Scrat
Barker, Sally	Arms Master	Scrat
Cumin, Bil	Cook’s Mate	Scrat

CADET	POSITION	SHIP
Farnsworth, Al	Arc Tauran Liaison	Frieda's Glory
Farnsworth, Al	Interpreter	Frieda's Glory
Major, Major	Cook's Mate	Scrat
Pickover, Bud	Captain	Athena Ascendant

This table uses the Cadet/Position combination as a primary key. Note that some cadets have more than one job.

To earn your posting as Data Minder First Class:

1. Describe the table's update, deletion, and insertion anomalies.
2. Put it in 2NF. Find any fields that don't depend on the entire primary key and move them into a new table.

How It Works

1. Describe the table's update, deletion, and insertion anomalies.

This table allows update anomalies because it contains repeated values. For example, if you change Sally Barker's Ship in row 2 to Athena Ascendant, then it would conflict with the Ship value in row 3 that says Sally is on the Scrat.

This table also allows deletion anomalies. If you delete the last row, then you no longer know that the Athena Ascendant exists. (Her crew probably won't mind a little cruising around without orders, but they'll be mad when headquarters stops sending paychecks.)

Finally, the table allows insertion anomalies because you cannot store information about a cadet without a ship or a ship without a cadet.

2. Put it in 2NF. Find any fields that don't depend on the entire primary key and move them into a new table.

This table has problems because some of the fields don't depend on the entire primary key. In particular, the Ship field depends on the Cadet (it's the ship where this cadet is assigned) but it does not depend on the Position. You could solve the problem if you could remove Position from the primary key, but we need both Cadet and Position to uniquely identify a record.

The solution is to split the table into two new tables, CadetPositions and CadetShips, and then move the ship information (which doesn't depend on the entire primary key) into the new table. Figure 7.6 shows the relational model for this new design.

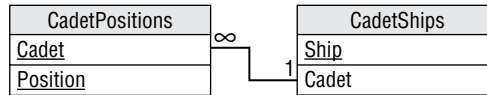


FIGURE 7.6

Figure 7.7 shows the new tables and their data.

CadetPositions		CadetShips	
Cadet	Position	Cadet	Ship
Ash, Joshua	Fuse Tender	Ash, Joshua	Frieda's Glory
Barker, Sally	Pilot	Barker, Sally	Scrat
Barker, Sally	Arms Master	Cumin, Bil	Scrat
Cumin, Bil	Cook's Mate	Farnsworth, Al	Frieda's Glory
Farnsworth, Al	Arc Tauran Liaison	Major, Major	Scrat
Farnsworth, Al	Interpreter	Pickover, Bud	Athena Ascendant
Major, Major	Cook's Mate		
Pickover, Bud	Captain		

FIGURE 7.7

You can easily verify that these tables are in 1NF.

The CadetPositions table is in 2NF trivially because every field is part of the primary key.

The CadetShips table is in 2NF because the only field that is not part of the primary key (Ship) depends on the single primary key field (Cadet).

THIRD NORMAL FORM (3NF)

A table is in *third normal form (3NF)* if:

1. It is in 2NF.
2. It contains no transitive dependencies.

A *transitive dependency* is when one non-key field's value depends on another non-key field's value. This probably sounds a bit confusing, but an example should help.

Suppose you and your friends decide to start a book club. To see what kinds of books people like, you put together the following table listing everyone's favorite books. It uses Person as the primary key. (Again, the Author field might violate 1NF if you consider it as containing multiple values: first and last name. For simplicity, and because you won't ever need to search for books by only first name or last name, we'll treat this as a single value.)

PERSON	TITLE	AUTHOR	PAGES	YEAR
Amy	Support Your Local Wizard	Duane, Diane	473	1990
Becky	Three to Dorsai!	Dickson, Gordon	532	1975
Jon	Chronicles of the Black Company	Cook, Glen	704	2007
Ken	Three to Dorsai!	Dickson, Gordon	532	1975
Wendy	Support Your Local Wizard	Duane, Diane	473	1990

You can easily show that this table is 1NF. It uses a single field as primary key, so every field in the table depends on the entire primary key, so it's also 2NF. (Each row represents that Person's favorite book, so every field must depend on that Person.)

However, this table contains a lot of duplication, so it is subject to modification anomalies. (At this point, you probably knew that!) If you discover that the Year for *Support Your Local Wizard* is wrong and fix it in row 1, it will conflict with the last row.

It's also subject to deletion anomalies (if Jon insults everyone and gets kicked out of the group so you remove the third row, you lose all of the information about *Chronicles of the Black Company*) and insertion anomalies (you cannot save Title, Author, Pages, and Year information about a book unless it's someone's favorite, and you cannot allow someone to join the group until they decide on a favorite).

The problem here is that some of the fields are related to others. In this example, Author, Pages, and Year are related to Title. If you know a book's Title, then you could look up its Author, Pages, and Year.

In this example, the primary key, Person, doesn't exactly drive the Author, Pages, and Year fields. Instead, it selects the Person's favorite Title and then Title determines the other values. This is a transitive dependency. Title depends on Person and the other fields depend on Title.

The main clue that there is a transitive dependency is that there are lots of duplicate values in the table.

You can fix this problem in a way similar to the way you put a table into 2NF: find the fields that are causing the problem and pull them into a separate table. Add an extra field to contain the original field on which those were dependent so that you can link back to the original table.

In this case, you could make a Books table to hold the Author, Pages, and Year fields. You would then add a Title field to link the new records back to the original table.

Figure 7.8 shows a relational model for the new design.

Figure 7.9 shows the new tables containing the original data.

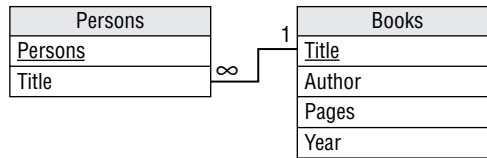


FIGURE 7.8

Persons		Books			
Person	Title	Title	Author	Pages	Year
Jon	Chronicles of the Black Company	Chronicles of the Black Company	Cook, Glen	704	2007
Amy	Support Your Local Wizard	Support Your Local Wizard	Duane, Diane	473	1990
Wendy	Support Your Local Wizard	Support Your Local Wizard	Duane, Diane	473	1990
Becky	Three to Dorsai!	Three to Dorsai!	Dickson, Gordon	532	1975
Ken	Three to Dorsai!	Three to Dorsai!	Dickson, Gordon	532	1975

FIGURE 7.9

Arranging Data in the Third Normal Form

Suppose you're helping to organize the 19,524th Jedi Olympics. (The first one was a long time ago in a galaxy far, far away.) Mostly the contestants stand around bragging about how they don't need to use violence because the Force is strong in them. You also often hear the phrases, "I was just following the Force," and "The Force made me do it."

But to keep television ratings up, there are some athletic events. The following table shows the day's schedule. It uses Contestant as the primary key.

CONTESTANT	TIME	EVENT	VENUE
Boyce Codd	2:00	Monster Mayhem	Monster Pit
General Mills	1:30	Pebble Levitating	Windy Plains Arena
Master Plethora	4:00	X-wing Lifting	Windy Plains Arena
Master Tor	1:00	Monster Mayhem	Monster Pit
Glenn	5:00	Poker	Dark Force Casino
Xzktpl Krffzk	5:00	Poker	Dark Force Casino

As part of your data processing padawan training:

1. Describe the data anomalies that this table allows.
2. Put the table in 3NF. If some fields are dependent on other non-key fields, then pull the dependent fields out into a new table.

How It Works

1. Describe the data anomalies that this table allows.

This table allows update anomalies because it contains lots of repeated values. For example, if you changed the Venue for Monster Mayhem in row 1, it would conflict with the Venue for Monster Mayhem in row 4. It also allows deletion anomalies (if Master Plethora is caught midi-chlorian doping and he drops out, you lose the fact that X-wing Lifting occurs in the Windy Plains Arena) and insertion anomalies (you cannot add a new contestant without an event or an event without a contestant).

2. Put the table in 3NF. If some fields are dependent on other non-key fields, then pull the dependent fields out into a new table.

Pull the dependent Venue field out into a new table. Add the field that it depends upon (Event) as a key to link back to the original table.

The problem is that the Event and Venue fields are dependent on each other in some manner.

The solution is to pull the dependent fields out and place them in a new table. Then add a field linking back to the field on which they depend. The next question is, “Which of these two related fields should be the one that you leave in the original table to use as the link?” Does Venue depend on Event? Or does Event depend on Venue? Or does it matter which one you consider dependent on the other?

In this example, it does matter.

Event determines Venue. In other words, each particular Event occurs in only one Venue, so if you know the Event, then you know the Venue. For example, all Pebble Levitating events occur in Windy Plains Arena. This means Venue is dependent on Event.

However, the reverse is not true. Venue does not determine Event. If you know the Venue, you do not necessarily know the Event. For example, the Windy Plains Arena is the venue for both Pebble Levitating and X-wing Lifting. This means Event is not dependent on Venue.

If there were a one-to-one mapping of Event to Venue, then each field would determine the other so you could use either as the key field. (Although to me it makes more intuitive sense to use Event as the key.)

Figure 7.10 shows the new model.

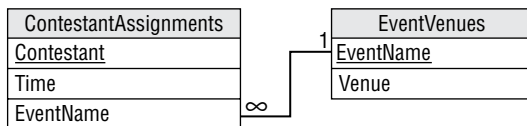


FIGURE 7.10

Figure 7.11 shows the tables containing the original data.

Contestants		
ContestantName	Time	Event
Boyce Codd	2:00	Monster Mayhem
Master Tor	1:00	Monster Mayhem
General Mills	1:30	Pebble Levitating
Glenn	5:00	Poker
Xzktpl Krffzk	5:00	Poker
Master Plethora	4:00	X-wing Lifting

EventVenues	
Event	Venue
Monster Mayhem	Monster Pit
Pebble Levitating	Windy Plains Arena
Poker	Dark Force Casino
X-wing Lifting	Windy Plains Arena

FIGURE 7.11

STOPPING AT THIRD NORMAL FORM

Many database designers stop normalizing the database at 3NF because it provides the most bang for the buck. It's fairly easy to convert a database to 3NF and that level of normalization prevents the most common data anomalies. It stores separate data separately so that you can add and remove pieces of information without destroying unrelated data. It also removes redundant data so the database isn't full of a zillion copies of the same information that waste space and make updating values difficult.

However, the database may still be vulnerable to some less common anomalies that are prevented by the more complete normalizations described in the following sections. These greater levels of normalization are rather technical and confusing. They can also lead to unnecessarily complicated data models that are hard to implement, maintain, and use. In some cases, they can give worse performance than less completely normalized designs.

Although you might not always need to use these super-normalized databases, it's still good to understand them and the problems that they prevent. Then you can decide whether those problems are potentially big enough to justify including them in your design. (Besides, they make great ice breakers at parties. "Hey everyone! Let's see who can put the guest list in 4NF the fastest!")

BOYCE-CODD NORMAL FORM (BCNF)

Even stating this one is kind of technical, so to understand it you need to know some terms.

Recall from Chapter 2, "Relational Overview," that a *superkey* is a set of fields that contain unique values. You can use a superkey to uniquely identify the records in a table.

Also recall that a *candidate key* is a minimal superkey. In other words, if you remove any of the fields from the candidate key, then it won't be a superkey anymore.

Now for a new term. A *determinant* is a field that at least partly determines the value in another field. Note that the definition of 3NF worries about fields that are dependent on another field that is *not* part of the primary key. Now, we're talking about fields that might be dependent on fields that *are* part of the primary key (or any candidate key).

A table is in *Boyce-Codd normal form (BCNF)* if:

1. It is in 3NF.
2. Every determinant is a candidate key.

For example, suppose you are attending the Wizards, Knights, and Farriers Convention hosted by three nearby castles: Castle Blue, Castle Green, and Le Château du Chevalier Rouge. Each attendee must select a track: Wizard, Knight, or Farrier. Each castle hosts three seminars, one for each track.

During the conference, you might attend seminars at any of the three castles, but you can only attend the one for your track. That means if you pick a castle, I can deduce which session you will attend there.

Here's part of the attendee schedule. The letters in parentheses show the attendee and seminar tracks to make the table easier to read and they are not really part of this data.

ATTENDEE	CASTLE	SEMINAR
Agress Profundus (w)	Green	Poisons for Fun and Profit (w)
Anabel (k)	Blue	Terrific Tilting (k)
Anabel (k)	Rouge	Clubs 'N Things (k)
Frock Smith (f)	Blue	Dealing with Difficult Destriers (f)
Lady Mismyth (w)	Green	Poisons for Fun and Profit (w)
Sten Bors (f)	Blue	Dealing with Difficult Destriers (f)
The Mighty Brak (k)	Green	Siege Engine Maintenance (k)
The Mighty Brak (k)	Rouge	Clubs 'N Things (k)

This table is susceptible to update anomalies because it contains duplicated data. If you moved the Poisons for Fun and Profit seminar to Castle Blue in the first record, then it would contradict the Castle value in row 5.

It's also vulnerable to deletion anomalies because the relationship between Castle and Seminar is stored implicitly in this table. If you deleted the second record, then you would lose the fact that the Terrific Tilting seminar is taking place in Castle Blue.

Finally, this table suffers from insertion anomalies. For example, you cannot create a record for a new seminar without assigning an attendee to it.

In short, this table has a problem because it has multiple overlapping candidate keys.

This table has two candidate keys: Attendee/Castle and Attendee/Seminar. Either of those combinations will uniquely identify a record.

The remaining combination, Castle/Seminar, cannot identify the Attendee, so it's not a candidate key.

As you have probably noticed, the Castle and Seminar fields have a dependency: Seminar determines Castle (but not vice versa). In other words, Seminar is a determinant of Castle.

This table is not in BCNF because Seminar is a determinant but is not a candidate key.

You can put this table in BCNF by pulling out the dependent data and linking it to the determinant. In this case, that means moving the Castle data into a new table and linking it back to its determinant, Seminar.

Figure 7.12 shows the new design.

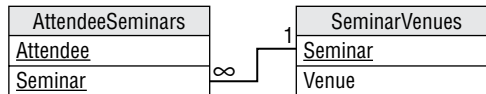


FIGURE 7.12

Figure 7.13 shows the new tables containing the original data.

AttendeeSeminars		SeminarVenues	
Attendee	Seminar	Castle	Seminar
Anabel (k)	Clubs 'N Things (k)	Rogue	Clubs 'N Things (k)
The Mighty Brak (k)	Clubs 'N Things (k)	Blue	Dealing with Difficult Destriers (f)
Frock Smith (f)	Dealing with Difficult Destriers (f)	Green	Poisons for Fun and Profit (w)
Sten Bors (f)	Dealing with Difficult Destriers (f)	Green	Siege Engine Maintenance (k)
Agress Profundus (w)	Poisons for Fun and Profit (w)	Blue	Terrific Tilting (k)
Lady Mismyth (w)	Poisons for Fun and Profit (w)		
The Mighty Brak (k)	Siege Engine Maintenance (k)		
Anabel (k)	Terrific Tilting (k)		

FIGURE 7.13

Now you can move the Poisons for Fun and Profit seminar to Castle Blue by changing a single record in the SeminarVenues table. You can delete Anabel's record for Terrific Tilting without losing the fact that Terrific Tilting takes place in Castle Blue because that information is in the SeminarVenues table. Finally, you can add a new record to the SeminarVenues table without assigning any attendees to it.

For another example, suppose you have an Employees table with columns EmployeeId, FirstName, LastName, SocialSecurityNumber, and Phone. Assume you don't need to worry about weird special cases such as roommates sharing a phone number or multiple employees with the same name.

This table has several determinants. For example, `EmployeeId` determines every other field's value. If you know an employee's ID, then all the other values are fixed. This doesn't violate BCNF because `EmployeeId` is also a candidate key.

Similarly, `SocialSecurityNumber` and `Phone` are each determinants of all of the other fields. Fortunately they, too, are candidate keys.

So far so good. Now for a stranger case. The combination `FirstName/LastName` is a determinant for all of the other fields. If you know an employee's first and last names, then the corresponding `EmployeeId`, `SocialSecurityNumber`, and `Phone` values are set. Fortunately, `FirstName/LastName` is also a candidate key, so even that doesn't break the table's BCNF-ness.

This table is in BCNF because every determinant is also a candidate key. Intuitively, the table is in BCNF because it represents a single entity: an employee.

The previous example was not in BCNF because it represented two concepts at the same time: attendees and the seminars they're attending, and the locations of the seminars. We solved that problem by splitting the table into two tables that each represented only one of those concepts.

Generally, if every table represents a single concept or entity, it will be in pretty good shape. It's when you ask a table to do too much that you run into problems.

Arranging Data in the BCNF

Consider an `EmployeeAssignments` table with the fields `EmployeeId`, `FirstName`, `LastName`, and `Project`. Each employee can be assigned to multiple projects and each project can have multiple employees. Ignore weirdnesses such as two employees having the same name.

To become a Data Wizard:

1. Explain why this table is not in BCNF. (Find a determinant that this not also a candidate key.)
2. Describe data anomalies that might befall this table.
3. Put this table in BCNF.

How It Works

1. Explain why this table is not in BCNF. (Find a determinant that this not also a candidate key.)

Intuitively, the problem with this table is that it includes two different concepts. It contains multiple pieces of employee data (`EmployeeId`, `FirstName`, `LastName`) together with employee project assignment data.

More technically, this table's candidate keys are `EmployeeId/Project` and `FirstName/LastName/Project`. (Take a few minutes to verify that these combinations specify the records uniquely, that you cannot remove any fields from them, and that the remaining combination `EmployeeId/FirstName/LastName` doesn't work.)

The problem occurs because these two candidate keys are partially overlapping. If you subtract out the overlapping field (`Project`), you get `EmployeeId` and `FirstName/LastName`. These are in the two

candidate keys because they specify the same thing: the employee. That means they are determinants of each other. Unfortunately, neither of these is a candidate key by itself, so the table is not in BCNF.

2. Describe data anomalies that might befall this table.

Suppose an employee is involved with multiple projects. If you change the employee's FirstName in one of that employee's rows, it will contradict the FirstName in the employee's other rows. This is a modification anomaly.

Suppose you delete the only record containing employee Milton Waddams. You no longer have a record of his EmployeeId. In fact, you no longer have a record of him at all. (Perhaps that's the way he got deleted from the database in the movie *Office Space*.) This is a deletion anomaly.

You also can't add an employee record without assigning the employee to a project, and you can't create a project without assigning an employee to it. These are insertion anomalies.

3. Explain how to put this table in BCNF.

The solution is to move one of the dependent fields into a new table. In this case, you could pull the FirstName and LastName fields out of the table and move them into a new EmployeeData table. Add an EmployeeId field to link the new table's records back to the original records.

Figure 7.14 shows the new model.

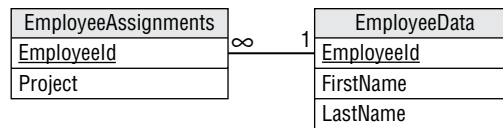


FIGURE 7.14

FOURTH NORMAL FORM (4NF)

Suppose you run a home fixit service. Each employee has a set of skills, and each drives a particular truck that contains useful special equipment. They are all qualified to use any of the equipment. The following table shows a really bad attempt to store this information.

EMPLOYEE	SKILLS	TOOLS
Gina Harris	Electric, Plumbing	Chop saw, Impact hammer
Pease Marks	Electric, Tile	Chain saw
Rick Shaw	Plumbing	Milling machine, Lathe

You should instantly notice that this table isn't even in 1NF because the Skills and Tools columns contain multiple values.

The following table shows an improved version. Here each row holds only one skill and tool.

EMPLOYEE	SKILL	TOOL
Gina Harris	Electric	Chop saw
Gina Harris	Electric	Impact hammer
Gina Harris	Plumbing	Chop saw
Gina Harris	Plumbing	Impact hammer
Pease Marks	Electric	Chain saw
Pease Marks	Tile	Chain saw
Rick Shaw	Plumbing	Milling machine
Rick Shaw	Plumbing	Lathe

Unfortunately, to capture all of the data about each employee, this table must include a lot of duplication. To record the fact that Gina Harris has the electric and plumbing skills and that her truck contains a chop saw and an impact hammer, you need four rows showing the four possible combinations of values.

In general, if an employee has S skills and T tools, then the table would need $S \times T$ rows to hold all the combinations.

This leads to the usual assortment of problems. If you modify the first row's Skill to Tile, it contradicts the second row, causing a modification anomaly. If Gina loses her impact hammer, then you must delete two rows to prevent inconsistencies. If Gina takes classes in Painting, then you need to add two new rows to cover all of the new combinations. If she later decides to add a spray gun to her toolbox, then you need to add three more rows.

Something strange is definitely going on here. And yet this table is in BCNF!

You can easily verify that it's in 1NF.

Next, note that every field must be part of the table's primary key. If you omit one field from the key, then there might be two records with the same values for all of the remaining fields and that's not allowed if those fields form the primary key.

The table is in 2NF because all of the non-key fields (there are none) depend on all the key fields (all of them). It's in 3NF because there are no transitive dependencies (every field is in the primary key so no field is dependent on a non-key field). It's in BCNF because every determinant is a candidate key. (The only determinant is Employee/Skill/Tool, which is also the only candidate key.)

In this table, the problem arises because Employee implies Skill and Employee implies Tool but Skill and Tool are independent. This situation is called an *unrelated multivalued dependency*.

A table is in *fourth normal form (4NF)* if:

1. It is in BCNF.
2. It does not contain an unrelated multivalued dependency.

A particular Employee leads to multiple Skills and for any given Skill there can be many Employees, so there is a many-to-many relationship between Employee and Skill. Similarly, there is a many-to-many relationship between Employee and Tool. (Note, there is no relationship between Skill and Tool.)

Figure 7.15 shows an ER diagram for the entities involved: Employee, Skill, and Tool.

The solution to the problem is to find the field that drives the unrelated multivalued dependency. In this case, Employee is the central field. It's in the middle of the ER diagram shown in Figure 7.15, and it forms one part of each of the many-to-many relationships.

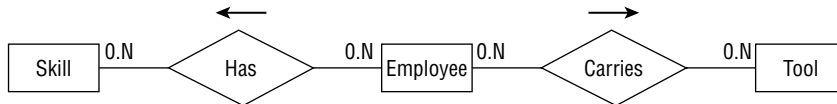


FIGURE 7.15

To fix the table, pull one of the other fields out into a new table. Add the central field (Employee) to link the new records back to the original ones.

Figure 7.16 shows the new model.

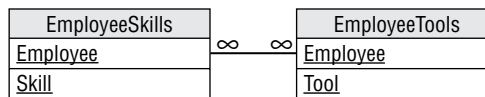


FIGURE 7.16

Figure 7.17 shows the original data in the new tables.

EmployeeSkills		EmployeeTools	
Employee	Skill	Employee	Tool
Gina Harris	Electric	Gina Harris	Chop saw
Gina Harris	Plumbing	Gina Harris	Impact hammer
Pease Marks	Electric	Pease Marks	Chain saw
Pease Marks	Tile	Rick Shaw	Milling machine
Rick Shaw	Plumbing	Rick Shaw	Lathe

FIGURE 7.17

Arranging Data in Fourth Normal Form

Consider the following artist's directory that lists artist names, genres, and shows that they will attend this year.

ARTIST	GENRE	SHOW
Ben Winsh	Metalwork	Makers of the Lost Art
Ben Winsh	Metalwork	Tribal Confusion
Harriette Laff	Textile	Fuzzy Mountain Alpaca Festival
Harriette Laff	Textile	Tribal Confusion
Harriette Laff	Sculpture	Fuzzy Mountain Alpaca Festival
Harriette Laff	Sculpture	Tribal Confusion
Mark Winslow	Sculpture	Green Mountain Arts Festival

A true database artist should be able to:

1. Identify the table's unrelated multivalued dependency.
2. Draw an ER diagram showing the table's many-to-many relationships.
3. Put the table into 4NF, and then draw a relational model for the result.
4. Display the new tables with their data.

How It Works

1. Identify the table's unrelated multivalued dependency.

In this example, there is a many-to-many relationship between Genre and Artist, and a second many-to-many relationship between Show and Artist.

2. Draw an ER diagram showing the table's many-to-many relationships.

Figure 7.18 shows an ER diagram for this situation.

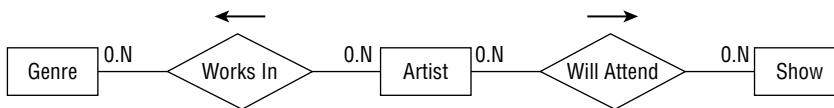


FIGURE 7.18

- Put the table into 4NF and draw a relational model for the result.

The central entity is Artist. One solution to this puzzle is to pull the Show field out into a new table and add an Artist column to link back to the original records. Figure 7.19 shows the result.

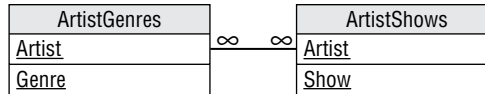


FIGURE 7.19

- Display the new tables with their data.

Figure 7.20 shows the new tables holding the original data.

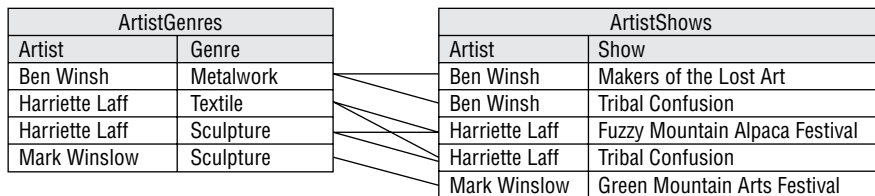


FIGURE 7.20

I want to mention two other ideas before leaving 4NF. First, you might not plan to normalize a database to this level and only worry about it when you run into problems. In this case, the problem is the multiplicative effect of multiple many-to-many relationships. The many combinations of Skill/Tool or Genre/Show fill the database with permutations of the same data, and that’s your hint that there is a problem.

The second idea is that there may be a better way to solve this problem. If you have two many-to-many relationships, you can separate them by pulling one of them out into a new table. Often, it’s better to pull them both out and make a central table to represent the common element in the relationships. For example, the solution to the Show-Artist-Genre example used Artist as a key to link the ArtistGenres and ArtistShows together.

That works, but what if you want to add more information about the artists such as their phone numbers, addresses, and online locations of their NFT marketplaces? Which table should hold that information? You could put it in either table, but then that table won’t be in 2NF anymore. (Check for yourself and maybe review the alligator wrestling example.) It’s the “one table trying to do too much” problem again.

A better solution would be to create a new Artists table to hold the new information and then link that table to *both* the ArtistGenres and ArtistShows tables. The moral of the story is, unless the concept in the middle of the multiple many-to-many relationships is a very simple linking concept with no other information, you should consider moving it into a new table.

FIFTH NORMAL FORM (5NF)

A table is in *fifth normal form (5NF, also called project-join normal form)* if:

1. It is in 4NF.
2. It contains no related multivalued dependencies.

For example, suppose you run an auto repair shop. The grease monkeys who work there may be certified to work on particular makes of vehicles (Honda, Toyota, DeLorean) and on particular types of engines (gas, diesel, hybrid, matter-antimatter).

If a grease monkey is certified for a particular make and for a particular engine, then that person *must* provide service for that make and engine (if that combination exists). For example, suppose Joe Quark is certified to repair Hondas and Diesel. Then he must be able to repair Diesel engines made by Honda.

Now, consider the following table showing which grease monkey can repair which combinations of make and engine.

GREASEMONKEY	MAKE	ENGINE
Cindy Oyle	Honda	Gas
Cindy Oyle	DeLorean	Gas
Eric Wander	Honda	Gas
Eric Wander	Honda	Hybrid
Eric Wander	DeLorean	Gas
Joe Quark	Honda	Diesel
Joe Quark	Honda	Gas
Joe Quark	Honda	Hybrid
Joe Quark	Toyota	Diesel
Joe Quark	Toyota	Gas
Joe Quark	Toyota	Hybrid

In this case, GreaseMonkey determines Make. For a given GreaseMonkey, there are certain Makes that this person can repair.

Similarly, GreaseMonkey determines Engine. For a given GreaseMonkey, there are certain Engines that this person can repair.

Up to this point, the table is very similar to the Employee/Skill/Tool table described in the previous section about 4NF. Here comes the difference.

In the Employee/Skill/Tool table, Skill and Tool were unrelated. In this new table, however, Make and Engine are related. For example, Eric Wander is certified in the Makes Honda and DeLorean. He is also certified in the Engines Gas and Hybrid. The rules state that he must repair Gas and Hybrid engines for Honda and DeLorean vehicles, if they provide those Engines. But DeLorean doesn't make a hybrid, so Eric doesn't need to service that combination.

There's the dependency between Make and Engine. While the GreaseMonkey determines the Make and Engine, Make also influences Engine.

So, how can you remove this dependency? Break the single table into three new tables that record the three different relationships: GreaseMonkey/Make, GreaseMonkey/Engine, and Make/Engine.

Figure 7.21 shows the new relational model.

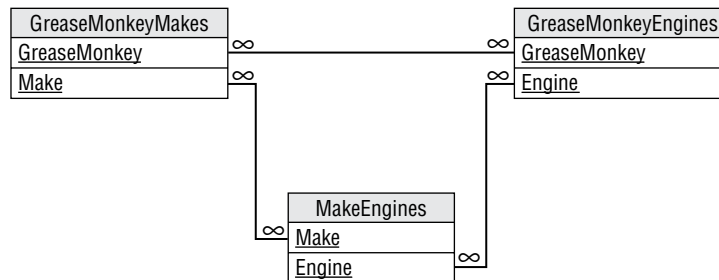


FIGURE 7.21

Figure 7.22 shows the new tables holding the original data. I haven't drawn lines connecting related records because it would make a big mess.

GreaseMonkey	Make	GreaseMonkey	Engine	Make	Engine
Cindy Oyle	Honda	Cindy Oyle	Gas	Honda	Diesel
Cindy Oyle	Hummer	Eric Wander	Gas	Honda	Gas
Eric Wander	Honda	Eric Wander	Hybrid	Honda	Hybrid
Eric Wander	Hummer	Joe Quark	Diesel	Hummer	Gas
Joe Quark	Honda	Joe Quark	Gas	Hummer	Diesel
Joe Quark	Toyota	Joe Quark	Hybrid	Toyota	Diesel
				Toyota	Gas
				Toyota	Hybrid

FIGURE 7.22

Working with the Fifth Normal Form

Remember the artist's directory from the previous sections about 4NF? The rules have changed slightly. The directory still lists artist names, genres, and shows that they will attend, but now each show allows only certain genres. Now, the Fuzzy Mountain Alpaca Festival includes only textile arts and Tribal Confusion includes only metalwork and sculpture. (Also, Ben Winsh started making sculptures.)

Here's the new schedule:

ARTIST	GENRE	SHOW
Ben Winsh	Metalwork	Makers of the Lost Art
Ben Winsh	Metalwork	Tribal Confusion
Ben Winsh	Sculpture	Makers of the Lost Art
Harriette Laff	Textile	Fuzzy Mountain Alpaca Festival
Harriette Laff	Sculpture	Tribal Confusion
Mark Winslow	Sculpture	Green Mountain Arts Festival

To prove you're a true database artist:

1. Identify the table's related multivalued dependency.
2. Draw an ER diagram showing the table's many-to-many relationships.
3. Put the table into 5NF and draw a relational model for the result.
4. Display the new tables with their data.

How It Works

1. Identify the table's related multivalued dependency.
Artist determines Genre, Artist determines Show, and Show determines Genre.
2. Draw an ER diagram showing the table's many-to-many relationships.
Figure 7.23 shows an ER diagram for this situation.
3. Put the table into 5NF and draw a relational model for the result.
The new model should make separate tables to store the relationships between Artist and Genre, Artist and Show, and Show and Genre. Figure 7.24 shows this 5NF model.
4. Display the new tables with their data.
Figure 7.25 shows the new tables holding the original data.

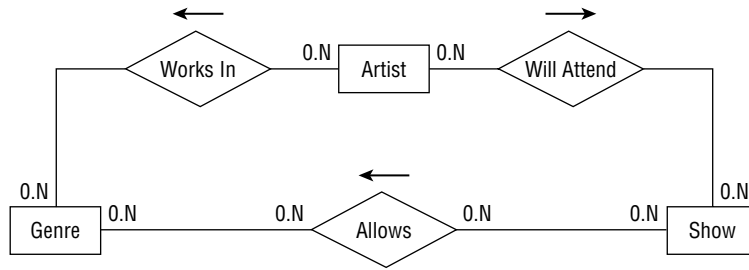


FIGURE 7.23

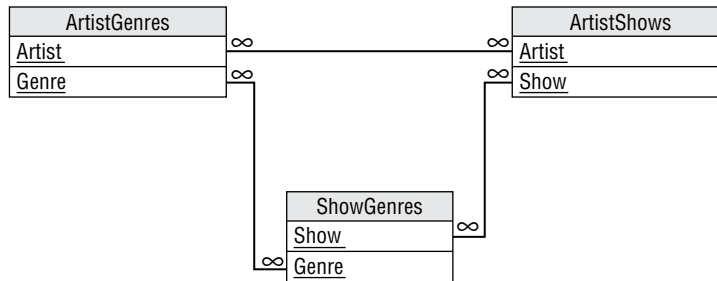


FIGURE 7.24

ArtistGenres		ArtistShows		ShowGenres	
Artist	Genre	Artist	Show	Show	Genre
Ben Winsh	Metalwork	Ben Winsh	Markers of the Lost Art	Fuzzy Mountain Alpaca Festival	Textile
Ben Winsh	Sculpture	Ben Winsh	Tribal Confusion	Green Mountain Arts Festival	Sculpture
Harriette Laff	Sculpture	Harriette Laff	Fuzzy Mountain Alpaca Festival	Markers of the Lost Art	Metalwork
Harriette Laff	Textile	Harriette Laff	Tribal Confusion	Markers of the Lost Art	Sculpture
Mark Winslow	Sculpture	Mark Winslow	Green Mountain Arts Festival	Tribal Confusion	Metalwork
				Tribal Confusion	Sculpture

FIGURE 7.25

DOMAIN/KEY NORMAL FORM (DKNF)

A table is in *domain/key normal form (DKNF)* if:

1. The table contains no constraints except domain constraints and key constraints.

In other words, a table is in DKNF if every constraint is a consequence of domain and key constraints.

Recall from Chapter 2, “Relational Overview,” that a field’s domain consists of its allowed values. A domain constraint simply means that a field has a value that is in its domain. It’s easy to check that a domain constraint is satisfied by simply examining all of the field’s values.

A key constraint means the values in the fields that make up a key are unique.

So if a table is in DKNF, then to validate all constraints on the data it is sufficient to validate the domain constraints and key constraints.

For example, consider a typical Employees table with fields FirstName, LastName, Street, City, State, and Zip. There is a hidden constraint between Street/City/State and Zip because a particular Street/City/State defines a Zip value and a Zip value defines City/State. You could validate new addresses by using a web service or with a table-level check constraint that looked up Street/City/State/Zip to make sure it was a valid combination.

This table contains a constraint that is neither a domain constraint nor a key constraint, so it is not in DKNF.

You can make the table DKNF by simply removing the Zip field. Now instead of validating a new Street/City/State/Zip, you only store the Street/City/State and you look up the address's ZIP Code whenever you need it.

It can be proven (although not by me) that a database in DKNF is immune to all data anomalies. So, why would you bother with lesser forms of normalization? Mostly because it can be confusing and difficult to build a database in DKNF.

Lesser forms of normalization also usually give good enough results for most practical database applications, so there's no need for DKNF under most circumstances.

However, it's nice to know what DKNF means so that you won't feel left out at cocktail parties when everyone else is talking about it.

Arranging Data in the Domain/Key Normal Form

Consider the following student/class assignment table:

STUDENT	CLASS	DEPARTMENT
Annette Silver	First Order Logic	Philosophy
Annette Silver	Real Analysis II	Mathematics
Janet Wilkes	Fluid Dynamics I	Physics
Janet Wilkes	Real Analysis II	Mathematics
Mark Hardaway	First Order Logic	Philosophy
Mark Hardaway	Topology I	Mathematics

This table has a dependency between Class and Department because each class is within a single department.

To ace this class:

1. Explain why this table is not in DKNF.
2. Explain how you could put the table in DKNF.
3. Draw a relational model for the result.
4. Show the table(s) you create containing the original data.

How It Works

1. Explain why this table is not in DKNF.

The dependency between Class and Department is not a domain constraint or a key constraint.

2. Explain how you could put the table in DKNF.

Pull the Department data out of the table and make a new table to relate Department and Class. Now instead of storing the Department data in the original table, you store only the Student and Class. When you need to know the Department, you look it up in the new table.

3. Draw a relational model for the result.

Figure 7.26 shows a relational model in DKNF.

4. Show the table(s) you create containing the original data.

Figure 7.27 shows the new tables holding the original data.

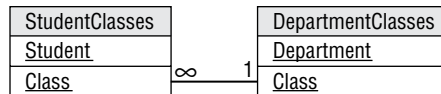


FIGURE 7.26

StudentClasses		DepartmentClasses	
Student	Class	Class	Department
Annette Silver	First Order Logic	First Order Logic	Philosophy
Mark Hardaway	First Order Logic	Fluid Dynamics I	Physics
Janet Wilkes	Fluid Dynamics I	Real Analysis II	Mathematics
Annette Silver	Real Analysis II	Topology I	Mathematics
Janet Wilkes	Real Analysis II		
Mark Hardaway	Topology I		

FIGURE 7.27

ESSENTIAL REDUNDANCY

One of the major data anomaly themes is redundancy. If a table contains a lot of redundant data, then it's probably vulnerable to data anomalies, particularly modification anomalies.

However, this is not true if the redundant data is within the keys. For example, look again at Figure 7.27. The StudentClasses table contains several repeated student names and class names. Similarly, the DepartmentClasses table contains repeated Department names. You might think that these create a modification anomaly hazard.

In fact, if you look at Figure 7.26, you'll see that all these fields are part of the tables' keys. Their repetition is necessary to represent the data that the tables hold. For example, the repeated Department values in the DepartmentClasses table are part of the data showing which departments hold which classes. Similarly, the repeated Student and Class data in the StudentClasses table is needed to represent the students' class assignments.

Although these repeated values are necessary, they do create a different potential problem. Suppose that you want to change the name of the class "Real Analysis II" to "Getting Real, the Sequel" because you think it will make more students sign up for it.

Unfortunately, you're not supposed to change the value of a primary key. If you could change the value, then you might need to update a large number of records and that could lead to problems like any other modification anomaly would.

The real problem here is that you decided that the class's name should be changed. Because you can't change key values, the solution is to use something else instead of the class's name for the key. Typically, a database will use an arbitrary ID number to represent the entity, and then move the real data (in this case the class's name) into another table. Because the ID is arbitrary, you should never need to change it.

Figure 7.28 shows one way to replace these fields used as keys with arbitrary IDs that you will never need to change.

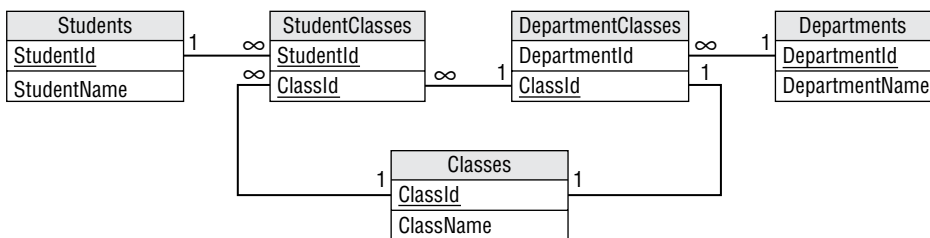


FIGURE 7.28

For bonus points, you can notice that you can combine the DepartmentClasses and Classes tables to give the simpler model shown in Figure 7.29. (The fact that they have a one-to-one relationship is a hint that you might be able to merge them.)

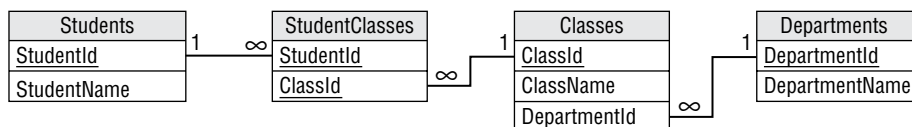


FIGURE 7.29

This is a reasonable model. Each table represents a single, well-defined entity: student, class, department, and the relationship between students and classes.

THE BEST LEVEL OF NORMALIZATION

Domain/key normal form makes a database provably immune to data anomalies, but it can be tricky to implement and it's not usually necessary. The higher levels of normalization may also require you to split tables into many pieces, making it harder and more time consuming to reassemble the pieces when you need them.

For example, the previous section explained that an Employees table containing Street, City, State, and Zip fields was not in DKNF because the Street/City/State combination duplicates some of the information in the Zip field. The solution was to remove the Zip field and to look up an employee's ZIP Code whenever it was needed. To see whether this change is reasonable, look at the costs and benefits.

The extra cost is that you must perform an extra lookup every time you need to display an employee's address with the ZIP Code. Nearly any time you display an employee's address you will need the ZIP Code, so you will probably perform this lookup a lot.

The benefit is that it makes the data less susceptible to data modification anomalies if you need to change a ZIP Code value. But how often do ZIP Codes change? On a national level, ZIP Codes change all the time, but unless you have millions of employees, your employees' ZIP Codes probably won't change all that frequently. This seems like a rare problem. It is probably better to use a table-level check constraint to validate the Street/City/State/Zip combination when the employee's data is created or modified and then leave well enough alone. On the rare occasion when a ZIP Code really does change, you can do the extra work to update all of the employees' ZIP Codes.

Often, 3NF reduces the chances of anomalies to a reasonable level without requiring confusing and complex modifications to the database's structure.

When you design your database, put it in 3NF. Then look for redundancy that could lead to anomalies. If the kinds of changes that would cause problems in your application seem like they may happen often, then you can think about using the more advanced normalizations. If those sorts of modifications seem rare, you may prefer to leave the database less normalized.

NOSQL NORMALIZATION

This chapter talks a lot about relational databases and how you normalize them, but what about NoSQL databases? (Here, I'm talking about nonrelational NoSQL databases, not relational pieces built into a "not only SQL" style database.) Should you normalize them too? Does that give you the same benefits as normalizing a relational database?

The short answer is, "No, NoSQL databases are not really normalized." That's practically the definition of a nonrelational database. They don't use tables, so the same concepts don't apply.

This is both the benefit and the curse of NoSQL databases. The benefits include:

- **Flexible data models**—You can store just about any kind of data, structured or unstructured. A relational database can store only structured data.
- **Changeable data models**—You can easily change the data model over time. You might plan to store certain kinds of data, but if your needs change, then you can add new data to the database. This may make NoSQL databases particularly attractive if you don't know what data you'll eventually need.
- **Scalable**—Because the pieces of a relational database are tied together, it can be complicated to distribute data across many servers. If the pieces of data in a nonrelational database are not related, then they can be stored just about anywhere.

Disadvantages include:

- **Poor standardization**—Different kinds of nonrelational databases store data differently and have different query languages.
- **Data anomalies**—Relational databases provide a lot of tools to prevent anomalies such as foreign key constraints and the structure of the database itself. Those are missing in NoSQL databases, so just about any kind of data anomaly is possible. If you delete the last piece of data that refers to a particular customer, then you have no knowledge that the customer exists.
- **Consistency**—NoSQL databases often focus on performance and scalability, possibly at the expense of consistency. For example, the database won't stop you from adding two pieces of data that contradict each other the way a relational database can.
- **Lack of atomicity**—There's usually no way to commit or rollback a set of transactions as a group. Someone might fetch an inconsistent version of the data while you're in the middle of changing it. (NoSQL databases do support the idea of *eventual* consistency.)

You really can't normalize these databases, but this doesn't mean that you can't have rules if you want them.

For example, suppose you have a document-oriented database that stores customer data in a JavaScript Object Notation (JSON) format. JSON lets you define elements on the fly, so you can store just about anything in a JSON document. Need to add a Birthdate field? Just do it!

That doesn't mean you can't create a rule saying there can be only one Customer element for any given customer. You might have to do some extra work to enforce that rule because the document won't do it for you.

Similarly, you can use a graph database to define a network. If you like, you can make a rule that all nodes shall have between zero and four neighbors, but you'll have to enforce that rule yourself because the database won't do it for you.

SUMMARY

Normalization is the process of rearranging a database's table designs to prevent certain kinds of data anomalies. Different levels of normalization protect against different kinds of errors.

If every table represents a single, clearly defined entity, then you've already gone a long way toward making your database safe from data anomalies. You can use normalization to further safeguard the database.

In this chapter, you learned about:

- Different kinds of anomalies that can afflict a database
- Different levels of normalization and the anomalies that they prevent
- Methods for normalizing database tables

The next chapter discusses another way that you can reduce the chances of errors entering a database. It explains design techniques other than normalization that can make it safer for a software application to manipulate the database.

Before you move on to Chapter 8, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

EXERCISES

1. Suppose a student contact list contains the fields Name, Email, Email, Phone1, PhoneType1, Phone2, PhoneType2, and MajorOrSchool. The student's preferred email address is listed in the first Email field and a backup address is stored in the second Email field. Similarly, the preferred phone number is in the Phone1 field and a backup number is in the Phone2 field. The MajorOrSchool field stores the student's major if they have picked one and the student's school (School of Engineering, School of Liberal Arts, School of Metaphysics, and so forth) otherwise.
 - a. Explain why this list isn't in 1NF.
 - b. Convert it into 1NF. Draw a relational diagram for it.
2. Consider the following table that lists errands that you need to run. The list shows the most important items at the top.

LOCATION	ITEMS
Grocery store	milk, eggs, bananas
Office supply store	paper, pencils, divining rod
Post Office	stamps
Computer store	flash drive, 8" floppy disks

- a. Explain why this list isn't in 1NF.
 - b. Convert this list into a single 1NF table. Be sure to define a primary key.
3. For the table you built for Exercise 2:
- a. Explain why the table isn't in 2NF.
 - b. Convert the table into 2NF.
-
4. Consider the following employee assignments table, which uses Employee as its primary key.

EMPLOYEE	PROJECT	DEPARTMENT
Alice Most	Work Assignment	Network Lab
Bill Michaels	Network Routing	Network Lab
Deanna Fole	Survey Design	Human Factors
Josh Farfar	Work Assignment	Network Lab
Julie Wish	Survey Design	Human Factors
Mandy Ponem	Network Routing	Network Lab
Mike Mix	New Services Analysis	Human Factors

- a. Explain why the table isn't in 3NF.
 - b. Convert the table into 3NF.
-
5. One of your friends has decided to start a breakfast club. What each member can cook depends on their skills and equipment. Your friend built the following table to record all the combinations.

PERSON	FOOD	TOOL
Alice	muffins	muffin tin
Alice	muffins	omelet pan
Alice	muffins	pancake griddle
Alice	omelets	muffin tin
Alice	omelets	omelet pan
Alice	omelets	pancake griddle

PERSON	FOOD	TOOL
Alice	pancakes	muffin tin
Alice	pancakes	omelet pan
Alice	pancakes	pancake griddle
Bob	muffins	omelet pan
Bob	omelets	omelet pan
Bob	pancakes	omelet pan
Cyndi	omelets	muffin tin
Cyndi	omelets	pancake griddle

Fortunately, you know all about normalization so you help your friend by:

- a. Condescendingly explaining why the table isn't in 5NF.
- b. Converting the table into 5NF.

6. In Figure 7.30, match the normal forms on the left with their corresponding rules on the right.

First Normal Form	<ul style="list-style-type: none"> It contains no transitive dependencies.
Second Normal Form	<ul style="list-style-type: none"> It does not contain an unrelated multi-valued dependency.
Third Normal Form	<ul style="list-style-type: none"> Each column must have a unique name. The order of the rows and columns doesn't matter.
Boyce/Codd Normal form	<ul style="list-style-type: none"> Each column must have a single data type. No two rows can contain identical values. Each column must contain a single value. Columns cannot contain repeating groups.
Fourth Normal form	<ul style="list-style-type: none"> All of the non-key fields depend on all of the key fields.
Fifth Normal form	<ul style="list-style-type: none"> Every determinant is a candidate key.
Domain/Key Normal Form	<ul style="list-style-type: none"> It contains no related multivalued dependencies. The table contains no constraints except domain constraints and key constraints.

FIGURE 7.30

8

Designing Databases to Support Software

The previous chapters showed you how to gather user requirements, build a database model, and normalize the database to improve its performance and robustness. You learned how to look at the database from your customers' perspective, from the end-users' perspective, and from a database normalization perspective. But there's one other viewpoint that you should consider before you open your database product and start slapping tables together: the programmer's.

You might not be responsible for writing a program to work with a database. The database might not ever directly interact with a program (although that's rare). In any case, the techniques for making a database easier for a program to use often apply to other situations. Learning how to help a database support software applications can make the database easier to use in general.

In this chapter, you learn:

- Steps that you can take to make the database more efficient in practical use
- Methods for making validation easier in the user interface
- Ways to easily manage nonsearchable data

This chapter describes several things that you can do to make the database more program-friendly.

A few of these ideas (such as multi-tier architecture) have been covered in earlier chapters. They are repeated in brief here to tie them together with other programming-related topics, but you should refer to the original chapters for more detailed information.

PLAN AHEAD

Any complicated task benefits from prior planning, and software development is no exception. (For a lot more information about software development, see my book *Beginning Software Engineering, Second Edition*, Wiley, 2022.) It has been shown that the longer an error remains in a project the longer it takes to fix it. Database design typically occurs early in the development process, so mistakes made here can be extremely costly. A badly designed database provides the perfect proving ground for the expression “Act in haste, repent at leisure.” Do your work up front or be prepared to spend a lot of extra time fixing mistakes.

Practically all later development depends directly or indirectly on the database design. The database design acts like a building’s foundation. If you build a weak foundation, the building on top of it will be wobbly and unsound. The Leaning Tower of Pisa is a beautiful result built on a weak foundation, but it’s the result of luck more than planning and people have spent hundreds of years trying to keep it from falling down. If you try to build on a wobbly foundation, you’re more likely to end up with a pile of broken rubble than an iconic building.

After you gain some experience with database design, it’s tempting to just start cranking out table and field definitions without any prior planning, but that’s almost always a mistake. Don’t immediately start building a database or even creating a relational object model. At least sketch out an ER diagram to better understand the entities that the database must represent before you start building.

DOCUMENT EVERYTHING

Write everything down. This helps prevent disagreements about who promised what to whom. (“But you promised that the database could look up a customer’s credit rating and Amazon password.”)

Good documentation also keeps everyone on the same wavelength. If you have done a good job of writing requirements, use cases, database models, design specifications, and all the other paperwork that describes the system, then the developers can scurry off to their own little burrows and start working on their parts of the system without fear of building components that won’t work together.

You can make programmers’ lives a lot easier if you specify table and field definitions in great detail. Record the fields that belong in each table. Also record each field’s properties: name, data type, length, whether it can be null, string format (such as “mm/dd/yyyy” or “###-###”), allowed ranges (1–100), default values, and other more complex constraints.

Programmers will need this information to figure out how to build the user interface and the code that sits behind it (and the middle tiers if you use a multi-tier architecture). Make sure that the information is correct and complete at the start so that the programmers don’t need to make a bunch of changes later.

For example, suppose Price must be greater than \$1.00. The programmers get started and build a whole slew of screens that assume Price is greater than \$1.00. Now, it turns out that you meant Price must be *at least* \$1.00 not *greater than* \$1.00. This is a trivial change to the design and to the

database, but the programmers will need to go fix every screen that contains the incorrect assumption. (Actually, good programming practices will minimize the problem, but you can't assume everyone is a top-notch developer.)

After you have all of this information, don't just put it in the database and assume that everyone can get the information from there. Believe it or not, some developers don't know how to use every conceivable type of database product (MySQL, SQL Server, Access, Informix, Oracle, DB2, Paradox, Sybase, PostgreSQL, FoxPro, MongoDB—there are hundreds), so making them dig this information out of the database can be a time-consuming hassle. Besides, documenting it all gives you something to show management to prove that you're making progress.

CONSIDER MULTI-TIER ARCHITECTURE

A multi-tier architecture can help isolate the database and user interface development so that programmers and database developers can work independently. This approach can also make a database more flexible and amenable to change. Unless you're a project architect, you probably can't decide to use this kind of architecture by yourself, but you can make sure it is considered. See Chapter 6, "Extracting Business Rules," for more details about multi-tier architectures.

CONVERT DOMAINS INTO TABLES

It's easy enough to validate a field against its domain by using check constraints. For example, suppose you know that the Addresses table's State field must always hold one of the values CA, OR, or WA. You can verify that a field contains one of those values with a field-level check constraint. In Microsoft Access, you could set the State field's Validation Rule property to:

```
= 'CA' Or = 'OR' Or = 'WA'
```

Other databases use different syntax.

Although this is simple and it's easy for you to change, it's not easily visible to programmers building the application. That means they need to write those values into their code to allow the user interface to validate the user's choice. Later if you change the list, the programmers need to change their code, too.

Even worse, someone needs to remember that the code needs to be changed! It's fairly common to change one part of an application and forget to make a corresponding change elsewhere. Those kinds of mistakes can lead to some bugs that are very hard to find.

A better approach is to move the domain information into a new table. Create a States table and put the values CA, OR, and WA in it. Then make a foreign key constraint that requires the Addresses table's States field to allow only values that are in the States table. Programmers can query the database at runtime to learn what values are allowed and can then do things such as making a combo box that allows only those choices. Now if you need to change the allowed values, you only need to update the States table's data and the user interface automatically picks up the change.

Wherever possible, convert database structure into data so everyone can read it easily.

Using Lookup Tables

Okay, this is really easy but it's important, so bear with me. Suppose your Phoenician restaurant offers delivery service for customers with ZIP Codes between 02154 and 02159. In Access, you could validate the customer's Zip field with the following field-level check constraint:

```
>='02154' And <='02159'
```

Unfortunately, that constraint is hidden from the programmers building the user interface. These checks may also not be in a form that's easy for the program to understand. Most programmers don't know how to write code to allow the user interface to open Access, read a field's check constraints, and parse an expression such as this one to figure out what it does.

How could you make this condition easier for programmers to discover and use at runtime?

How It Works

Reading and parsing check constraints is hard, but it's fairly easy for a program to read the values from a table. The answer is to make a DeliveryZips table that lists the ZIP Codes in the delivery area:

ZIP
02154
02155
02156
02157
02158
02159

This seems less elegant than the field-level check constraint, but it's a lot easier for the program to understand. It also works with any kind of database, so the program doesn't need to know how check constraints are represented in Access, MariaDB, PostgreSQL, or whatever kind of database you're using.

KEEP TABLES FOCUSED

If you keep each table well focused, then there will be fewer chances for misunderstandings. Developers will have an easier time keeping each table's purpose straight, so different parts of the application will use the data in the same (intended) way.

Modern object-oriented programming languages also use objects and classes that are similar to the objects and classes used in semantic object modeling and that are similar to the entities and entity sets used by entity-relationship diagrams. If you do a good job of modeling and keep object and table

definitions well focused, then those models practically write the code by themselves. There are still plenty of other things for the programmers to do, but at least they'll be able to make programming object models that closely resemble the database structure.

USE THREE KINDS OF TABLES

One tip that can help you keep tables focused is to note that there are three basic kinds of tables. The first kind stores information about objects of a particular type such as Students, Employees, or QuarterlyReports. These hold the bulk of the application's data.

The second kind of table represents a link between two or more objects. For example, the association objects described in Chapter 5, "Translating User Needs into Data Models," represent a link between two types of objects.

Figure 8.1 shows a relational diagram to model employees and projects. Each employee can be assigned to multiple projects and each project can involve multiple employees. The EmployeeRoles table provides a link between the two object tables. It also stores additional information about the link. In this case, it stores the role that an employee played on each project.

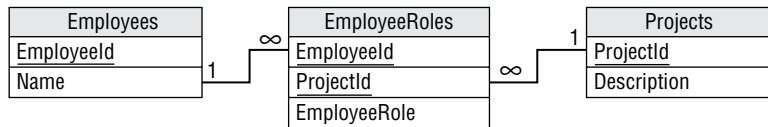


FIGURE 8.1

The third basic kind of table is a lookup table. These are tables created simply to use in foreign key constraints. For example, the States table described in the earlier section "Convert Domains into Tables" is a lookup table.

When you build a table, ask yourself whether it represents an object, a link, or a lookup. If you cannot decide or if the table represents more than one of those, then the table's purpose may not be clearly defined.

Well-Focused Tables

Take a look at the following table of extra-terrestrial animals:

ANIMAL	SIZE	PLANET	PLANETARYMASS
Hermaflamingo	Medium	Virgon 4	1.21
Shunkopotamus	Large	Dilbertopia	0.88
Mothalope	Medium	Xanth	0.01

ANIMAL	SIZE	PLANET	PLANETARYMASS
Shunkopotamus	Large	Virgon 4	1.21
Platypus	Small	Australia	1.00

The table's primary key is the combination of Animal/Planet. (The PlanetaryMass field is measured in Earth masses.)

This isn't a well-focused table.

1. Which ideas is this table is trying to capture?
2. What types of ideas are these (object, link, or lookup)?
3. Suggest a better design that keeps the table's purposes separate.

How It Works

1. Which ideas is this table is trying to capture?

This table is trying to capture three different ideas: information about the animals (Animal and Size), information about planets (Planet and PlanetaryMass), and the associations between the animals and planets. The fact that this table holds information about three different concepts is a sign that it is not well focused.

(Also note this table is not in 2NF because it has non-key fields that do not depend on the entire key. Recall that the primary key is Animal/Planet. The Size field depends on Animal but not Planet, and the PlanetaryMass field depends on Planet but not Animal. But you knew that!)

2. What types of ideas are these (object, link, or lookup)?

Information about the animals and information about the planets represent objects. Information about the associations between animals and planets is a link.

3. Suggest a better design that keeps the table's purposes separate.

The key is to move the three kinds of information into three different tables. Figure 8.2 shows one relational design that separates the three sets of information into three tables.

Figure 8.3 shows the tables holding their original data.

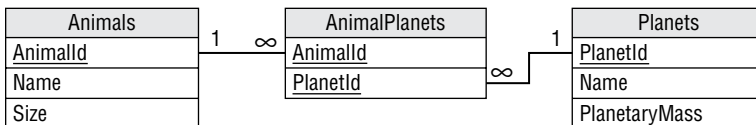


FIGURE 8.2

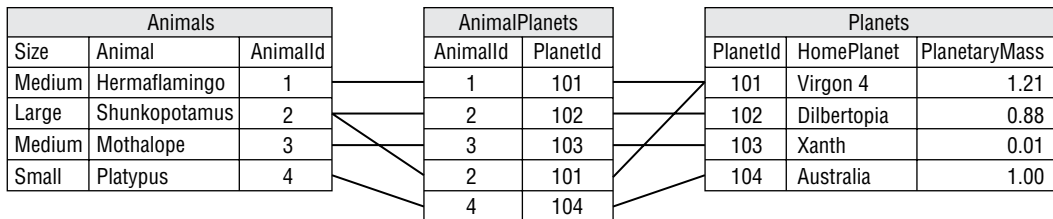


FIGURE 8.3

USE NAMING CONVENTIONS

Use consistent naming conventions when you name database objects. It doesn't matter too much what conventions you use as long as you use something consistent.

Some database developers prefix table names with `tbl` and field names with `fld` as in, "The `tblEmployees` table contains the `fldFirstName` and `fldLastName` fields." For simple databases, I prefer to omit the prefixes because it's usually easy enough to tell which names represent objects (tables) and which represent attributes (fields).

Some developers also use a `lnk` prefix for tables that represent a link between objects as in, "The `lnkAnimalsPlanets` table links `tblAnimals` and `tblPlanets`." These developers are also likely to use `lu` or `lup` as a prefix for lookup tables.

Some developers prefer to use `UPPERCASE_LETTERS` for table names and `lowercase_letters` for field names. Others use `MixedCase` for both.

WARNING Some database engines might have their own views on names. For example, due to reasons that I'll explain in Chapter 18, "PostgreSQL in Python," PostgreSQL works most easily with names that are in lowercase.

Do a little research and perhaps write a test program or two before you define your naming conventions.

Some prefer to make table names singular (the `Employee` table) and others prefer to make them plural (the `Employees` table).

As I said, it doesn't matter too much which of these conventions you use as long as you pick some conventions and stick to them.

However, there are three "mandatory" naming rules that you should follow or the other developers will snicker behind your back and openly mock you at parties.

First, don't use special characters in table names, field names, or anywhere else in database objects. For example, some databases (such as Access) allow you to include spaces in table and field names. For example, you can make a field named "First Name." Those databases also provide some mechanism for making these weird names readable to the database. For example, in Access you need to use square brackets to surround a field with a name that contains spaces as in "[First Name]." This produces hard-to-read expressions in check constraints and anywhere else that you use the field. It also makes programmers using the field take similar annoying steps, and that makes their code less readable too. Just don't do it.

Second, if two fields in different tables contain the same data, give them the same name. For example, suppose you use an ID field to link the Employees table to the EmployeePhones table. Don't call this linking field `Id` in the Employees table and `EmpId` in the EmployeePhones table. That's just asking for trouble. Call the field `EmployeeId` in both tables. (A corollary to this rule is that you cannot name an ID field something vague such as `Id`. It might make sense in the main table such as Employees, but that name won't make sense in a related table such as EmployeePhones.)

The third "mandatory" naming rule is to use meaningful names. Don't abbreviate to the point of obscurity. It shouldn't take a team of National Security Agency cryptographers to decipher a table's field names. `StudPrfCrS` is much harder to read than `StudentPreferredCourses`. Don't be afraid to spell things out so that everyone can understand them. (The exception here seems to be the military where everyone would understand the phrase "SecInt visited NavSpecWarGru" but saying "the Secretary of the Interior visited the Naval Special War Group" would brand you as an outsider.)

The section "Poor Naming Standards" in Chapter 10, "Avoiding Common Design Pitfalls," has more to say about naming conventions and includes a few links that you can follow to learn about some specific standards that you can adopt if you like.

ALLOW SOME REDUNDANT DATA

Chapter 7, "Normalizing Data," explained that it is not always best to normalize a database as completely as possible. The higher forms of normalization usually spread data out into tables that are linked by their fields. When a program needs to display that data, it must reassemble all of that scattered data, and that can take some extra time.

For example, if you allow customers to have any number of phone numbers, email addresses, postal addresses, and contacts, then what seems to the user like a simple customer record is actually spread across the Customers, CustomerPhones, CustomerEmails, CustomerAddresses, and CustomerContacts tables.

In some cases, it might be better to restrict the database's flexibility somewhat to gain speed and simplicity. For example, if you allow the customers to have only two phone numbers, one email address, and one contact, then you cut the number of tables that make up the customer's information from five to two. The database won't be as infinitely flexible, and it won't be quite as completely normalized, but it will be easier to use.

Usually, it's also best not to store the same data in multiple ways because that can lead to modification anomalies. For example, you don't really need a Balance field in a customer's record if you can recalculate the balance by adding up all of the customer's credits and debits.

However, suppose you're running an Internet service that allows customers to download music so a typical customer makes dozens or even hundreds of purchases a month. After a year or two, adding up all of a customer's credits and debits could be time consuming. In this case, you might be better off adding a Balance field to the customer's record and exercising a little extra caution when updating the account.

NoSQL databases have no trouble holding duplicate and summary data because they don't need to obey relational database constraints. A JavaScript Object Notation (JSON) or Extensible Markup Language (XML) format customer document can define as many phone numbers, email addresses, social media accounts, favorite made-for-television movies, and shoe sizes as you want, and the number can be different for every customer.

DON'T SQUEEZE IN EVERYTHING

Just because you're using a database doesn't mean every piece of data that the system uses must be squeezed in there somewhere. Databases provide tools for storing and retrieving some strange pieces of data such as audio, video, images—just about anything that you can cram into a computer file. That doesn't mean you should go crazy and store every file on your computer within the database.

For example, suppose an application must locate and play thousands of audio files. You could store all of them in the database, or you could place the files in a directory tree somewhere and then store the locations of those files in the database. That makes the database simpler, smaller, and possibly more efficient because it doesn't need to store all those files. It also makes it a lot easier to update the files. Instead of loading a new file into the database, you can simply replace the file on the disk.

This technique can also be useful for managing large amounts of shared data such as web pages. You don't need to copy Wikipedia pages into your database. (In fact, it would probably be a copyright violation.) Instead, you can store the URLs pointing to the pages that you need. In this case you give up control of the data, but you also don't have to store and maintain it yourself. If the data is updated, you'll see the new data the next time you visit its URL.

There are only a couple of drawbacks to this technique. First, you lose the ability to search inside any data that is not stored inside the database. I don't know of any databases that let you search inside video, audio, or JPEG data, however, so you probably shouldn't lose much sleep over giving up an ability that you don't have anyway. I wouldn't move textual data outside the database in this way, however, unless you're sure that you'll never want to search inside it.

Second, you give up some of the security provided by the database. For example, you lose the database's record-locking features, so you might have trouble allowing multiple concurrent users to update the data safely.

First Normal Form

Suppose you're building a database of amusing commercials (for example, see www.renderforest.com/blog/funny-commercials if you have some time to waste). The Commercials table includes the fields Name, Product, Description, Length, Video (the commercial), and Still (a representative frame to remind you what the commercial is about).

Figure out which of these fields should remain in the database and which might be moved outside into the filesystem.

1. To figure out which fields should remain in the database, identify those that you might want to search. Include fields with simple values (such as numbers and short strings) that are easy to store in a database and that would not make a very big file on the disk.
2. To figure out which fields might be moved outside of the database into the filesystem, identify the fields that contain large chunks of nonsearchable data.

How It Works

1. In this database, you might want to search the Name, Product, Description, and Length fields.
2. You cannot search the Video or Still fields whether you want to or not, so you might as well move them into the filesystem. The Video field will contain particularly large amounts of data, so moving it outside of the database might even make the database more efficient.

However, tools that can identify images and guess what they contain are becoming more reliable, so you might be able to search the images themselves someday. It would still probably be better to preprocess each image and save that kind of information in a searchable Keywords field rather than trying to examine images on the fly.

SUMMARY

Although the book's focus and your database design effort is on databases, a database rarely lives in total isolation. Usually someone writes a program to interact with it. Often, the database is a backend for a complicated user interface.

To get the most out of your database, you need to consider it in its natural habitat. In particular, you should think about the applications and programmers who will interact with it. Often, a few relatively small changes to the design can make life easier for everyone involved.

In this chapter, you learned to:

- Plan ahead and document everything.
- Convert domains into tables to help user interface programmers and to make maintaining domain information easier.
- Keep tables well focused and make each perform a single task.
- Use some redundancy and denormalized tables to improve performance.

The last several chapters dealt with database design techniques and considerations. Those chapters explained general techniques for building a data model, and then modifying it to make it more efficient.

The next chapter switches from general discussion to more specific techniques. It summarizes some of the methods described in the previous chapters and explains some common relational database design patterns that you might find useful in providing specific data features.

Before you move on to Chapter 9, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

EXERCISES

- Suppose you sell ocean cruises. Customers first decide what kind of Ship they want to travel on: Luxury Liner, Schooner, Tuna Boat, or Barge. Depending on that choice, they might select different classes of cabins. Luxury Liners provide 1st through 5th class. Schooners have 1st and 2nd class (basically, a single or a double), Tuna Boats have a single class (which they euphemistically call 1st Class) where you share a single large bunkroom with the rest of the crew. Barges have no class.

You could validate the Trips record's Ship and Cabin fields by using a table-level check constraint, but because you're a team player, you would rather build a foreign key constraint so the user interface can read the allowed values from the tables.

Build such a table and display its data. Explain how this table will be used in the foreign key constraint.

- Figure 8.4 shows a relational diagram showing the relationships between students, the classes they are taking, and the departments that hold the classes. For each table in this diagram, tell which of the three types of table it is: object, link, or lookup.

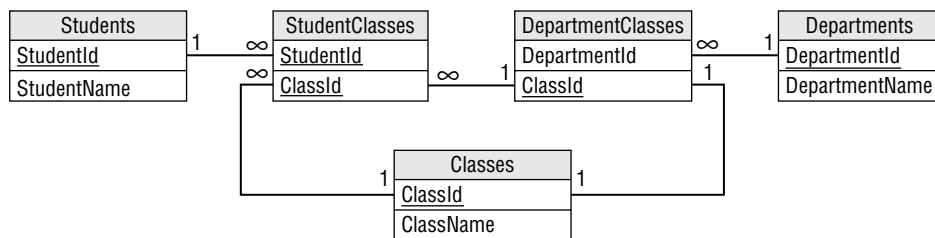


FIGURE 8.4

- The following table stores information about checkers matches. Explain why it lacks focus and how you would fix it.

PLAYER1	PLAYER1RANK	PLAYER2	PLAYER2RANK	MATCHTIME
Smith	10	Jones	3	1:00
Marks	9	Lars	4	1:00
Aft	8	Cook	5	2:00
Mauren	7	Juno	6	2:00

4. Assume you have a large database that tracks how closely airplanes are to their scheduled departure and landing times. It tracks these values by plane (which is associated with a particular airline) and airport. It also records the weather at the starting and landing airports.

Which of the following daily values should you store in a redundant variable and which should you calculate as needed?

- a. Average minutes late for an airline at a particular airport
 - b. Average minutes late for all airlines at a particular airport
 - c. Average minutes late for an airline across the country
 - d. Average minutes late for all airlines across the entire country
5. Assume that you need these numbers quickly several times per day.
-



Using Common Design Patterns

The previous chapters described general techniques for building database designs. For example, Chapter 5, “Translating User Needs into Data Models,” explained how to build semantic object models and entity-relationship diagrams for a database, and how to convert those models into relational designs. Chapter 7, “Normalizing Data,” explained how to transform those designs to normalize the database.

This chapter takes a different approach. It focuses on data design scenarios and describes methods for building them in a relational model.

In this chapter, you will learn techniques for:

- Providing different kinds of associations between objects
- Storing data hierarchies and networks
- Handling time-related data
- Logging user actions

This chapter does not provide designs for specific situations such as order tracking or employee payroll. Appendix B, “Sample Relational Designs,” contains those sorts of examples.

This chapter focuses on a more detailed level to give you the techniques that you need to build the pieces that make up a design. You can use these techniques as the beginning of a database design toolbox that you can apply to your problems.

The following sections group these patterns into three broad categories: associations, temporal data, and logging and locking.

ASSOCIATIONS

Association patterns represent relationships among various data objects. For example, an association can represent the relationship between a rugby team and its opponents during matches.

The following sections describe different kinds of associations.

Many-to-Many Associations

It's easy to represent a many-to-many association in an ER diagram. For example, a Student can be enrolled in many Courses and a Course includes many Students, so there is a many-to-many relationship between Students and Courses. Figure 9.1 shows an ER diagram modeling this situation.

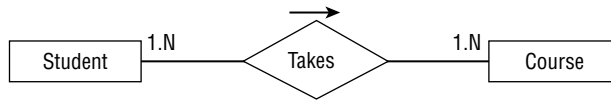


FIGURE 9.1

Unfortunately, relational databases cannot handle many-to-many relationships directly. To build this kind of relationship in a relational database, you need to add an association table to represent the relationship between students and courses. Simply create a table called StudentCourses and give it fields StudentId and CourseId. Figure 9.2 shows this structure.

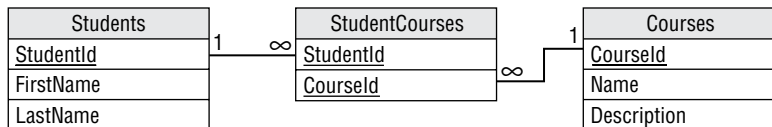


FIGURE 9.2

To list all of the courses for a particular student, find the StudentCourses records with the required StudentId. Then use each of those records' CourseId values to find the corresponding Courses records.

To list all of the students enrolled in a particular course, find the StudentCourses records with the required CourseId. Then use each of those records' StudentId values to find the corresponding Students records.

Multiple Many-to-Many Associations

Sometimes, a many-to-many relationship contains extra associated data. For example, the previous section explained how to track students and their current course enrollments. Suppose you also want to track student enrollments over time. In other words, you want to know each student's enrollments for each year and semester. In this case, you actually need to make multiple many-to-many associations between students and courses. You need whole sets of these associations to handle each school semester.

Fortunately, this requires only a small change to the previous solution. The StudentCourses table shown in Figure 9.2 can already represent the relationship of students to courses. The only thing missing is a way to add more records to this table to store information for different years and semesters.

The solution is to add Year and Semester fields to the StudentCourses table. Figure 9.3 shows the new model.

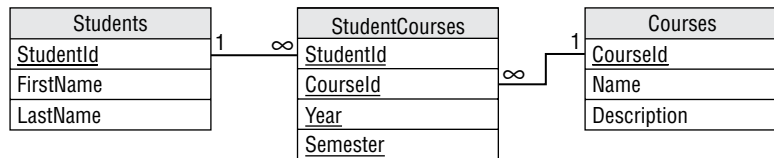


FIGURE 9.3

Now, the StudentCourses table can store multiple sets of records representing different years and semesters.

If you need to store extra information about each semester, you could make a new Semesters table to hold that information. Then you could add the Year and Semester fields to this new table and use them as a foreign key in the StudentCourses table.

Many-to-Many Relations

Suppose you're coordinating a weeklong tour called "Junk Yards of the Napa Valley." Each day, the participants can sign up for several tours of different junkyards. They can also sign up for dinner at a fine restaurant or winery.

Build a relational model to record this information.

1. Build Participants, Tours, and Restaurants tables.
2. Study the relationships between Participants and Tours, and between Participants and Restaurants. Determine whether they are many-to-many or some other kind of relationship.
3. Build a relationship table to represent each many-to-many relationship. Be sure to include enough fields to distinguish among similar combinations of the involved tables. (For example, Bill really liked the trip to Annette's Scrap and Salvage on Tuesday, so he took that tour again on Thursday.) Your model needs a ParticipantTours table and a ParticipantRestaurants table. To distinguish among repeats such as Bill's, add a Date field to each table.

How It Works

1. There's no real trick in Step 1. Just be sure to give each table an ID field so it's easy to refer to its records.
2. To understand Step 2, remember that each participant can go on many tours and each tour can have many participants, so the Participants/Tours relationship is many-to-many. During the week, each

participant can eat at several restaurants and each restaurant can feed many participants, so the Participants/Restaurants relationship is also many-to-many.

- To model the two many-to-many relationships, the model needs a ParticipantTours table and a ParticipantRestaurants table. To distinguish among repeats (a customer takes the same tour twice or visits the same restaurant twice), add a Date field to each table.

Figure 9.4 shows the final relational model.

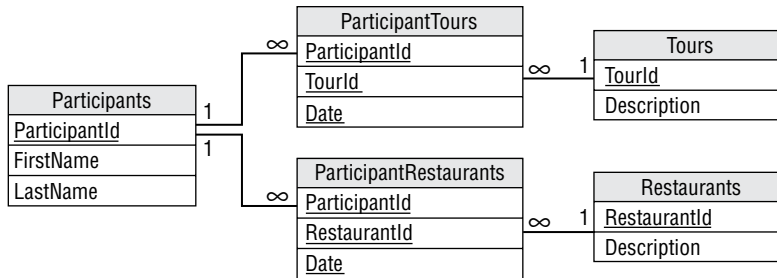


FIGURE 9.4

Multiple-Object Associations

A multiple-object association is one where many different kinds of objects are collectively associated to each other. For example, making a movie requires a whole horde of people, including a director, a bunch of actors, and a huge number of crew members with improbable titles like gaffer, dolly grip, best boy, and python wrangler. (No, I didn't invent any of those.) You could model the situation with the ER diagram shown in Figure 9.5.

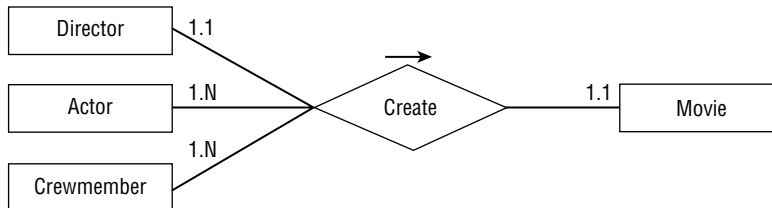


FIGURE 9.5

If this collection of people always worked as a team, then this situation would be easy to implement in a relational model. You would assign all of the people a TeamId, and then build a Movies table with a TeamId field to tell who worked on that movie.

Unfortunately, this idea doesn't quite work because all these people can work on any number of movies in any combination.

You can solve this problem by thinking of the complex multi-object relationship as a combination of simpler relationships. In this case, you can model the situation as a one-to-one Director/Movie relationship, a many-to-many Actor/Movie relationship, and a many-to-many Crewmember/Movie relationship.

Figure 9.6 shows the new ER diagram.

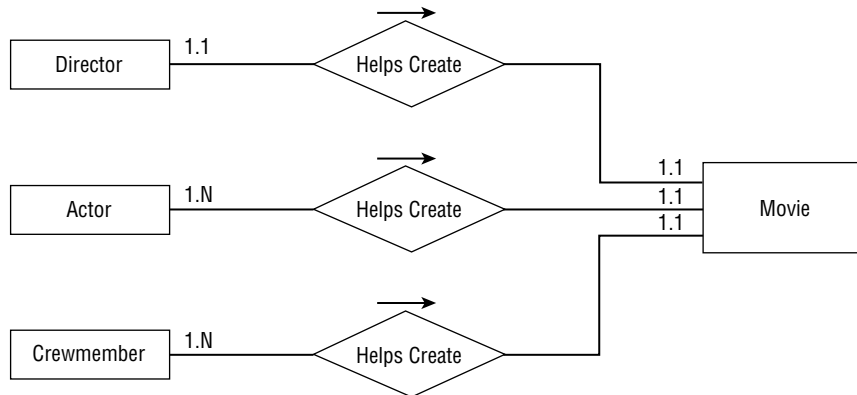


FIGURE 9.6

You can convert this simpler diagram into a relational model, as shown in Figure 9.7.

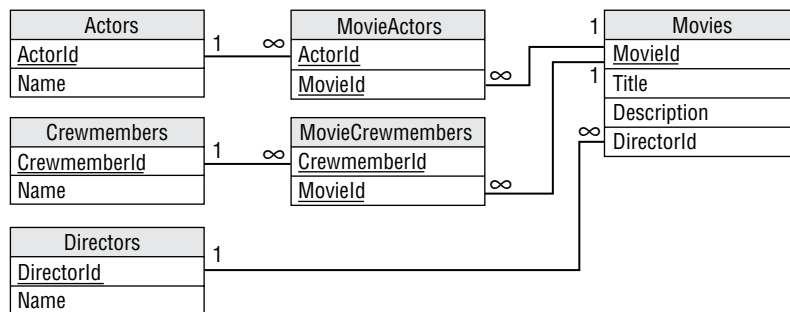


FIGURE 9.7

Notice that this model uses two association tables to represent the two many-to-many relationships. The relationship between Directors and Movies doesn't require an association table because this is a simpler one-to-one relationship.

Building Multiple-Object Associations

Consider another aspect of the “Junk Yards of the Napa Valley” tours. You have multiple tour guides and multiple vehicles. A Trip represents a specific instance of a tour by a guide, vehicle, and a group of participants.

Build a relational model to hold this data.

1. Build Guides, Vehicles, Tours, Participants, and Trips tables.
2. Study the relationships between Trips and Guides, Vehicles, Tours, and Participants. Determine whether they are many-to-many or some other kind of relationship.
3. Build an ER diagram to show these relationships.
4. Build a relationship table to represent each many-to-many relationship.
5. Draw the relational model.

How It Works

1. There’s no real trick in this. Just be sure to give each table an ID field so that it’s easy to refer to its records.
2. Each guide can lead several trips but each trip has a single guide, so Guides/Trips is a one-to-many relationship.

Each vehicle can go on many trips but each trip has a single vehicle, so Vehicles/Trips is a one-to-many relationship.

A tour represents a destination. A destination can be the target of many trips but each trip visits only one destination, so Tours/Trips is a one-to-many relationship.

Finally, each participant can go on many trips and each trip can have many participants, so Participants/Trips is a many-to-many relationship.

3. Figure 9.8 shows these relationships in an ER diagram.
4. This model has only one many-to-many relationship: Participants/Trips. To handle it, the model needs a ParticipantsTrips table. Figure 9.9 shows the relational model.

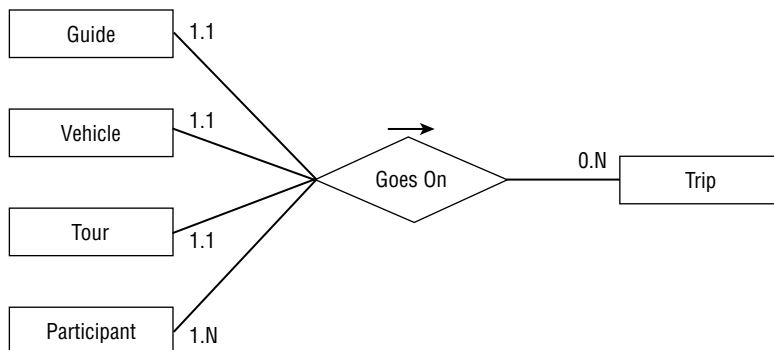


FIGURE 9.8

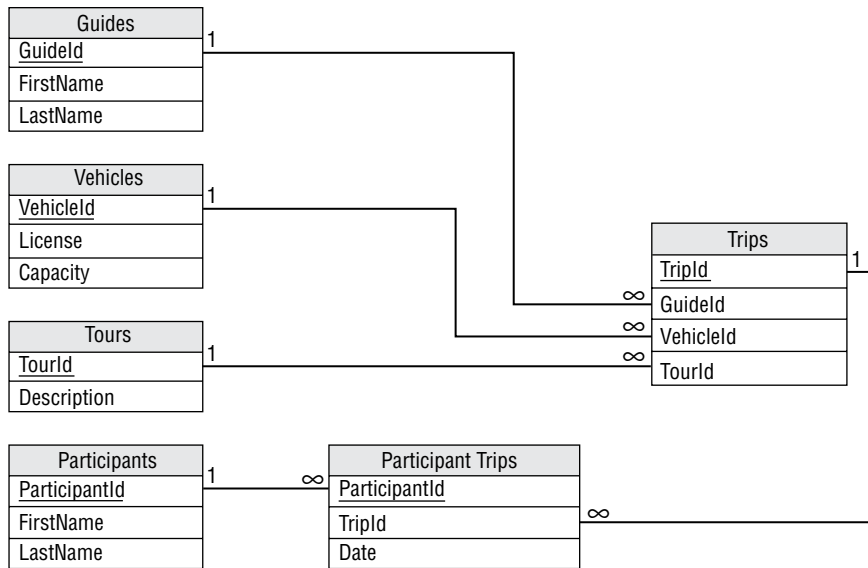


FIGURE 9.9

Repeated Attribute Associations

Some entities have multiple fields that represent either the same kind of data or a very similar kind of data. For example, it is common for purchase orders and other documents to allow you to specify a daytime phone number and an evening phone number. Other contact-related records might allow you to specify even more phone numbers for such things as cellphones, emergency contacts, fax numbers, pagers, and others.

Figure 9.10 shows a semantic object model for a `PERSON` class that allows any number of `Phone` attributes.

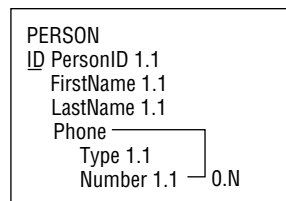


FIGURE 9.10

In this model, we want to allow any number of instances of the repeated attribute even if they have the same type. For example, we would allow an order to include multiple emergency contact phone numbers.

To allow any number of repeated attributes in a relational model, build a new table to contain the repeated values. Use the original table's primary key to link the new records back to the original table.

Figure 9.11 shows how to do this for the `PERSON` class shown in Figure 9.10.

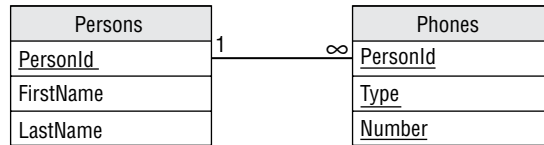


FIGURE 9.11

Because the Phones table's primary key includes all of the table's fields, the combination of PersonId/Type/Number must be unique. That means a person can only use a phone number for a particular purpose once. That makes sense. It would be silly to list the same phone number as a work number twice for the same person. However, a person could have the same number for multiple purposes (daytime and evening numbers are the same cell phone) or have multiple phone numbers for the same purpose (office and receptionist numbers for work phone).

You can use the primary keys and other keys to enforce other kinds of uniqueness. For example, to prevent someone from using the same number for different purposes, make PersonId/Number a unique key. To prevent someone from providing more than one number for the same purpose (for example, two cell phone numbers), make PersonId/Type a unique key.

For another example, suppose you want to add multiple email addresses to the Persons table, allow each person to have any number of phone numbers and email addresses of any type, but not allow duplicate phone numbers or email addresses. (For example, you cannot use the same phone number for Home and Work numbers.)

NOTE *This isn't a typical real-world scenario. You usually wouldn't want to prevent someone from using the same value for both daytime and evening phone numbers. This example just shows how you can allow repeated attributes while prohibiting certain kinds of duplication.*

Just as you created a Phones table, you would create an Emails table with Type and Address fields, plus a PersonId field to link it back to the Persons table. To prevent an email address from being duplicated for a particular person, include those fields in the table's primary key. Figure 9.12 shows the new relational model.

Reflexive Associations

A *reflexive association* or *recursive association* is one in which an object refers to an object of the same class. You can use reflexive associations to model a variety of situations ranging from simple one-to-one relationships to complicated networks of association.

The following sections describe different kinds of reflexive associations.

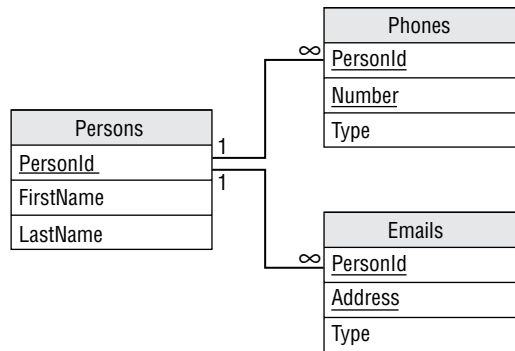


FIGURE 9.12

One-to-One Reflexive Associations

As you can probably guess, in a one-to-one reflexive association an object refers to another single object of the same class. For example, consider the `PERSON` class's `Spouse` field. A `Person` can be married to exactly one other person (at least in much of the world), so this is a one-to-one relationship. Figure 9.13 shows an ER diagram representing this relationship.

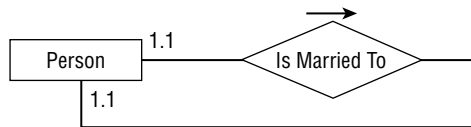


FIGURE 9.13

Figure 9.14 shows a relational model for this relationship.

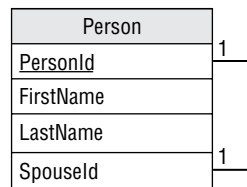


FIGURE 9.14

Unfortunately, this design does not require that two spouses be married to each other. For example, Ann could be married to Bob and Bob could be married to Cindy. That might make an interesting television show, but it would make a confusing database.

Another approach would be to create a `Marriage` table to represent a marriage. That table would hold the IDs of the spouses involved in the marriage. Figure 9.15 shows this design.

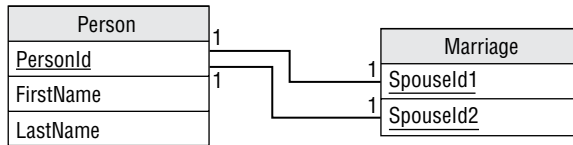


FIGURE 9.15

In this design, the Person table refers to itself indirectly.

For another example, suppose you’re making a database that tracks competitive rock-paper-scissors matches (see <https://wrpsa.com/rock-paper-scissors-tournaments>). You need to associate multiple competitors with each other to show who faced off in the big arena. You also want to record who won and what the winning moves were.

You would start by making a Competitors table with fields Name and CompetitorId.

Next, you would make a CompetitorMatches table to link Competitors. This table would contain CompetitorId1 and CompetitorId2 fields, and corresponding FinalMove1 and FinalMove2 fields to record the contestants’ final moves. To distinguish among different matches between the same two competitors, the table would also include Date and Time fields.

Figure 9.16 shows the relational model.

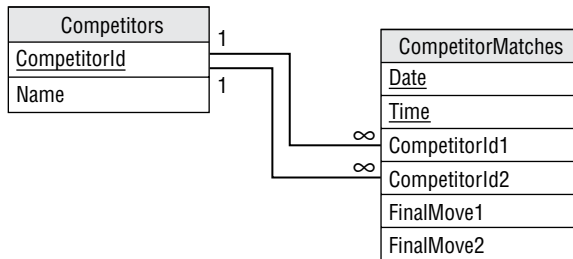


FIGURE 9.16

One-to-Many Reflexive Associations

Typically, employees have managers. Each employee is managed by one manager and a manager can manage any number of employees, so there is a one-to-many relationship between managers and employees.

But a manager is just another employee, so this actually defines a one-to-many relationship between employers and employees. Figure 9.17 shows an ER diagram for this situation.

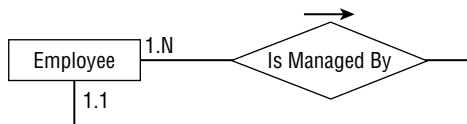


FIGURE 9.17

Figure 9.18 shows a relational model that handles this situation.

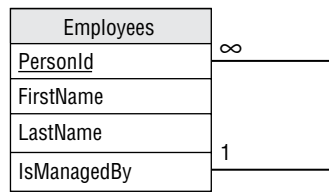


FIGURE 9.18

Hierarchical Data

Hierarchical data takes the form of tree-like structures. Every object in the hierarchy has a “parent” object of the same type. For example, a corporate organizational chart is a hierarchical data structure that shows which employee reports to which other employee. Figure 9.19 shows the org chart for a fictional company.

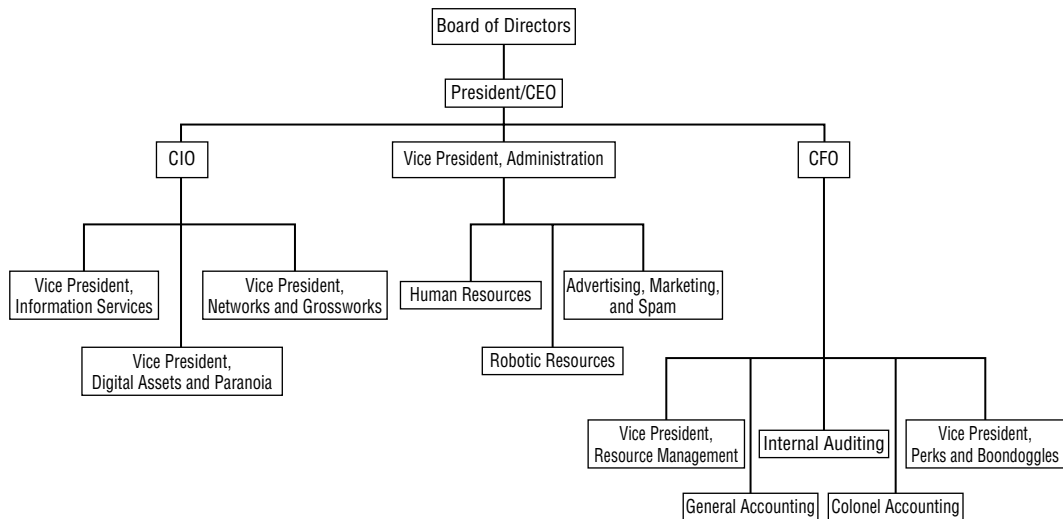


FIGURE 9.19

Hierarchical data is an instance of a one-to-many reflexive association as described in the previous section. Generally, people think of the “Is Managed By” relationship as being relatively flat, so managers supervise front-line employees but no one needs to manage the managers. In that case, the hierarchy is very short.

An org chart can also represent the infinitesimally different concept of “Reports To.” I guess this is more palatable to managers who don’t mind reporting to someone even if they don’t need help managing their own work (although I’ve known a few managers who could have used some serious help in that respect).

The “Reports To” hierarchy may be much deeper (physically, not necessarily intellectually) than the “Manages” hierarchy, but you can still model it in the same way. Figure 9.20 shows an `EMPLOYEE` class that can model both hierarchies simultaneously.

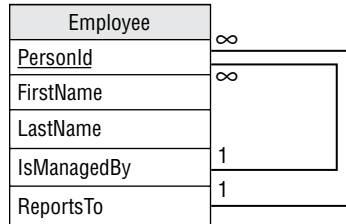


FIGURE 9.20

Notice that the relationships used to implement a hierarchy are “upward-pointing.” In other words, each object contains a reference to an object higher up in the hierarchy. This is necessary because each object has a single “parent” in the hierarchy but may have many “children.” Though you can list an object’s parent in a single field, you cannot list all of its children in a single field.

Working with Hierarchical Data

The following table contains information about a corporate org chart.

PERSONID	TITLE	REPORTSTO
1	Mgr. Pie and Food Gags	9
2	Dir. Puns and Knock-Knock Jokes	6
3	Dir. Physical Humor	9
4	Mgr. Pratfalls	3
5	President	null
6	VP Ambiguity	5
7	Dir. Riddles	6
8	Dir. Sight Gags	3
9	VP ShtickShtick	5

Use this data to reconstruct the org chart graphically.

1. Find the record that represents the root node.
2. For each node on the bottom level of the tree so far (initially, this is just the root node), find all of the records that have that node as a parent (ReportsTo). Attach them below their parent. For example, the first time around you would find the people who report to President. Their records have ReportsTo equal to President's PersonId: 5. These people are VP Ambiguity and VP Shtick. Attach them below President.
3. Repeat until you have processed every record.

How It Works

1. The root node is the one that has no parent. In the table, it's the one where ReportsTo is null: President.
2. Draw the root node.
 - a. Find the people who report to President. Their records have ReportsTo equal to President's PersonId: 5. These people are VP Ambiguity and VP Shtick. Attach them below President.
 - b. Find the people who report to VP Ambiguity. They are Dir. Puns and Knock-Knock Jokes and Dir. Riddles. Draw them below VP Ambiguity.
 - c. Find the people who report to VP Shtick. They are Mgr. Pie and Food Gags and Dir. Physical Humor. Draw them below VP Shtick.
 - d. Find the people who report to Dir. Puns and Knock-Knock Jokes. There are none, so Dir. Puns and Knock-Knock Jokes is a leaf node. (It has no children.)
 - e. Find the people who report to Dir. Riddles. There are none, so Dir. Riddles is a leaf node.
 - f. Find the people who report to Mgr. Pie and Food Gags. There are none, so Mgr. Pie and Food Gags is a leaf node.
 - g. Find the people who report to Dir. Physical Humor. They are Mgr. Pratfalls and Dir. Sight Gags. Draw them below Dir. Physical Humor.
 - h. Find the people who report to Mgr. Pratfalls. There are none, so Mgr. Pratfalls is a leaf node.
 - i. Find the people who report to Dir. Sight Gags. There are none, so Dir. Sight Gags is a leaf node.

At this point, every record is represented on the tree, so we're done.

3. Figure 9.21 shows the finished org chart.

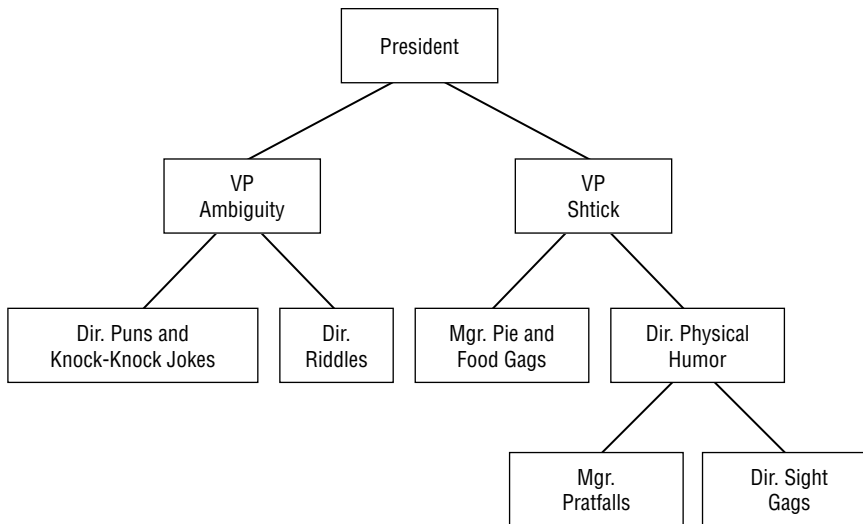


FIGURE 9.21

This method of building a tree works, but it might require a program to jump all over the database fetching the tree's nodes one at a time, possibly from widely separated parts of the disk drive. If the database is small, that's not a problem. If the database is large, however, then this might force the database to reload the same pages of the hard drive many times, and that can greatly slow the process.

In that case, it might be better to load all the tree information all at once and then process it in memory to build the tree. Alternatively you can consider NoSQL databases.

Hierarchical Data with NoSQL

The preceding section explained how to represent hierarchical data in a relational database. It works fairly well, but relational databases aren't truly intended to work with hierarchical data. However, certain kinds of NoSQL databases are.

A graph database is, as I'm sure you can guess, designed to work with graphs. A graph (which is also called a network) represents objects connected by relationships. For example, a street network represents locations in the real world connected by streets.

The trees described in the previous section are a special case of a graph where each node has at most one parent. That means a NoSQL graph database can represent trees easily. Because those kinds of databases are designed to work with graphs, they might also provide features for searching graphs and therefore trees. For example, you might be able to query the database to find out who reports directly or indirectly to VP Shtick.

The *JavaScript Object Notation (JSON)* and *Extensible Markup Language (XML)* file formats are naturally hierarchical, so they can easily store tree-like data. By themselves these are just file formats, but they have been around long enough that there are tools for working with those formats. For example, many programming languages have libraries that can read, write, and search JSON and XML files extremely quickly.

Unless you need relational capabilities, you'll probably get better performance if you store hierarchical data in a NoSQL graph database or in a JSON or an XML file.

Network Data

A network (or graph) contains objects that are linked in an arbitrary fashion. References in one object point to one or more other objects.

For example, Figure 9.22 shows a street network. Each circle represents a node in the network. An arrow represents a link between two nodes. The numbers give the approximate driving times across the links. Notice the one-way streets with arrows pointing in only one direction.

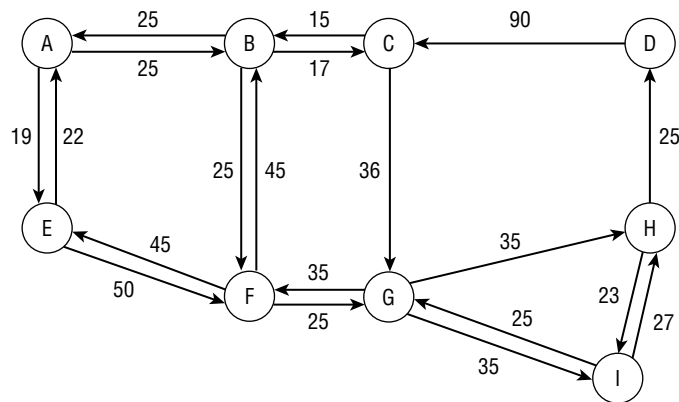


FIGURE 9.22

An object cannot use simple fields to refer to an arbitrary number of other objects, so this situation requires an intermediate table describing the links between objects.

Figure 9.23 shows an ER diagram describing the network's Node object. Notice that the Connects To relationship has a `LinkTime` attribute that stores the time needed to cross the link.

The section "Many-to-Many Associations" earlier in this chapter showed how to build a relational model for many-to-many relationships. That method just needs a small twist to make it work for a many-to-many reflexive relationship.

Instead of creating two tables to represent the related objects, just create a single Nodes table. Then create an intermediary Links table to represent the association between two nodes. That object represents the network link and holds the LinkTime data.

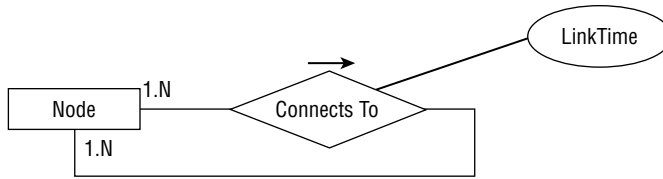


FIGURE 9.23

Figure 9.24 shows this design. In addition to a NodeId, the Nodes table contains X- and Y- coordinates for drawing the node.

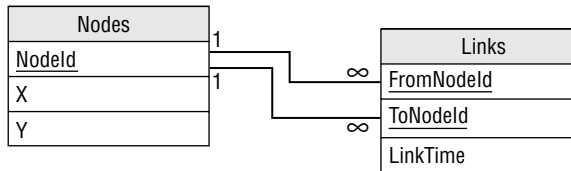


FIGURE 9.24

Note that you need to use some care when you try to use this data to build a network in a program. One natural approach is to start with a node, follow its links to new nodes, and then repeat the process, following those nodes' links. Unfortunately, if the network contains loops, the program will start running around in circles like a dog chasing its tail and it will never finish.

A better approach is to select all of the Nodes records and make program objects to represent them. Then select all of the Links records. For each Links record, find the objects representing the “from” node and the “to” node and connect them. This method is fast, requires only two queries, and best of all, eventually stops.

For a concrete example, consider the small network shown in Figure 9.25. As before, the numbers next to links show the links' times.

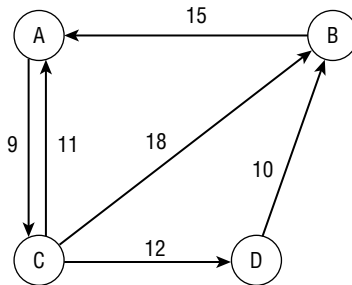


FIGURE 9.25

Start by making a Nodes table with fields NodeId, X, and Y. The following table shows the Nodes table's values. The X and Y fields are blank here because we're not really going to draw the network, but a real program would fill them in.

NODEID	X	Y
A		
B		
C		
D		

Next make a Links table with fields FromNode and ToNode, plus a LinkTime field. Look at Figure 9.25 to see which nodes are connected to which others and what their LinkTime values should be. The following table shows the Links table's data.

FROMNODE	TONODE	LINKTIME
A	C	9
B	A	15
C	A	11
C	B	18
C	D	12
D	B	10

Network Data with NoSQL

Two sections ago, I mentioned NoSQL graph databases. They work well with trees because trees are a special kind of graph. They also work with more general graphs (which are also called networks), and they have the same advantages that they do for trees.

JSON and XML files are naturally hierarchical, so they make storing tree-like data easy. They're not really intended to work with network data, however. In particular, they have trouble with loops in the link data. You can still use them, but you might have to use special tools or do some of the work in your code. You can still store networks in these kinds of files, but it may be easier to use a graph database.

Unless you need relational capabilities, you'll probably get better performance if you store network data in a NoSQL graph database.

TEMPORAL DATA

As its name implies, temporal data involves time. For example, suppose you sell produce and the prices vary greatly from month to month. (For instance, tomatoes are expensive in the winter, while pumpkins are practically worthless on November 1.) To keep a complete record of your sales, you need to track not only orders but also the prices at the time each order was placed.

The following sections describe a few time-related database design issues.

Effective Dates

One simple way to track an object that changes over time is to add fields to the object giving its valid dates. Those fields give the object's effective or valid dates.

Figure 9.26 shows a relational model for temporal produce orders or orders for any other products with prices that change over time.

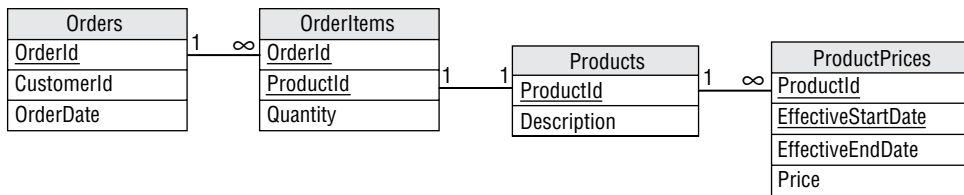


FIGURE 9.26

The Orders table contains an OrderId field and a Date, in addition to other order information such as CustomerId. The OrderId field provides the link to the OrderItems table.

Each OrderItems record represents one line item in an order. Its ProductId field provides a link to the Products table, which describes the product purchased on this line item. The Quantity field tells the number of items purchased.

The ProductPrices table has a ProductId field that refers back to the Products table. The Price field gives the product's price. The EffectiveStartDate and EffectiveEndDate fields tell when that price was in effect.

To reproduce an order, you would follow these steps:

1. Look up the order in the Orders table and get its OrderId. Record the OrderDate.
2. Find the OrderItems records with that OrderId. For each of those records, record the Quantity and ProductId. Then:
 - a. Find the Products record with that ProductId. Use this record to get the item's description.
 - b. Find the ProductPrices record with that ProductId and where $\text{EffectiveStartDate} \leq \text{OrderDate} \leq \text{EffectiveEndDate}$. Use this record to get the item's price at the time the order was placed.

The result is a *snapshot* of how the order looked at the time it was placed. By digging through all of these tables, you should be able to reproduce every order as it appeared when it was entered into the system.

For another example, suppose you want to store one address for each employee but you want to track addresses over time. You don't want to track any other employee data temporally.

To build a relational model to hold this information, start by creating a basic Employees table that holds EmployeeId, FirstName, LastName, and other fields as usual.

Next, design an Addresses table to hold the employee addresses. Create Street, City, State, and Zip fields as usual. Include an EmployeeId field to link back to the Employees record and EffectiveStartDate and EffectiveEndDate fields to track temporal data.

Figure 9.27 shows the resulting relational model.

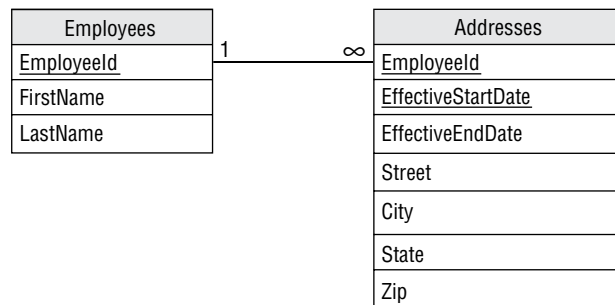


FIGURE 9.27

Deleted Objects

When you delete a record, the information that the record used to hold is gone forever. If you delete an employee's records, you lose all of the information about that employee, including the fact that he was fired for selling your company's secrets to the competition. Because the employee's records were deleted, they could potentially get a job in another part of the company and resume spying with no one the wiser.

Similarly when you modify a record, its previous values are lost. Sometimes that doesn't matter, but other times it might be worthwhile to keep the old values around for historical purposes. For example, it might be nice to know that an employee's salary was increased by only 0.25 percent last year, so you might consider a bigger increase this year.

One way to keep all of this data is to never, ever delete or modify records. Instead, you use effective dates to "end" the old record. If you're modifying the record rather than deleting it, you would then create a new record with effective dates starting now.

For example, suppose you hired Hubert Phreen on 4/1/2027 for a salary of \$45,000. On his first anniversary, you increased his salary to \$46,000 and on his second you increased it to \$53,000. He then grew spoiled and lazy, so he hasn't gotten a raise since. The following table shows the records

this scenario would generate in the `EmployeeSalaries` table. Using this data, you can tell what Hubert's current salary is and what it was at any past point in time.

EMPLOYEE	SALARY	EFFECTIVESTARTDATE
Hubert Phreen	\$45,000	4/1/2027
Hubert Phreen	\$46,000	4/1/2028
Hubert Phreen	\$53,000	4/1/2029

Deciding What to Temporalize

If you decide to use effective dates instead of deleting or modifying records, then you will end up with a bigger database. Depending on how often these sorts of changes occur, it might be a *much* bigger database.

Disk space and cloud storage is relatively inexpensive these days (as little as \$0.022 or so per GB and dropping every year), so that may not be a big issue. If the database is enormous, however, you may want to be selective in what tables you make temporal.

For example, in the model shown in Figure 9.26, only the `ProductPrices` table has effective dates. That would make sense for that example if you don't allow changes to orders after they are created. That greatly reduces the amount of data that you will have to duplicate to record changes.

Before you rush out and add effective dates to everything in sight, carefully consider what data is worth saving in this manner. Be sure to decide which tables to make temporal as early as possible because retrofitting effective date fields can be very difficult, particularly for any programs that access the data. Any queries that request data from tables with effective dates must be parameterized to get the right dates. This is definitely a case where you want thorough planning before you start to build.

If the application will generate too much temporal data, then you may want to consider moving older records into long-term storage. You can buy separate hard drives to hold older data and tuck them away somewhere on a less used server. It might take you longer to pull this data back when you need it, but this approach will keep the main database leaner and faster.

If you're using a cloud database, then you can move older data into your provider's long-term storage plans. This typically makes storing data cost less, but you might pay an access fee to retrieve data and there may be minimum duration requirements. For example, to use Google's Nearline storage, you need to keep the data in storage for at least 30 days.

FOLLOW THE LINES A piece of media such as a hard drive or flash drive is online when it is connected and available. It is offline when it is unplugged or not ready.

The term nearline is a portmanteau (the non-luggage kind) of “near” and “online.” It’s data that is not exactly online but not really offline either. In cloud storage terms, it’s data that is intended to be accessed infrequently, perhaps every few months. (In pre-cloud days, this might be data stored on flash drives that are kept in your desk drawer. You can pull this data up fairly quickly, but not as quickly as you can retrieve data from your hard drive.)

The term coldline continues the pattern. This is data that is intended to be accessed very infrequently, perhaps once a year. (In pre-cloud days, this might be data stored on 8” floppy disks and kept in a spiderweb-covered filing cabinet in a corner of the computer room. You can retrieve this data if you must, but it might take an hour or two to find the right disk. Plus spiders creep you out.)

The term archive means data that you don’t plan to use again but that you want to keep around just in case. (In pre-cloud days, this data was stored at inconvenient locations where it was sometimes lost or misfiled. Think of the Indiana Jones warehouse.)

The following table shows approximate prices for Google’s cloud storage for a server in Iowa. (I found these prices at <https://cloud.google.com/storage/pricing#storage-pricing>, but they are subject to change. In fact, they’re likely to change before you even see this book. I can’t even guarantee that Google won’t move that page.)

PLAN	MINIMUM DURATION	STORAGE PER GB	RETRIEVAL PER GB
Standard	None	\$0.0200	\$0.00
Nearline	30 days	\$0.0100	\$0.01
Coldline	90 days	\$0.0040	\$0.02
Archive	365 days	\$0.0012	\$0.05

The exact prices depend on many factors such as the frequency of transactions, average availability of the servers in hours per day and days per week, number of CPUs, GPUs, and threads, number of nodes, number of certain kinds of operations, and the data center’s physical location. (Presumably the

bytes are bigger in Texas and more fashionable in Milan or something.) In general, longer-term plans cost less for storage and more for retrieval. You can do your own calculations to see which plan will be most cost-effective for your scenario.

LOGGING AND LOCKING

Two techniques that I've found useful in a number of database applications are audit trails and turnkey records. Audit trails let you log changes to key pieces of data. Turnkey records let you easily control access to groups of related records.

Audit Trails

Many databases contain sensitive data, and it is important to make sure that the data is safe at all times. While you cannot always prevent a user from incorrectly modifying the data, you can keep track of who made a modification. Later, if it turns out that the change was unauthorized, you can hunt down the perpetrator and wreak a terrible vengeance.

NOTE *For example, in 2007 State Department contractors “inappropriately reviewed” the passport files of then Senator and presidential candidate Barack Obama. The offenders were probably just curious, but it violated the department’s privacy rules, so two people were fired and one reprimanded. Keeping an audit trail of access to sensitive data lets you know when someone is not playing by the rules.*

One way to provide a record of significant actions is to make an *audit trail* table. This table has fields Action, Employee, and Date to record what was done, who did it, and when it happened. For some applications, this information can be nonspecific, for example, recording only that a record was modified and by whom. In other applications, you might want to record the fields that were modified together with the old and new values.

A similar technique works well with the effective dates described in the section “Temporal Data” earlier in this chapter. If you never delete or update records, then you can add a CreatedBy field to a table that you want to audit and fill in the name of the user who created the record. Later, if someone modifies the record, you will be able to see who made the modification in the new version of the record.

You may still want a separate AuditEvents table, however, to record actions other than creating, deleting, and modifying records. For example, you might want to keep track of who views records (as in the State Department passport case), generates or prints reports (so you know that they were printed), sends emails, or prints letters. You might even want to record user log in and log out times depending on your security needs, accountability requirements, and general level of paranoia.

Turnkey Records

When a user needs to modify a record, a relational database locks that record so that other users can't see an inconsistent view of the data. This prevents others from seeing a half-completed operation. (This isn't necessarily true in NoSQL databases, which only guarantee *eventual* consistency.)

Relational databases also provide transactions that allow one user to perform a series of actions atomically—as if they were a single operation. This effectively locks all the records involved in those actions until the transaction is complete.

These features work well, but their record-locking behaviors can lead to a couple of problems.

First, most databases won't tell you who has a record locked. If someone is in the middle of editing a record in the Employees table, you won't be able to edit that record. Unfortunately, the database won't tell you that Frank has the record locked, so you can't go down the hall and ask him to release it. Or worse, you'll discover that Frank left his computer locked and went home for an early weekend so you're stuck until Monday. In that case, it will require database administrator powers and an act of Congress to unlock the record so you can get some work done.

A second issue is that a complicated series of locks adds to the database's load.

One technique that I've found useful for addressing these problems is to use a *turnkey record* to control access to a group of tables.

Suppose normalization has spread a work order's data across several tables holding basic information, addresses, phone numbers, email addresses, the actual items ordered, and other stuff. Now, suppose the system is designed to assign work orders to users for processing. It would be nice to lock a work order's data while a user is working on it so others can't blunder in and make conflicting changes. Unfortunately, it's wasteful to lock all of those records.

To use a turnkey record, add a LockedBy field to a table that is central to the work order. This is probably the table that contains the work order's basic information.

Now to “reserve” the work order for use by a particular user, the program sets this record's LockedBy field to the user's name. That token means that this user has permission to mess with all of the work order's records in all of its tables without actually locking anything. Because the user's name is in the database, other parts of the program can tell that the record is locked and by whom. The program can even allow an administrator with appropriate privileges to clear that field so you can fix the work order after Frank has gone home.

The one drawback to this method is that if Frank's computer crashes while he has the work order reserved, then it remains reserved just as if Frank has left for an endless vacation. To recover, you'll need to add an option in the program to clear those sorts of zombie reservations.

A similar technique gives most of the same benefits while removing the problems that come with a LockedBy field. Suppose you assign each work order to a particular person who then works on it, and no one else is allowed to work on that order.

To handle this case, add an `AssignedTo` field to the order. Some agent (either human or automated) sets the `AssignedTo` field and after that the field doesn't need to be changed. In that case, if Frank's computer crashes, his record is still assigned to him after he reboots. Because Frank is still the one who should work on the job, you don't need to clear this field. (In practice, however, there will always be a situation where someone needs to foist a job off on someone else for some weird reason, so you should allow some way for an administrator to step in and fix it if necessary.)

SUMMARY

This chapter described some common patterns that you can use to solve particular database design problems. For example, if you need to build a database that includes many-to-many relationships, you can use the pattern described in the section “Many-to-Many Associations” to implement that part of your relational database design.

In this chapter, you learned to model:

- Many-to-many relationships and multiple-object associations
- Repeated attribute associations
- Reflexive or recursive associations
- Temporal data
- Logging and locking

The next chapter does the opposite of this one. It describes common mistakes that people make when designing databases and explains ways to avoid those mistakes.

Before you move on to Chapter 10, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

EXERCISES

1. Parcheesi is a board game for two to four players. Make an ER diagram to record information about Parcheesi matches.
2. Build a relational model to record information about Parcheesi matches. Be sure to include a way to tell who finished first through fourth.
3. Chess enthusiasts often like to walk through the moves of past matches to study how the play developed. They even give names to the most famous of these matches. For example, the “Immortal Game” was played on June 21, 1851 by Adolf Anderssen and Lionel Kieseritzky (see http://en.wikipedia.org/wiki/Immortal_game).

The following text shows the first six moves in the Immortal Game:

```
e4 e5 f4 exf4 Bc4 Qh4+?! Kf1 b5?! Bxb5 Nf6 Nf3 Qh6
```

(If someone showed me this string and I wasn't thinking about chess at the time, I'm not sure whether I would guess it was an assembly program, encrypted data, or a program written in *Malbolge*. See <https://en.wikipedia.org/wiki/Malbolge>.)

Of course, a database shouldn't store multiple pieces of information in a single field, so the stream of move data should be broken up and stored in separate records. In chess terms, a *ply* is when one player moves a piece and a *move* is when both players complete a ply.

Figure 9.28 shows a semantic object model for a `CHESS_MATCH` class that stores the move information as a series of `Move` attributes, each containing two `Ply` attributes. The `Movement` field holds the actual move information (Qh4+?!) and `MoveName` is something like "The Sierpinski Gambit" or "The Hilbert Defense." `Commentary` is where everyone else pretends they know what the player had in mind when he made the move.

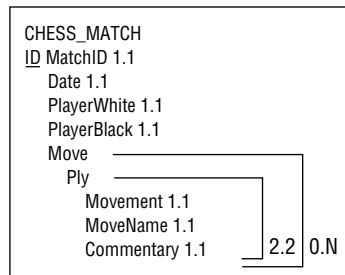


FIGURE 9.28

For this exercise, draw an ER diagram to show the relationships between the `Player`, `Match`, `Move`, and `Ply` entity sets.

4. Build a relational model to represent chess relationships among `Players`, `Matches`, `Moves`, and `Plies` tables. How can you represent the one-to-two relationship between `Moves` and `Plies` within the tables' fields? How would you implement this in the database?
5. Consider the relational model that you built for Exercise 4. The `Moves` table doesn't contain any data of its own except for `MoveNumber`. Build a new relational model that eliminates the `Moves` table. (Hint: Collapse its data into the `Plies` table.) How does the new model affect the one-to-two relationship between `Moves` and `Plies`?
6. Suppose you are modeling a network of pipes and you want to record each pipe's diameter. Design a relational model that can hold this kind of pipe network data.
7. Suppose you run a wine and cheese shop. Wine seems to get more expensive the longer it sits on your shelves, but most cheeses don't last forever. Build a relational model to store cheese inventory information that includes the date by which it must be sold. Assume that each batch of cheese gets a separate sell-by date.

8. Modify the design you made for Exercise 7 assuming each *type* of cheese has the same shelf life.

9. Suppose you need to store 10 TB of data for several years and you expect to access around 1 TB of that data per month. Build a table showing the cost for storage plans with prices in the following table. (These are repeated from the section “Deciding What to Temporalize” earlier in this chapter.)

PLAN	MINIMUM DURATION	STORAGE PER GB	RETRIEVAL PER GB
Standard	None	\$0.0200	\$0.00
Nearline	30 days	\$0.0100	\$0.01
Coldline	90 days	\$0.0040	\$0.02
Archive	365 days	\$0.0012	\$0.05

Which plan is the least expensive?

10

Avoiding Common Design Pitfalls

Chapter 9, “Using Common Design Patterns,” described some common patterns that you might want to use while designing a database. This chapter takes an opposite approach: it describes some common pitfalls that you don’t want to fall into while designing a database. If you see one of these situations starting to sprout in your design, stop and rethink the situation so that you can avoid a potential problem as soon as possible.

In this chapter, you will learn to avoid problems with:

- Normalization and denormalization
- Lack of planning and standards
- Mishmash and catchall tables
- Performance anxiety

The following sections describe some of the most common and troublesome problems that can infect a database design.

LACK OF PREPARATION

I’ve nagged about this in earlier chapters, but it’s time to nag again. Database design is often one of the first steps in development. It’s only natural for developers to want to rush through the design to get some serious coding done. It gives them something to show management to prove that they aren’t *only* playing computerized mahjong and reading friends’ MySpace pages and Facebook posts. Coding is also more fun than working on plans, designs, use cases, documentation, and all the other things that you need to do before you can roll up your sleeves and get to work.

Before you start cranking out tables and code, you need to do your homework. Some things that you need to do before you start wiring up the database are:

- Making sure you understand the problem
- Writing requirements documents to state the problem
- Building use cases to see if you have solved the problem
- Designing a solution
- Testing the design to see if it satisfies the use cases
- Documenting everything

Remember, the time you spend on up-front design almost always pays dividends down the road.

POOR DOCUMENTATION

This is part of preparation but it is so important and underappreciated that it deserves its own section. Many developers think of documentation as busy work or a chore to keep managers who have no real talents off their backs while they build elegant data structures of intricate beauty and complexity.

I confess that occasionally that's a handy attitude, but the real purpose of documentation is to keep everyone on the project focused on the same goals. The documentation should tell people where the project is headed. It should also spell out the project's design decisions so everyone knows how the pieces will fit together.

If the documentation is weak, different people will make different and possibly contradicting assumptions. Eventually, those assumptions will collide and you'll have to resolve the conflict. That will require developers to go back and fix work that they made under the wrong assumptions. That leads to more work, more errors, and copious bickering over whose fault it was.

The real fault was poor documentation.

POOR NAMING STANDARDS

In a sense, naming standards are part of documentation. When done properly, an object's name should give you a lot of information about the object. For example, if I tell you to build an `Employees` table, you probably know a lot about that table without even being told. You know that it will need name, address, phone number, email address, and Social Security Number information (in the United States, at least). In most companies, it will also need employee ID, hire date, title, salary, and payroll information (such as deductions, bank account for automatic deposit, and so forth). Somehow, it should probably link to a manager and possibly to projects. You got all of that from the single word "Employees."

Now, suppose I told you to build a `People` table, but I really want to use the table to hold employee data. You'd probably only put about half of the necessary fields in this table. You'd get the name and address stuff right, but you'd completely miss the business-related fields.

The problems become worse when you start working with multiple related tables and fields. For example, suppose you use employee IDs to link a bunch of tables together, but one table calls the linking field EmpNo, another calls it EmployeeId, and a third calls it Purchaser.

This might seem like a small inconvenience—and in isolation it is—but together a lot of little inconveniences can add up to a real headache. Inconsistent naming makes developers think harder about names than the things they represent, and that makes the team less productive and less accurate.

I have worked on projects where poor naming conventions made small changes take days instead of hours because developers had to jump back and forth through the code to figure out what was happening. Inconsistent naming by itself is unlikely to sink a project, but it is enough to nudge an already leaky ship toward the rocky shoals of disaster.

Pick names for the fields that will be used in more than one table and stick to them so that the same concept gets the same name everywhere. Then use those names consistently. Consistency is more important than following particular arcane formulae to generate names, although I will mention two useful conventions for naming database objects such as tables and fields.

First, don't use keywords such as TABLE, DROP, and INDEX. Though these may make sense for your application and they may be technically allowed by the database, they can make programming confusing. If one of these words really fits well for your project, try adding a word or two to make it even more descriptive. For example, if your database will hold seating assignments and it really makes sense to have a field named Table, try naming it TableNumber or AssignedTable instead.

Second, don't put special characters such as spaces in table or field names even if the database allows it. Although there are ways to use these sorts of names, it makes working with the database a lot more confusing. Remember, the point of good naming conventions is to reduce confusion.

For some more information on naming conventions, search online for “database naming conventions” or take a look at some of these sites:

- [https://en.wikipedia.org/wiki/Naming_convention_\(programming\)](https://en.wikipedia.org/wiki/Naming_convention_(programming))
- http://vyaskn.tripod.com/object_naming.htm
- www.geeksforgeeks.org/database-table-and-column-naming-conventions

Picking good names for tables is like a vocabulary test. You need to think of a word or short phrase that sums up as many of the features in the related items as possible so that someone else who looks at your table's name will immediately understand the characteristics that you're trying to record. The following table shows some examples.

A TABLE THAT HOLDS:	SHOULD BE CALLED:
Magazines, newspapers, and comic books	Periodicals
Things that your company sells, including physical items and services	Products
People who work in your restaurant, including servers, bussers, cooks, and greeters	Employees

A TABLE THAT HOLDS:	SHOULD BE CALLED:
Things that cost you money, such as groceries, gasoline, and fencing lessons	PersonalExpenses
Things that you pay for but that are for work purposes, such as stationery, stamps, and phone calls	BusinessExpenses

THINKING TOO SMALL

Too often developers design a perfectly reasonable database only to discover during the final project stages that it cannot handle the load being dumped on it. Make some calculations, estimate the database's storage and transaction loads, calculate the likely network traffic, and then multiply by five. For some applications, such as online web applications that can have enormous spikes in load over just a few hours, you might want to multiply by 10 or more.

Be sure you use a realistic model of the users' computers and networks. It's fairly common in software development to give the programmers building a system great big, shiny, powerful computers so they can be more productive. (It takes a lot of horsepower to play those interactive role-playing games quickly so you can get back to work.)

Unfortunately, customers often cannot afford to buy every user a new computer every two years. (Five developers \times \$3,000 = \$15,000. That's not exactly pocket money, but it's small change compared to \$2,000 \times 200 users, for a total of \$400,000.) Make sure your calculations are based on the hardware that the users will really have, not on the dream machine that you are using.

If your customers are "civilians" who download your app or who run it over the Internet, you should assume that the average user's hardware is at least a few years out of date. They may have a decent Internet connection and a few gigabytes of memory, but they may not have the latest solid-state disks and external graphics processing units.

If you don't think your architecture can handle the load, you should probably rethink things a bit. You might be able to buy a more powerful server with more disk space, move to a faster network, or split the data across multiple servers. If those tricks don't work, you might need to consider a three-tier architecture with different middle-tier objects running on separate computers. You might also need to think about moving some of the more intense calculations out of database code and moving them into code running on separate servers. In addition, you might also need to redesign the database to use turnkey records instead of using the database's record-locking tools. Finally, you might even need to split the database into disjoint pieces that can run on different computers.

Recent advances in cloud computing may help with some of these issues. Some cloud platforms can automatically distribute the load and increase compute and storage resources when they are needed. You can build your Pizza Cut app (which figures out the best way to slice a pizza so that the pepperoni is evenly divided among the slices) in a relatively small cloud database. When it becomes

massively successful, you can upgrade your service to gain more database space and processing power. If you want to use a cloud database to deal with huge potential growth, then you need to plan that in from the start.

Solving these problems may be difficult, but you should at least plan ahead and be prepared to face them. A sure way to disappoint customers is to get them all excited, release the database, and then tell them they can't use it for four months while you rethink its performance problems.

NOT PLANNING FOR CHANGE

As you design the database, look for places that might have to change in the future. You don't need to build features that might never be needed, but you don't want to narrow the design so those features cannot be implemented later.

In particular, look for exceptions in the data. Customers often think in terms of paper forms, and those are easy to modify. It's easy to cross out headings and scribble in the margins of a paper form. It's a lot harder to do that in a computerized system. (Drawing on the screen with dry erase markers doesn't work very well.)

Whenever you see two or more things that have a lot in common, ask the customers if those are enough or whether you'll sometimes need to add more. Listen for words such as "except," "sometimes," and "usually." Those words often hint at changes yet to come.

For example, suppose a customer says, "This field holds the renter's front binding tension, unless he's goofy-footed." Here the word "unless" tells you that this one field might not be good enough to hold all of the data. You'd better find out what "goofy-footed" means and change the database accordingly.

For another example, suppose the customer says, "The order form must hold two addresses, one for shipping and one for billing. Unless, of course, we're billing a split order." This says that two address fields (or groups of fields) may not be enough. At this point you probably need to pull the address data into a new table so you can accommodate any number of addresses, including the ones the customer hasn't remembered yet.

For a third example, suppose you're building a coaching tool for youth soccer teams. Figure 10.1 shows your initial design.

Let's review this design and identify any pieces that are likely to change later.

It's often useful to look at fields that allow a single value and ask yourself if they might need to change later. In this design, there are several such fields. It's also particularly useful to look at one-to-one relationships and this design contains some of those, too.

First, do we need to change the one-to-many relationship between Games and GamePlayers to a many-to-many relationship? Probably not. If the group of players is the same in any two games, that's more or less coincidence rather than a more formal arrangement such as an "A team" and a "B team." (I've known some frighteningly serious youth soccer coaches who might just do that—seriously scary individuals with clipboards yelling at the tops of their lungs at four-year-olds.)

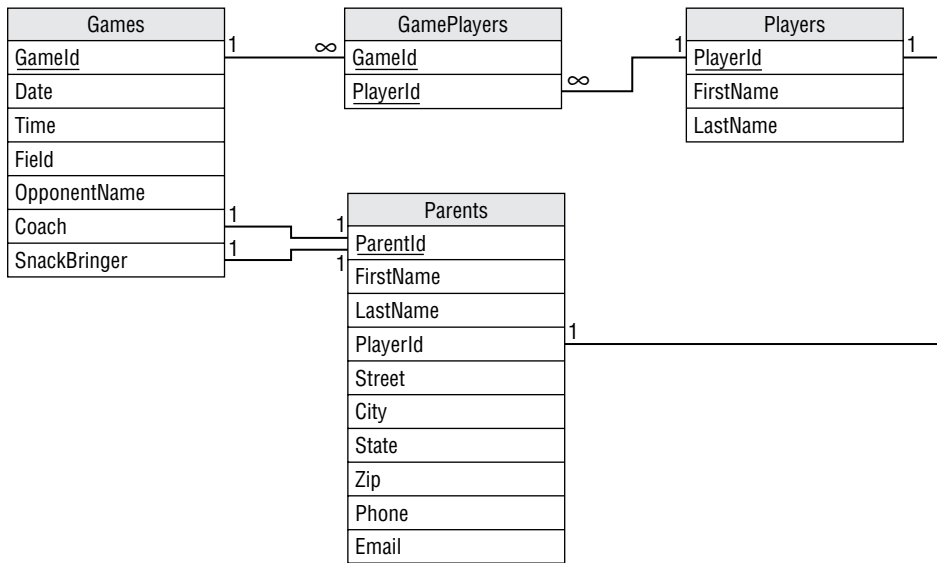


FIGURE 10.1

The many-to-one relationship between GamePlayers and Players, together with the Games/ GamePlayers relationship, helps model the many-to-many relationship between Games and Players, so it probably shouldn't change.

What about the one-to-one relationship between Parents and Players? This link implies that a parent can have only one player and that might not be a good assumption. What if a parent has two players on the same team? For the younger age brackets, you won't see players of different ages on the same team, but you will find twins on the same team.

This relationship also implies that each player has a single parent (which is unlikely until cloning techniques become more practical). You could add information about a second parent (in fact, that's a very common approach), but if a player's parents are separated and remarried, you might need up to four parents and sometimes you might need to contact any subset of them to figure out if a player will be at a game. It might make the most sense to just allow a player to have any number of parents and not worry about the details. (And don't even think about requiring players to have the same last names as their parents! The combinations, including hyphenated last names, are too numerous to contemplate.)

NoSQL? NO PROBLEM!

Because NoSQL databases don't need to use the tight constraints that relational databases do, they might be able to handle the unknown number of parents problem more easily. A document database can simply hold whatever information is available for each player. Some might list only one parent and others might list as many as needed.

The first one-to-one relationship between Games and Parents means that one parent will be the coach for that game. Is that a reasonable assumption? Will you ever need to worry about multiple coaches or assistant coaches? For a youth league, it's probably good enough to assume there is only one main coach and not worry about any others, so I wouldn't change that. But this is a good question to ask.

By far the most important piece of information in this database tells who is bringing a game's snacks. The second one-to-one relationship between Games and Parents means one parent will bring snacks for the game. Is that a reasonable assumption? In my experience, there has always been only one snack-bringer per game. As in the case with the coach, you can probably at least assume there is a main caterer and if someone else wants to bring extra cupcakes for a player's birthday, we just won't worry about it. But again, a good question.

One final place to look for these kinds of changes is in the fields within a table. In this design, the field that begs for multiple values is Phone. Lots of people have multiple phones and sometimes you might need to call several of them to track someone down (such as the all-important snack bringer!). I would split the Phone field off into a new table to allow parents to have any number of phone numbers.

Figure 10.2 shows the new and improved design.

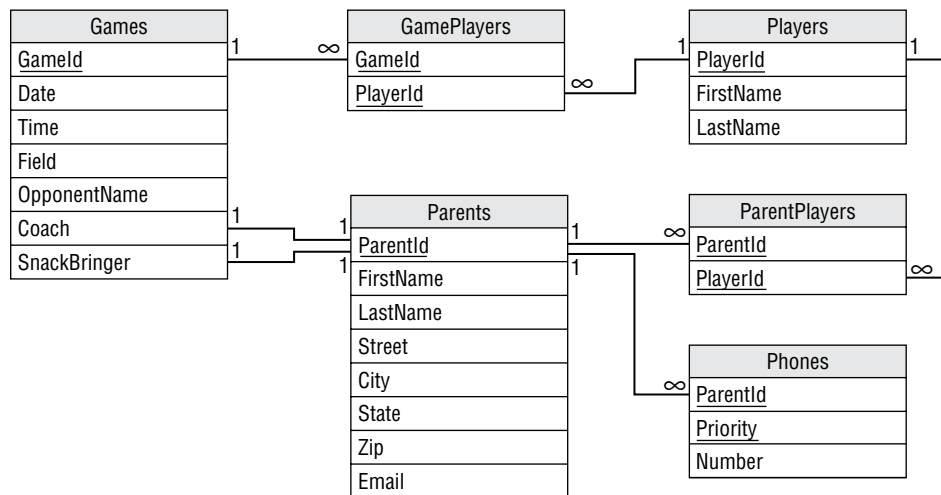


FIGURE 10.2

Again, you don't need to use psychic powers to build all of the features that the customer will need in the next 15 years, but keep your eyes open during requirements gathering and try not to close off opportunities for later change.

TOO MUCH NORMALIZATION

Taken to extremes, too much normalization can lead to a database that scatters related data all over the place for little additional benefit. It can make the design confusing and can slow performance.

When you normalize, think about what a change will cost and what benefits it will provide. Think about how the data will be accessed. If data is only read and written through stored procedures or middle-tier code, that code can help play a role in keeping the data consistent and might allow you to get away with slightly less normalization in the database's tables. Putting every table in fifth normal form or domain/key normal form isn't always necessary to keep the data safe.

I once worked on a project where a certain database developer (who coincidentally had just taken a class in database normalization) wanted to split every data value out into a separate table. For example, a customer record would contain little more than a `CustomerId`. Then a `Values` table would hold the actual data in its three fields `Id`, `ValueName`, and `ValueData`. To look up a customer's name, you would search the `Values` table for a record with `Id` equal to the customer's ID and `ValueName` equal to "Name." In some bizarre otherworldly sense, this table is very normalized and it lets you do some amazing things. For example, you could decide to add a new `EarSize` field to the customer data without changing the tables at all. However, that design doesn't reflect the structure of the data, so it would be next to impossible to use. (If you really think that you might need to add an `EarSize` field, then you might consider a NoSQL document database so you can add just about anything at any time.)

INSUFFICIENT NORMALIZATION

Although too much normalization can make the database slower than necessary, poor performance is rarely the reason a software project fails. Much more common reasons for failure are designs that are too complex and confusing to build, and designs that don't do what they're supposed to do. A database that doesn't ensure the data's integrity definitely doesn't do what it's supposed to do.

Normalization is one of the most powerful tools you have for protecting the data against errors. If the database refuses to allow you to make a mistake, then you won't have trouble with bad data later. Adding an extra level of indirection to gather data from a separate table adds only milliseconds to most queries. It's very hard to justify allowing inconsistent data to enter the database to save one or two seconds per user per day.

This doesn't mean you need to put every table in fifth normal form, but there's no excuse for tables that are not in at least third normal form. It's much too easy to normalize tables to that level for anyone to claim that it's not necessary.

If the code needs to parse data from a single field (Hobbies = "sail boarding, skydiving, knitting"), break it into multiple fields or split its values into a new table. If a table contains fields with very similar names (`JanPayment`, `FebPayment`, `MarPayment`), pull the data into a new table. If two rows might contain identical values, figure out what makes them logically different and add that explicitly to the table so that you can make a primary key. If some fields' values don't depend on the entire key, consider spreading the record across multiple tables.

If you use a NoSQL database, then you probably cannot normalize it the way you can with a relational database, but you should still protect it against invalid data. You need to write code that validates the data going into the database to prevent it from becoming inconsistent.

INSUFFICIENT TESTING

This problem is closely related to “Thinking Too Small” and “Too Much Normalization.” Some developers perform little or no testing before releasing a database into the wild. They run through a few tests to check correctness (the better ones go through all of the use cases) and assume that everything will work in the field. Then when customers try to use it under realistic conditions, the whole thing falls apart. They discover bugs that the testers missed and the performance is unacceptable.

Be sure to test the database and any attached applications thoroughly. Fully testing every nook and cranny of a system takes a lot of work, but it’s necessary. You need to be sure you exercise every piece of code, every table, and every constraint. You also need to perform load testing to see if the database can handle the expected load.

If you don’t find all of the bugs and bottlenecks, sooner or later the user will, guaranteed!

BOTHERSOME BUGS

It’s also pretty safe to assume that every nontrivial application contains at least some bugs no matter how much testing you perform. In that case, you cannot find every bug, so you might be tempted to not even try. The goal isn’t actually to catch every bug, but to find enough of them and to find the most likely to occur, so the probability of the users finding a bug is very small. The bugs will still be hiding in there, but if you only get one or two user complaints per year, you’re doing pretty well.

PERFORMANCE ANXIETY

Many developers focus so heavily on performance that they needlessly complicate things. They make a simple solution complicated and harder to build and maintain all in the name of speed. They denormalize tables to avoid using any more tables than necessary, and they build business rules into the database so they don’t need to use stored procedures or other code to implement them separately.

Modern hardware and software is pretty fast, however. Often, these CPU-pinching measures save only milliseconds on a one-second query. Think hard about whether a convoluted design will really save all that much time before you make things so complicated that you can’t build, debug, and maintain the application. If you’re not sure, either run some tests and find out or go with the simpler version and change it later if absolutely necessary. Usually, performance is acceptable, but a database that contains contradictory data is not.

THINKING SMALL

I once worked on a huge database application where a simple change to the data might require five or more minutes of recalculation. After about three days digging through horribly convoluted code and database structure, I found the problem. The original developers had used a bunch of tricks to perform calculations in some sneaky ways to save a little bit of time here and there. Then they had done something really silly that made them perform the same calculations again and again more than a hundred thousand times. They were so busy worrying about tripping over the blades of grass that they wandered blindly into a patch of poison ivy. I managed to speed things up a little, but a lot of their time-saving tricks were so buried in the underlying design that there wasn't much we could do without a total rewrite.

The moral is, you don't need to be stupid about design and ignore obvious chances to improve performance, but don't be so focused on the little things that they cloud the grander design.

First, make it work. Then make it work fast.

MISHMASH TABLES

Sometimes, it's tempting to build tables that contain unrelated values. The classic example is a DomainValues table that contains allowed values for fields in tables scattered throughout the database. For example, suppose the State, Brand, and Medium fields take values from lists. State can take values CA, CT, NV, and so forth; Size can take values Grande, Enorme, Gigantesco, and Intergalattico; and Medium can take values Oil, Acrylic, Pastel, and Crayon. You could build a DomainValues table with fields TableName, FieldName, and Value. Then it would hold records such as TableName = Artwork, FieldName = Medium, Value = Crayon. You would use this magic table to validate foreign keys in all the other tables.

This one-table-to-rule-them-all approach will work, but it's more of a headache than it's worth. The table is filled with unrelated values and that can be confusing. It might seem that having one table rather than several would simplify the database design, but this single table does so many things that it can be hard to keep track of them all. Just think about drawing the database design's ER diagram with this single table connected to dozens of other tables.

Tying this table to a whole bunch of others can make it a chokepoint for the entire system. It can also lead to unnecessary redundancy if multiple tables contain fields that have the same domains.

It's better to use separate tables for each of the domains that you need. In this example, just build separate States, Sizes, and Media tables. Although this requires more tables, the pieces of the design are simpler, smaller, and easier to understand.

Remember the rule that one table should do one clearly defined thing and nothing else. Although this kind of mishmash table has an easily defined purpose, it does not do just one thing.

Mishmash Bash

Consider the following mishmash DomainValues table.

TABLE	FIELD	VALUE
Customers	State	CO
Customers	State	KS
Customers	State	WY
Employees	State	CO
Employees	State	KS
Employees	State	WY
OrderItems	Size	Large
OrderItems	Size	Medium
OrderItems	Size	Small
Orders	ShippingMethod	Overnight
Orders	ShippingMethod	Priority
Orders	ShippingMethod	Snail Mail

How could you avoid building this mishmash table? What tables would use the new domain tables? To find out:

1. Figure out which records represent similar items.
2. Move similar records into smaller validation tables.
3. Define foreign keys to use the new tables for validation.

How It Works

1. Figure out which records represent similar items.

This table contains domain values for four tables (Employees, Customers, OrderItems, and Orders), so you might think that you need four domain tables. However, the table really only holds domain values for three kinds of fields: states, sizes, and shipping methods. Those define the groups of similar items, which means you only need three new domain tables.

continues

(continued)

2. Move similar records into smaller validation tables.

The States table contains the following data:

STATE
CO
KS
WY

The Size table contains the following data:

SIZE
Large
Medium
Small

Finally, the ShippingMethod table contains the following data:

SHIPPINGMETHOD
Overnight
Priority
Snail Mail

3. Define foreign keys to use the new tables for validation.
Instead of making every table validate domains against the mishmash table, the database now uses the following foreign key constraints:
 - Customers.State = States.State
 - Employees.State = States.State
 - OrderItems.Size = Sizes.Size
 - Orders.ShippingMethod = ShippingMethods.ShippingMethod

NOT ENFORCING CONSTRAINTS

When you design a table, you should write down the domains and other constraints for every field. Most database designers can handle that, but it's surprising how often those restrictions don't make it into the actual database.

When you start building the database, go through the list of field constraints and check them off as you implement them. Often, these are as simple as making a field required or writing field-level check constraints for a field.

You can rely on middle-tier objects and user interface code to enforce some of these, but the database is the final authority, so why not take advantage of its capabilities? You might want to allow the middle tier to verify that a flyball team's number of dogs is no more than 3 because it's a possibly changing business rule. It seems unlikely that a team would ever include -1 dogs, however, so let the database enforce that rule at least.

Databases can also often verify field formats. For example, some databases can verify that a phone number string has the format ###-###-####. You might want the user interface to validate this type of format, too, to save round-trips to the database, but there's no reason not to let the database do whatever it can to ensure that garbage doesn't slip into the data.

The database is pretty good at enforcing these simple rules. Let it do its job so it can feel appreciated.

OBSESSION WITH IDs

ID numbers are nice. They are relatively small and efficient, and it's easy to ensure that they are always unique. However, they don't have any real meaning. You can probably recite your name, address, and phone number easily enough, but do you remember your employee ID, utility company account number, and driver's license number? Unless you have a better memory than mine (which is likely, since I have a pretty poor memory) or someone took a shortcut when they defined their keys (in some states, your driver's license number is the same as your Social Security number), you probably don't remember all of these.

It's okay to have some tables with keys that are not artificial IDs, particularly if the data provides a nice unique key readymade for you. Books have International Standard Book Numbers (ISBNs) that uniquely identify them so, if you're tracking books, use ISBN instead of creating a new meaningless number. Products often have stock-keeping unit (SKU) or product numbers that are just as useful as an artificial number. Even keys that include multiple fields can be perfectly fine and give acceptable performance.

Three obvious times when you should create an artificial primary key are when:

- You might need to change the value of the natural key. (You shouldn't change primary key values and some databases won't even let you.)
- The natural key doesn't guarantee uniqueness.
- Adding an automatically generated surrogate key makes integration with other systems easier.

GUID GAMES

A *universally unique identifier* (UUID) is an artificially generated value that you can use to represent whatever you want. It's sort of like a serial number for anything. They also go by the name *globally unique identifier*, which is abbreviated *GUID*. (I prefer that to UUID because you pronounce GUID to rhyme with “squid.” I don't know how to pronounce UUID.)

The algorithms used to generate GUIDs make it extremely unlikely that two values will be duplicates. For example, I can generate a GUID to represent my bicycle and you can generate one to represent your sandwich and we are almost certain to get two different values.

There are two reasons I'm mentioning GUIDs now. First, this gives you another way to generate IDs for records that are not only unique within your table, but that are also unique across the entire universe. Second, many databases now have a GUID data type so they can efficiently store, index, and retrieve these values.

For more information on GUIDs, see https://en.wikipedia.org/wiki/Universally_unique_identifier. You can also find GUID generators online— for example, at www.guidgenerator.com.

Before you create a new key field, ask yourself whether it's really necessary.

IDs Undone

Consider the following tables:

- Customers with fields FirstName, LastName, Street, and so on
- ChessMatches with fields Date, WhitePlayerId, and BlackPlayerId
- Books with fields Title, Author, Year, Pages, Price, and Publisher
- InventoryItems with fields Name, Vendor, Description, and QuantityInStock
- WeatherReadings with fields Date, Time, Temperature, and Humidity

To figure out which of these probably needs an artificial primary key:

1. Look for a natural key.
2. Decide whether you will ever need to change the key's value.
3. Decide whether the key guarantees uniqueness.
4. Use the results from steps 1 through 3 to decide which tables need artificial IDs.

How It Works

1. Look for a natural key:
 - *Customers with fields FirstName, LastName, Street, and so on:* FirstName/LastName is a natural key.
 - *ChessMatches with fields Date, WhitePlayerId, and BlackPlayerId:* Date/WhitePlayerId/BlackPlayerId is a natural key.
 - *Books with fields Title, Author, Year, Pages, Price, and Publisher:* Title/Author is a natural key.
 - *InventoryItems with fields Name, Vendor, Description, and QuantityInStock:* Name/Vendor is a natural key.
 - *WeatherReadings with fields Date, Time, Temperature, and Humidity:* Date/Time is a natural key.
2. Decide whether you will ever need to change the key's value:
 - *Customers with fields FirstName, LastName, Street, and so on:* Depending on your application, you might need to change FirstName or LastName values. For example, when some friends of mine got married they decided to change both of their last names to something completely different. This means FirstName/LastName may not make a good primary key.
 - *ChessMatches with fields Date, WhitePlayerId, and BlackPlayerId:* The application should never need to change Date/WhitePlayerId/BlackPlayerId after the data for a match is entered.
 - *Books with fields Title, Author, Year, Pages, Price, and Publisher:* The application should never need to change Title/Author after the data for a book is entered.
 - *InventoryItems with fields Name, Vendor, Description, and QuantityInStock:* There's some chance that a vendor will rename a product. In that case, Name/Vendor might not make a good primary key, although you could also treat the renamed product as a new product instead and just stop using the old record.
 - *WeatherReadings with fields Date, Time, Temperature, and Humidity:* The application should never need to change Date/Time after the weather data for a reading is entered.
3. Decide whether the key guarantees uniqueness:
 - *Customers with fields FirstName, LastName, Street, and so on:* FirstName/LastName is probably not enough to guarantee uniqueness. There are just too many John Smiths and Maria Garcias out there.
 - *ChessMatches with fields Date, WhitePlayerId, and BlackPlayerId:* Unless you allow two players to play more than one match at the same time (which is possible), Date/WhitePlayerId/BlackPlayerId is unique.
 - *Books with fields Title, Author, Year, Pages, Price, and Publisher:* It's extremely unlikely that the same author (or two authors with the same name) would publish two books with the same title, but Title/Author does not absolutely guarantee uniqueness. You could make the key more unique by adding Year, but it's still not an absolute guarantee.

- *InventoryItems with fields Name, Vendor, Description, and QuantityInStock*: It's very unlikely that a particular vendor will have more than one product with the same name (and you might not want to do business with such a confused vendor), so Name/Vendor guarantees uniqueness.
 - *WeatherReadings with fields Date, Time, Temperature, and Humidity*: Unless you take more than one reading at the same time, Date/Time guarantees uniqueness.
4. Use the results from steps 1 through 3 to decide which tables need artificial IDs:
- *Customers with fields FirstName, LastName, Street, and so on*: A customer's FirstName/LastName value might change and these fields don't guarantee uniqueness, so this table really needs an ID field.
 - *ChessMatches with fields Date, WhitePlayerId, and BlackPlayerId*: Date/WhitePlayerId/BlackPlayerId is a fine primary key.
 - *Books with fields Title, Author, Year, Pages, Price, and Publisher*: It is unlikely that you would have trouble with Title/Author/Year as a primary key, but it's easy enough to use the book's ISBN code to remove all doubt. Adding that data will make it a lot easier to integrate with other systems that already use ISBN, such as online bookstores.
 - *InventoryItems with fields Name, Vendor, Description, and QuantityInStock*: Name/Vendor is unlikely to cause trouble but there is a chance. Most products have part numbers, SKUs, Universal Product Codes (UPCs—those values that go with the barcode), or other ID values that make excellent primary keys, so I would add one of those. That might also make integration with other systems easier.
 - *WeatherReadings with fields Date, Time, Temperature, and Humidity*: The combination Date/Time guarantees uniqueness and you shouldn't need to change those values later, so this table shouldn't need an artificial key.
-

NOT DEFINING NATURAL KEYS

Closely related to obsession with IDs is not defining natural keys. A natural key is a key that you might actually use to search the data. If a table is only there to provide detail for another table, then an ID makes a reasonable link between the two, but if you will be searching a table for natural values such as names or phone numbers, then those might make good keys.

For example, suppose the Addresses table uses CustomerId to link back to Customers records. Typically to look up an address, you will look up the customer's record, and then look at its related address records. It's unlikely that you will know a customer's address and need to look up their name, so you probably don't need to define a key on the Addresses table's Street, City, State, or Zip fields. In the unusual circumstance where you do know the customer's address but not their name, you can still look up the record, it will just take longer.

In contrast, suppose the Customers table uses CustomerId for its primary key. What are the chances that you'll need to look up customers by ID? Unless the ID is something special (such as phone

number if you're running a phone company), then that number is meaningless to mere mortals, so you're more likely to look up a customer by name. You can make that faster by making the name a key. It can't be the primary key because it doesn't guarantee uniqueness and you might need to change a customer's name, but making it a non-primary key will make searches by customer name faster. If that's the search you perform the most, this key is a worthwhile addition to the database.

You might also consider using other fields as keys, too. For example, you might want to be able to list customers by city or ZIP Code to prepare mass mailings (postal spam). In that case, making City and ZipCode keys might also be useful if you perform those searches often.

SUMMARY

This chapter described some common mistakes that people make while designing databases. If you don't pay attention to the ideas described in this chapter, you might end up rediscovering the importance of proper planning, documentation, and testing by painful experience.

Some of these lessons I've learned the hard way, some by studying others' mistakes, and some through research. Take my word for it when I say it's a lot easier (and more humorous) to learn about these issues here instead of through firsthand experience.

In this chapter, you learned the importance of:

- Advanced preparation through thorough requirements gathering
- Good design practices such as using naming conventions and making a design before building the database
- Anticipating changes and increased database load
- Using the database's tools to ensure that values are within their allowed domains
- Avoiding artificial keys if they are unnecessary and making natural keys even if they cannot be a primary key

This chapter ends the book's main discussion of general database design topics. The next few chapters explain some practical database implementation issues, paying extra attention to the Microsoft Access and MySQL database management systems.

Before you move on to Chapter 11, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

EXERCISES

1. Suppose your client runs a ski rental shop and he's that most dreaded of all clients: one who thinks he knows something about databases. He has designed a Customers table that looks like this:

NAME	ADDRESS	CITY	ZIP	PHONE1	PHONE2	STUFF
Sue Rank	2832 Shush Ln. #2090	Boiler	72010	704-291-2039		Downhill, Snowboarding
Mark Bosc	276 1st Ave	East Fork	72013	704-829-1928	606-872-3982	X-country

The Stuff field contains a list of Downhill, X-country, Snowboarding, and so forth. Your client plans to get the customer's state from the Zip value when he's sending out mailings.

Your client wants you to build Orders and other tables to go with this one.

For this exercise, explain to your client what's wrong with this table, paying particular attention to the ideas covered in this chapter.

- Suppose you're building a system to track rentals for a company that owns two flying car rental stores and plans to open a third next year. What special considerations do you need to ponder that might not be as important if the client were, for example, a well-established party rental store. (They rent chairs, punch bowls, bouncy castles, and other stuff for large parties.)

- What's wrong with an Addresses table that includes these fields:

- CustomerId
- StreetName
- StreetNumber
- StreetPrefix
- StreetSuffix
- StreetPreDirectional
- StreetPostDirectional
- ApartmentOrSuite
- Floor
- City
- Neighborhood
- State
- Zip
- PlusFour

The Zip and PlusFour fields hold detailed ZIP Code data. For example, if a customer's ZIP+4 is 02536-2918, then the Zip field would hold 02536 and the PlusFour field would hold 2918.

- Consider the relational design shown in Figure 10.3.

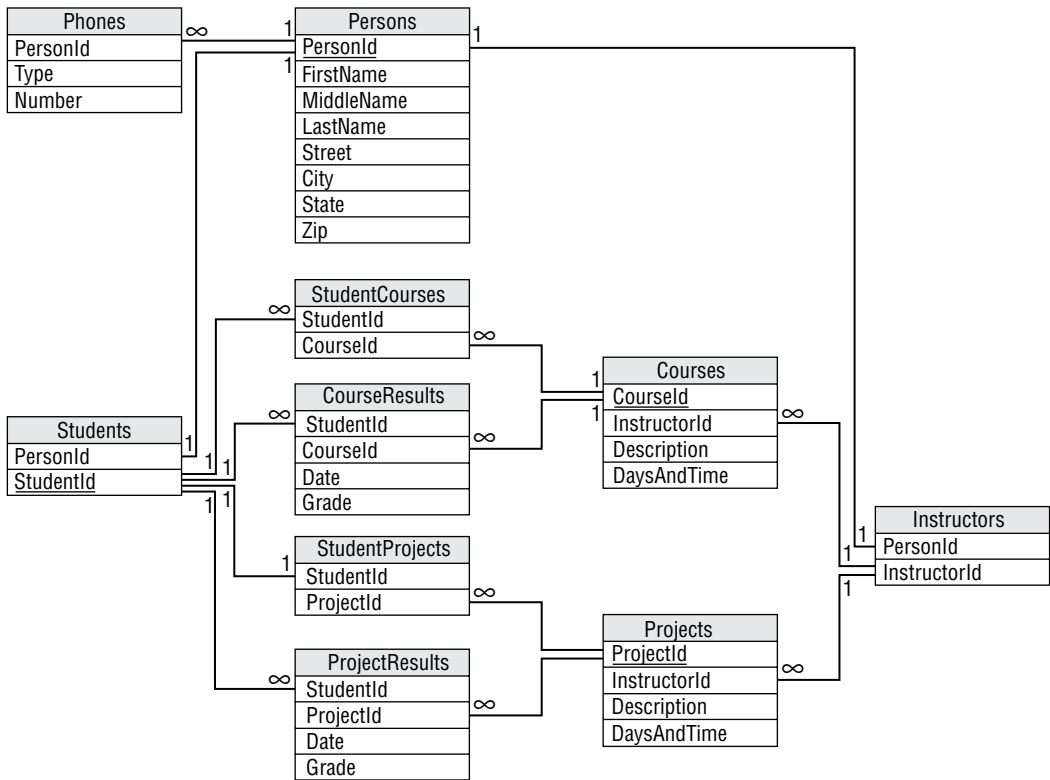


FIGURE 10.3

List the database constraints that you would place on the fields in this model and explain how you would implement each of those constraints. (Feel free to add new tables if necessary.)

PART 3

A Detailed Case Study

- ▶ **Chapter 11:** Defining User Needs and Requirements
- ▶ **Chapter 12:** Building a Data Model
- ▶ **Chapter 13:** Extracting Business Rules
- ▶ **Chapter 14:** Normalizing and Refining

The chapters in the next part of the book walk through a fictitious case study for the Pampered Pet, a pet supply store. These chapters work through the steps for building the company a new database, from requirements gathering to implementation.



Chapter 11 begins the development process by gathering user requirements. It walks through the project's initial steps, including meeting the customers, picking the customers' brains, and examining the database's performance, security, and data integrity needs. It finishes by developing use cases and requirements documents.

Chapter 12 uses the requirements gathered in Chapter 11 to build initial data models. It builds semantic object models and entity-relationship models to satisfy the project's requirements and makes some improvements on the initial models.

Chapter 13 examines the models so far and determines which business rules should be implemented inside the database and which should be extracted into middle-tier or user interface code.

Chapter 14 converts the models built in Chapter 13 into a normalized relational model.

As you work through these chapters, you might find it useful to build an example of your own. Pick some other business, fictitious or otherwise, and work through the corresponding phases of database design and development as you read along.

11

Defining User Needs and Requirements

The first step in designing and building a database is gathering user requirements. You cannot build a database to solve your users' needs unless you understand those needs. This chapter walks you through the process for the Pampered Pet.

In this chapter, you see examples of:

- Identifying user requirements
- Determining what the database's main entities are
- Defining use cases to verify that requirements have been met

The scenarios described here do not necessarily present the most efficient possible outcome. Ideally, your customers know exactly what they need and give you their full cooperation while spelling out the requirements in crystal-clear detail. Things don't always go that way, however (in fact, I've never seen it happen that way), so neither do the steps described here.

Perhaps you'll get lucky and things will go more smoothly than some of the examples described here, but you should realize that at least sometimes people skills are as important as database design skills during this phase.

MEET THE CUSTOMERS

Requirements gathering for this project begins with a series of meetings in the Pampered Pet's back room (where they hold pet training courses, so it smells a bit funny). Occasionally, customers have an agenda for these introductory meetings, but as often as not they won't have been through the process of building a database before, so it'll be up to you to keep things moving in the proper direction.

The initial meetings with the Pampered Pet are mostly to introduce you and the customers so you can get to know each other. For this example, four key players attend the first meeting:

- Bill Wye “The Pet Store Guy,” the founder and owner of the Pampered Pet. Bill is the one who decided the company needed a complete database. He’s the Executive Champion. Because this is a fairly small company and he’s at the very top of the food chain, there will be no serious disputes over whether the project should go forward, although his help might be needed to keep unenthusiastic participants pulling in the right direction.
- Alicia Myth, the store’s manager. Alicia has been working at the store since it was opened and knows just about everything there is to know about the business. She spends more time keeping things organized and running than anyone else at the store and knows more about the day-to-day business than anyone else, even Bill.
- Charlie “Ice” Walker is a trainer specializing in aggressive dogs. He also works shifts at the store and knows a lot about day-to-day operations. He doesn’t care as much about selling as he does about training. He has a very “whatever” attitude about the new database system.
- Sveta Clark is a dog and exotic bird trainer who also works at the store about half the time. Sveta isn’t convinced that the store needs a new computerized system and just wants to be left alone to do her job the way she always has. She’s definitely more comfortable with animals than people.

During the first meeting, you realize that Bill isn’t going to play much of a role in this project. He’s the one who initiated the whole thing, and he can help keep things moving if they get bogged down, but he’s not going to be directly involved on a daily basis. (He’s too busy with his classic car collection.)

Charlie and Sveta are not particularly enthusiastic about the whole thing, but with Alicia’s encouragement and an occasional nudge from Bill, they’ll cooperate. Unfortunately, Alicia’s time will be largely taken up by running the store, so Charlie and Sveta will be your Customer Representatives. Alicia will be around to make critical decisions, break ties, and generally look menacing if Charlie and Sveta become difficult.

The purpose of the initial meeting is mostly to let you and the customers meet and get comfortable with each other. There’s a chance that you’ll get some serious work done, but it’s more likely that this meeting will stay at a fairly high level.

A couple of questions that you should try to address right away are:

- What do the customers expect to get from the new system?
- Why do they think they need a new database system?
- Does this system enhance or replace an existing system?
- Are there other systems with which this one must interact?

These are big-picture executive-level questions that Bill can answer, and because he probably won’t be present at the more detailed meetings still to come, he should answer them now. In the ensuing discussion, during which Alicia talks more than anyone else, you realize that Bill doesn’t really know

what he'll get out of the database. Although Bill made the final decision, it was Alicia who thought a new database system would help.

Alicia thinks the new system can help better track inventory (there have been times when they've run out of products without realizing it). She also hopes it can streamline payroll (it currently takes quite a while for her to track everyone's hours), and she believes that it can help the company figure out how to reach new customers and better market their training courses.

There is no existing system and no other systems with which this one must interact. The current process is manual and uses paper order forms, paper shift assignments, and paper timesheets.

You should address these issues as soon as possible to give the customers the right expectations. A new database can help with inventory tracking and streamline payroll. It will also help identify which customers take which courses, but it's not really a marketing tool. It will tell you about existing customers but not about people who have never interacted with the company. Alicia seems a little disappointed, but she still sees some worthwhile benefits and is ready to get started.

After the initial meeting, where you and the key players get to know one another a bit, you begin a series of meetings where you try to pick Charlie's and Sveta's brains to define the project's requirements.

PICK THE CUSTOMERS' BRAINS

Sometimes, customers have already prepared requirements documents before they bring in database designers and other developers, but often gathering requirements is part of the development process. In this project, that means locking Charlie and Sveta in a room and picking their brains.

This is where you pull out your prepared list of questions (described in the "Bring a List of Questions" section in Chapter 4, "Understanding User Needs.") The following sections run through a series of sample questions and discuss the answers that Charlie and Sveta give.

Determining What the System Should Do

Charlie says he doesn't know what the system should do and Sveta doesn't care. For goals, they basically repeat what Bill and Alicia said during the kickoff meeting.

This could be a problem. If these two really won't join the team, they won't be much help, and right now they're all you've got. You could ask Alicia to have a word with them, but it would be better if they enlist semi-voluntarily.

Rather than making waves right away, you decide to make some educated guesses and see if you can get Charlie and Sveta more interested in the project.

Instead of starting with basic system features, you decide to jump to something that Charlie and Sveta might find more interesting: training course information. You quickly sketch out a form that displays course information, including the general description, dates, locations, and the instructor. You draw a little smiley face picture for the guide's picture and write Charlie's name beneath.

Training is the real reason why Charlie and Sveta work here and it's what they love, so this gets their attention. They start giving you useful information about what's involved in defining a course: general description (which includes the type of animal: dog, cat, bird, fish), locations, times, dates, price, and maximum number of participants.

You let Charlie and Alicia brainstorm and tell stories about courses they've run and funny pets they've met so they can build some enthusiasm. Before lunch you snap their pictures with your camera phone. Then during lunch you email the pictures to yourself and paste them into the quickly assembled form shown in Figure 11.1. A quick mockup makes the application seem more real to customers.

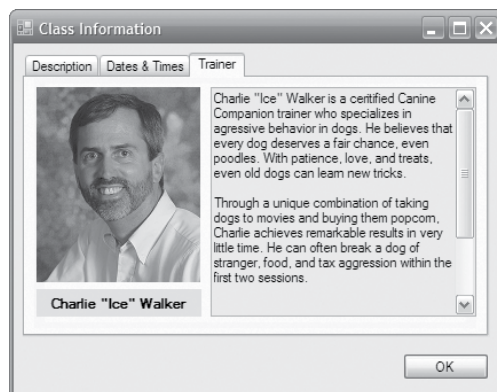


FIGURE 11.1

This might seem like a somewhat silly exercise, but a little glitz and showmanship can go a long way toward uniting a project team. A simple mockup such as this one shows the customers something tangible early on that they can understand. Database models and relational designs are your eventual goal, but they are too abstract to generate much excitement with all but the nerdiest customers. Putting a customer's face on a mockup will almost certainly grab and hold their attention.

After lunch, you present your mockup and let Charlie and Sveta suggest improvements, which might be as far as you get during this session. Before the next session, you build an improved mockup with Sveta's picture so she doesn't feel left out and to show that you listened to their suggestions and made appropriate changes.

The following list describes a few things to consider when you build this kind of mockup:

- **Decide what tool you want to use**—Use a tool that you find comfortable. I used an older version of Visual Basic to build the form shown in Figure 11.1 (you can download the free Community edition at <https://visualstudio.microsoft.com/downloads>), but you should use whatever tool you like the most. If you've programmed before, you can use a programming tool. If you like building web pages, build one. If you have experience with Microsoft Word or some other word processor, use it. Use drawing tools or even paper and colored markers. You're going to rebuild all of this later anyway, so use whatever is easiest for you.

- **Put an image or graphic on the form**—Graphics make a form more interesting, so add some graphics to your mockup. Use familiar images such as the customers' pictures, company logos, or relevant clipart. (A dog leaping to catch a disc or a cat stalking a mouse would work for the Pampered Pet.) You can add interest with a special font for the company's name and with colors. Don't go overboard, though. You want the graphics to tie the form to the customer, not to distract from the form's purpose.
- **Add details that will be meaningful to the customer**—Use terms and concepts that are familiar to the customer. Use the customer's company name and logo. This doesn't have to be perfect (the lawyers will eventually tell you that you're not using exactly the right shade of blue anyway), and the information doesn't have to be perfectly correct. Just use anything that the customers will find familiar and comfortable so they can see how this project connects with their situation.

MAPPING MINNESOTA

On one project for the Minnesota Department of Transportation, I gave the login form the shape of the state of Minnesota. When they first saw the demo project, they were amazed. They had never seen shaped forms before.

Determining How the Project Should Look

In later sessions with Charlie and Sveta, you can start defining what the finished project will look like. Sketch out screens or let them do it for you. These don't need to be perfect. The goal is to figure out what data the system must contain to provide those screens, not to do the user interface designer's job.

After this step, you should have a rough list of forms that the application should contain. In this case, they might include:

- **Login**—Log in with username and password.
- **Orders**—Write up sales orders for customers.
- **Inventory**—View and update inventory levels.
- **Courses**—Create and edit courses.
- **Employees**—Enter and edit employee information. Employees include salespeople and trainers.
- **Shifts**—Assign work shifts.
- **Customers**—Enter and edit information about customers, particularly courses they're taking.

You should sketch out these forms to make them easier for the customers to understand. The sketches not only help customers visualize the forms and help them remember what information should be on them, but they also help you understand what data they should contain.

The program also needs the following reports:

- **Weekly Work Schedule**—Displays the work shifts for the week.
- **Course Schedule**—Displays courses scheduled during a user-entered period of time.
- **Course Roster**—Prints a course roster for a trainer to use during a course.
- **Reorder Items**—Lists items that need to be reordered.
- **Sales Stats**—Lists employees sorted by amount of sales.
- **Item Sales**—Lists high and low selling items.
- **List Customers**—Displays a list of customers selected by as yet unknown criteria. (This sort of feature is very useful to users. Fortunately, most databases will allow the user to perform ad hoc queries so they can invent new reports long after the application has been built.)

Determining What Data Is Needed for the User Interface

Review the form sketches and figure out where the data should come from for the forms. For example, Figure 11.2 shows a customer order form mocked up in Microsoft Word.

Customer Order		4/1/27	
		Sold By: <u>Sveta Clark</u>	
Items:			
Description	Price Each	Quantity	Total Price
Doggy Diet food, senior	\$34.95	1	\$34.95
Squeaky toy	\$2.99	3	\$8.97
Misc. mouse, small	\$1.99	4	\$7.96
Class: Mouse Socialization, 4/1/10 – 4/29/10	\$79.50	1	\$79.50
		Subtotal	\$131.38
		Tax	\$6.57
		Shipping	\$0.00
		Grand Total	\$137.95
Purchased By:			
Name:	<u>Robert Terwilliger</u>		
Street:	<u>1265 Petlover Ln</u>		
	<u>Apt. 12</u>		
City:	<u>Menagerie</u>		
State:	<u>WI</u>	ZIP:	<u>73827</u>
Email:	_____		
	Phone:	_____	
Ship To:			
<input checked="" type="checkbox"/> Same As Above			
Name:	_____		
Street:	_____		

City:	_____		
State:	_____	ZIP:	_____

FIGURE 11.2

At this point, you don't need to figure out exactly where every piece of data will be stored in the database—you just need to discover what data is needed on the form. By now, you can probably guess that there will be a Customers table for information such as Name and Street that doesn't change with every order, an Orders table to hold order-specific information such as the date and shipping information, and an OrderItems table to hold information about the items that make up the order.

The following list shows the main types of data needed for each of the forms identified in the previous section.

- **Login**—Username and password. The program will use the database's integrated security, so this data doesn't need to be stored in the database.
- **Order**—Customer data (name, address), order data (date, shipping address), order items (item, quantity), employees (sold by).
- **Inventory**—Inventory item data (UPC, description, buy price, sell price, quantity in stock, quantity to require reorder, reorder amount, vendor information).
- **Course**—Course information (description, trainer, price, dates, times, location, animal type), customer information.
- **Employee**—Employee information (name, address, Social Security number, skills).
- **Shift**—Employee, work shifts (date, time).
- **Customer**—Customer data (name, address, shipping address, orders, courses, email for newsletter).

Determining Where the Data Should Come From

You can start making a list of the entity sets that will take part in the new system as soon as you have a decent understanding of how the system will work. Be sure you don't grow emotionally attached to this very first design, however, because things are likely to change as you learn more.

For now, you can assume there will be entity sets corresponding to the forms listed in the previous section. You don't need to include a table for login information because the program will use the database's security features to log in. The program will prompt for a username and password, and then try to connect to the database using them. If the connection fails, the program doesn't let the user do anything else.

The initial list of entities includes Order, Inventory, Course, Employee, Shift, and Customer.

At this point, you might realize that an order could include any number of items, so you know that you will need a separate OrderItem entity to handle the one-to-many relationship.

Determining How the Pieces of Data Are Related

Think about how these entities are related. The forms that you sketched out show relationships among pieces of data so they can help. For example, the order form shown in Figure 11.2 contains information about the customer, the order, and the order items, so those entities must be related.

The following list describes relationships defined by the initial forms:

- **Order**—This entity relates customer data, order data, order items, and employees. Some order items might be training courses, so it also relates course data.
- **Inventory**—This entity is fairly self-contained. Other entities such as Order refer to it, but it doesn't need to refer to others.
- **Course**—This entity relates basic course information, trainer information, and customer information.
- **Employee**—This entity is fairly self-contained. Other entities such as Course refer to it, but it doesn't refer to others.
- **Shift**—This entity relates work shifts and employee data.
- **Customer**—This entity is fairly self-contained. Other entities such as Order and Course refer to it but it doesn't refer to others.

Reports also define relationships among pieces of data. The following list shows where data comes from for the previously defined reports:

- **Weekly Work Schedule**—This report relates employees and shifts.
- **Course Schedule**—This report relates basic course information and trainers. It doesn't need to list customer information.
- **Course Roster**—This report relates basic course information, trainers, and customers.
- **Reorder Items**—This report uses only inventory data.
- **Sales Stats**—This report relates inventory and employees.
- **Item Sales**—This report uses only inventory data.
- **List Customers**—Because the selection criteria for these reports are not yet defined, you can't know exactly which entities might be involved. However, you can make some guesses. Users will probably want to search for customers based on the items they purchased and courses they took. It's conceivable that they would want to search for customers who purchased items from a particular employee (perhaps so they can apologize), but most of the reports seem to relate customer, order, inventory, and course data.

Where's the Data?

Having a general idea of what data is required to build a report is very different from actually building the report. Often when you try to identify the fields needed to build a report, you will find holes in your understanding of the project. To fill in some of those holes, follow these steps for the Sales Stats and Item Sales reports:

1. Determine exactly where the data comes from.
2. Decide whether it seems likely that the database can build these reports quickly enough to satisfy the users.

How It Works

1. Determine exactly where the data comes from.

To generate the Sales Stats report, you need to figure out how much each employee sold during a certain period of time. For each employee in the Employees table, you need to find the corresponding Orders items with dates within the desired time period. You will then have to look up the corresponding order items, calculate their total prices, and add up the results.

To generate the Item Sales report, you must figure out how many of each type of item was sold during a particular time period. To get this data by date, you'll have to search the Orders table for orders placed during the time period. For each order, you'll look up the order's items and add up the number of each item sold.

2. Decide whether it seems likely that the database can build these reports quickly enough to satisfy the users.

If a user wants to look at Sales Stats or Item Sales figures for the past month, these queries shouldn't be a problem. They will probably involve a reasonable number of records, so the report will run quickly.

If the user wants to look at data for the last year, this might take slightly longer but should still be reasonable for the Pampered Pet. If the store were a large chain, these reports could take much longer. (Walmart has more than two million employees worldwide, but they probably keep their data spread out on a bunch of different databases and only look at summaries on a global scale.)

In each of these cases, the report will probably run reasonably quickly if it doesn't cover too big a time span. Once you realize this, you might want to inform the customers so they can decide how important these reports are. You might also want to tell them that reports covering longer time periods (for example, the previous year or year-to-date) will take longer than reports covering the previous month.

Determining Performance Needs

The system must be fast enough to be usable by an employee working with a customer. This is a typical case of "fast enough is fast enough." Unfortunately, "fast enough" isn't a verifiable quantity. You need to write down an explicit definition of what "fast enough" means so you can tell if you're meeting your goal.

For an interactive application, a good rule of thumb is a 5-second response. If the application takes longer than 5 seconds to respond to a request, users grow impatient. (They check their email, wander off to get coffee, bump into each other in the break room and start to chat, and pretty soon the 10-second response takes 30 minutes.)

So, the goal of the system will be to respond to interactive requests within 5 seconds 90 percent of the time. Reports that are run daily should finish within 5 minutes and reports run less often can take as long as 15 minutes. (These are very generous limits for this application and should be easy to achieve, but they should be acceptable for most applications.)

Although this is an important application, it does not need 24/7 support. The data is fairly important, however, particularly the data concerning future events such as orders that you have not yet delivered, customer enrollment in courses, and future work shifts. Because the system doesn't need to run 24 hours a day, it can be shut down nightly for backups (and to save a little electricity). A manual system (using pens, notebooks, sales slips, manual credit forms, and so forth) should be in place in case of a power failure or in case the system crashes for some other reason.

POWERED DOWN

Have you ever been in a store during a power failure or computer system crash and they couldn't process credit cards or even take cash? Not long ago my local McDonald's was closed most of a day because their computers were down. You'd think they could make and sell hamburgers for cash without a computer. During a recent power failure, my local grocery store had emergency power for its registers and computers so, while they were frantically moving cheese and salads into refrigerated trucks, they could still sell everything else.

Determining Security Needs

At first, Charlie and Sveta don't see any need for extra levels of security. You point out that things such as setting prices, reordering inventory, and assigning work shifts can only be performed by Alicia (the store manager), so the system needs at least two classes of users: general employees and managers.

It will probably also be useful to have a third class of user to perform system administration tasks, although Alicia may perform that role as well.

The general employees will make and fulfill orders. Managers (Alicia) will set prices, run reports, and assign work shifts. System administrators will change system parameters, define reports, and perform backups.

Charlie and Sveta probably won't like the idea of an audit trail keeping track of their every move, but most of what they do (creating orders) should include their usernames anyway in the Sold By field, so perhaps audit trails may not be necessary.

You can also point out that, of course, everyone trusts Charlie and Sveta, but who knows if we can rely on the cobra trainer that the store may hire someday?

This is a place where you should look for future modifications. As long as Alicia makes all of the systemwide changes (changing prices and assigning work shifts), you probably don't need audit trails. Orders hold a username in the Sold By field so that you know who created an order. If the system doesn't allow users to change orders after they are placed, there aren't too many opportunities for Charlie and Sveta to cause serious damage, which means any other changes are Bill's or Alicia's fault.

However, if the store ever gets a second manager, it might be handy to record who makes what changes.

Only Bill can really decide whether audit trails are necessary. They're not too hard to implement, at least if you plan for them in advance, so let's include them in this design.

Determining Data Integrity Needs

To properly address this issue, start making a list of the fields that belong to each of the database entities. For example, the following table gives the data types and constraints for the fields in the Order entity.

FIELD	REQUIRED?	DATA TYPE	DOMAIN
Date	Yes	Date	Any date.
FirstName	Note 1	String	Any first name. Not validated.
LastName	Note 1	String	Any first name. Not validated.
Street	Note 1	String	Any street name and number. Not validated.
City	Note 1	String	Any city name. Not validated?
State	Note 1	String	Foreign key to States table.
Zip	Note 1	String	Valid ZIP Code. Not validated?
Email	No	String	Valid email address. If provided, send the customer a monthly email newsletter.
HomePhone	Note 2	String	Valid 10-digit phone number.
CellPhone	Note 2	String	Valid 10-digit phone number.
SameAsAbove	Yes	Boolean	If unchecked, and we're shipping, then the Ship To fields are required.
ShipToFirstName	Note 3	String	Any first name. Not validated.
ShipToLastName	Note 3	String	Any first name. Not validated.
ShipToStreet	Note 3	String	Any street name and number. Not validated.
ShipToCity	Note 3	String	Any city name. Not validated?
ShipToState	Note 3	String	Foreign key to States table.

FIELD	REQUIRED?	DATA TYPE	DOMAIN
ShipToZip	Note 3	String	Valid ZIP Code. Not validated?
SoldBy	Yes	Reference	Reference to employee information.
Description	Yes	String	Foreign key to Inventory table.
PriceEach	Yes	Currency	Taken from Inventory table.
Quantity	Yes	Integer	Greater than zero.
TotalPrice	Yes	Currency	Calculated from PriceEach and Quantity.
Subtotal	Yes	Currency	Calculated from the Items.
Tax	Yes	Currency	Calculated from the Subtotal.
Shipping	Yes	Currency	Greater than or equal to 0.
GrandTotal	Yes	Currency	Subtotal + Tax + Shipping.

- **Note 1**—This field is required if the customer is signing up for a course or if we’re shipping products to the customer.
- **Note 2**—If the customer is signing up for a course or if we’re shipping products to the customer, at least one of the HomePhone and CellPhone fields is required.
- **Note 3**—This field is required if we are shipping products to the customer and SameAsAbove is false.

The customer contact fields (FirstName, LastName, Street, City, State, and Zip) are only required if we are shipping the order to the customer or if the customer is enrolled in a course. If we are shipping, either the Same As Above box must be checked or the user must fill in the Ship To address fields.

Inventory

Build a table similar to the previous one for the InventoryItem entity.

1. List the fields.
2. Determine which are required.
3. Determine their data types.
4. Determine their domain requirements.

How It Works

The following table describes the fields in the InventoryItem entity.

FIELD	REQUIRED?	DATA TYPE	DOMAIN
UPC	Yes	String	Valid UPC values.
Description	Yes	String	Any description.
BuyPrice	No	Currency	Greater than 0.
SellPrice	Yes	Currency	Greater than 0.
QuantityInStock	Yes	Integer	Greater than or equal to 0.
StockLocation	Yes	String	Where the item is stored when not on display.
ShelfLocation	Yes	String	Where the item is stored when on display.
ReorderWhen	No	Integer	Greater than 0. If null, don't reorder automatically.
ReorderAmount	No	Integer	Greater than or equal to 0. If null, someone must specify the amount when reordering.
Vendor	No	Reference	Vendor information (name, address, and so on).

The UPC, Description, SellPrice, and QuantityInStock fields are required. The BuyPrice and Vendor will be filled in after the first purchase. When the database first goes into use, however, those values may not be known so they cannot be required.

The Vendor field contains a bunch of data such as name, address, and phone number, so it is a reference to another entity that we need to add to the model.

WRITE USE CASES

One of the most important parts of identifying customer requirements is writing use cases, which help drive the database toward its final goals and help keep developers on track. They let you test whether the project is moving closer to completion and let you verify that you have met your goals after you're finished.

The following list shows use cases for the Pampered Pet database:

- Login
 - Log in successfully.
 - Log in unsuccessfully.

- Orders
 - Create a new order for a new customer (see the “Create a customer record” use case).
 - Create a new order for an existing customer.
 - Modify a pending order.
 - Cancel a pending order.
 - Fulfill an order (ship the items).
- Inventory
 - View all inventory.
 - View low inventory.
 - View excess inventory.
 - Add an item to inventory.
 - Remove an item from inventory.
 - Modify an item in inventory.
 - Reorder low inventory.
 - List best- and worst-selling items in the last week, month, quarter, and year.
- Courses
 - Create a course.
 - Modify a course.
 - Delete a course.
 - Display and print a list of current and future courses.
 - Select a course and display its information.
 - Enroll a customer in a course.
 - Remove a customer from a course.
 - “Delete” a course after it has been completed by marking the record as inactive.
 - Print a flyer about a course.
 - Print a course roster.
- Employees
 - Create a new employee.
 - Modify an employee.
 - “Delete” an employee by marking the record as inactive.

- List employees and total sales for the last week, month, quarter, and year sorted by employee or total sales.
- Verify that sensitive information (salary, Social Security number, and so on) are visible only to manager and administrator.
- Shifts
 - Assign work shifts for a week.
 - Display and print work shifts for a week.
 - Copy shifts from one week to a new week for all employees.
 - Copy shifts from one week to a new week for one employee.
 - Modify work shifts.
 - Reassign a shift to a different employee.
 - Verify that no one can modify work shifts for past weeks.
- **Customers**—Information about customers, particularly courses they’re taking.
 - Create a customer record.
 - Modify a customer record.
 - Print or email general customer mailing.
 - Print or email customer mailing based on criteria (for example, customers who took or are taking a particular course, or customers who have a certain kind of pet).
 - Display and print customers selected by ad hoc criteria.
 - “Delete” a customer record by marking the record as inactive.

Note that it’s important to write use cases *before* you build the database. Developers who build use cases after the fact tend to slant the tests toward what the application *can actually do* rather than what it *should do*. (Sort of like the politician who predicts prosperity if they’re elected and then after a year of recession says, “See, I told you there were tough times ahead!”)

What’s the Use?

Each use case must be specified in enough detail that someone can try it out and decide whether the database or project has passed the test. Ideally, the instructions should be simple enough that less experienced developers or even users can try them out while the more experienced developers concentrate on making excuses and fixing the problems that the use cases uncover.

Write out a detailed description for the use case “Display and print customers selected by ad hoc criteria.” Give the use case these sections:

1. Goals
2. Summary
3. Actors

4. Normal Flow

5. Alternative Flow

Refer to Chapter 4 for more information on use cases if necessary.

How It Works

1. Goals

Allow the user to display and print customer lists using ad hoc criteria.

2. Summary

Allow the user to enter criteria to select customers. Display a list of the selected customers. Let the user print the list. Let the user jump from a customer in the list to that customer's detailed information.

3. Actors

Only the Manager and Administrator should be able to use this feature.

4. Normal Flow

Here's where the bulk of the test case begins. The following steps test the normal flow:

- a. User selects Reports ⇨ List Customers from menu.
- b. Customer List screen appears.
- c. User selects fields (Zip, TotalPurchases, LastPurchaseDate) and operators (<, >, >=) from combo boxes. User clicks List button.
- d. The screen displays a list of the selected customers.
- e. User selects Data ⇨ Details from the menu to display a dialog box where the user can check the fields that should be displayed in the list. The list clears after new selections are made. The user can click List again to rebuild the list.
- f. The user double-clicks a customer's entry to open that customer's detailed information.
- g. The user selects File ⇨ Print from the menu to print the list with the selected fields. (Be sure to test lists containing 1, 2, and 3 pages.)

5. Alternative Flow

These steps test unusual or exceptional circumstances. The following steps test unusual conditions:

- a. Criteria select no customers.
 - b. User enters no criteria (should select all customers).
 - c. User selects no fields for the list (should make empty list). (Question: Always include customer name?)
 - d. User tries to print empty list (should refuse).
-

WRITE THE REQUIREMENTS DOCUMENT

The requirements document is the first blueprint detailing the project's scope. It sets the tone for future development and guides developers as the project progresses, so it's a very important document.

Unfortunately, it's also fairly long. This chapter identified about a half dozen main tables (later chapters will define more) and around 50 use cases, which means a reasonable requirements document would probably take 50 to 75 pages. To save space, a full requirements document for the Pampered Pet database isn't included here.

DOCUMENT DIMENSIONS

Fifty or so pages is about the shortest requirements document that I've seen on a formal project. I've worked on some projects with requirements documents around 500 pages long stored in multiple ring binders.

Before wrapping up, however, this section shows two key pieces of the requirements document: the mission statement and the executive overview.

The mission statement is a *very* brief declaration of the project's overall purpose. Sometimes, it is the only part of the requirements document that upper management reads, so it needs to be written for "big-picture" executives. (Insert your own joke about "big-picture" executives here.) Ideally, it should include at least a little content so the executives can discuss the project in the clubhouse after a hard round of golf.

The mission statement for the Pampered Pet database might read:

MISSION STATEMENT

The Pampered Pet Database will allow management to better track and understand its customers, orders, and inventory. It will provide streamlined administration of currently manual processes such as inventory tracking and work shift assignment to allow key personnel to dedicate additional time to more productive uses. Its data-tracking capabilities will allow management to better identify customer purchasing trends so the company can position itself to take best advantage of emerging industry trends. The database will truly allow the Pampered Pet to move aggressively into 21st century data management and forecasting.

Seriously, this mission statement in all of its polysyllabic execuspeak splendor isn't quite as silly as it sounds. It gives your Executive Champion some useful information and buzzwords that can be used to fight for resources if necessary and to defend the project from possible outside interference.

That probably won't happen on this project because the Pampered Pet is a small company and no one can really challenge your Executive Champion Bill.

For more hands-on managers such as Bill Wye, you should also provide an executive summary. This gives him a little more information if he needs it or he's just curious while not flooding him with so many details that his eyes glaze over. It explains what and why but not how.

The following bulleted list makes a concise executive summary that identifies the project's key points.

EXECUTIVE SUMMARY

The Pampered Pet Database will allow management to better:

- Track customer orders and fulfillment.
- Identify customers with particular purchasing histories.
- Identify customers with a history of taking training courses.
- Streamline work shift assignment.
- Identify products that are hot.
- Identify products that are underperforming.
- Identify salespeople who are over- or underperforming.
- Manage inventory tracking.
- Optimize inventory levels.

DEMAND FEEDBACK

It's important to get feedback at every stage of development. Remember, the longer a mistake is in the system, the harder it is to fix. Mistakes made during requirements gathering can throw the whole effort out of whack.

Unfortunately customers, particularly those who know the most about their business, are often very busy and may not feel they have time to look over the requirements documents thoroughly and provide feedback. Because this feedback is so important, you might need to push a bit. In this example, that means pestering Alicia mercilessly until she makes time to review the plan so far. When she has a chance to look things over thoroughly, she finds the following mistakes:

- Some inventory items such as live food (for example, crickets, mealworms, feeder guppies) and pet feed have expiration dates. That means:
 - The InventoryItem entity needs a new ExpirationDate field.
 - The system needs a new Expiring Inventory report.

- Employees don't always show up for their shifts on time and sometimes leave early, particularly if business is slow. That means the work shift data is not enough to determine the hours that an employee actually worked. In turn, that means the database needs a new TimeEntry entity.
- The system should have a report that shows how much money each employee earned for the store during a particular week, month, quarter, or year.
- The system should print payroll checks and record the date on which they were printed.
- Alicia mentions that they might want to provide direct deposit at some point.

This may seem like a lot of changes, but it's really not so bad. The basic database structure is close to correct. The only real changes are one new field, one new entity, and a couple of reports. This chapter won't make these changes, but the following chapter will include them.

SUMMARY

Any project begins with requirements gathering. If you don't have a good understanding of the customers' needs, then you have little chance of building an effective solution for the customers' problems.

This chapter described the requirements gathering phase for the Pampered Pet database project. In this chapter, you saw examples of:

- Meeting with customers to identify requirements
- Building a mockup to increase customer understanding, buy-in, and enthusiasm
- Defining the database's main entities and their relationships
- Determining data integrity requirements for entities
- Defining and writing use cases
- Obtaining feedback and refining the requirements

After you gather requirements information, you're ready for the next stage of design and development: building a data model. The following chapter describes this phase for the Pampered Pet database project.

Before you move on to Chapter 12, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

EXERCISES

1. Make a table showing the data integrity needs for the Course entity. Note any special requirements and conditions, which fields are required, and any relationships with other entities.
2. Make a table showing the data integrity needs for the Employee entity. Note any special requirements and conditions, which fields are required, and any relationships with other entities.

3. Make a table showing the data integrity needs for the Shift entity. Note any special requirements and conditions, which fields are required, and any relationships with other entities.

4. Make a table showing the data integrity needs for the Customer entity. Note any special requirements and conditions, which fields are required, and any relationships with other entities.

5. Make a table showing the data integrity needs for the TimeEntry entity. Note any special requirements and conditions, which fields are required, and any relationships with other entities.

6. Make a table showing the data integrity needs for the Vendor entity. Note any special requirements and conditions, which fields are required, and any relationships with other entities.

12

Building a Data Model

The previous chapter described requirements gathering for the Pampered Pet database project. It took the basic requirements and used them to build the fundamental entities that will take part in the database's operations.

This chapter builds more formal data models describing those entities. Semantic object models emphasize the entities' fields, and entity-relationship diagrams emphasize the relationships among them.

In this chapter, you'll see examples of:

- Converting requirements entities into semantic objects
- Splitting off repeated data into new objects
- Converting requirements entities and semantic objects into entity-relationship diagrams
- Converting semantic object models and entity-relationship diagrams into relational models

SEMANTIC OBJECT MODELING

Semantic object models have the advantage that they are relatively close in structure to the kinds of entity definitions that you typically get out of requirements gathering. They focus on the attributes that objects have. That is the same type of information that you get by studying the customer's needs and user interface mockups, and then figuring out where those mockups will get their data.

Building an Initial Semantic Object Model

To build a semantic object model, review the tables showing data integrity needs that were presented in the section "Determining Data Integrity Needs" in Chapter 11, "Defining User

Needs and Requirements.” The chapter’s text showed the data needed by the Order and Inventory-Item entities. The exercises built tables showing the data needed by the Course, Employee, Shift, Customer, TimeEntry, and Vendor entities. Chapter 11 also discussed the relationships among those entities.

To convert the data requirements tables in Chapter 11 into semantic objects, simply convert the entity’s pieces of data into attributes. Then add object attributes to represent relationships with other object classes.

For example, the following table summarizes the Course entity’s fields given in Chapter 11.

FIELD	REQUIRED?	DATA TYPE	DOMAIN
Title	Yes	String	Any string
Description	Yes	String	Any string
MaximumParticipants	Yes	Integer	Greater than 0
Price	Yes	Currency	Greater than 0
AnimalType	Yes	String	One of: Cat, Dog, Bird, and so on
Dates	Yes	Dates	List of dates
Time	Yes	Time	Between 8 a.m. and 11 p.m.
Location	Yes	String	One of: Room 1, Room 2, yard, arena, and so on
Trainer	No	Reference	The Employee teaching the course
Students	No	Reference	Customers table

One non-obvious detail is that Dates values should be after the current date when you create a new Course entity. It doesn’t make sense to schedule a new Course in the past (unless you’re training Temporal Terriers or Retrodoodles). After you create the Course, however, the record will continue to exist, so you can’t make this a database constraint. Instead, the user interface should verify the date when you create or reschedule a Course.

The Course entity has two relationships, one to the employee teaching the course (Trainer) and a second to the customers taking the course (Students). Figure 12.1 shows the corresponding semantic object class.

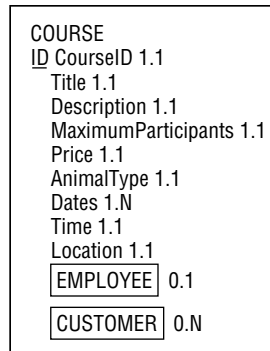


FIGURE 12.1

A Little Class

Define a semantic `EMPLOYEE` class.

1. Write down the data requirements for the Employee entity.
2. Convert the entity's fields into attributes.
3. Add object attributes to represent relationships between Employee and other entities.

How It Works

1. The following table shows the data requirements for the Employee entity identified in Chapter 11.

FIELD	REQUIRED?	DATA TYPE	DOMAIN
FirstName	Yes	String	Any first name.
LastName	Yes	String	Any last name.
Street	Yes	String	Any street name and number. Not validated.
City	Yes	String	Any city name. Not validated?
State	Yes	String	Foreign key to States table.
Zip	Yes	String	Valid ZIP Code. Not validated?
Email	No	String	Valid email address. If provided, send the customer a monthly email newsletter.
HomePhone	No	String	Valid 10-digit phone number.

FIELD	REQUIRED?	DATA TYPE	DOMAIN
CellPhone	No	String	Valid 10-digit phone number.
SocialSecurityNumber	Yes	String	Valid Social Security number.
Specialties	No	String	Zero or more of: Dog, Cat, Horse, Bird, Fish, and so on.

- The FirstName, LastName, Street, City, State, Zip, Email, HomePhone, CellPhone, SocialSecurityNumber, and Specialties fields all turn into attributes in the `EMPLOYEE` class. Obviously in different countries you'll need to use different address and phone number formats, and you'll need to replace Social Security number with whatever is appropriate. We'll ignore these issues to keep things simple.
- The Employee entity is related to the entities Course (an employee teaches courses), Shift (an employee is assigned to work a shift), and Time Entry (an employee actually works sometimes). Figure 12.2 shows the initial model for the `EMPLOYEE` class.

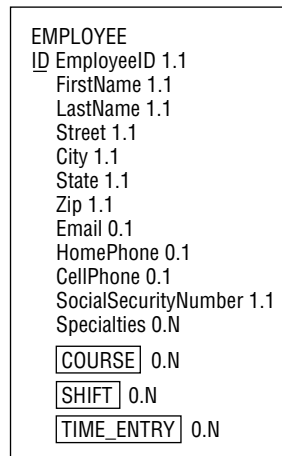


FIGURE 12.2

Improving the Semantic Object Model

Figure 12.3 shows a first attempt at building a semantic object model for the major entities identified so far.

Notice that the relationships in Figure 12.3 are two-way. If object A is related to object B, then object B is related to object A. For example, in this model the `EMPLOYEE` class contains an object attribute referring to `COURSE` and the `COURSE` class contains an object attribute referring to `EMPLOYEE`.

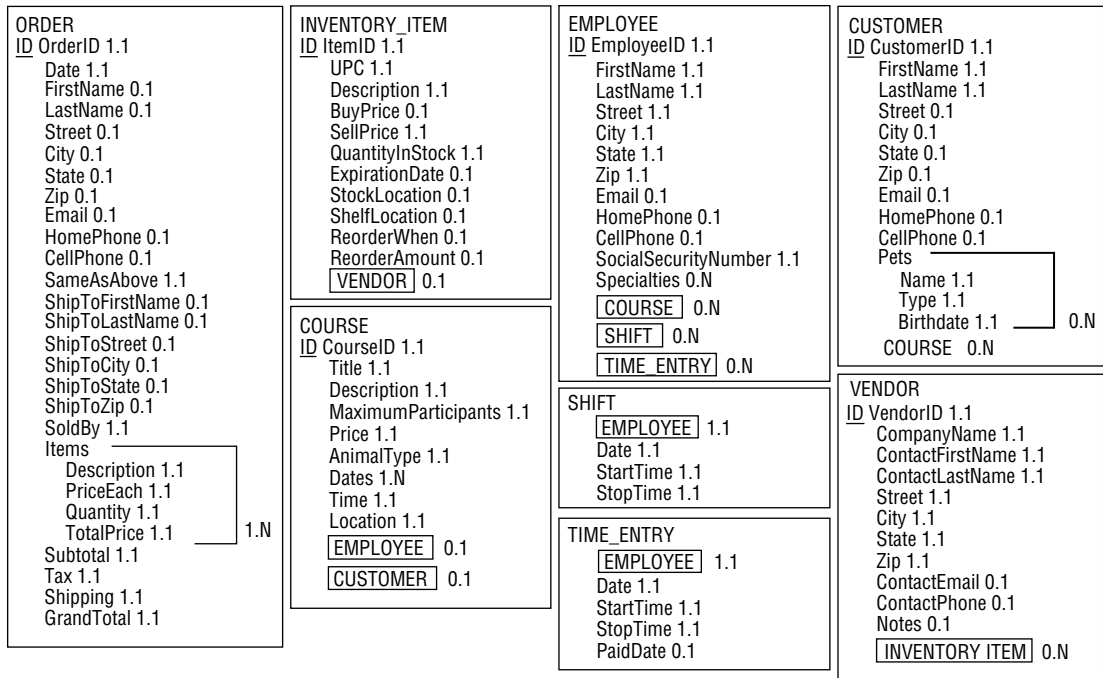


FIGURE 12.3

A quick look at Figure 12.3 uncovers several problems. First, the `ORDER` class contains two addresses: the customer’s address and a shipping address. They are the same kind of data, so they should be represented by a repeating multivalued attribute.

This model doesn’t acknowledge the relationship between orders and customers. A customer places an order, but there’s no link between the `ORDER` and `CUSTOMER` classes. The model should be changed to make that relationship explicit.

Furthermore, one of the addresses contained in the `ORDER` class is actually the customer’s address. That address is already represented in the `CUSTOMER` class, so it’s not needed in `ORDER`.

The `ORDER` class’s second address is the shipping address. It probably makes sense to leave that address in the `ORDER` class rather than moving it into `CUSTOMER` because it tells where that particular order should be shipped. If this address is missing, the order should be shipped to the customer’s address.

Because `ORDER` and `CUSTOMER` both contain addresses, it makes sense to create a new `ADDRESS` class to hold address data for both of those classes.

Figure 12.3 also shows that the `CUSTOMER`, `EMPLOYEE`, and `VENDOR` classes share several attributes in common. They all include name, address, email, and phone information, which makes intuitive sense because customers, employees, and vendors are all types of people.

To recognize the relationship among customers, employees, and vendors, it makes sense to build a PERSON parent class that holds name, address, email, and phone information. The CUSTOMER, EMPLOYEE, and VENDOR classes then become subclasses of the PERSON class.

Finally, the phone information in the CUSTOMER, EMPLOYEE, and VENDOR classes is not exactly identical. The CUSTOMER and EMPLOYEE classes include both home and cell numbers, whereas the VENDOR class has only a single contact phone number. That makes sense (most vendors don't want you calling their employees at home), but it's easy to generalize the model slightly and allow the PERSON class to hold any number of phone numbers of various kinds. A VENDOR object may never need a home phone number, but it doesn't hurt to allow the possibility.

Figure 12.4 shows the improved model.

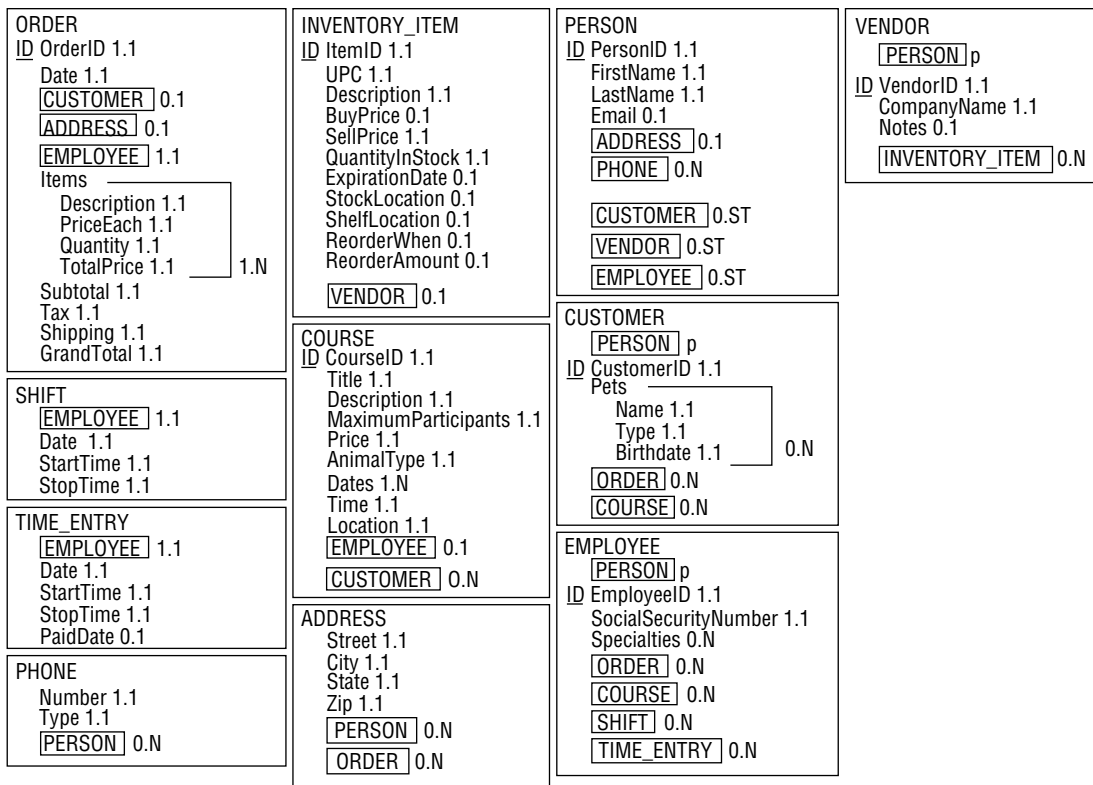


FIGURE 12.4

Note that some of these steps used to improve the model actually make the database more normalized. Many people think of normalization as a step that occurs after the data model is complete, but it really occurs throughout the data modeling process. As you see parts of the database that need to be normalized (and with experience you'll see them earlier and earlier), go ahead and fix them even if the model isn't complete yet.

ENTITY-RELATIONSHIP MODELING

Though semantic object models are fairly easy to build from lists of the database's main objects and their properties, they have the disadvantage that their structure doesn't closely match that of a relational database. Although the objects typically map into relational tables, the semantic object model doesn't emphasize the relationships among the entities. It also allows data arrangements that don't fit the relational model, such as attributes that are repeated any number of times within the same entity.

An entity-relationship model has a structure that's closer to the one used by relational databases, so it makes some sense to convert the semantic object model into a set of ER diagrams.

Building an ER Diagram

To start converting the semantic object model into ER diagrams, consider a particular semantic class and build a corresponding entity set. Connect it to other entity sets representing the class's object attributes.

Finally, consider the class's group attributes. If a group attribute is repeated, you should probably move it into a new entity connected to the original one. If a group attribute occurs only once, you might still think about moving the data into a new entity to either allow repetition later or to make similar data uniform across other entities. If a `STUDENT` class contains a single `Address` group attribute, it might be worth moving the `Address` data into a new table that holds address data for all kinds of entities (`Instructor`, `Employee`, and so forth).

For example, the `ORDER` class shown in Figure 12.4 is one of the more complicated classes, having relationships with three other classes: `CUSTOMER`, `ADDRESS`, and `EMPLOYEE`. To start building the ER diagram, you would create an `Order` entity set and connect it to `Customer`, `Address`, and `Employee` sets.

The `ORDER` class has one repeating group attribute: `Items`. Move that data into a new `InventoryItem` entity set and connect it to the `Order` entity.

For each of the relationships, think about how many of each type of entity could be associated with a single instance of the other entity type. For example, the `Order` entity is related to the `Customer` entity. A single `Order` must have exactly one `Customer`, so the `Customer` end of the relationship gets cardinality 1.1. Looking at the relationship from the other end, a single `Customer` might have 1 or more `Orders`, so the `Order` end of the relationship gets cardinality 1.N.

Similarly, you can find the cardinalities for the `Order/Employee`, `Order/Address`, and `Order/InventoryItem` relationships.

Figure 12.5 shows the ER diagram for the `Order` entity and its relationships.

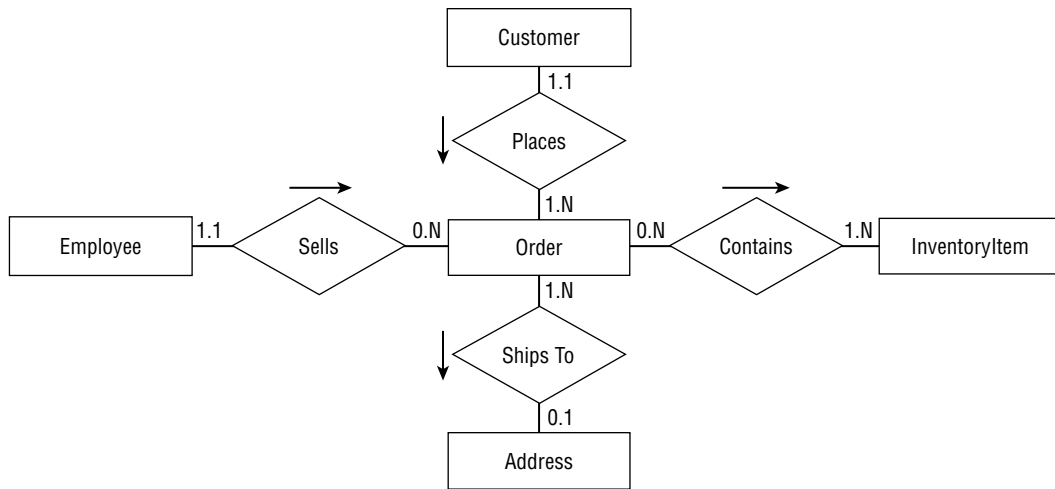


FIGURE 12.5

A Matter of Course

Make an ER diagram representing the `COURSE` class shown in Figure 12.4.

1. Make a Course entity set.
2. Make entity sets corresponding to the `COURSE` class's object attributes and connect them to the Course entity.
3. Consider any group attributes and decide whether to move them into new entities.

How It Works

1. Simply create a rectangle to hold the new Course entity.
2. The `COURSE` class has object references to `EMPLOYEE` and `CUSTOMER`, so you should create Employee and Customer entities and connect them to Course.
3. The `COURSE` class doesn't have any group attributes, so you don't need to move them into new entities. Figure 12.6 shows an ER diagram for the Course entity and its relationships.

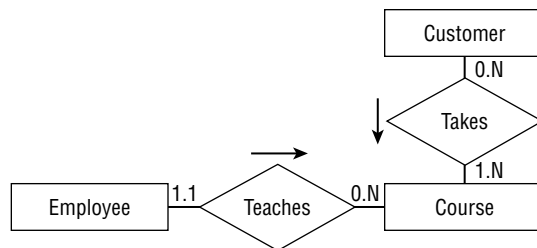


FIGURE 12.6

Building a Combined ER Diagram

After you build separate ER diagrams for each of the classes defined by the semantic object model, you can combine them into one big diagram. The individual diagrams are enough to let you understand the entities' relationships on a local level, but a combined diagram can help show larger patterns of relationship.

Sometimes, it can be tricky arranging the entities so their relationships don't overlap and are easy to read. In that case, it is sometimes useful to leave parts of the model out and show them in separate diagrams.

Figure 12.7 shows the combined ER diagram for the bottom-level classes modeled in Figure 12.4. To keep things a bit simpler, the diagram displays the Customer, Employee, and Vendor entities but does not show the fact that they are subclasses of the `PERSON` parent class.

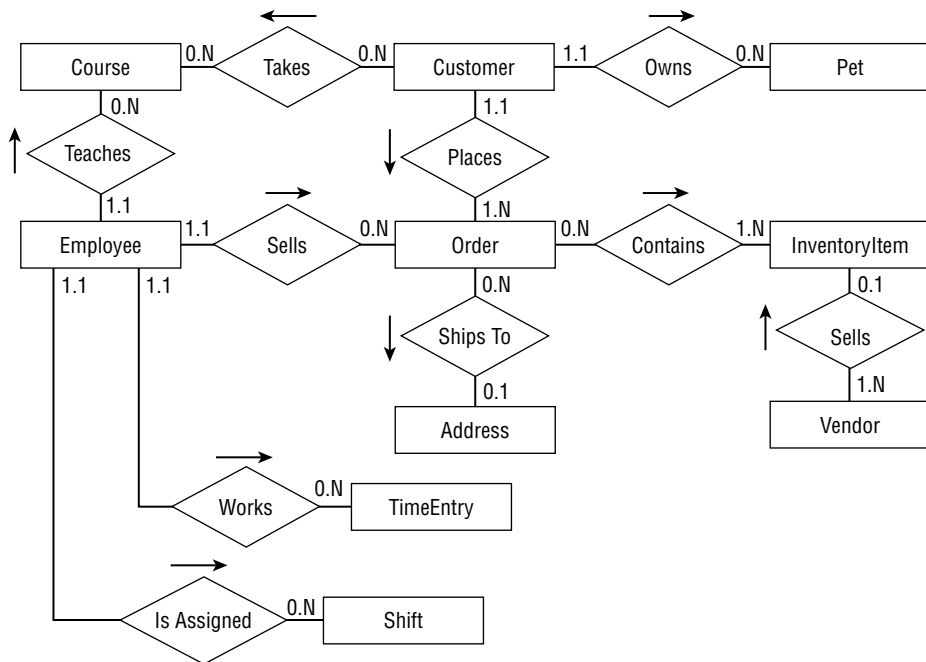


FIGURE 12.7

The diagram shown in Figure 12.7 uses more descriptive and business-oriented terms wherever possible. For example, from a purely theoretical perspective, you could say that an Employee “has a” Shift, “has a” TimeEntry, and “has a” Course. That would be more uniform but would make the diagram much harder to read.

OWNERS PWNEED

The phrase “Customer Owns Pet” is a bit tricky where I live in Boulder, Colorado. Here people decided that pet owners would be more responsible and caring if they were called “guardians” instead of “owners,” so all of the city documents were changed appropriately. My tax dollars hard at work! I suppose for cities such as Boulder, San Francisco, Berkeley, and others we’ll have to make a special edition of the book that changes the “Owns” relationship to “Is The Responsible And Caring Guardian Of.”

Figure 12.8 shows the inheritance hierarchy containing the `Person`, `Customer`, `Employee`, and `Vendor` classes. You could squeeze this onto the diagram shown in Figure 12.7, but it would make the result more complicated. (I think this part of the model is easier to understand in two pieces.)

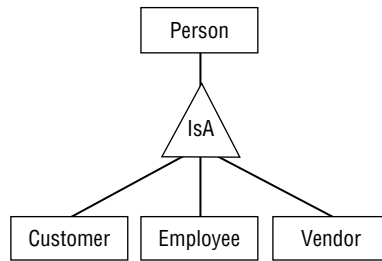


FIGURE 12.8

Figure 12.9 shows the entities representing the last remaining classes shown in Figure 12.4. This figure shows the relationship between the `Person` parent class and the `Address` and `Phone` entities. (You could easily add this to Figure 12.8 but to me the two seem logically separate. One shows inheritance and the other shows entity relationships.)

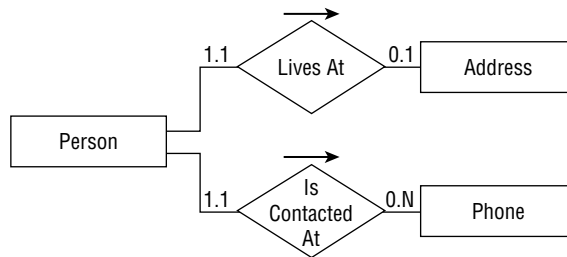


FIGURE 12.9

Improving the Entity-Relationship Diagram

If you look closely at Figure 12.7, you'll find two many-to-many relationships. First, a Customer may take many Courses while a Course may have many Customers enrolled. Second, an Order might contain many InventoryItems and an InventoryItem can be part of many Orders.

Entity-relationship diagrams have no trouble modeling many-to-many relationships, but a relational model cannot. To see why not, consider the relationship between Customer and Course. To build this relationship in a relational model, one of the tables must contain information linking it to the other.

To link a single Customer record to many Course records, you would need to list many Course IDs in the Customer record. Because a customer might take any number of courses, that would require the Customer record to contain an indefinite number of fields, and that's not allowed in a relational model.

Now suppose you try to make a single Course record hold information linking it to several Customer records. That would require the Course record to contain an indefinite number of fields, and that's not allowed in a relational model.

The way out of this dilemma is to create an intermediate entity to represent the combination of a particular customer and a particular course. Then you can connect the Customer and Course entities to the new one with one-to-many relationships, which can be represented in a relational model.

Figure 12.10 shows this new piece of the entity-relationship puzzle. Now a Customer is associated with any number of CustomerCourse entities, each of which is associated with a single Course. Similarly, a Course is associated with any number of CustomerCourse entities, each of which is associated with a single Customer.

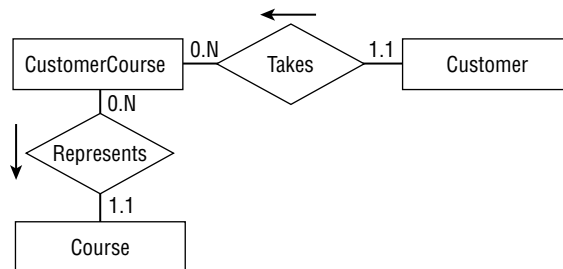


FIGURE 12.10

Broken Relationships

Restructure the other many-to-many relationship shown in Figure 12.7 between Order and InventoryItem so that it doesn't require a many-to-many relationship.

1. Create a new intermediate entity.
2. Associate the new entity with the old ones.

How It Works

1. To connect the Order and InventoryItems entity sets, create a new OrderItem entity set.
2. Connect the Order and InventoryItems entity sets with the new one.

One order can contain one or more items so the OrderItem end of the Order/OrderItem relationship has cardinality 1.N. One OrderItem is associated with exactly one Order so the Order end of this relation has cardinality 1.1.

One inventory item can be used in zero or more orders, so it may be represented by many OrderItems. That means the OrderItem end of the InventoryItem/OrderItem relationship has cardinality 0.N. A single order item represents a particular inventory item, so the InventoryItem end of this relationship has cardinality 1.1.

Figure 12.11 shows an ER diagram representing the Order/OrderItem/InventoryItem relationships.

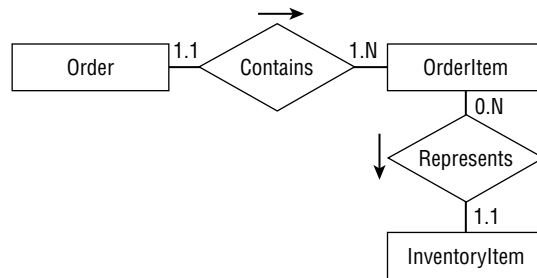


FIGURE 12.11

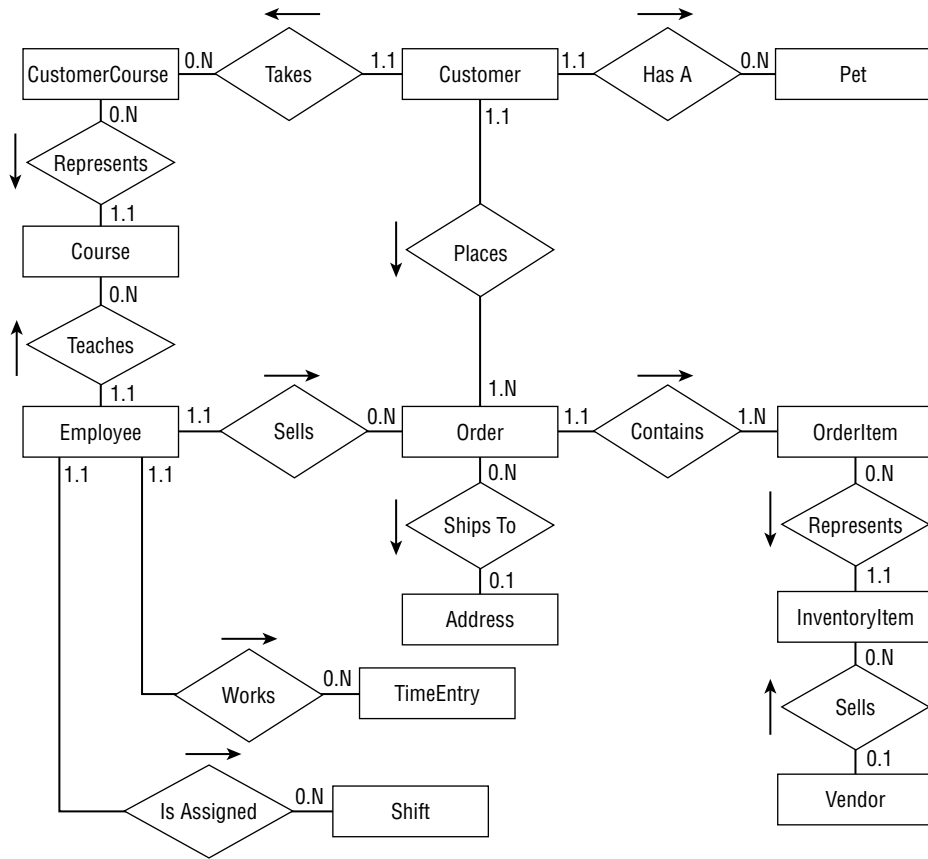
Figure 12.12 shows the new larger ER diagram from Figure 12.7 with the many-to-many relationships replaced by intermediate tables.

The changes to remove the many-to-many relationships are another step that normalizes part of the database. They remove the need for repeated columns in tables by replacing them with intermediate tables. The entity-relationship model can represent many-to-many relationships, so you don't really need to remove them at this stage. Instead, you could wait and remove them when you build the relational model in the next step. However, the diagram shown in Figure 12.7 makes these relationships easy to see, so this is a reasonable time to straighten them out, and it will make building the relational model easier in the following chapters.

RELATIONAL MODELING

The semantic object model made it easy to study the classes that will make up the database and allowed some normalization. The entity-relationship model emphasized the entities' relationships and made it easy to remove many-to-many associations.

Now it's time to use what you've learned by building the semantic object and entity-relationship models to create a relational model.


FIGURE 12.12

Start by making a table for each of the models' classes and entity sets. Look at the final ER diagrams shown in Figures 12.8, 12.9, and 12.12, and make tables for the entities drawn in rectangles. The following list shows the tables that you need to create:

- CustomerCourses
- Customers
- Pets
- Courses
- Employees
- Orders
- OrderItems
- Addresses

- InventoryItems
- TimeEntries
- Shifts
- Vendors
- Persons
- Phones

Refer to the semantic object model in Figure 12.4 to find the basic fields that each table needs.

Next, consider the tables that are directly related in the ER diagrams. Figure 12.12 contains several one-to-many relationships. To implement those in the relational model, you need one of the related tables to include a field that leads back to a field in the other table. The table at the “one” end of the one-to-many relationship cannot hold an unknown number of fields linking to the “many” records, so the fields must work the other way around.

In the table on the “one” side of the relationships, identify the primary key fields. Remember, to qualify as a primary key, the fields must guarantee uniqueness so that no two records in the table can have exactly the same primary key values.

Because those fields will be used as the record’s primary key, they should not be values that you will want to modify later. For example, a combined FirstName/LastName key is a bit risky because people do occasionally change their names.

The key values will also be contained in the table on the “many” side of the relationship, so it’s better if the key doesn’t include a lot of data. The FirstName/LastName pair might be a moderately long text string. Though it won’t hurt database performance too much if you use such a key, it’s easier to work with a single field key.

If the table on the “one” side of the relationship doesn’t contain an easy-to-use natural key, add one. Name it after the table and add “Id” at the end.

For example, consider the relationship between the Address and Order entities. Figure 12.12 shows that this is a one-to-many relationship with the Address entity on the “one” side. That entity contains `Street`, `City`, `State`, and `Zip` attributes. Even if you allow only a single customer per street address, using those fields as the primary key would be risky because you might need to change them later. For example, an employee might misspell the customer’s street name when creating the customer’s record. Even worse, a customer might move to a new address. In both of those cases, it would be seriously annoying to have to delete the customer’s record and create a new one just to update the address. (I once had an Internet service provider that couldn’t seem to figure out how to change a customer’s email address without closing the account and opening a new one. I’d send them a copy of this book if I thought they’d read it.)

Because this table has no natural primary key, add an `AddressId` field to it and use that to link the tables together.

Now add an `AddressId` field to the “many” side of the relationship. In this example, that means adding a new field to the `Orders` table.

Finally, draw the link between the two tables, place a 1 next to the “one” end of the relationship, and add a ∞ next to the “many” end.

Figure 12.13 shows the resulting relational model for these two tables. Note that this version considers only those two tables. In the more complete model, these tables will need additional ID fields to link them to other tables.

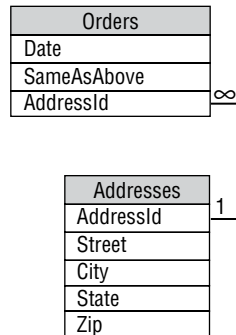


FIGURE 12.13

Identifying IDs

Figure out what fields to add to represent the relationship between the Orders and OrderItems tables.

1. Identify the “one” side of the one-to-many relationship. Find or create a primary key for that table.
2. Give the table representing the “many” side of the relationship fields to refer to the first table’s primary key.
3. Draw the tables and the new link. Include the information about the relationship between Orders and Addresses shown in Figure 12.13.

How It Works

1. The Orders table represents the “one” side of this one-to-many relationship because one order can include many OrderItems. There is no natural primary key in an Order entity (if the same customer placed another order on the same date, it could have exactly the same values), so add a new OrderId field to the Orders table.

A single order can have many order items, so OrderId isn’t enough to uniquely identify the records in the OrderItems table. Add a SequenceNumber field to the primary key to uniquely identify the records. This field also lets you display the items for an order in sorted order. The record with SequenceNumber = 1 comes first, the record with SequenceNumber = 2 comes next, and so forth. Displaying the items in the same order in which they were originally entered is generally comforting to the users.

2. The OrderItems table represents the “many” side of the one-to-many relationship. Give it an OrderId field that refers back to the Orders table’s OrderId field. Notice that the OrderItems table’s primary key includes both the OrderId and SequenceNumber fields, but the Orders table refers only to the OrderId field. A program listing an order’s items would use the OrderId value to fetch all of the related OrderItems records, and then would use the SequenceNumber field to sort them.
3. Figure 12.14 shows a relational diagram that includes the previous relationship between Orders and Addresses shown in Figure 12.13 plus the new relationship between Orders and OrderItems.

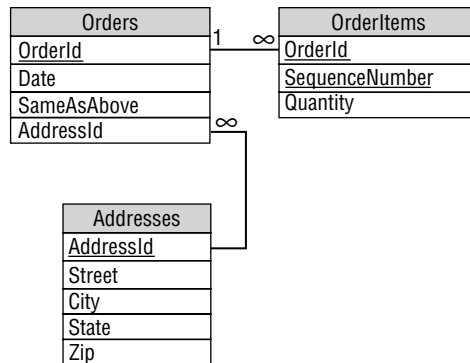


FIGURE 12.14

PUTTING IT ALL TOGETHER

Continue examining the relationships shown in the ER diagram in Figure 12.12. Find or create a primary key in the table on the “one” side of the relationship and add a corresponding field in the table on the “many” side.

You can make arranging the tables easier if you place them in roughly the same arrangement that the corresponding entities occupied in the entity-relationship diagram. (Figures 12.13 and 12.14 show the Addresses table below the Orders table because the Address entity is below the Order entity in Figure 12.12.)

You’ll probably need to move the tables around a little because they won’t be the same size as the rectangles in the ER diagram, but starting with that arrangement should make drawing relationships between the tables easier.

Figure 12.15 shows the resulting relational model.

As a quick check, examine each relational link. The end touching a primary key field should have cardinality 1. The field’s name should be the table’s name plus “Id.” The exception to this naming convention is in the InventoryItems table, which uses the natural primary key field UPC.

The end of a link touching the foreign key field on the “many” side of the relationship should have the same name as the ID field and should have cardinality 1 (for a one-to-one relationship) or ∞ (for a one-to-many relationship).

Notice that the two intermediate tables used to represent many-to-many relationships, CustomerCourses and OrderItems, contain little more than keys linking to the two tables that they connect. For example, the CustomerCourses table contains only a CustomerId field linking to the Customers table and a CourseId field linking to the Courses table.

The OrderItems table includes its two linking ID fields plus the SequenceNumber field to make the primary key unique and to allow sorting. It also contains a Quantity field to indicate the number of

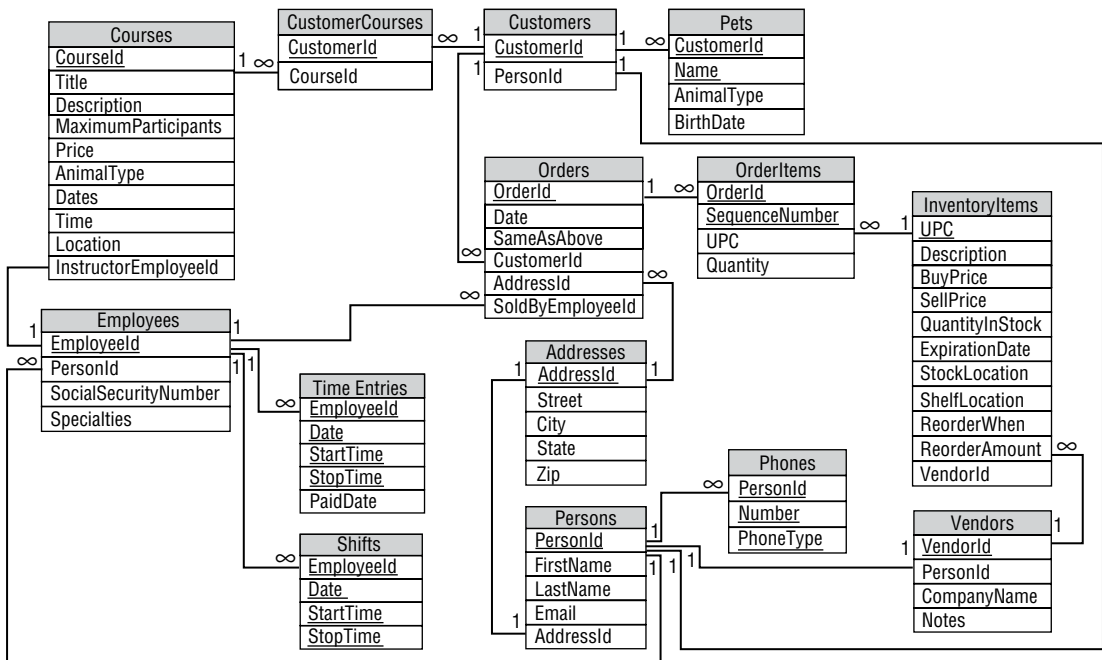


FIGURE 12.15

that type of item included in the order (for example, 12 scratch posts).

SUMMARY

This chapter explains the data modeling steps for the Pampered Pet database project. It showed how to build a semantic object model and how to convert that into an entity-relationship model. Along the way, it showed how to improve the models by normalizing parts of them.

In this chapter, you saw examples of:

- Building semantic objects
- Moving repeated semantic group attributes and some other group attributes into new classes
- Converting a semantic object model into an entity-relationship model
- Representing a many-to-many relationship with two one-to-many relationships
- Improving models by normalizing parts of them
- Converting semantic object models and ER diagrams into a relational model
- Adding ID fields to tables
- Converting entity relationships into relational links

Not all projects use both semantic object models and entity-relationship models. Many developers prefer one or the other and don't bother with the extra work of creating two models. Some even jump straight to a relational model. Each of these types of models has its strengths and weaknesses, however, so it's often useful to work through all three kinds.

Figure 12.15 shows a pretty reasonable relational model for the Pampered Pet database, but it's still not perfect. If you look closely, you might be able to identify a few places where the tables are not well normalized. (Can you spot the tables that are not even in first normal form?)

Chapter 13, "Extracting Business Rules," shows how to improve the model by isolating business rules that are likely to change in the future. Chapter 14, "Normalizing and Refining," further normalizes the database and puts the finishing touches on it.

Before you move on to Chapter 13, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

EXERCISES

Consider the entity-relationship diagram shown in Figure 12.12 and think about possible changes that the Pampered Pet might later want to make to the database. Easy changes include adding or removing non-identifier fields from an entity. Harder changes would require adding or removing entities or changing the relationships among them. For each of the following changes, explain how you would make the change and how hard it would be.

1. The Pampered Pet opens a café that serves snacks for pets and their owners. How would you handle the new food items for sale?

2. Management decides that only certain employees with special training can teach courses. How would you model this new type of employee?

3. The Pampered Pet opens a new store. Management wants to track customers company-wide but wants to track sales by store. How would you handle that?

4. New courses are offered off-site at parks, lakes, and other locations. How would you store the addresses of these off-site courses?

5. The Pampered Pet offers free clinics and outings such as dog and llama hikes. How would you store information about these freebies?

6. You need to allow more than one address on an order. How would you store the new addresses?

7. You need to store a phone number for each order. How would you store these phone numbers?

8. Management decides they want to track the department that sold each item. How would you track item departments?

9. Management decides to track customer addresses as they change over time. How would you remember old addresses?
-
10. The Pampered Pet starts holding sales and offering discounts to employees. Where do you store the discount information?
-
11. Figure 12.16 shows an ER diagram for a Robot Wars style competition (for example, see www.robotwars.tv). A competitor builds one or more robots either alone or with others (so robots can have one or more builders). Each robot can fight in any number of matches (if it survives), and a match involves several robots. Finally, each match has a single winner (the last robot “standing”). Unfortunately, the design shown in Figure 12.16 includes two many-to-many relationships. Draw a new ER diagram that replaces those relationships with one-to-many relationships.
-

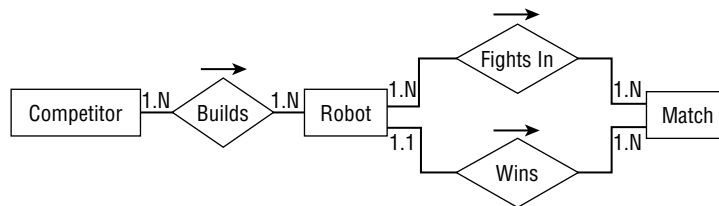


FIGURE 12.16

12. Build a relational model for the solution you built for Exercise 11.
-

13

Extracting Business Rules

The previous chapters have built up a basic design for the Pampered Pet database. They gathered customer requirements, built a semantic object model and entity-relationship diagrams, and converted those into a relational model.

This chapter further refines the design by identifying business rules in the relational model and isolating them so they will be easy to modify in the future if necessary.

In this chapter, you'll see examples that identify:

- Required fields and other field-level constraints that are unlikely to change
- Sanity checks that are also unlikely to change
- Business rules that are more complicated or likely to change in the future

IDENTIFYING BUSINESS RULES

The text and exercises in Chapter 12, “Building a Data Model,” listed the fields required for the initial database design. For each field, that chapter gave the field's data type, whether the field is required, and its domain. That information describes most of the project's business rules.

Domain information usually includes simple “sanity check” constraints. These are conditions that basically verify the “laws of physics” for the database. For example, an item's cost cannot be less than \$0.00. Exercise 5 in Chapter 12 discussed free pet clinics and outings, so it's possible that an item might be free, but it's hard to imagine the Pampered Pet's management charging less than nothing for a product.

Other sanity check conditions include field data types (the number of items is an integer, not a string or date) and whether a field is required. It might also include simple pattern validation. For example, the database might require a phone number to have a 10-digit format, as in 951-262-3062, or a format similar to (0)30-23125 457, or whatever your phone number format is.

Because these sanity checks will never change, they can be coded directly into the database by setting field conditions (data type and required) and check constraints.

Primary key information is also built into the database. For example, making the `InventoryItems` table's primary key be its `UPC` field ensures that every record has a non-null `UPC` value and that every record has a unique `UPC` value.

Other domain information is either more complicated or more likely to change over time. Those conditions should be isolated as much as possible to make them easier to change in the future.

The following sections describe the constraints on each of the tables defined in the relational model shown in Figure 12.15. They explain which of those constraints can be built into the database and which should be isolated as business rules.

Courses

The `Courses` table's required fields include `CourseId` (the primary key), `Title`, `Description`, `MaximumParticipants`, `Price`, `AnimalType`, `Dates`, `Time`, and `Location`. The `InstructorEmployeeId` is not required and only gets filled in after an employee is assigned to teach the course. The required fields can be built into the database.

Sanity checks include:

- `MaximumParticipants >= 0` and `MaximumParticipants < 100`
- `Price > 0`
- `Dates` no earlier than the current date
- `Time` between 8 a.m. and 11 p.m.

The sanity checks can be built into the database as field-level check constraints.

This table has two fields that take values from enumerated lists:

- `AnimalType` comes from the list holding `Cat`, `Dog`, `Bird`, `Spider`, and so on.
- `Location` comes from the list holding `Room 1`, `Room 2`, `Back Yard`, `arena`, and so on.

If you coded the lists of choices allowed for `AnimalType` and `Location` into field-level check constraints, you would have to make nontrivial changes if the allowed values changed. To make the design more flexible, the choices should be placed in lookup tables `AnimalTypes` and `Locations`. Then the fields in the `Courses` table can refer to those values as foreign key constraints.

This table contains one slightly more complicated validation:

- `Price > 0`. The price must be at least 0. Management has not said it could be equal to 0, but that's a change they could make in the future.

This rule could be implemented in the user interface or in middle-tier code that fetches and updates course data.

It's tempting to make this a field-level check constraint. After all, it would be relatively easy to do so and the change isn't inevitable. However, suppose the company offers free clinics for a while, and then

decides to no longer offer them. At that point, there would be old course entries with Price = 0 and you could not change the field-level check constraint back to Price > 0 because it would contradict existing data.

The management will get the most flexibility for this constraint if it is kept separate from the database's structure. It could be implemented in middle-tier routines that fetch and update Courses data.

Figure 13.1 shows the Courses table with its new lookup tables. The lookup tables are drawn with dashed rectangles so the main tables stand out from them. This figure omits all of the other tables and their relationships.

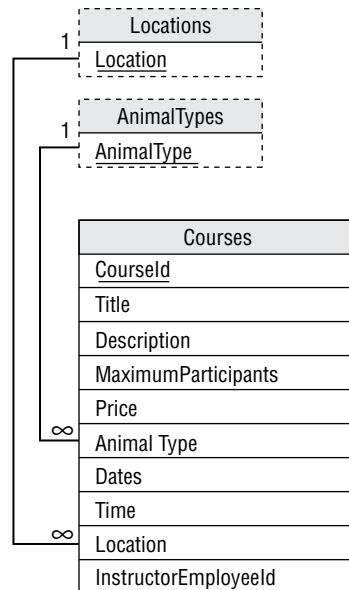


FIGURE 13.1

Address Constraints

Identify the various kinds of constraints on the Addresses table and determine which should be implemented in the database and which should be provided elsewhere as business rules.

1. Identify the primary key and the required fields.
2. Identify sanity checks that will never change.
3. Create lookup tables for fields with a fixed set of allowed values.
4. Identify more complicated business rules and rules that may change in the future.

How It Works

1. The Addresses table's required fields are AddressId (the primary key), Street, City, State, Zip. All of these fields should be marked as required in the database.

2. In the United States at least, the Zip value must always have the form 12345 or 12345-6789. Verifying the format could be implemented as a simple field-level check constraint.
3. The State field's values must be one of the standard state abbreviations. Those abbreviations should be added to a States table, and then this field can refer to it as a foreign key.
4. The relationship between City, State, and Zip is complex. It probably doesn't make a lot of sense to validate every possible combination because that would require a huge lookup table. It's unlikely that the Pampered Pet will do business in every state, so most of that data would never be used anyway. A simpler solution would be to either validate the address and ZIP Code with a web service, or ignore the problem and hope the user doesn't mistype the ZIP Code.

An alternative would be to build a CityStateZips lookup table to hold the nearby City/State/Zip combinations, and then warn the user if a record has a set of values that is not in the list. This way if the user misspells a local town or enters a Zip value that doesn't match that town, the program will let the user fix it. If a customer wants an order shipped a thousand miles away, the program won't know about the City/State/Zip combination and would let the user enter it anyway.

Database constraints are unyielding. A value is either allowed or it isn't. By itself, the database won't ask the user "Are you sure?" and then accept an unusual value. That means this constraint cannot be implemented in the database. It must be implemented in a middle-tier routine or in the user interface.

Figure 13.2 shows the Addresses table with its States lookup table and the CityStateZips pseudo-lookup table. Both are drawn in dashed rectangles to indicate that they are lookup tables. The relationship between the Addresses table and the CityStateZips table is also dashed because it is implemented in a middle-tier routine, a web service call, or the user interface rather than inside the database.

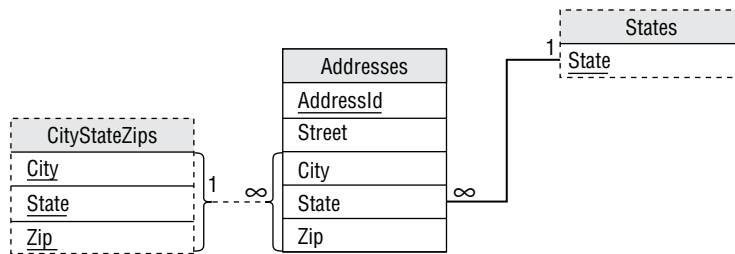


FIGURE 13.2

CustomerCourses

This table is an intermediate table representing the many-to-many relationship between the Courses and Customers tables. It contains only two fields, CustomerId and CourseId. Both of these are required because they are part of the table's primary key. They are also used as foreign key constraints matching values in the Courses and Customers tables. That's about as constrained as a field can get, so there isn't much more to do with this table.

Customers

The version of the Customers table shown in Figure 12.15 contains only two fields. CustomerId is the primary key. PersonId is used in a foreign key constraint matching values in the Persons table so it's completely constrained.

Pets

The Pets table's CustomerId and Name fields form the primary key. Its other fields, AnimalType and BirthDate, are also required, so they should be flagged as required, in the database.

It wouldn't hurt to put a sanity check constraint on the BirthDate field to verify that new dates are not past the current date. Some pets live a really long time, however, so it's hard to set a safe lower limit on BirthDate. Macaws can live 65 or more years and aggressive little yippy dogs live practically forever—at least that seems to be the case with my neighbor's dogs. Some turtles and bowhead whales have been known to live for hundreds of years, and the jellyfish *Turritopsis dohrnii* is known as the “immortal jellyfish.” We may want to warn the user if a BirthDate is more than 20 years in the past, but we can't code an earliest allowed BirthDate into the database.

The AnimalType field can only take a value from a list of allowed values, so those values should be placed in a lookup table. The design already calls for an AnimalTypes lookup table (see the section about the Courses table earlier in this chapter), so this table can refer to that one.

Employees

The Employees table's primary key is its EmployeeId field. The PersonId and SocialSecurityNumber fields are also required.

The SocialSecurityNumber field must have a format similar to 123-45-6789. That pattern can be verified by a field-level check constraint.

PersonId is a foreign key constraint referring to the Persons table, so it is completely constrained.

Orders

The Orders table's primary key is OrderId. The Date, SameAsAbove, CustomerId, and SoldByEmployeeId fields are also required.

CustomerId is a foreign key constraint referring to the Customers table, so it is constrained. Similarly, SoldByEmployeeId is a foreign key constraint referring to the Employees table, so it is also constrained.

The AddressId field is optional. It is a foreign key constraint referring to the Addresses table, so if it is present it is completely constrained. (This field is fairly confusing. If the SameAsAbove field has the value True, then the customer wants the order shipped to the customer's address. If the SameAsAbove value is False and AddressId is present, then the customer wants the order shipped to that address. If SameAsAbove is False and AddressId is null, then the customer is picking up the order and doesn't want it shipped. All of this must be implemented in the user interface but doesn't affect the database design.)

OrderItems

Like the CustomerCourses table, this table is an intermediate table used to implement a many-to-many relationship.

This table's OrderId and SequenceNumber fields make up the primary key. The UPC and Quantity fields are also required.

UPC is used as a foreign key constraint referring to the InventoryItems table, so it is completely constrained.

For sanity checking, a field-level check constraint should also verify that Quantity > 0.

InventoryItems

This table's UPC field is its primary key. The Description, SellPrice, QuantityInStock, StockLocation, and ShelfLocation fields are also required.

Sanity checks include:

- If present, BuyPrice >= \$0.00.
- SellPrice >= \$0.00.
- QuantityInStock >= 0.
- If present, ExpirationDate > some early date guaranteed to be earlier than any expiration date such as the date the database is put into use.
- On new records, if present, ExpirationDate > the current date.
- If present, ReorderWhen > 0. (If null, reorder only occurs manually.)
- If present, ReorderAmount >= 0. (If null, someone must specify the amount.)

VendorId is a foreign key constraint referring to the Vendors table so, if its value is not null, the value is completely constrained.

The StockLocation and ShelfLocation fields can only take certain values, so those values should be moved into the lookup tables StockLocations and ShelfLocations.

TimeEntries

This table's EmployeeId, Date, StartTime, and StopTime fields make up its primary key.

EmployeeId is a foreign key constraint referring to the Employees table, so it is completely constrained.

The table should also contain the following field-level sanity check constraints:

- Date >= some early date such as the date the database is put into use.
- StartTime >= 6 a.m.
- StopTime <= 11 p.m.
- If present, PaidDate >= Date.

Shifts

The Shifts table is similar to the TimeEntries table except it doesn't have a PaidDate field. Its EmployeeId, Date, StartTime, and StopTime fields make up its primary key.

EmployeeId is a foreign key constraint referring to the Employees table, so it is completely constrained.

The table should also contain the following field-level sanity check constraints:

- Date >= some early date such as the date the database is put into use.
- StartTime >= 6 a.m.
- StopTime <= 11 p.m.

Persons

This table's PersonId is its primary key. The FirstName and LastName fields are also required.

Email should have a valid email format. Unfortunately, it's pretty hard to define valid email formats, so this should be considered a more complicated business rule. You can verify that the address looks like someone@somewhere.extension, but you can only really verify that the @ and. are present, and you can't stop the user from mistyping the other pieces.

Validation should be provided by a middle-tier routine that saves Persons records, a stored procedure, or user interface code. Then if management decides to change the way this field is validated, you can make the change reasonably easily.

This table's AddressId field is a foreign key constraint referring to the Addresses table so, if it is present, it is completely constrained.

Phones

This table's primary key includes all of its fields: PersonId, Number, and PhoneType.

PersonId is also a foreign key constraint referring to the Persons table, so it is completely constrained.

Number should have a valid phone number format such as 262-3062 or 951-262-3062. This won't change (unless the store starts accepting international orders), so it can be checked in a field-level check constraint.

PhoneType must be one of several values such as Home, Cell, or Work. Those values should be moved into a PhoneTypes lookup table so this field can refer to it as a foreign key constraint.

Vendors

The Vendors table's primary key is its VendorId field. The PersonId and CompanyName fields are also required.

PersonId is a foreign key constraint referring to the Persons table, so it is completely constrained.

The Notes field is completely unconstrained and optional.

DRAWING A NEW RELATIONAL MODEL

Figure 13.3 shows the new relational model.

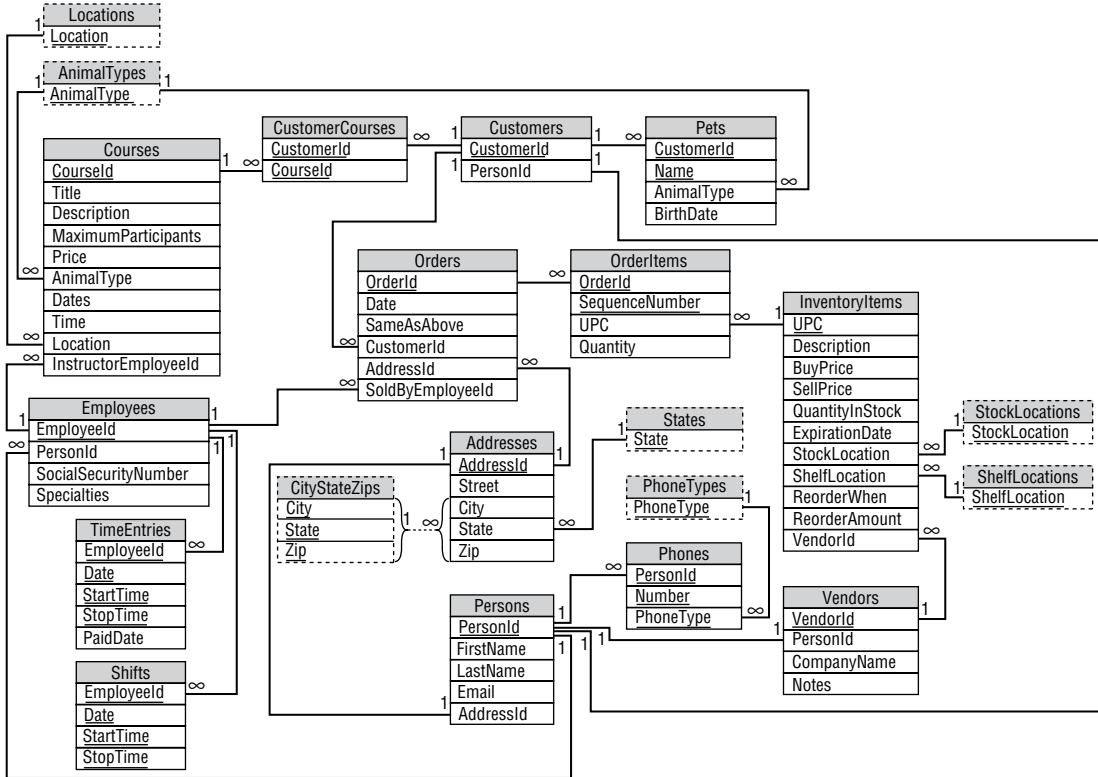


FIGURE 13.3

SUMMARY

This chapter showed you how to classify table constraints in the Pampered Pet database. You learned how to:

- Identify the primary key and the required fields (to implement in the database).
- Identify sanity checks that will never change (to implement in the database).
- Create lookup tables for fields with a fixed set of allowed values.
- Identify more complicated business rules and rules that may change in the future.

After identifying these constraints and adding lookup tables, we examined a new relational design for the database.

Even at this point, however, the database isn't perfect. The next chapter makes a few final changes to make the database more flexible and robust.

Before you move on to Chapter 14, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

EXERCISES

For these exercises, consider the relational model for a Robot Wars competition shown in Figure 13.4 (see the exercises at the end of Chapter 12 and www.robotwars.tv). Use your judgment while working through the exercises.

NOTE The original Robot Wars television show was broadcast on British television so I'm sure most of the contestants used British addresses. For consistency, I'm going to use U.S. formats, but feel free to use a different format if you like.

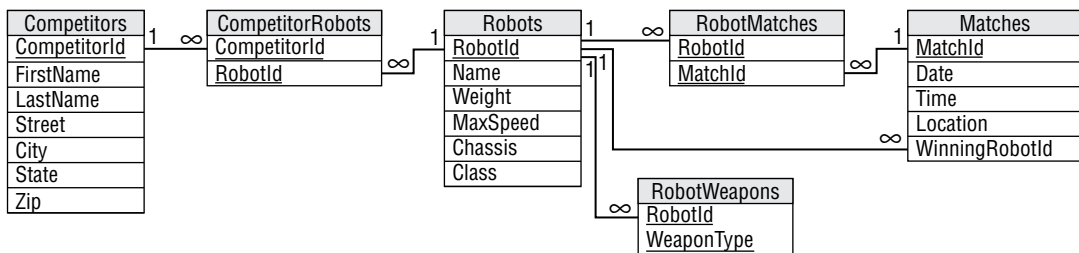


FIGURE 13.4

1. Identify the tables' primary keys and required fields.
2. Identify the tables' sanity checks that will never change.
3. Define lookup tables for fields with a fixed set of allowed values.
4. I can think of three somewhat more complicated business rules that should be implemented, but in general it would be hard to identify business rules without knowing more about the competition. See if you can think of the three I came up with and make up some others. What sorts of things would make interesting business rules that should not be built into the database?
5. Draw a new relational model showing the new tables.

14

Normalizing and Refining

Chapters 11 through 13 walked you through the steps of designing a preliminary database for the Pampered Pet. They showed how to gather requirements, build semantic object and entity-relationship models, and convert those into a relational model. Chapter 13 illustrated how to identify rules that should be built into the database and more complex or volatile rules that should be isolated as business rules.

Even after all of this work, the database isn't perfect. This chapter puts the finishing touches on the database by normalizing it appropriately.

In this chapter, you'll see examples of:

- Improving the design to make the database more flexible
- Identifying tables that are insufficiently normalized
- Normalizing tables to prevent data anomalies
- Not normalizing where normalization would be more trouble than it's worth

IMPROVING FLEXIBILITY

Figure 14.1 shows the relational design built in Chapter 13, "Extracting Business Rules."

This design is fairly reasonable, and I've seen worse designs in working databases, but it can use a couple of improvements. Later sections in this chapter discuss normalization, but first there's a big flaw to fix.

If you think about the design long enough and you walk through the use cases, you'll notice that there's a problem with the course data. Currently, the design allows many customers to take a course, and it allows a course to hold many customers. However, the design allows only one instance of any given course. If you run a Puppy Socialization course in April, you cannot run the same course again in May because the course's dates would have already passed.

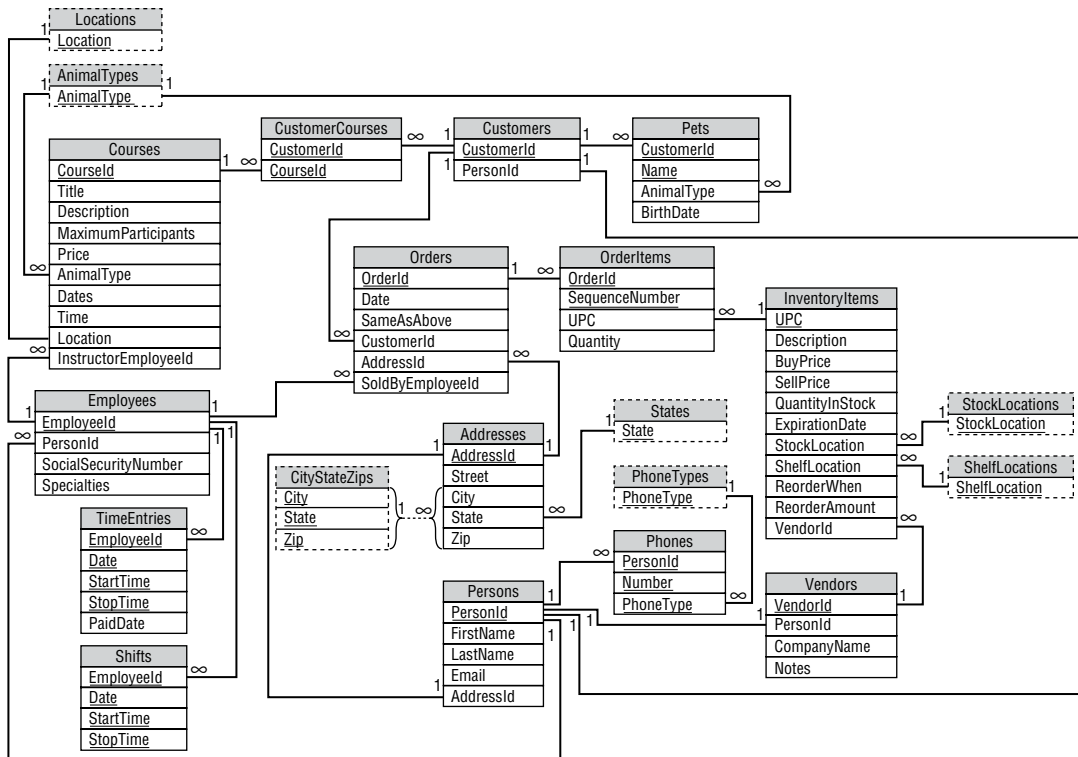


FIGURE 14.1

Furthermore, the same customer couldn't take both the April and May courses (some dogs are slow learners) because that would require identical records in the *CustomerCourses* table.

Instead, you would need to make a completely new *Courses* record for the May course. That wouldn't be the end of the world, and some applications would do exactly that, but now the table contains multiple records that truly represent different offerings of the same thing. You can tell that there's a problem because the records would have so many duplicated fields: *Title*, *Description*, *Price*, and *AnimalType*. The fields that would change between offerings are *MaximumParticipants*, *Dates*, *Time*, *Location*, and *InstructorEmployeeId*.

The customers taking the course would also change (except for those who fail the first time and retake the course), so the course offering would need some kind of new ID value to link it to the *CustomerCourses* table.

The database needs a new type of object to represent a course offering. The course offerings will link to a *Courses* record that provides all of their shared data.

One course can have many offerings, but an offering corresponds to only one course, so this is a one-to-many relationship, and you can add it to the database the way you always build a one-to-many relationship. First, add an ID field to the "one" table. Then refer to the ID field in the "many" table as a foreign key.

Now, the CustomerCourses table should link to the new CourseOfferings table instead of the Courses table (because the customer takes an offering of a course not the abstract course description).

Figure 14.2 shows the design with the new table added.

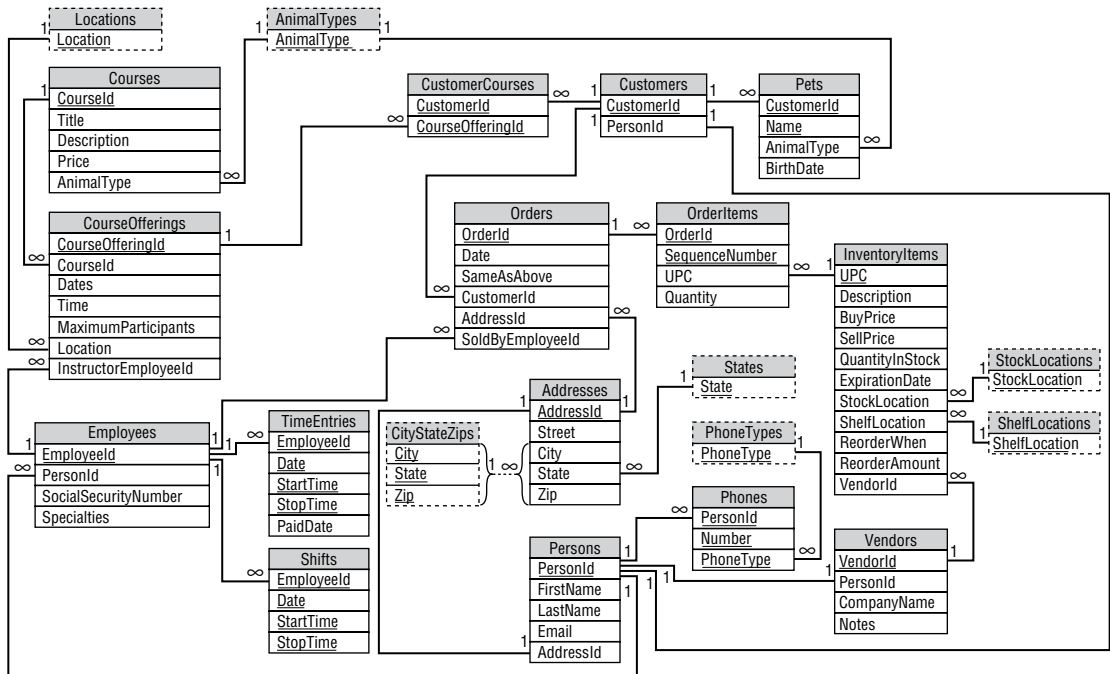


FIGURE 14.2

VERIFYING FIRST NORMAL FORM

The previous chapters were pretty careful about building their tables so they're almost certainly in 1NF, right? Not always. With some experience and attention to detail, you should be able to build tables in 1NF almost all of the time and 3NF most of the time, but occasionally a few normalization bugs slip through.

Recall the rules for 1NF:

1. Each column must have a unique name.
2. The order of the rows and columns doesn't matter.
3. Each column must have a single data type.
4. No two rows can contain identical values. (The table has a unique primary key.)
5. Each column must contain a single value.

You can easily verify the first four rules. For example, the Courses table's columns all have different names, the order of rows and columns doesn't matter, each column in the Courses table has a single data type, and the CourseId field is the primary key so no two records can have the same CourseId value and therefore, they are different.

The rule that usually catches people is rule 5: each column must contain a single value. Sometimes, the data in a field contains more than one logical piece of data. In that case, the field should be broken into pieces.

If you know how many pieces there will be, you can use multiple fields in the same table. For example, if you have a Name field that should be broken into FirstName and LastName, you know there are only two pieces to the field and you can just replace it with two new fields.

If the field's data could contain any number of values, you should move the values into a new table and link back to the original table. For example, suppose the Customers table contains a Children field that lists the customer's children's names separated by commas. In that case you can't just add a bunch of Child fields to the Customers table. (In the 1700s, Valentina and Feodor Vassilyev had 69 children, so handling them would require a very large, mostly empty table. And there's no limit to how many kids someone might adopt.) Instead, you need to add a CustomerChildren table that uses CustomerId to find the Customer associated with a particular record.

If you look carefully at each of the fields shown in Figure 14.2, you'll find a couple that might contain multiple data values. You can ignore simple compound values such as the Pets table's BirthDate field and the Phones table's Number field. Though you can think of a birth date as containing a day, month, and year, the Pampered Pet will probably never need to look at those values separately. I suppose someone might want to make a list of all pets born on the 13th of any month, but that would be pretty strange. Similarly, unless the requirements call for you to be able to list customers in a given area code, it isn't worth breaking up the phone number field.

The first field that truly holds more than one value is the CourseOfferings table's Dates field. This single field is supposed to hold a list of the dates when a course offering takes place. For example, a particular offering might occur every Wednesday for six weeks.

If every course is only offered on a weekly schedule, such as every Wednesday, you could encode that in a single field by simply giving the day of the week. If there might be exceptions (for example, six Mondays, skipping Labor Day), that system doesn't work as well.

To solve this problem, you can break the Dates field into pieces. If the Pampered Pet doesn't require every course to have the same number of sessions, you need to move the values into a new CourseOfferingDates table that refers back to its course offering.

Figure 14.3 shows the design with the new CourseOfferingDates table.

One hint that a field might contain multiple values is that its name is plural. If the field represents a number, such as the CourseOfferings table's MaximumParticipants field, then it probably represents a single value. If the field represents a group of values, however, it should probably be broken apart.

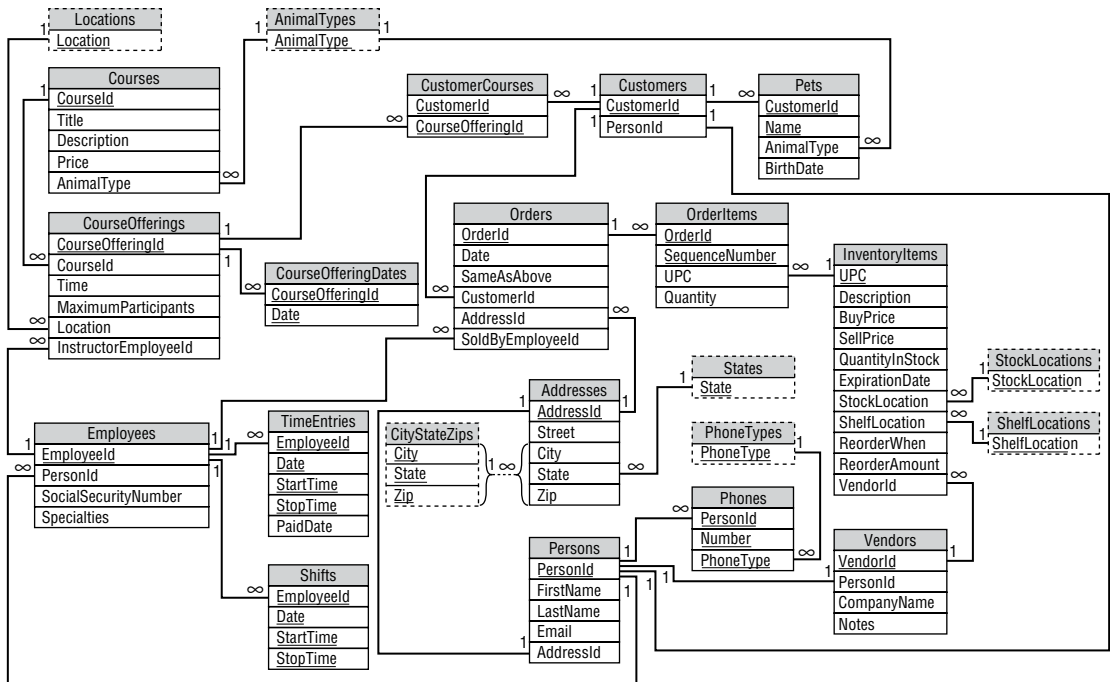


FIGURE 14.3

Normal

There's one other table in Figure 14.3 that isn't in 1NF. Find and fix it.

1. Look for fields that don't represent a single value.
2. Decide whether the field contains a fixed, known number of values or an unknown number of values. If the field contains a fixed number of values, split it into the required number of fields. If the field contains an unknown number of values, move the values into a new table.
3. Perform other modifications to the design if this change requires them. (Hint: In this case, you'll need to create a new lookup table.)

How It Works

1. All of the fields shown in Figure 14.3 represent a single value except the Employees table's Specialties field. This field lists the employee's areas of expertise. They might include animal types, products, problems, and so forth. A typical value might be, "Cat, Dog, Parasites" (as in treating parasites, not training them). This is also the only field in Figure 14.3 that has a plural name other than MaximumParticipants, which represents a single number.
2. The Specialties field could contain any number of values, so it cannot be broken into new fields within the Employees table. Instead, the model needs a new EmployeeSpecialties table that refers back to Employees. The new table will have fields EmployeeId and Specialty.

3. The new table's Specialty field can contain only specific values, such as Cat, Dog, and Penguin, so the model should validate the field by making it a foreign key that refers to a lookup table. In this case, the new table should be called Specialties and will have a single field Specialty. Figure 14.4 shows the new design.

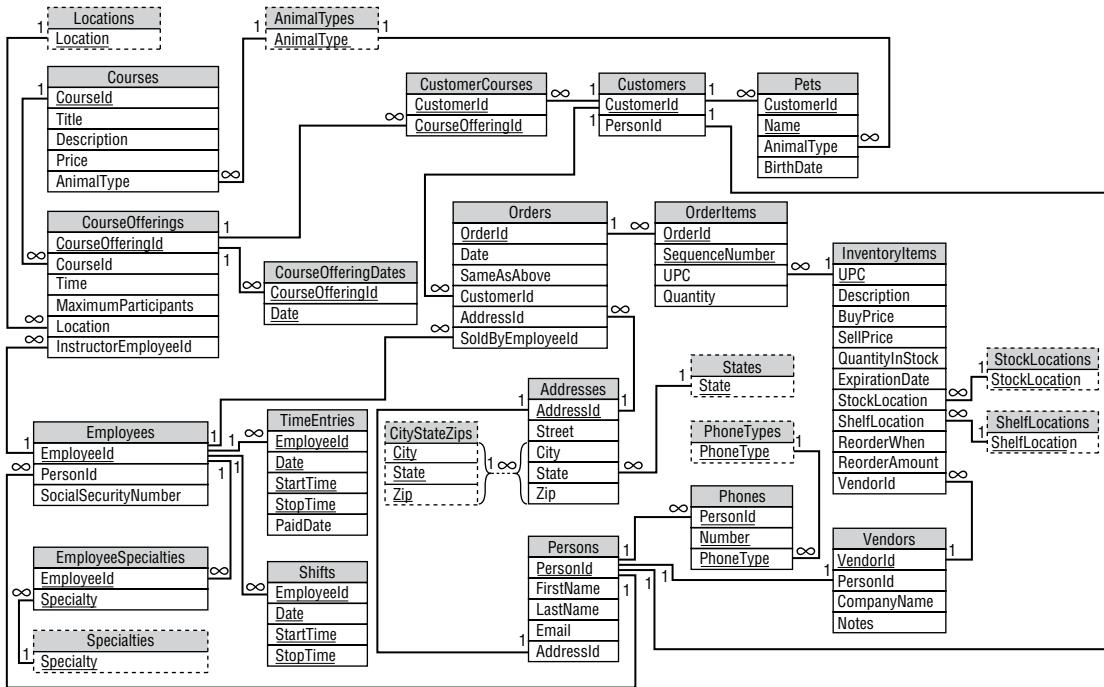


FIGURE 14.4

VERIFYING SECOND NORMAL FORM

Recall the rules for 2NF:

1. The table is in 1NF.
2. All of the non-key fields depend on all of the key fields.

Many of the tables have a one-field primary key, so every other field must depend on the entire primary key. In the intermediate tables representing many-to-many relationships and the lookup tables, every field is part of the primary key, so rule 2 doesn't apply.

The only tables remaining to consider are Pets, OrderItems, and TimeEntries.

Pets

The Pets table's primary key contains the combination CustomerId/Name. Its other fields are AnimalType and BirthDate. AnimalType depends on both CustomerId and Name because you need to know both CustomerId and Name to deduce the pet's AnimalType.

Another way to think of this is to notice that if you know the CustomerId alone, you cannot guess the pet's AnimalType because the customer might have a cat and a fish. (Until the cat eats the fish, like my cat, Victor, did.) Similarly, if you know the pet's Name, you cannot determine the AnimalType because different customers might have different kinds of pets with the same name. You can use similar arguments to show that BirthDate depends on both CustomerId and Name.

A third way to look at this, which may be less intuitive but that is easy to apply mechanically, is to ask yourself whether the database could contain any record with the same CustomerId, a different Name, and a different AnimalType. If CustomerId alone determines animal type, then every record with the same CustomerId must have the same AnimalType.

In this case, the database could hold a record with the same CustomerId, a different Name, and a different AnimalType (someone could own a fish named Phred and a dog named Pheidaux), so you know that AnimalType depends on CustomerId.

Similarly, you can ask whether another record could have the same Name, different CustomerId, and different AnimalType. I used to have a dog named Snortimer and, believe it or not, I've met two other people with cats named Snortimer, so that situation is possible. That means AnimalType also depends on CustomerId.

You can use similar arguments to show that BirthDate depends on both CustomerId and Name.

This all assumes a customer doesn't give multiple pets the same name. There's a wonderfully silly Sandra Boynton song named "Fifteen Animals" about someone who has 15 pets all named Bob, except the turtle, which is named Simon James Alexander Ragsdale III. For this customer, you'll probably have to assign the pets serial numbers or something: Bob-1, Bob-2, Bob-3, and so forth.

OrderItems

Verify that the OrderItems table is in 2NF. For each of the table's primary keys (OrderId and SequenceNumber) and each of the non-key fields (UPC and Quantity), see if another record could have a different value for the key field and a different value for the non-key field. Consider all four combinations:

1. OrderId and UPC
2. OrderId and Quantity
3. SequenceNumber and UPC
4. SequenceNumber and Quantity

How It Works

1. Could there be two records with the same SequenceNumber, different OrderId, and different UPC? Yes. Two orders could contain different items listed first. Then the OrderItems records will have the same SequenceNumber (because the items are listed first), different OrderId (because they're two separate orders), and different UPC (because the orders are for different things).
2. Could there be two records with the same SequenceNumber, different OrderId, and different Quantity? Yes. Two orders could contain different quantities of the same item. For example, one customer could order one toy mouse, and a second customer could order two toy mice. Then the OrderItem records will have the same SequenceNumber (because they are the first items for each order), different OrderId (because they are different orders), and different Quantity (because the first customer ordered one toy mouse and the second customer ordered two).
3. Could there be two records with the same OrderId, different SequenceNumber, and different UPC? Yes. Suppose an order contains two different items. Then the OrderItems records will have the same OrderId (because they're part of the same order), different SequenceNumber (because they're different line items in the order), and different UPC (because the two items are different).
4. Could there be two records with the same OrderId, different SequenceNumber, and different Quantity? Yes. Suppose an order contains two items with different quantities (for example, one hamster wheel and two rawhide bones). Then the OrderItems table will contain two records for this order with the same OrderId, different SequenceNumber (1 and 2), and different Quantity (1 and 2).

Because the table can hold all of these combinations, all of the non-key fields depend on every primary key field, so the table is in 2NF.

TimeEntries

The TimeEntries table's primary key includes the fields EmployeeId, Date, StartTime, and StopTime. Its only remaining field is PaidDate. To see that PaidDate depends on all of the primary key fields, ask yourself whether you could deduce the PaidDate value if you are missing one of the key values.

If EmployeeId is missing, another employee might have worked the same shift and may or may not be paid.

If Date is missing, the employee might have worked similar hours on another date and may or may not have been paid.

The StartTime and StopTime fields are a bit trickier. If StartTime is missing, the table could contain another record for the same employee on this date with the same StopTime but a different StartTime. However, the business rules require that this table cannot have two records for the same employee on the same date with overlapping times, so this record would not be allowed.

Does that mean the PaidDate field doesn't depend on StopTime, so the table isn't in 2NF? Not really. Though the table isn't allowed to hold a record with overlapping times, the table's structure doesn't prevent it; a business rule does.

To see this in another way, consider the wrestling match schedule described in the “Second Normal Form (2NF)” section of Chapter 7, “Normalizing Data.” The following table shows part of a schedule of matches. The table’s primary key is the Time/Wrestler combination.

TIME	WRESTLER	CLASS	RANK
1:30	Annette Cart	Pro	3
2:00	Sydney Dart	Amateur	1
3:45	Annette Cart	Pro	3

The problem with this table is that Class and Rank depend only on Wrestler and not on Time. Annette Cart is ranked third professionally no matter when she wrestles. That makes the table vulnerable to data anomalies. For example, if you change the first entry’s Class to Amateur, it contradicts the third entry.

Now consider again the TimeEntries table. Because of the “no overlapping time” business rule, this table cannot hold two records with the same EmployeeId, Date, and StopTime. It cannot hold two records that correspond to the two wrestling schedule records for Annette Cart, and that means it cannot suffer from the same kind of modification anomaly as the wrestling schedule table.

Similarly, the table is safe from update anomalies because you cannot add two records with the same EmployeeId, Date, and StopTime.

VERIFYING THIRD NORMAL FORM

Recall the rules for 3NF:

1. The table is in 2NF.
2. It contains no transitive dependencies.

A *transitive dependency* is when one non-key field’s value depends on another non-key field’s value. It takes a bit more work to find transitive dependencies than it does to detect other errors.

Because transitive dependencies occur when two non-key fields are related, you only need to consider tables that have at least two non-key fields. In this example, those tables are Courses, CourseOfferings, Pets, Orders, OrderItems, InventoryItems, Employees, Vendors, Addresses, and Persons.

Most of these tables are easy to check. For example, consider OrderItems. Its non-key fields are UPC and Quantity. Are these fields related? Of course not. The type of item a customer is buying does not determine the number of items bought, or vice versa. (If the store has only one Jack Russell Terrier, you can buy only one, but that’s an inventory issue, not a database design issue. Of course, Jack Russells are so energetic it might be insane to buy more than one, but again, that’s not a database design issue.)

In the Courses table, the Title, Description, Price, and AnimalType fields don't depend on each other. If the business requirements stated that all Dog courses had the same price, then things would be different, but in this example there are no such restrictions.

The Orders table might give you pause because the CustomerId might be related in some sense to the AddressId. Remember that an Orders record has an AddressId field only if the customer wants the order shipped to an address other than the customer's usual address, however, so the relationship is not really there. If the order included the customer's home address every time, then there would be a relationship between the CustomerId and the AddressId.

The Addresses table also contains the standard weird relationship between City, State, and Zip. Chapter 13, "Extracting Business Rules," already considered this relationship (see the sidebar "Address Constraints" in Chapter 13) and decided to live with a lookup table for local addresses rather than building an enormous lookup table for every City/State/Zip combination.

The last tricky table is CourseOfferings. The maximum number of participants for a course is determined by the location where it is taught. For example, the store's back conference room can hold only 20 people (according to the fire marshal), so that's the maximum size for any course taught there.

This transitive dependency means that any course in a particular location will have the same MaximumParticipants value. To remove this dependency, you should create a new table that lists the MaximumParticipants value for each location, and then remove MaximumParticipants from the CourseOfferings table. However, the design already contains a Locations lookup table that holds location names. You can use that table if you add the MaximumParticipants field to it.

Note that it's not always a good idea to combine tables in this manner. In this case, however, the new version of the table holds data for a single, clearly defined purpose: to describe locations. Because the table's fields both fit this purpose, it makes sense to put them in the same table.

Figure 14.5 shows the new design. The new version of the Locations table is no longer simply a lookup table, so it's not drawn with a dashed rectangle as it was in Figure 14.4.

Notice that the Locations table acts as a lookup table for the CourseOfferings table's Location field. The Locations table is not only a lookup table, however, so it's not drawn with a dashed rectangle.

One way to see that this is not simply a lookup table is to think about removing it. You could remove the pure lookup tables from the database and the database would still function, although you would need to implement some field-level check constraints. If you removed the Locations table, you would lose all of the MaximumParticipants data.

Note that the business rules could have indicated that different courses using a particular location might be able to have different numbers of participants. For example, a seminar on piranha feeding takes less room than a hands-on elephant training workshop, so more people and pets will fit in the room. You could model that situation by making the Locations table use Location and AnimalType as its primary key, but this example seems complicated enough already.

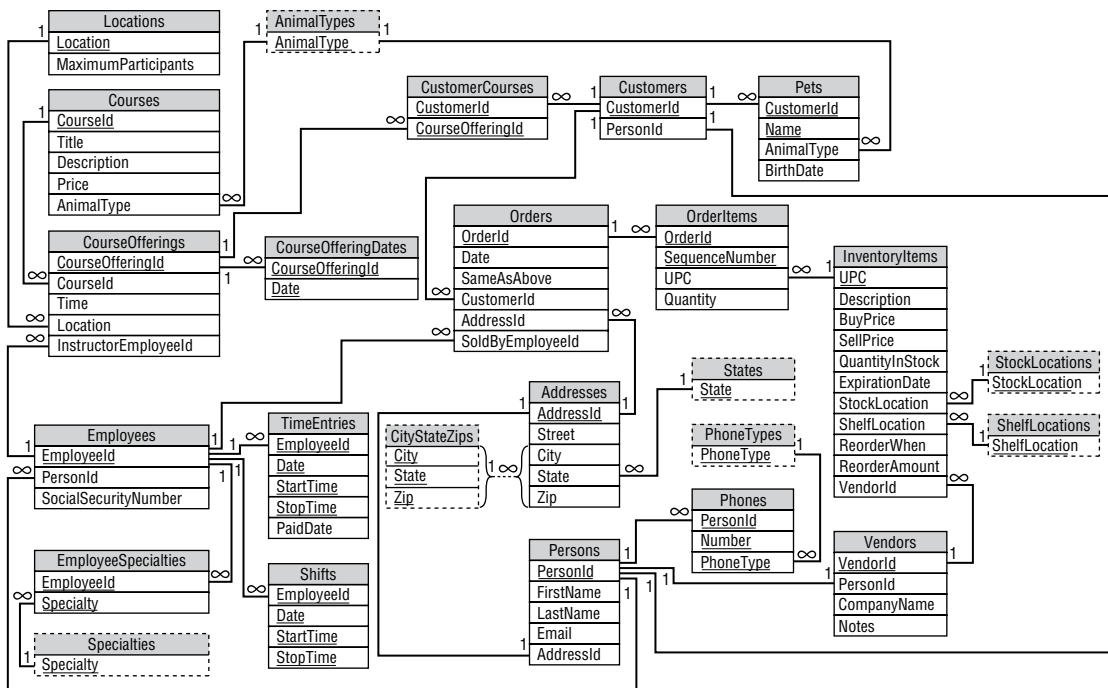


FIGURE 14.5

SUMMARY

This chapter refined the Pampered Pet database to increase its flexibility. It also normalized the database's tables to make the database more resistant to data anomalies.

This chapter showed examples of:

- Increasing the database's flexibility by allowing multiple offerings of a particular course
- Putting the CourseOfferings table into 1NF by moving course dates into a new CourseOfferingDates table
- Putting the Employees table into 1NF by moving employee specialty information into a new EmployeeSpecialties table
- Putting the CourseOfferings table into 3NF by moving information about the maximum number of participants at a location into a new Locations table

Of course, you might have spotted these problems earlier and been muttering under your breath about how silly the design was for the last few chapters, but sometimes these problems sneak through to the bitter end.

At this point, the database is in pretty good shape, and you should be able to build it with some confidence that it can successfully fend off some serious data anomalies.

However, Figure 14.5 doesn't show the complete picture. It shows the table structures and lookup tables, but it doesn't show the many additional constraints that were identified in Chapters 11 and 12. Those must be implemented as field- and table-level check constraints.

This chapter ends the Pampered Pet database case study. The next part of the book describes examples that build local and cloud databases that demonstrate several different database management systems in Python and C#.

Before you move on to Chapter 15, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

EXERCISES

For these exercises, consider the following aquarium show schedule. The show times with asterisks match with the shows with asterisks. For example, the 11:15 show in Sherman's Lagoon is "Sherm's Shark Show" and the 1:15 show is "Meet the Rays." (Yeah, I think it's a strange way to list shows, too, but I saw a real schedule very similar to this one recently.)

SHOW	VENUE	SEATING	TIMES
Sherm's Shark Show/Meet the Rays*	Sherman's Lagoon	375	11:15, 1:15*, 3:00, 6:00*
Deb's Daring Dolphins/The Walter Walrus Comedy Hour*	Peet Amphitheater	300	11:00, 12:00, 2:00*, 5:27*, 6:30
Flamingo Follies/Wonderful Waterfowl*	Ngorongoro Wash	413	2:00, 3:00*

1. Explain why this table isn't in 1NF. Make a relational design that uses one table in 1NF. Show the data in the new table.
2. Explain why the solution to Exercise 1 isn't in 2NF. Make a relational design that fixes it. Show the data in the new tables.
3. Explain why the solution to Exercise 2 isn't in 3NF. Make a relational design that fixes it. Show the data in the new tables.
4. If you made the fewest changes possible while converting the original table into 1NF, 2NF, and 3NF, the new tables probably use ShowName, Time, and Venue Name as primary keys. That bodes ill if you need to change a show's name (for example, if Pete Penguin holds out for equal billing in "The Walter Walrus Comedy Hour"), a time, or a venue's name (the trustees decide to sell naming rights and change the name of "Peet Amphitheater" to "Pampered Pet Cove").

Modify the design to make those kinds of changes easier. Show the data in the new tables.

PART 4

Example Programs

- **Chapter 15:** Example Overview
- **Chapter 16:** MariaDB in Python
- **Chapter 17:** MariaDB in C#
- **Chapter 18:** PostgreSQL in Python
- **Chapter 19:** PostgreSQL in C#
- **Chapter 20:** Neo4j AuraDB in Python
- **Chapter 21:** Neo4j AuraDB in C#

- **Chapter 22:** MongoDB Atlas in Python
- **Chapter 23:** MongoDB Atlas in C#
- **Chapter 24:** Apache Ignite in Python
- **Chapter 25:** Apache Ignite in C#

The chapters in this part of the book describe example programs that demonstrate various database engines in Python and C#. I've designed the examples to include a variety of database types both locally on your computer and in the cloud. I've also picked database products that you can try for free.

Chapter 15 provides an overview of the examples and explains how to install the two programming languages that the examples use: Python (in the form of Jupyter Notebook) and C# (which is included in Visual Studio). Be sure to read this chapter if you want to install Python or C# so that you can build the examples on your computer.

Chapters 16 through 25 describe the example programs. You can download the examples' source code at www.wiley.com/go/beginningdbdesign2e.

15

Example Overview

Up until now, the chapters have dealt with databases in general. The chapters in this part of the book describe example programs that show how to connect to various types of databases and use them in very simple ways.

TOOL CHOICES

There are hundreds of database engines floating around the Internet that you can use to power your programs. Multiply those by the *many* hundreds of programming languages and the handful of possible operating systems that you can use, and the total number of combinations is enormous. To keep things manageable, I chose a few example databases, two programming languages (Python and C#), and one operating system (Windows 11).

NOTE *This is not a Python or C# book, so I won't explain how to use those languages. The example chapters assume that you already know Python or C#, although they don't assume you're a ninja super-programmer.*

If you don't know either of those languages but you know some other curly bracket language like C++ or Java, then you'll probably have no trouble understanding the C# chapters. If you aren't familiar with a curly bracket language, then the Python chapters might be a bit easier to follow.

I selected these databases and programming languages partly because they let you get started for free. They don't require you to provide a credit card and they don't expire. Instead, they restrict the size or structure of the databases that you can build.

I chose Windows 11 because that's what I'm using.

It is likely that you can get the examples to run with other databases, languages, and operating systems with a little work, but I make no promises and I won't be able to help much if you run into trouble. If you try some other combination and get stuck, search online for others who have tried your chosen database and language. You can also look on the database vendor's website or contact them to see if they have examples. As long as you're not trying to do anything too strange, they may be able to help you.

Table 15.1 summarizes the examples, the kind of database each demonstrates, and whether it runs locally or in the cloud.

TABLE 15.1 Examples, the database each demonstrates, and the location where they run

CHAPTERS	DATABASE	DATABASE TYPE	LOCATION
16–17	MariaDB	Column-oriented	Local
18–19	PostgreSQL	Relational	Local
20–21	Neo4j AuraDB	Graph	Cloud
22–23	MongoDB Atlas	Document	Cloud
24–25	Apache Ignite	Key-Value	Local

Even with these limited selections, other options are possible. For example, Chapter 18, “PostgreSQL in Python,” and Chapter 19, “PostgreSQL in C#,” describe examples that use PostgreSQL to run a relational database on your local computer, but Amazon Web Services (AWS), Microsoft Azure, and other providers can also run PostgreSQL in the cloud (just not for free). With a few modifications, you should be able to run any of these either locally or in the cloud.

My tech editor, John Mueller, also tells me that PostgreSQL and Python both work quite easily on Linux systems, so the examples shown here should work with little or no modification (as long as you use the same version of Python that I used).

You should also be able to use many other database products that are eagerly awaiting your business online. Some offer free versions and some have limited trials. This doesn't mean getting them to work will be easy, however. It took me between a few hours and a few days to figure out how to get the examples to connect to their databases.

If you get stuck trying to use a new database, first search for online examples. There's an excellent chance that someone out there has already tried the exact combination of operating system, programming language, and database engine that's currently ruining your day. If that doesn't work, you could contact the database vendor. They have a vested interest in getting people to use their product, so they'll probably help if they can.

The following sections explain how to install the two development environments I used for the examples: Jupyter Notebook (Python) and Visual Studio (C#).

JUPYTER NOTEBOOK

There are many versions of Python that you can use. I've decided to use Jupyter Notebook because it's fairly easy to get up and running. It runs in a browser but can also run offline in case you need to update your local database while you're climbing Mount Everest or something. (The only clouds you can access from the peak are the fluffy white kind that hold rain and snow, but there is a wireless network at the base camp, so at least you can synchronize your data and achieve “eventual consistency” on your way down.)

Jupyter Notebook is a web application that lets you build programs in a literate programming style. In *literate programming*, pieces of code can be interspersed with chunks of documentation and program output such as text, graphs, pictures, and other artifacts. We're not going to need all of that, however. We're just using Notebook as an easy place to write Python code.

If you use some other flavor of Python, much of the example code should be the same. In fact, I wouldn't be surprised if all of the examples work for you with minimal changes, but I make no promises.

For a brief overview of Jupyter Notebook and some installation instructions, go to the “Jupyter/IPython Notebook Quick Start Guide” at <https://jupyter-notebook-beginner-guide.readthedocs.io>. The links on the left lead to some basic information, installation instructions, and notes on running Notebook.

The installation instructions have only two steps:

1. Install a modern browser. Firefox and Chrome work well. (I'm using Chrome for this.)
2. Go to www.anaconda.com/download and download and install Anaconda, which includes Jupyter Notebook. The download is pretty big (currently a bit under 600 MB for Windows), so be sure you have a fast Internet connection.

NOTE I installed Anaconda 2021.11, which included Jupyter Notebook version 6.4.5. You may need to tweak the code slightly if you have different versions.

After you've installed Notebook, you can start it by opening the Windows Start menu and searching for **Jupyter Notebook (Anaconda)**. That does two things. First, it starts a notebook server running in a console window like the one shown in Figure 15.1. That window should remain as long as you want to run notebooks.

The second thing the Start menu command does is open a browser window holding a directory navigator, like the one shown in Figure 15.2.

You can use the directory navigator to move around the filesystem. Click a directory's link to open it. Use the New menu's Folder command to create a new folder. Use its Python 3 (ipykernel) command to create a new notebook.

```

Jupyter Notebook (Anaconda) x + - □ ×
[W 2022-12-05 08:31:56.378 LabApp] 'notebook_dir' has moved from NotebookApp to ServerApp. This config will be passed to
ServerApp. Be sure to update your config before our next release.
[W 2022-12-05 08:31:56.378 LabApp] 'notebook_dir' has moved from NotebookApp to ServerApp. This config will be passed to
ServerApp. Be sure to update your config before our next release.
[I 2022-12-05 08:31:56.383 LabApp] JupyterLab extension loaded from C:\ProgramData\Anaconda3\lib\site-packages\jupyterlab
[I 2022-12-05 08:31:56.383 LabApp] JupyterLab application directory is C:\ProgramData\Anaconda3\share\jupyter\lab
[I 08:31:56.388 NotebookApp] The port 8888 is already in use, trying another port.
[I 08:31:56.388 NotebookApp] Serving notebooks from local directory: C:\Users\rod
[I 08:31:56.388 NotebookApp] Jupyter Notebook 6.4.5 is running at:
[I 08:31:56.388 NotebookApp] http://localhost:8889/?token=b52f9b159827614a6d455d6c0527575524582db7214d0370
[I 08:31:56.388 NotebookApp] or http://127.0.0.1:8889/?token=b52f9b159827614a6d455d6c0527575524582db7214d0370
[I 08:31:56.388 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 08:31:56.408 NotebookApp]

To access the notebook, open this file in a browser:
file:///C:/Users/rod/AppData/Roaming/jupyter/runtime/nbserver-25440-open.html
Or copy and paste one of these URLs:
http://localhost:8889/?token=b52f9b159827614a6d455d6c0527575524582db7214d0370
or http://127.0.0.1:8889/?token=b52f9b159827614a6d455d6c0527575524582db7214d0370

```

FIGURE 15.1

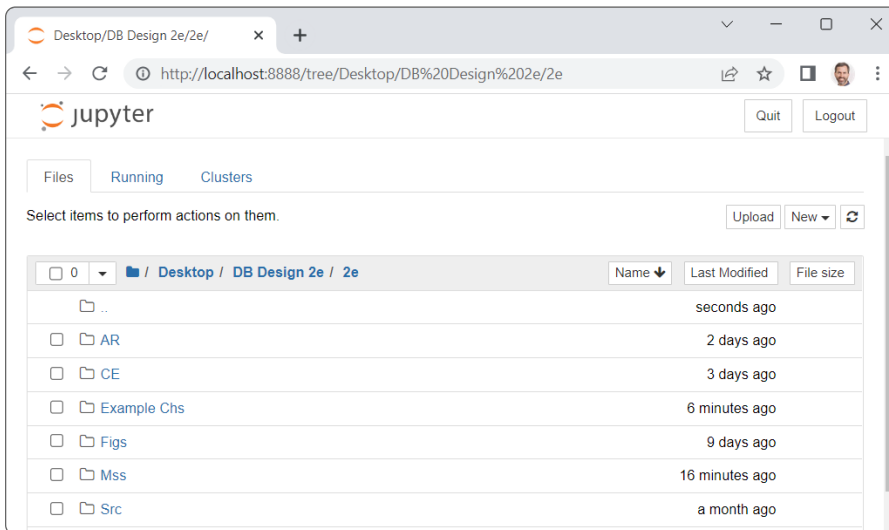


FIGURE 15.2

Figure 15.3 shows a notebook that executes a `print` statement. The notebook displays the output from the statement below the code block.

If you haven't used Jupyter Notebook before, you may want to spend a few minutes experimenting with it.

One nice thing about Jupyter Notebook is that you can define cells that contain documentation or code and then run the cells individually. The examples in the following chapters use that ability to describe the code in pieces. If you work through the examples, you can place each piece of code in its own cell and then execute them.

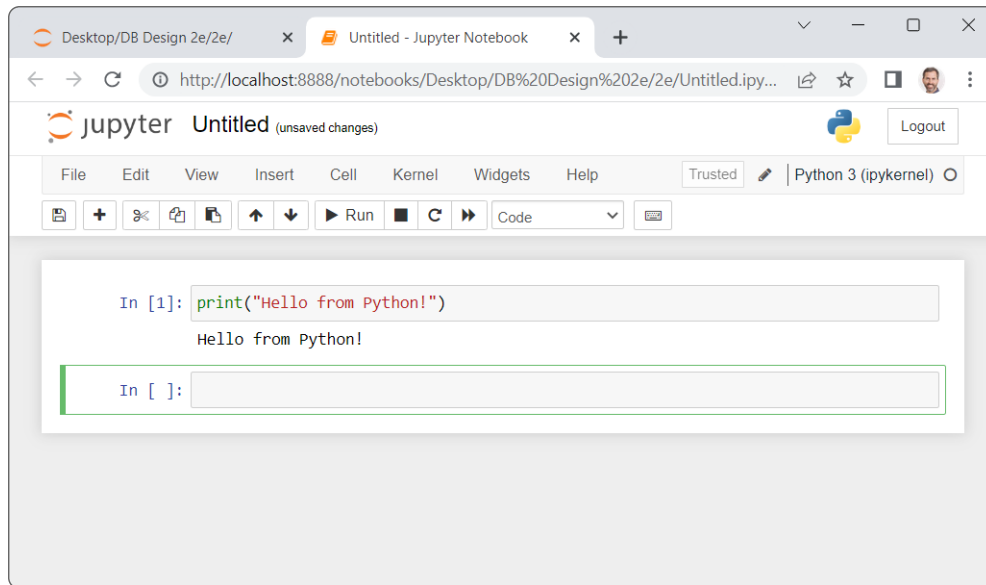


FIGURE 15.3

VISUAL STUDIO

Jupyter Notebook is a development environment for building and running Python programs. Similarly, Visual Studio is a development environment for creating C# programs.

Python and Jupyter Notebook provide relatively few advanced features (although you can add more complicated tools via packages) to help you get started relatively quickly. In contrast, Visual Studio is an extremely powerful development environment that contains a host of tools such as a debugger, code profiler, object browser, and refactoring tools. Fortunately, you don't need to use most of those to get started.

To download Visual Studio, go to <https://visualstudio.microsoft.com/downloads>, look in the Community column, and click the Free Download button.

NOTE Microsoft sometimes moves their download page and changes the way it looks. If the download address shown doesn't work, just search online for **download visual studio** and go to the appropriate Microsoft download page. Be sure to install the free Community edition. You won't need the features in the more expensive editions for the examples in this book. (In fact, you might never need those features.)

After you download the installer, double-click it to start it running. The installer itself is only a couple of megabytes, but when it runs it downloads a huge amount of data, so start the process on a fast Internet connection. The whole process normally takes between half an hour and an hour, but it may take much longer depending on the speed of your computer and your Internet connection. Visual Studio also requires between a few gigabytes and 60 GB or so of disk space depending on which options you install.

NOTE These examples use Microsoft Visual Studio Community 2022 (64-bit). Visual Studio automatically installs new minor builds, but it doesn't change the major release (in this case version 2022) automatically. The code should work for new minor builds, but you may need to modify the code slightly if you install a more recent major release.

Normally, the installation is smooth. If it seems like the installation is stuck, have patience and let it run for at least a few hours. Visual Studio installations have always worked smoothly for me, but some took a lot longer than I expected. If you're sure you're having real problems, search online for help.

DATABASE ADAPTERS

A typical database application consists of three main parts.

Database Engine The *database engine* includes the application that knows how to manipulate the database files. It can build tables, create indexes, perform CRUD operations, execute SQL code, fetch records, and so forth. It's written in C++, or assembly language, or Sanskrit for all we know. We don't really care because we don't use this code directly. Instead, we use a database adapter to interact with the database engine.

Database Adapter The *database adapter* or *connector* is a library that allows your program to communicate with the database engine. It might be written in the same language that you're using, but that's not always the case. It provides methods and objects that your programming language can understand so that you can work with the database.

Your Program This is your program written in Python, C#, COBOL, Lisp, or whatever language you use.

Figure 15.4 shows the idea graphically.

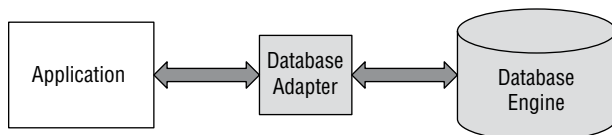


FIGURE 15.4

Internally, your application might be divided into client and server layers to provide extra flexibility, but any part of your program that interacts with the database does so through the adapter. (Technically, you might be able to interact directly with the database engine, but you would have to basically reproduce the features of the adapter, and I'm much too lazy to do that!)

For example, the example described in Chapter 18 uses a Python program running in Jupyter Notebook, the database adapter is Psycopg, and the database engine is PostgreSQL, as shown in Figure 15.5. For this example, all of these are running on the local computer with no cloud required.

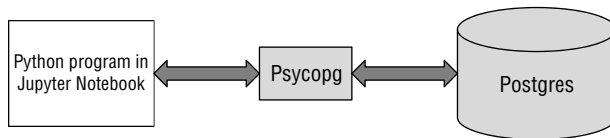


FIGURE 15.5

Before you can use a database adapter, you need to install it. Fortunately, it's relatively easy to install packages in Jupyter Notebook or Visual Studio.

Packages in Jupyter Notebook

If you're an experienced Python programmer, then you can use `pip` to install packages. For example, the following command installs the `pyignite` package that contains the data adapter for Apache Ignite:

```
$ pip install pyignite
```

PIP INSTALLS PACKAGES

Pip is a popular package management system used to install Python packages. When pip installs a package, it looks for other packages that the first package depends on and installs them too. It also looks for dependencies of those packages and so on until it has found a set of consistent packages. Because the final set of packages could be large, you might want to use pip when you have a reasonably fast Internet connection.

Pip was originally called `pyinstall` (many Python packages begin with "py") but later renamed pip. The original author, Ian Bicking, says that pip stands for "Pip Installs Packages."

If you're not familiar with pip, don't despair! You can execute `pip` commands inside Jupyter Notebook. The following command installs `pyignite`:

```
!pip install pyignite
```

You also don't need to worry about breaking the system by accidentally reinstalling a package. If you run the command again, pip installs an updated version if one exists. If you're already running the package's latest version, it displays a message similar to the one shown in Figure 15.6.

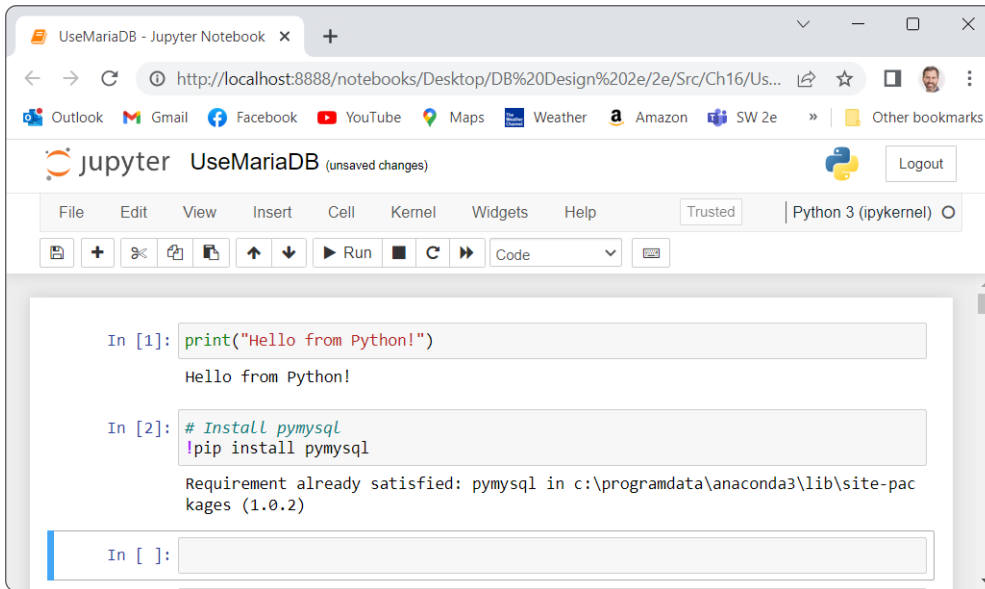


FIGURE 15.6

Packages in Visual Studio

The latest version of Visual Studio uses NuGet (pronounced “new get”) to manage packages. NuGet is a free, open source package manager designed to let developers share code.

To use NuGet in Visual Studio, first create a new project. Then open the Project menu and select Manage NuGet Packages to display the form shown in Figure 15.7.

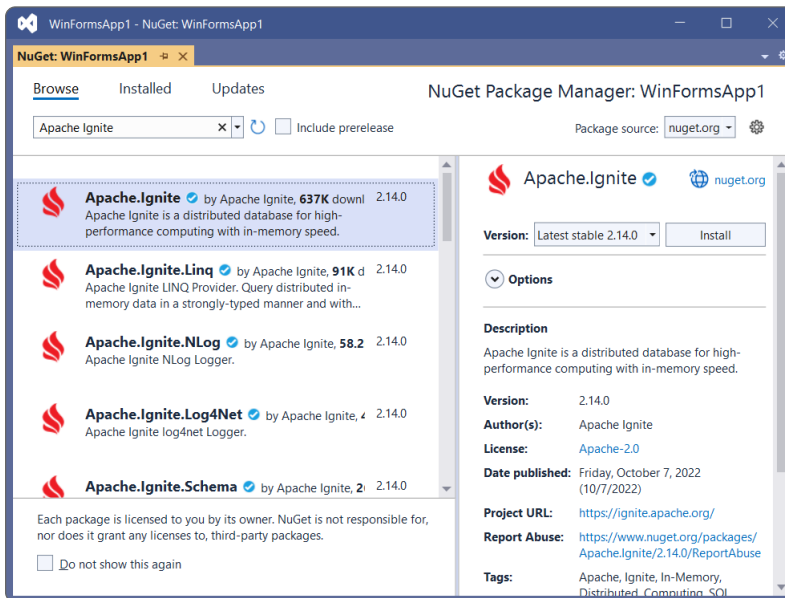


FIGURE 15.7

Click the Browse tab in the upper left, type keywords in the text box below, and press Enter. Click one of the packages to see more information on the right. In Figure 15.7, I entered Apache Ignite and clicked the Apache. Ignite entry so that you can see that package's information on the right.

If you like what you see, click Install to add the package to your project. Visual Studio displays some information about the package that you want to install in the Output window, but that's just the beginning of the process. Like pip, the NuGet package manager recursively checks for dependencies and installs or updates them if necessary. Figure 15.8 shows the packages that NuGet plans to install.

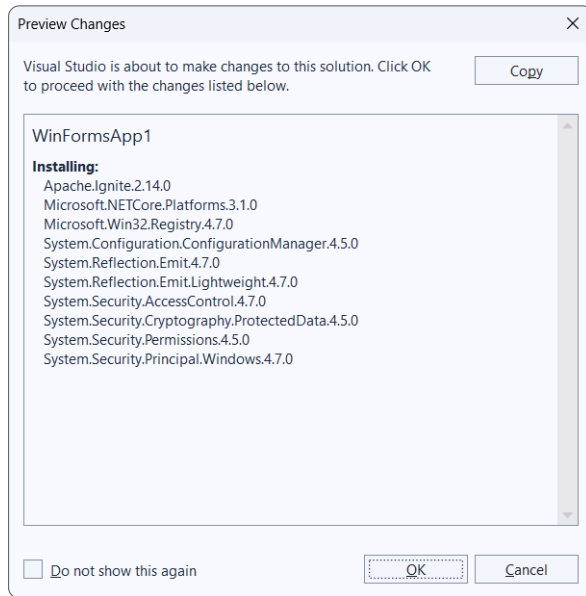


FIGURE 15.8

Because the final set of packages installed packages could be large, you might want to install packages when you have a fast network connection. I didn't follow my own advice and tried this installation on a fairly slow connection, so it took about 10 minutes to get to Figure 15.8. After I clicked OK, the installation only took about 4 seconds and Visual Studio added its results to the Output window.

After you have installed a package, you can use the NuGet package manager to view it. Open the manager as before and click the Installed tab. Be sure to clear the search box or you probably won't see anything. Figure 15.9 shows the information for the Apache.Ignite package that I just installed.

Now you can click Uninstall to remove the package. If you need a specific version of the package for some reason, you can open the Version drop-down, select the version you want, and click Update.

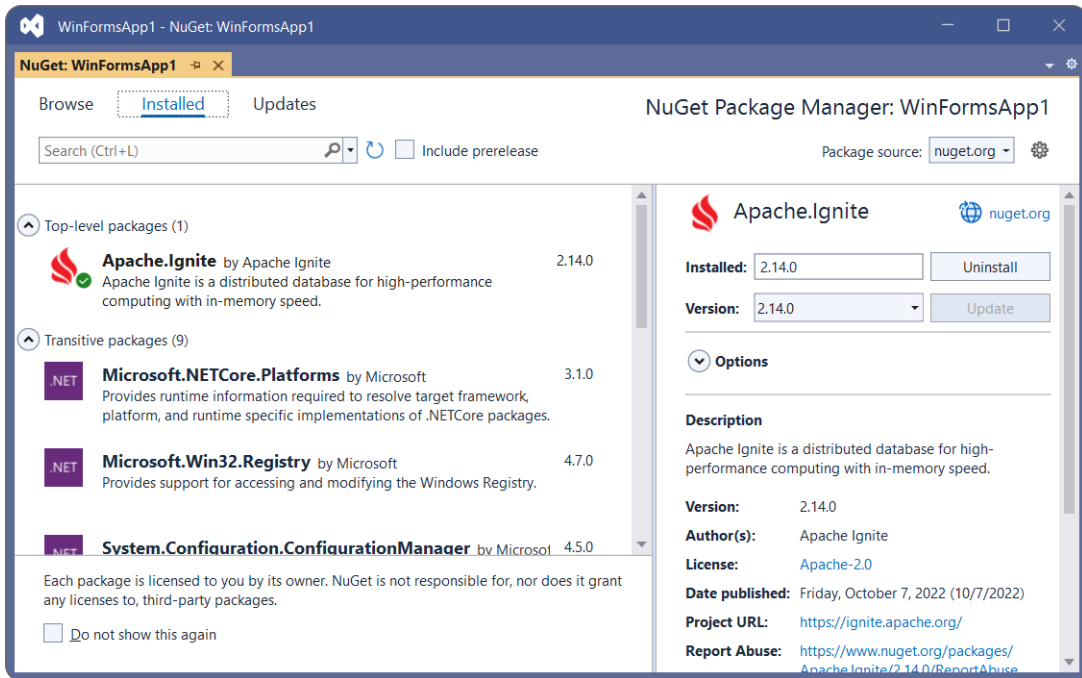


FIGURE 15.9

PROGRAM PASSWORDS

The example programs contain usernames and passwords. That's fine for basic examples but could be dangerous in a production application. Any hacker who got ahold of the program's code could easily see the username and password. It's particularly easy to see the values in a Python program, but even in compiled programs, those values are ripe for the plucking.

This means you need to protect the source code as if it were as important as the password (because it is). It also means that you shouldn't use the same username and password on multiple systems. Otherwise a hacker who gets them on one system gets them for the other systems too.

A better approach is to have the user enter a username and password when the program starts. Then the code doesn't need to contain those values for a hacker to find. That also lets different people log in as different users; that way, you can control their access and disable a user whose password has been compromised.

SUMMARY

The example programs described in the following chapters use Jupyter Notebook (to run Python programs) and Visual Studio (to run C# programs). This chapter briefly explained how you can download and install those development environments.

This chapter also explained how a program interacts with a database through an adapter or connector and how you can install adapters in Jupyter Notebook and Visual Studio. With that knowledge and one or both of the development environments installed, you should be ready to build the examples and experiment with them. Even if you don't want to build the example programs yourself, you can read the following chapters to see how they work.

If you read through both the Jupyter Notebook and C# chapters that follow, you might notice some redundancy. In particular, pairs of chapters mostly share “Summary” and “Exercises” sections. Sorry about that. I tried to strike a balance between conciseness and repeating information for readers who skip one language or the other.

The next chapter describes the first example, which uses Python and MariaDB to build a simple relational database. Before you move on to Chapter 16, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

EXERCISES

1. If you installed Jupyter Notebook, use it to write a Hello World program. (This is just to verify that you have installed Jupyter Notebook correctly, know how to launch it, and can create a notebook.)
2. If you installed Visual Studio, use it to write a C# Hello World Console App (.NET Framework). (This is just to verify that you have installed Visual Studio correctly, know how to launch it, and can create a console app.)
3. What does a database adapter do?
4. What is literate programming?
5. What tool do you use to install packages in Python? In Jupyter Notebook?
6. What tool do you use to install packages in Visual Studio?

16

MariaDB in Python

This example uses the MariaDB database, which is a descendant of the very popular MySQL database. It is a relational database that has a columnar storage engine to give it column-oriented database features. Those features don't change the way you can use MariaDB as a relational database, but they make large column-oriented queries more efficient. That makes MariaDB particularly useful for data warehousing and large-scale data analysis and analytics.

MariaDB ORIGINS

MySQL comes in a free and open source version and also as a product provided by several vendors. It was owned by the Swedish company MySQL AB, which was purchased by Sun Microsystems, which was later acquired by Oracle, who now sells a non-free version of MySQL.

When Oracle acquired Sun, MySQL co-founder Michael Widenius forked the open source version of MySQL to create MariaDB. (If you don't know the term, forked means that Widenius created a new branch or fork in the code tree to start MySQL. It's not a euphemism for something naughty.)

So, MariaDB is a direct descendant of MySQL.

To add a human interest angle to the story, Widenius named MySQL after his older daughter My, MaxDB after his son Max, and MariaDB after his younger daughter Maria. If he invents any new databases, he may need to have more children.

This example builds a database of extraterrestrial animals as described in Chapter 8, "Designing Databases to Support Software." Figure 16.1 shows the design with some data added to the tables.

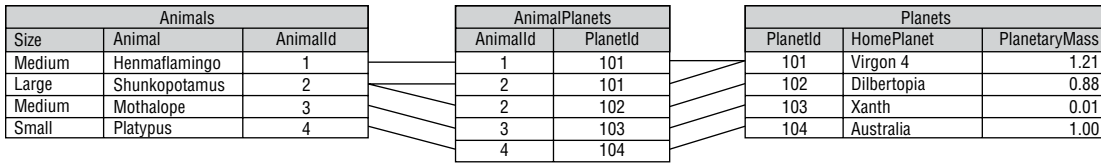


FIGURE 16.1

INSTALL MariaDB

To install MariaDB, first go to <https://mariadb.com/downloads/community> and select the MariaDB Community tab. Then pick the version that you would like (I used the most recent production version) and your operating system. Scroll down slightly and click Download. The installer file is pretty large (around 66 MB for 64-bit Windows), so you'll probably want to download it over a fast Internet connection.

Execute the installer and follow its instructions. When you get to the stage shown in Figure 16.2, pick a nice secure password for the database root. (Root is an all-powerful user who has complete access to the entire database, so this password should be hard to guess and you need to keep it safe.) Don't use an obvious password like "secret," don't leave it on a sticky note (unless you put that in a safe place), and don't publish your password in a database design book. (I used TheSecretPassword.)

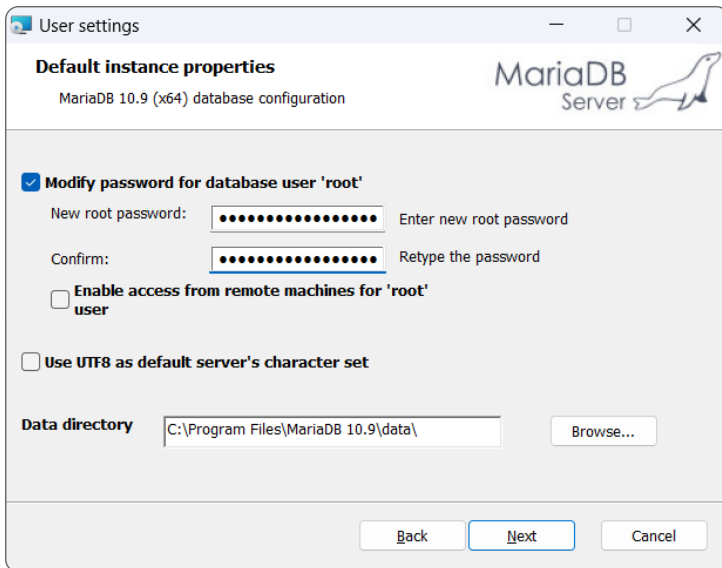


FIGURE 16.2

RUN HeidiSQL

You can search online for database management tools that can work with MariaDB if you like. In my exhaustive 20-second search, I found a few dozen options but none looked trivial to install and none were obviously free, so I decided to do without.

Then I noticed that MariaDB had installed HeidiSQL, a free open source database administration tool that works with MySQL, Microsoft SQL Server, PostgreSQL, SQLite, and probably others. Because it's already installed and free, why not use it?

HeidiSQL ORIGINS

Like MariaDB, HeidiSQL has its origins in MySQL. The original author, Ansgar Becker, forked the HeidiSQL code from his MySQL-Front 2.5 application.

The name was suggested by a friend in honor of Heidi Klum, and Becker adopted the name partly in nostalgia for the anime television series *Heidi, Girl of the Alps*. (Presumably if Becker has a daughter, he'll name her Heidi.)

I'll briefly explain how to use HeidiSQL to build a database, but this chapter also shows how to build the database with Python code. When you're writing a program to work with a database, it's often useful to build the database in code or with scripts. That approach lets you easily reset the database to a known state so that it contains specific test records that you can work with.

When you launch HeidiSQL, the session manager shown in Figure 16.3 appears.

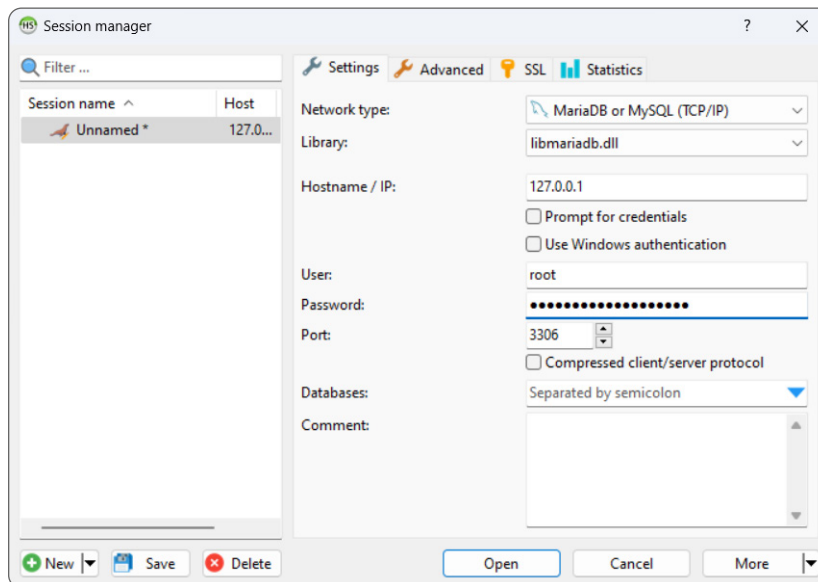


FIGURE 16.3

This example runs locally on your computer, so you should leave the Hostname / IP field with its default setting 127.0.0.1. Enter the password that you gave the root account when you installed MariaDB and click Open to see a window similar to Figure 16.4.

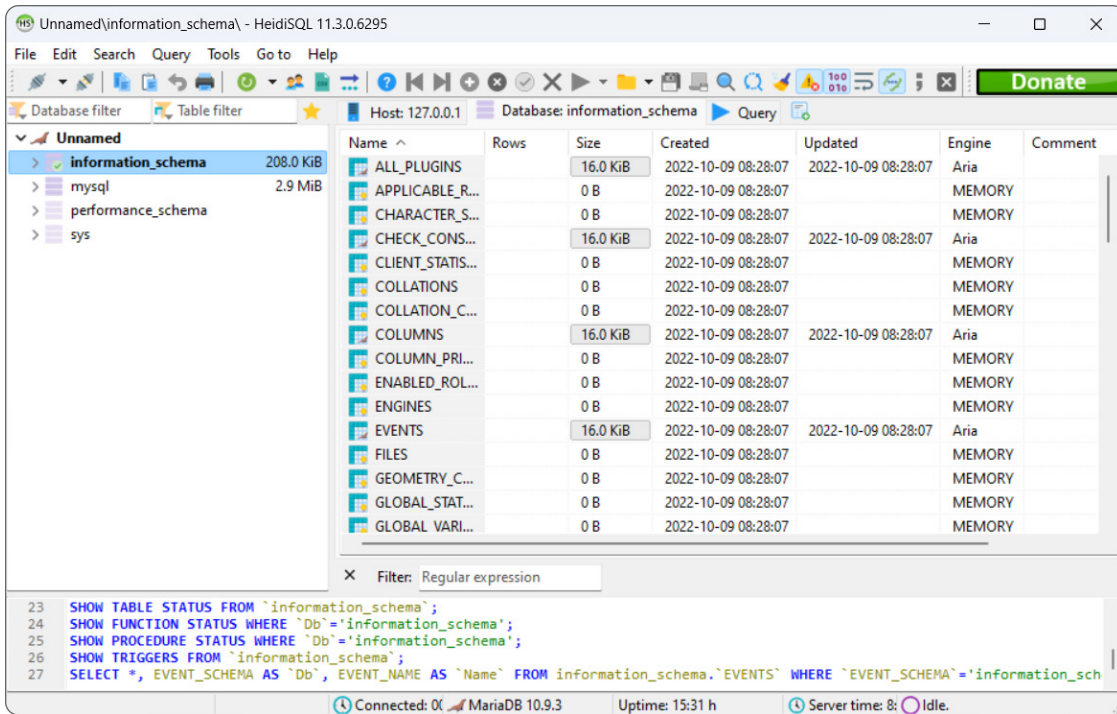


FIGURE 16.4

You can click the system databases on the left if you want to see what they contain.

To create a new database, right-click the Unnamed entry in the list on the left, open the Create New menu, and select Database. In the dialog box shown in Figure 16.5, enter the database name. Select a collation order from the ridiculously long list to determine how the database orders sorted text and click OK.

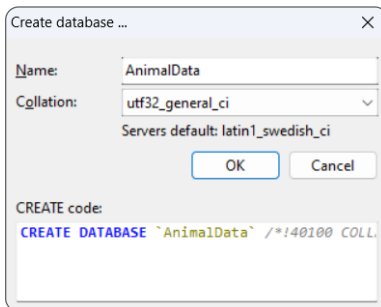


FIGURE 16.5

To create a table, right-click the new database, open the Create New menu, and select Table to display the window shown in Figure 16.6.

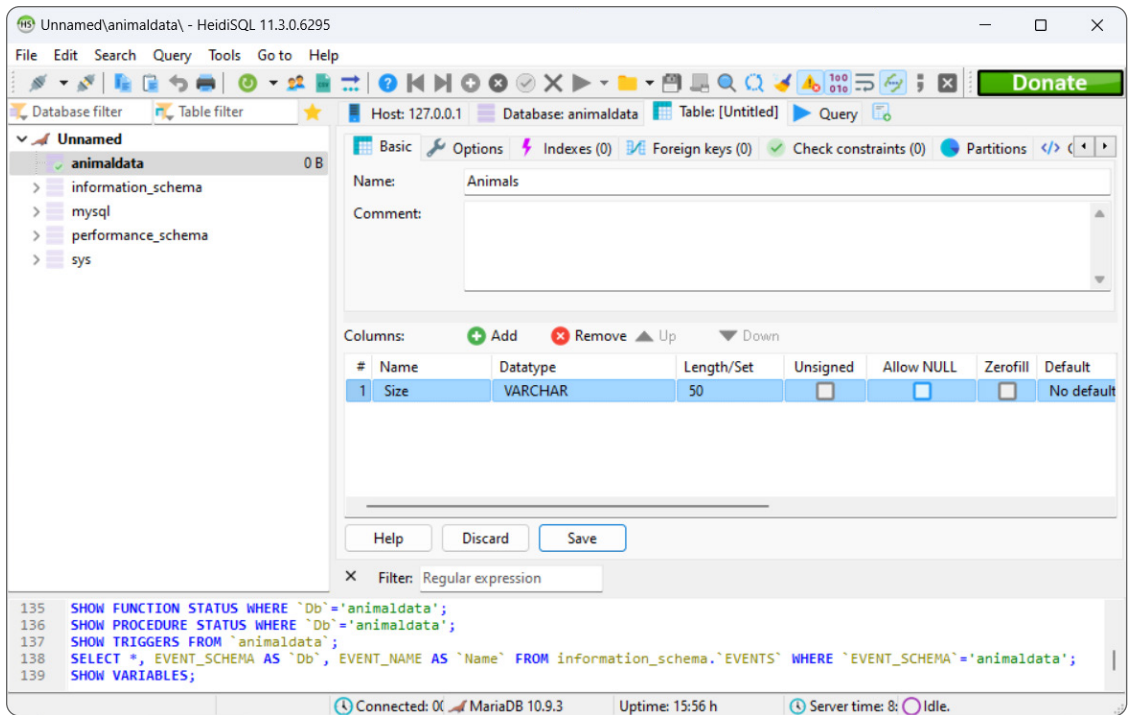


FIGURE 16.6

Click the Add button in the middle (below the table's Name and Comment fields) to create a new column. Then use the column list to set the column's properties. For example, click the column's name and type a new name, click its data type to pick a new type from a drop-down list, and deselect the Allow NULL box to make the field required.

You can continue to use HeidiSQL to build the entire database if you like. At this point, I'm going to stop working with HeidiSQL and write a Python program that creates the database shown in Figure 16.1.

CREATE THE PROGRAM

To create a Python program to work with the extraterrestrial animals MariaDB database, create a new Jupyter Notebook and then add the code described in the following sections.

The following sections describe each of the example program's code cells. Each cell connects to the database, performs some actions, and then closes the database connection.

If you like, you can join most of the cells together so you open the connection at the beginning and then close it at the end. You'll notice that the connection is slightly different when you're creating the database and when you're working with it, but I'm sure you can figure that out.

Install pymysql

Either use pip to install pymysql or enter the following code into a new notebook cell and execute it:

```
!pip3 install pymysql
```

The result should be similar to the following, possibly with different version numbers:

```
Collecting pymysql
  Downloading PyMySQL-1.0.2-py3-none-any.whl (43 kB)
Installing collected packages: pymysql
Successfully installed pymysql-1.0.2
```

Create the Database

MariaDB has a good online knowledge base, so it's easy to find explanations of the SQL commands that it supports. For example, if you search for **MariaDB create user**, you can find a page that explains the `CREATE USER` command. I'm going to be lazy and use the root user for this example, but in general it's better to create a separate database user for each flesh-and-blood user so that they can log in separately and so that you can minimize their privileges. It's generally safer if normal users cannot create and destroy databases, tables, and other users at will.

To create the database, enter the following code in a new notebook cell and execute it:

```
# Create the AnimalData database.
import pymysql

# Connect to the database server.
conn = pymysql.connect(
    host="localhost",
    user="root",
    password="TheSecretPassword")

# Create a cursor.
cur = conn.cursor()

# Drop the database if it exists.
cur.execute("DROP DATABASE IF EXISTS AnimalData")

# Create the database.
cur.execute("CREATE DATABASE AnimalData")

# List the available databases.
cur.execute("SHOW DATABASES")
databases = cur.fetchall()
for database in databases:
    print(database)

# Close the connection.
conn.close()
```

This code first imports the `pymysql` module. It then creates a connection object attached to the database server. Replace the password shown here with the one you used when you installed MariaDB. Notice that this code does not specify a database because we haven't created one yet.

LOUDY WITH A CHANCE OF DATA

This example uses a connect string where the host is `localhost`, which represents your local system. If the database's server is located on a different computer, you would specify the server's computer as either an IP address (as in `8.8.8.8`, which is the primary DNS server for Google DNS) or an Internet domain (as in `whitehouse.gov`).

This lets the program access the database over a network, so it's sort of "cloudy," but it's not really the cloud unless the server is hosted on a cloud provider.

Next, the code creates a cursor. Databases use cursors to execute commands and return results. This code uses the cursor's `execute` method to execute the following SQL statement:

```
DROP DATABASE IF EXISTS AnimalData
```

SQL is reasonably English-like, so you can probably guess that this drops the database named `AnimalData` (for example, if you used HeidiSQL to create it). The `IF EXISTS` clause tells the cursor to not try to drop the database if it doesn't exist. Trying to drop a database that doesn't exist (or more generally, to work with any object that doesn't exist) will make the program throw a temper tantrum.

Next, the code executes the following SQL statement to create the database:

```
CREATE DATABASE AnimalData
```

The code then uses the cursor to execute the `SHOW DATABASES` query command to list the available databases. It uses the cursor's `fetchall` method to retrieve the query's results and then loops through the results and prints them.

The code finishes by closing the database connection to release the connection's resources.

A CLOSE BY ANY OTHER NAME

Different databases may dispose of connections differently. With some, you should close the connection, as is done here. Sometimes, the connection isn't actually destroyed; it's just returned to a connection pool so that other programs can reuse it.

Some databases provide a `release` method (or some other method) to explicitly return the connection to the pool.

Many databases are optimized to open and close or reuse connections, so programs should not hold them open while they are not using them.

The following text shows the output from the previous code:

```
('animaldata',)
('information_schema',)
('mysql',)
('performance_schema',)
('sys',)
```

Each of these values is a tuple (a sort of unchangeable list) containing a single item that holds the name of a database.

COMMA CONFUSION

The commas at the ends of the tuples are there because of the way Python interprets parentheses. If a tuple contains only one item, Python ignores the parentheses and just uses the item instead of creating a tuple. For example, the code (1) just represents the value 1. Adding a comma at the end turns it into a tuple, so (1,) represents a tuple that contains the single value 1.

The first returned item is the database that we just created. The others represent system databases and schemas that MariaDB needs to manage its data. If you compare this output to Figure 16.6, you'll see the same list of databases.

Define Tables

Enter the following code into a new notebook cell to create the database's tables:

```
# Create the tables.
import pymysql

# Connect to the database server.
conn = pymysql.connect(
    host="localhost",
    user="root",
    password="TheSecretPassword",
    database="AnimalData")

# Create a cursor.
cur = conn.cursor()

# Drop any existing tables.
cur.execute("DROP TABLE IF EXISTS AnimalPlanets")
cur.execute("DROP TABLE IF EXISTS Animals")
cur.execute("DROP TABLE IF EXISTS Planets")

# Create the Animals table.
cmd = """CREATE TABLE Animals
(
```

```

        Size TEXT NOT NULL,
        Animal TEXT NOT NULL,
        AnimalId INT PRIMARY KEY
    ) ""
cur.execute(cmd)

# Create the Planets table.
cmd = """CREATE TABLE Planets
(
    PlanetId INT PRIMARY KEY,
    HomePlanet TEXT NOT NULL,
    PlanetaryMass FLOAT NOT NULL
) ""
cur.execute(cmd)

# Create the AnimalPlanets table.
# Foreign keys:
#   AnimalPlanets.AnimalId = Animals.AnimalId
#   AnimalPlanets.PlanetId = Planets.PlanetId
cmd = """CREATE TABLE AnimalPlanets
(
    AnimalId INT,
    PlanetId INT,
    CONSTRAINT fk_animals
        FOREIGN KEY (AnimalId) REFERENCES Animals (AnimalId)
        ON DELETE RESTRICT
        ON UPDATE RESTRICT,
    CONSTRAINT fk_planets
        FOREIGN KEY (PlanetId) REFERENCES Planets (PlanetId)
        ON DELETE RESTRICT
        ON UPDATE RESTRICT,
    PRIMARY KEY (AnimalId, PlanetId)
) ""
cur.execute(cmd)

# Close the connection.
conn.close()

```

The code first creates a database connection much as before. This time, however, it includes the database name `AnimalData` so it will work with that database.

Next, the code executes three `DROP TABLE` statements to delete the `AnimalPlanets`, `Animals`, and `Planets` tables if they already exist. Note, you must drop the tables in a valid order. In this database, the `AnimalPlanets` table has foreign key constraints matching fields in the `Animals` and `Planets` tables. If the `AnimalPlanets` table contains data and you drop either of the other tables first, then the `AnimalPlanets` records will violate their constraints and the program will crash.

DEADLY DROPS

The SQL statement `DROP TABLE` immediately destroys the table and all of its data with no do-overs or mulligans. Be sure you really want to discard all the data before you execute this command.

Dropping tables (if they exist) and then re-creating them is a common technique for building test databases. It lets you remove any experimental changes to the tables so that you can start with a known state. (This is the main reason why I didn't use HeidiSQL to build the database. If you build the database with a database management tool and then later decide to modify the table structure, you need to use the tool to rebuild the database's structure. It's often faster and easier to modify a program or script and rerun it to rebuild the database from scratch.)

Next, the code executes three `CREATE TABLE` statements to build the tables. The `CREATE TABLE` statement can be pretty complicated, but you can look at the code to get an idea of how it works. The first two statements just list the tables' fields, giving them their data types and some special properties such as `NOT NULL` to indicate that a field is required and `PRIMARY KEY` to indicate that a field is the table's primary key.

NOTE *Like many other database engines, MariaDB requires that primary key values be unique and not null.*

The third `CREATE TABLE` statement is a bit more complicated. It defines the table's fields, and then defines two foreign key constraints. The field name after the `FOREIGN KEY` keywords indicates the field in this table. The table and field names after the `REFERENCES` keyword tell what table and field the first field must match. In this example, the `fk_animals` constraint requires that each `AnimalPlanets.AnimalId` value must match some `Animals.AnimalId` value.

Similarly, the `fk_planets` constraint requires that `AnimalPlanets.PlanetId` values must match some `Planets.PlanetId` value.

The `ON UPDATE RESTRICT` and `ON DELETE RESTRICT` clauses mean that the database will prevent (restrict) any updates or deletions that violate the constraints. Other choices can make a change cascade to related records or null out related fields.

This `CREATE TABLE` statement finishes with a `PRIMARY KEY` statement that makes the pair of fields `AnimalId` and `PlanetId` the table's primary key.

Create Data

To add some data to the tables, create a new notebook cell and enter the following code:

```
# Create some data.
import pymysql

# Connect to the database server.
conn = pymysql.connect(
    host="localhost",
    user="root",
    password="TheSecretPassword",
```



```
        database="AnimalData")

# Create a cursor.
cur = conn.cursor()

# Delete any previous records.
cur.execute("DELETE FROM AnimalPlanets")
cur.execute("DELETE FROM Animals")
cur.execute("DELETE FROM Planets")

# Add records to the Animals table.
cmd = """INSERT INTO Animals
        (Size, Animal, AnimalId) VALUES
        ('Medium', 'Hermaflamingo', 1)"""
cur.execute(cmd)

cmd = """INSERT INTO Animals
        (Size, Animal, AnimalId) VALUES
        ('Large', 'Skunopotamus', 2)"""
cur.execute(cmd)

cmd = """INSERT INTO Animals
        (Size, Animal, AnimalId) VALUES
        ('Medium', 'Mothalope', 3)"""
cur.execute(cmd)

cmd = """INSERT INTO Animals
        (Size, Animal, AnimalId) VALUES
        ('Small', 'Platypus', 4)"""
cur.execute(cmd)

# Add records to the Planets table.
cur.executemany("""
    INSERT INTO Planets (PlanetId, HomePlanet, PlanetaryMass)
    VALUES (%s, %s, %s)""",
    [
        (101, "Virgon 4", 1.21),
        (102, "Dilbertopia", 0.88),
        (103, "Xanth", 0.01),
        (104, "Australia", 1.0)
    ])

# Add records to the AnimalPlanets table.
cur.executemany("""
    INSERT INTO AnimalPlanets (AnimalId, PlanetId)
    VALUES (%s, %s)""",
    [
        (1, 101),
        (2, 101),
        (2, 102),
        (3, 103),
        (4, 104)
    ])

```

```
# Commit the changes.
conn.commit()

# Close the connection.
conn.close()
```

This code creates a database connection and defines a cursor as usual. It then deletes all the records from the database's three tables. Emptying a table before adding test records is another common testing technique.

DANGEROUS DELETIONS

The SQL statement `DELETE FROM Animals` deletes all the records in the table `Animals`. Usually deletions are more targeted. For example, you might use the statement `DELETE FROM Animals WHERE Animal='Platypus'` to delete only the platypus's record.

If you accidentally forget the `WHERE` clause, the database won't warn you before you delete every record in the table. Before you execute any `DELETE` statement, double-check to see if it has the correct `WHERE` clause.

Next, the code creates four `INSERT INTO` statements and executes them one at a time to add records to the `Animals` table. That works but is somewhat verbose.

The code then uses a more concise method to add several records at once. It calls the cursor's `executemany` method, passing it an `INSERT` statement that contains the value `%s` as a placeholder for field values. It also passes the method an array of tuples, giving the values for each record. This approach lets the cursor insert many records all at once, which makes the code shorter. It also allows the database to optimize the insertion operation, so it is likely to be faster than inserting the records one at a time, although you won't notice the difference for such a small example.

After it has added the data, the code calls the connection object's `commit` method. This is important! If you don't commit the changes, then the newly added values are removed and you're left scratching your head while staring at code that you know should work. (Not that I have firsthand experience of that or anything.)

Fetch Data

To verify that the preceding code actually adds data to the database, create a new cell and add the following code:

```
# Fetch data.
import pymysql

# Connect to the database server.
conn = pymysql.connect(
    host="localhost",
    user="root",
    password="TheSecretPassword",
    database="AnimalData")
```

```

# Create a cursor.
cur = conn.cursor()

# Select each table's records.
print("*** Animals ***")
cmd = "SELECT * FROM Animals"
cur.execute(cmd)
rows = cur.fetchall()
for row in rows:
    print(row)

print("\n*** Planets ***")
cmd = "SELECT * FROM Planets"
cur.execute(cmd)
rows = cur.fetchall()
for row in rows:
    print(row)

print("\n*** AnimalPlanets ***")
cmd = "SELECT * FROM AnimalPlanets"
cur.execute(cmd)
rows = cur.fetchall()
for row in rows:
    print(row)

# Select matching records.
cmd = """SELECT Size, Animal, HomePlanet, PlanetaryMass
FROM Animals, Planets, AnimalPlanets
WHERE
    Animals.AnimalId = AnimalPlanets.AnimalId AND
    Planets.PlanetId = AnimalPlanets.PlanetId
ORDER BY Animal"""

print("\n*** Results ***")
cur.execute(cmd)
rows = cur.fetchall()
for row in results:
    print(row)

# Close the connection.
conn.close()

```

This code connects to the database as usual. It then creates three queries that fetch the data from the tables. It passes the queries into the cursor's `fetchall` method and loops through the returned tuples of results and displays each row's data.

The final query selects values from the tables using the ID fields to link corresponding records.

The following text shows the results:

```

*** Animals ***
('Medium', 'Hermaflamingo', 1)
('Large', 'Skunopotamus', 2)
('Medium', 'Mothalope', 3)

```

```
('Small', 'Platypus', 4)

*** Planets ***
(101, 'Virgon 4', 1.21)
(102, 'Dilbertopia', 0.88)
(103, 'Xanth', 0.01)
(104, 'Australia', 1.0)

*** AnimalPlanets ***
(1, 101)
(2, 101)
(2, 102)
(3, 103)
(4, 104)

*** Results ***
('Medium', 'Hermaflamingo', 'Virgon 4', 1.21)
('Medium', 'Mothalope', 'Xanth', 0.01)
('Small', 'Platypus', 'Australia', 1.0)
('Large', 'Skunopotamus', 'Virgon 4', 1.21)
('Large', 'Skunopotamus', 'Dilbertopia', 0.88)
```

SUMMARY

MariaDB is a direct descendant of the extremely popular MySQL database. It is a standard relational database that also provides column-oriented features, making it useful for data warehousing and large-scale data analysis and analytics.

The example described in this chapter creates a new database, defines some tables for it, and adds a little test data to those tables. That sort of application is useful for getting the database up and running, but in a real application you'd probably also need a user interface to let the user query and modify the data.

The next chapter describes a similar example that uses C# to build and manipulate a MariaDB database. Before you move on to Chapter 17, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

EXERCISES

1. Name Michael Widenius's children and the database tools named after them.

2. Name the database management tool that this chapter briefly demonstrates.

3. What is the main advantage of a database management tool over using SQL to build a database?

4. What is the main disadvantage of a database management tool over using SQL to build a database?

5. Add a new notebook cell and add code that does the following:
 - a. Connect to the database and create a cursor as usual.
 - b. Enter an infinite loop that does the following:
 - i. Prompt the user for an animal ID. If the user enters a blank string, break out of the loop. Otherwise, parse the value as an integer. If the user enters a non-integer, display an error message and continue the loop.
 - ii. Prompt the user for a planet ID. If the user enters a blank string, break out of the loop. Otherwise, parse the value as an integer. If the user enters a non-integer, display an error message and continue the loop.
 - iii. Build and execute a SQL command that uses the values to create a new record in the AnimalPlanets table. If something goes wrong, display an error message and continue the loop.
 - c. After the loop ends, commit the changes and close the connection.
 - d. Run the cell and create a new record (1, 104). Test the program by entering non-integers, huge integers, integers that are not present in the tables, and blank values. Run the data fetching cell to verify that the new record is present.

6. What exceptions can you encounter when the cell in Exercise 5 tries to add a new AnimalPlanets record?

17

MariaDB in C#

This example demonstrates the MariaDB database in a C# application. If you skipped Chapter 16, “MariaDB in Python,” which built a similar example in Python, go to that chapter and read the beginning and the first two sections, which are described in the following list:

- “Install MariaDB” explains how to install MariaDB.
- “Run HeidiSQL” explains how to use the HeidiSQL database management tool to create a MariaDB database.

When you reach the section “Create the Program” in Chapter 16, return to this chapter and read the following sections.

CREATE THE PROGRAM

To create a C# program to work with the MariaDB extraterrestrial animals database, create a new C# Console App (.NET Framework) and then add the code described in the following sections.

Jupyter Notebook lets you execute cells individually, but C# won't let you do that. It will, however, let you group related code into methods, so that's what we'll do here.

Add code to the main method so that it looks like the following:

```
using MySQLConnector;
...
static void Main(string[] args)
{
    CreateDatabase();
    // CreateTable();
    // CreateData();
    // FetchData();

    Console.ReadLine();
}
```

Initially, the main method only calls `CreateDatabase` (described shortly) and then waits for the user to press Enter. You'll uncomment the other statements as you work through the following sections, which explain how the other methods work.

Install MySqlConnection

To install a database connector, open the Project menu and select Manage NuGet Packages. If you select the Browse tab and search for MariaDB, you'll find more than 150 offerings. Most of them will probably work, but I decided to use the tool called MySqlConnection. Figure 17.1 shows that package selected in the NuGet Package Manager.

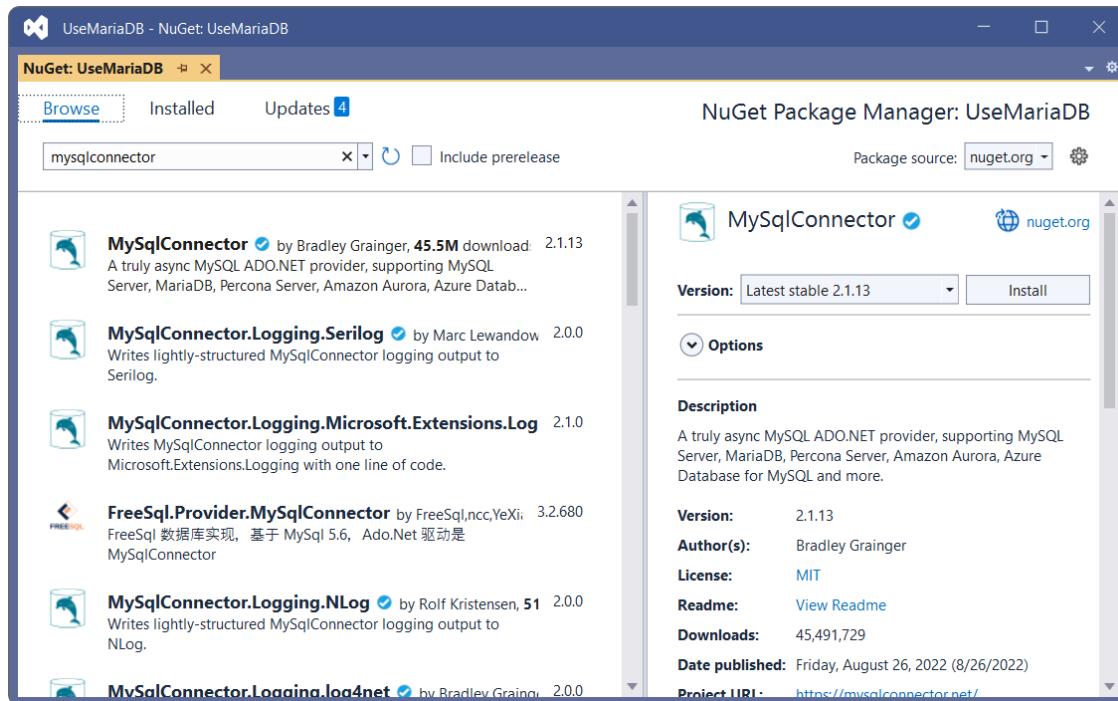


FIGURE 17.1

You can see in the figure that I installed version 2.1.13. When you build your program, the latest stable version will have definitely changed. In fact, it's different now when I'm editing just a few weeks after I built the initial program. Try the latest stable version. If that version won't work with the example code, try selecting version 2.1.13 to see if that works.

Create the Database

The following code shows the `CreateDatabase` method:

```
// Create the AnimalData database.
private static void CreateDatabase()
{
```



```

// Connect to the database server.
string connectString =
    "server=127.0.0.1;" +
    "uid=root;" +
    "pwd=TheSecretPassword";
using (MySqlConnection conn = new MySqlConnection(connectString))
{
    // The connection must be open before you can use it.
    conn.Open();

    // Create a command.
    using (MySqlCommand cmd = conn.CreateCommand())
    {
        // Drop the database if it exists.
        cmd.CommandText = "DROP DATABASE IF EXISTS AnimalData";
        cmd.ExecuteNonQuery();

        // Create the database.
        cmd.CommandText = "CREATE DATABASE AnimalData";
        cmd.ExecuteNonQuery();

        // List the available databases.
        cmd.CommandText = "SHOW DATABASES";
        MySqlDataReader reader = cmd.ExecuteReader();
        while (reader.Read())
        {
            Console.WriteLine(reader[0].ToString());
        }
    } // End using MySqlCommand cmd
} // End using MySqlConnection conn

Console.WriteLine();
}

```

This code first creates a database connect string that includes the server (127.0.0.1 means the local machine), username (root), and password. Replace the password shown here with the password that you used for the root user when you installed MariaDB.

The code passes the connect string to the constructor when it creates a `MySqlConnection` object. That object uses system resources, so it provides a `Dispose` method to free them when you're done with the connection. To ensure that you don't forget to call `Dispose`, the code creates the connection inside a `using` statement, so the program will call `Dispose` automatically when the code block ends.

Next, the program opens the connection. It then uses it to create a `MySqlCommand` object. Like `MySqlConnection`, `MySqlCommand` has a `Dispose` method so the program creates the command inside a `using` statement.

The code then sets the command object's `CommandText` property to the SQL statement `DROP DATABASE IF EXISTS AnimalData`. As I'm sure you can guess, that statement drops the `AnimalData` database if it exists.

The code calls the command object's `ExecuteNonQuery` method to execute the statement. That object has a few other `ExecuteXxx` methods that execute different kinds of statements. This time we're using `ExecuteNonQuery` because the `DROP TABLE` statement is not a query and does not return data.

The program then performs similar steps, setting the command's text to `CREATE DATABASE AnimalData` and executing that non-query.

After it has created the database, the code executes the `SHOW DATABASES` command. This command returns results, so the program calls the command's `ExecuteReader` method to perform the query and obtain a data reader object that holds the results.

The code calls the reader's `Read` method to advance to the first result row. That method returns true if as it retrieves a row and it returns false if it has run out of rows, so the `while` statement continues until the program has processed all the returned data.

Inside the loop, the program writes the value in the reader's first field to the console window. The `SHOW DATABASES` command only returns the names of the databases, so the reader only has one field.

If you run the program now, it deletes the database (if it exists), creates the new database, displays a list of the available databases, and then waits until you press Enter before closing the program. The following text shows the program's output:

```
animaldata
information_schema
mysql
performance_schema
sys
```

In this output, you can see the new table `animaldata` and the system tables.

Define Tables

The following code shows the `CreateTables` method:

```
// Create the tables.
private static void CreateTables()
{
    // Connect to the database server.
    string connectionString =
        "server=127.0.0.1;" +
        "uid=root;" +
        "pwd=TheSecretPassword;" +
        "database=AnimalData";
    using (MySqlConnection conn = new MySqlConnection(connectionString))
    {
        // The connection must be open before you can use it.
        conn.Open();

        // Create a command.
        using (MySqlCommand cmd = conn.CreateCommand())
        {
            // Drop any existing tables.
            cmd.CommandText = "DROP TABLE IF EXISTS AnimalPlanets";
```

```

cmd.ExecuteNonQuery();

cmd.CommandText = "DROP TABLE IF EXISTS Animals";
cmd.ExecuteNonQuery();

cmd.CommandText = "DROP TABLE IF EXISTS Planets";
cmd.ExecuteNonQuery();

// Create the Animals table.
cmd.CommandText = @"CREATE TABLE Animals (
    Size TEXT NOT NULL,
    Animal TEXT NOT NULL,
    AnimalId INT PRIMARY KEY)";
cmd.ExecuteNonQuery();

// Create the Planets table.
cmd.CommandText = @"CREATE TABLE Planets (
    PlanetId INT PRIMARY KEY,
    HomePlanet TEXT NOT NULL,
    PlanetaryMass FLOAT NOT NULL)";
cmd.ExecuteNonQuery();

// Create the AnimalPlanets table.
// Foreign keys:
//     AnimalPlanets.AnimalId = Animals.AnimalId
//     AnimalPlanets.PlanetId = Planets.PlanetId
cmd.CommandText = @"CREATE TABLE AnimalPlanets (
    AnimalId INT,
    PlanetId INT,
    CONSTRAINT fk_animals
        FOREIGN KEY (AnimalId) REFERENCES Animals (AnimalId)
        ON DELETE RESTRICT
        ON UPDATE RESTRICT,
    CONSTRAINT fk_planets
        FOREIGN KEY (PlanetId) REFERENCES Planets (PlanetId)
        ON DELETE RESTRICT
        ON UPDATE RESTRICT,
    PRIMARY KEY (AnimalId, PlanetId)
)";
cmd.ExecuteNonQuery();
} // End using MySqlCommand cmd
} // End using MySqlConnection conn

Console.WriteLine("Created tables");
}

```

This code creates a database connection much as before, except this time it includes the database's name. It then opens the connection and uses it to create a command object.

Next, the code executes three `DROP TABLE` statements to delete the `AnimalPlanets`, `Animals`, and `Planets` tables if they already exist. Note, you must drop the tables in a valid order. In this database, the `AnimalPlanets` table has foreign key constraints matching fields in the `Animals` and `Planets` tables. If the `AnimalPlanets` table contains data and you drop either of the other tables first, then the `AnimalPlanets` records will violate their constraints and the program will crash.

DEADLY DROPS

The SQL `DROP TABLE` statement immediately destroys the table and all of its data with no do-overs or mulligans. Be sure you really want to discard all of the data *before* you execute this command.

Dropping tables (if they exist) and then re-creating them is a common technique for building test databases. It lets you remove any experimental changes to the tables so that you can start with a known state. (This is the main reason I didn't build the database by using HeidiSQL, which is described in the preceding chapter. If you build the database with a database management tool and then later decide to modify the table structure, you need to use the tool to rebuild the database's structure. It's often faster and easier to modify a program or script and rerun it to rebuild the database from scratch.)

Next, the code executes three commands to create the new tables. The C# code uses multiline string literals (beginning with the `@` character) for the command text, so the text contains embedded carriage returns, line feeds, and spaces. Fortunately, the database engine ignores whitespace so that causes no problems.

The third `CREATE TABLE` statement is a bit more complicated than the first two because it also defines foreign key constraints for the table. The first constraint requires that any value in the `AnimalPlanets.AnimalId` field must exist in some `Animals.AnimalId` field. The second constraint requires that any `AnimalPlanets.PlanetId` field must exist in some `Planets.PlanetId` field.

The `ON UPDATE RESTRICT` and `ON DELETE RESTRICT` clauses mean that the database will prevent (restrict) any updates or deletions that violate the constraints. Other choices can make a change cascade to related records or null out related fields.

This final `CREATE TABLE` statement also sets the table's primary key to be the combination of the `AnimalId` and `PlanetId` fields.

NOTE Like many database engines, MariaDB requires that primary key values be unique and not null.

Create Data

The following code shows the `CreateData` method:

```
// Create the data.
private static void CreateData()
{
    // Connect to the database server.
    string connectionString =
        "server=127.0.0.1;" +
        "uid=root;" +
```

```
        "pwd=TheSecretPassword;" +
        "database=AnimalData";
using (MySQLConnection conn = new MySqlConnection(connectString))
{
    // The connection must be open before you can use it.
    conn.Open();

    // Create a command.
    using (MySQLCommand cmd = conn.CreateCommand())
    {
        // Delete any previous records.
        cmd.CommandText = "DELETE FROM AnimalPlanets";
        cmd.ExecuteNonQuery();

        cmd.CommandText = "DELETE FROM Animals";
        cmd.ExecuteNonQuery();

        cmd.CommandText = "DELETE FROM Planets";
        cmd.ExecuteNonQuery();

        using (MySQLTransaction trans = conn.BeginTransaction())
        {
            cmd.Transaction = trans;

            // Add records to the Animals table.
            cmd.CommandText = @"INSERT INTO Animals
                (Size, Animal, AnimalId) VALUES
                ('Medium', 'Hermaflamingo', 1)";
            cmd.ExecuteNonQuery();

            cmd.CommandText = @"INSERT INTO Animals
                (Size, Animal, AnimalId) VALUES
                ('Large', 'Skunkopotamus', 2)";
            cmd.ExecuteNonQuery();

            cmd.CommandText = @"INSERT INTO Animals
                (Size, Animal, AnimalId) VALUES
                ('Medium', 'Mothalope', 3)";
            cmd.ExecuteNonQuery();

            cmd.CommandText = @"INSERT INTO Animals
                (Size, Animal, AnimalId) VALUES
                ('Small', 'Platypus', 4)";
            cmd.ExecuteNonQuery();

            // Add records to the Planets table.
            cmd.CommandText = @"INSERT INTO Planets
                (PlanetId, HomePlanet, PlanetaryMass) VALUES
                (@planet_id, @home_planet, @planetary_mass)";
            cmd.Prepare();

            cmd.Parameters.AddWithValue("@planet_id", 101);
            cmd.Parameters.AddWithValue("@home_planet", "Virgon 4");
```

```
cmd.Parameters.AddWithValue("@planetary_mass", 1.21);
cmd.ExecuteNonQuery();

cmd.Parameters[0].Value = 102;
cmd.Parameters[1].Value = "Dilbertopia";
cmd.Parameters[2].Value = 0.88;
cmd.ExecuteNonQuery();

cmd.Parameters[0].Value = 103;
cmd.Parameters[1].Value = "Xanth";
cmd.Parameters[2].Value = 0.01;
cmd.ExecuteNonQuery();

cmd.Parameters[0].Value = 104;
cmd.Parameters[1].Value = "Australia";
cmd.Parameters[2].Value = 1.0;
cmd.ExecuteNonQuery();

// Add records to the AnimalPlanets table.
cmd.CommandText = @"INSERT INTO AnimalPlanets
    (AnimalId, PlanetId) VALUES (@animal_id, @planet_id)";
cmd.Prepare();

cmd.Parameters.Clear();
cmd.Parameters.AddWithValue("@animal_id", 1);
cmd.Parameters.AddWithValue("@planet_id", 101);
cmd.ExecuteNonQuery();

cmd.Parameters.Clear();
cmd.Parameters.AddWithValue("@animal_id", 2);
cmd.Parameters.AddWithValue("@planet_id", 101);
cmd.ExecuteNonQuery();

cmd.Parameters.Clear();
cmd.Parameters.AddWithValue("@animal_id", 2);
cmd.Parameters.AddWithValue("@planet_id", 102);
cmd.ExecuteNonQuery();

cmd.Parameters.Clear();
cmd.Parameters.AddWithValue("@animal_id", 3);
cmd.Parameters.AddWithValue("@planet_id", 103);
cmd.ExecuteNonQuery();

cmd.Parameters.Clear();
cmd.Parameters.AddWithValue("@animal_id", 4);
cmd.Parameters.AddWithValue("@planet_id", 104);
cmd.ExecuteNonQuery();

// Commit the changes.
trans.Commit();
} // End using MySqlTransaction trans
} // End using MySqlCommand cmd
} // End using MySqlConnection conn

Console.WriteLine("Created data");
}
```

This code creates a database connection and defines a command as usual. It then deletes all the records from the database's three tables. Emptying a table before adding test records is another common testing technique.

DEADLY DELETIONS

The SQL statement `DELETE FROM Animals` deletes all the records in the table `Animals`. Usually deletions are more targeted. For example, you might use the statement `DELETE FROM Animals WHERE Animal='Platypus'` to delete only the platypus's record.

If you accidentally forget the `WHERE` clause, the database won't warn you before you delete every record in the table. Before you execute any `DELETE` statement, double-check to see if it has the correct `WHERE` clause.

Next, the code creates a transaction object so it can insert records within a transaction. When it is finished, it can commit or roll back those insertions.

Like many database objects, the transaction object has a `Dispose` method, so the code creates it inside a `using` statement so that it is automatically disposed when the `using` block ends.

Before it can use the command with the transaction, the code sets the command's `Transaction` property to the transaction object.

The program then demonstrates three techniques for creating new records. First, it sets the command object's `CommandText` property to an SQL `INSERT` statement that includes the values that will be inserted. It then calls the command's `ExecuteNonQuery` method to perform the insertion. The code repeats this sequence a few times to add records to the `Animals` table.

For the second insertion technique, the code sets the command's text to an `INSERT` statement where the values are replaced by named placeholders that start with an `@` symbol. It then calls the command's `Prepare` method to make the database compile the command for faster execution.

To insert a record, the program must use the command's `Parameters` collection. The code first uses the collection's `AddWithValue` method to add parameters to the collection specifying their names and values. After it has added the values, the program calls the command's `ExecuteNonQuery` method to add the record.

Now that the parameters have been created, the code replaces their values with the values for a new record and calls `ExecuteNonQuery` again. The program repeats that step to create the other `Planets` records.

The program uses the third technique to insert records into the `AnimalPlanets` table. It sets the command text as before and prepares the command. For each record, it then clears the `Parameters` collection, uses its `AddWithValue` method to create parameters, and calls `ExecuteNonQuery` to create the record.

After it has created all of the new records, the code calls the transaction object's `Commit` method to make the new records permanent.

Fetch Data

The following `PrintReader` method displays a query's results:

```
// Display a query's results.
private static void PrintReader(MySqlDataReader reader)
{
    while (reader.Read())
    {
        Console.Write(reader[0].ToString());
        for (int i = 1; i < reader.FieldCount; i++)
        {
            Console.Write(", " + reader[i].ToString());
        }
        Console.WriteLine();
    }
    reader.Close();
}
```

This method loops through a `MySqlDataReader` object's results. For each returned row, the code displays the first field's value. It then loops through the remaining values, displaying each with a comma in front.

The code starts a new line after each row and closes the reader after it has displayed all the rows. (You cannot change a command object's text if there is an open reader associated with that command, so the code closes the reader.)

The following code shows the `FetchData` method, which uses the `PrintReader` method to check the database's contents:

```
// Fetch the data.
private static void FetchData()
{
    // Connect to the database server.
    string connectionString =
        "server=127.0.0.1;" +
        "uid=root;" +
        "pwd=TheSecretPassword;" +
        "database=AnimalData";
    using (MySqlConnection conn = new MySqlConnection(connectionString))
    {
        // The connection must be open before you can use it.
        conn.Open();

        // Create a command.
        using (MySqlCommand cmd = conn.CreateCommand())
        {
            // Select each table's records.
            Console.WriteLine("\n*** Animals ***");
            cmd.CommandText = "SELECT * FROM Animals";
            PrintReader(cmd.ExecuteReader());

            Console.WriteLine("\n*** Planets ***");
            cmd.CommandText = "SELECT * FROM Planets";
        }
    }
}
```



```

        PrintReader(cmd.ExecuteReader());

        Console.WriteLine("\n*** AnimalPlanets ***");
        cmd.CommandText = "SELECT * FROM AnimalPlanets";
        PrintReader(cmd.ExecuteReader());

        // Select matching records.
        Console.WriteLine("\n*** Results ***");
        cmd.CommandText = @"SELECT Size, Animal, HomePlanet, PlanetaryMass
            FROM Animals, Planets, AnimalPlanets
            WHERE
                Animals.AnimalId = AnimalPlanets.AnimalId AND
                Planets.PlanetId = AnimalPlanets.PlanetId
            ORDER BY Animal";
        PrintReader(cmd.ExecuteReader());
    } // End using MySqlCommand cmd
} // End using MySqlConnection conn
}

```

This code connects to the database and creates a command object as usual. It then uses the command to execute three queries that fetch the data from the tables. It calls the command's `ExecuteReader` method and passes the result to the `PrintReader` method to display the results.

The code then uses the same steps to execute a more complicated query that selects matching records from all three tables.

The following text shows the results:

```

*** Animals ***
Medium, Hermaflamingo, 1
Large, Skunkopotamus, 2
Medium, Mothalope, 3
Small, Platypus, 4

*** Planets ***
101, Virgon 4, 1.21
102, Dilbertopia, 0.88
103, Xanth, 0.01
104, Australia, 1

*** AnimalPlanets ***
1, 101
2, 101
2, 102
3, 103
4, 104

*** Results ***
Medium, Hermaflamingo, Virgon 4, 1.21
Medium, Mothalope, Xanth, 0.01
Small, Platypus, Australia, 1
Large, Skunkopotamus, Virgon 4, 1.21
Large, Skunkopotamus, Dilbertopia, 0.88

```

SUMMARY

MariaDB is a direct descendant of the extremely popular MySQL database. It is a standard relational database that also provides column-oriented features, making it useful for data warehousing and large-scale data analysis and analytics.

The example described in this chapter creates a new database, defines some tables for it, and adds a little test data to those tables. This sort of application is useful for getting the database up and running, but in a real application you'd probably also need a user interface to let the user query and modify the data.

The next chapter describes an example that uses Python and PostgreSQL to build another relational database. Before you move on to Chapter 18, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

EXERCISES

1. What is the command object's method that executes statements that do not return records such as `DROP DATABASE` and `CREATE TABLE`? What method executes a query and lets you loop through multiple rows of results?
2. If you try to delete a database or table that does not exist, the database throws an exception. How could you modify the statement `DROP TABLE AnimalPlanets` to make the database ignore the statement if there is currently no `AnimalPlanets` table?
3. What is the main advantage of a database management tool over using SQL to build a database?
4. What is the main disadvantage of a database management tool over using SQL to build a database?
5. Add a new `LetUserCreateRecords` method to the example program that does the following:
 - a. Connect to the database and create a command as usual.
 - b. Enter an infinite loop that does the following:
 - i. Prompt the user for an animal ID. If the user enters a blank string, break out of the loop. Otherwise, parse the value as an integer. If the user enters a non-integer, display an error message and continue the loop.
 - ii. Prompt the user for a planet ID. If the user enters a blank string, break out of the loop. Otherwise, parse the value as an integer. If the user enters a non-integer, display an error message and continue the loop.
 - iii. Build and execute a SQL command that uses the values to create a new record in the `AnimalPlanets` table. If something goes wrong, display an error message and continue the loop.

- c. Add code near the end of the `main` method that calls the `LetUserCreateRecords` method. After that method returns, call `FetchData` again to display the updated data.
 - d. Run the program and create a new record (1, 104). Test the program by entering non-integers, huge integers, integers that are not present in the tables, and blank values.
 - 6. What exceptions can you encounter when the cell in Exercise 5 tries to add a new `AnimalPlanets` record?
-

18

PostgreSQL in Python

This example uses the very popular PostgreSQL relational database. If you compare this example to the one described in Chapter 16, “MariaDB in Python,” you’ll see many differences in detail. Those are due mostly to the approaches taken by the two database adapters used by the different examples. This example also demonstrates some new database techniques, such as making the database automatically generate ID values and easily creating many records all at once.

PostgreSQL (pronounced post-gres-kyoo-el) is also known as Postgres. (The nickname Postgres is particularly popular with those who find it hard to pronounce PostgreSQL. It’s also shorter and, therefore, gives you slightly fewer chances to make typos.) It’s a free open source relational database management system.

WHAT’S IN A NAME?

Postgres was originally named POSTGRES because it was a successor to the earlier database Ingres. Later, it was renamed PostgreSQL when SQL support was added. Since then, the name PostgreSQL and the nickname Postgres have stuck.

This example builds a small order database for a coffee supply store named the Brew Crew. Customers place orders that have order items. Figure 18.1 shows the design with some data added to the tables.

First, I’ll tell you where you can download and install Postgres. Then I’ll explain how to install the Psycopg data adapter for Python.

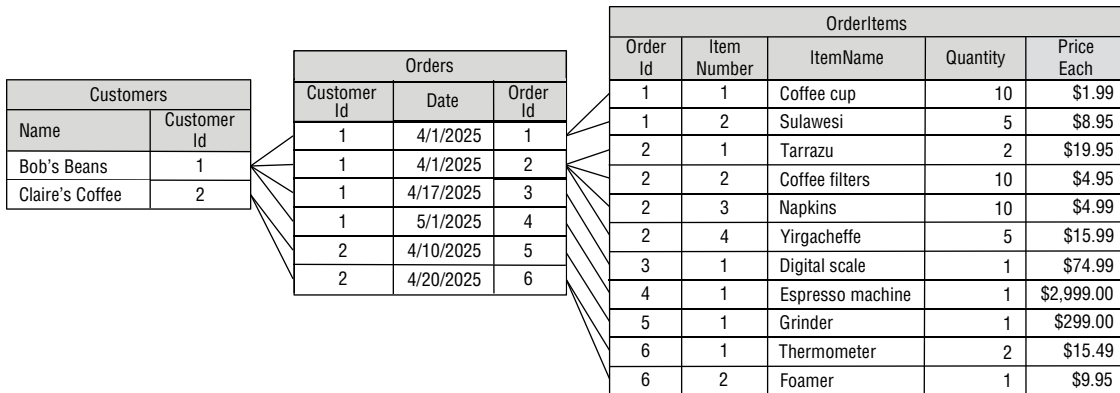


FIGURE 18.1

INSTALL PostgreSQL

Download Postgres from www.postgresql.org/download. You will need to click a link, download an installer, and run it. (It should only take a few minutes if you have a fast network connection.) Install the version that's appropriate for your computer. I installed postgresql-14.4-1-windows-x64 (Version 14.4-1 for 64-bit Windows x86).

When you install the database, use a good superuser password just as you used a good root password when you installed MariaDB in Chapter 16. (I used MyPassword. You should use something better.)

Before you can use the database, the engine must be running so that it can watch for requests sent to it by the database adapter. Normally, installing the database should also install a server daemon so that the database will be running when you need it.

To create the database, we'll use the database management program pgAdmin 4 that is installed along with Postgres. To start that program, look in the folder created by the installer and launch pgAdmin. (On my system, the folder is called `PostgreSQL 14`.)

LINUX TIP

Tech editor John Mueller suggests Linux users may prefer to build the database by using the command line as described at www.linode.com/docs/guides/use-postgresql-relational-databases-on-ubuntu-12-04.

As I mentioned in Chapter 16, "MariaDB in Python," and Chapter 17, "MariaDB in C#," I generally prefer to use scripts and code to create databases rather than using tools such as pgAdmin 4, but it's also useful to have some experience with these tools so that you know what they can do. If you prefer to work from the command line or in code, go ahead. If you haven't used this kind of tool before, you might consider it a character-building exercise.

This tool lets you do things such as building databases, creating tables, defining constraints, and examining the records in tables. It's similar to the HeidiSQL program mentioned in Chapter 16. We'll use only a tiny fraction of its capabilities later in this chapter.

RUN pgAdmin

Before you can start dumping data into tables, you need to do a little setup. First, you need to create a Postgres user. Then you need to create the database and define the tables in it. Only then can you start working with data.

We'll use the pgAdmin program for the first steps. We'll only use Python at the end to add and retrieve data.

Design the Database

The Brew Crew orders database contains three tables: customers, orders, and order_items. (We'll talk about the capitalization issue shortly.)

As you can probably guess, the customers table holds information about customers, the orders table holds information about orders, and the order_items table holds information about the items in the orders. Figure 18.2 shows the relational design.

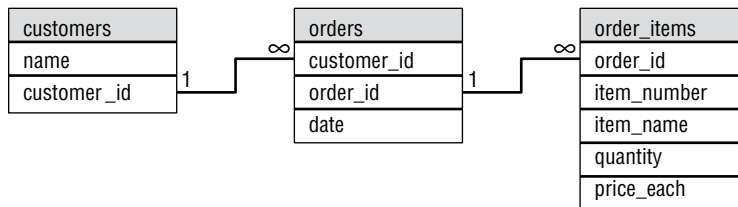


FIGURE 18.2

This isn't a very complete database and a real business would need a whole lot more, but it's enough for a meaningful example program.

Create a User

When you installed Postgres, it created a user named `postgres`. That user has superuser powers so it can do anything to any database. It's generally not a good idea to run your programs as a superuser just in case the program does something wrong like deleting an unrelated database, destroying other users, launching nuclear weapons, or whatever. It's better to work with a user who has the minimum powers necessary to get the job done. To do that, we'll create a new user named `brew_master`.

TIP Utilize users with the minimum possible permissions.

To create the user, launch pgAdmin. Expand the PostgreSQL branch, right-click Login/Group Roles, expand the Create submenu, and select Login/Group Role, as shown in Figure 18.3.

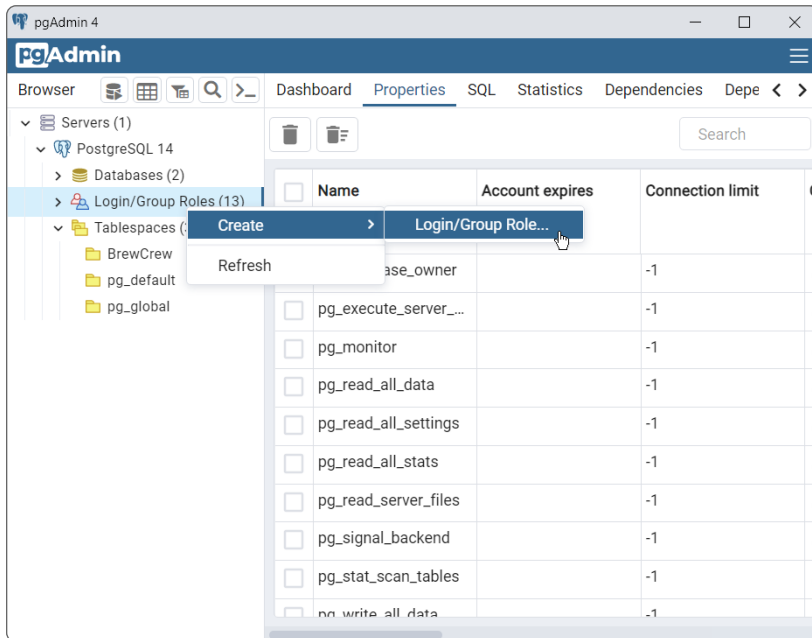


FIGURE 18.3

On the General tab, enter the name **brew_master**. Select the Definition tab and enter a password for the new user. For this example, I used the password **brew_password**. Click the Privileges tab and select Can Login. (Otherwise, the user won't be able to log in to the database. While that gives the database maximum protection, it's not very useful for the user.)

If you switch to the SQL tab, you'll see that pgAdmin uses the following SQL statement to create this user:

```
CREATE ROLE brew_master WITH
    LOGIN
    NOSUPERUSER
    NOCREATEDB
    NOCREATEROLE
    INHERIT
    NOREPLICATION
    CONNECTION LIMIT -1
    PASSWORD 'xxxxxxx';
```

After you've finished viewing the SQL code, click Save to create the user.

Create the Database

When you installed Postgres, it created a database named `postgres`. Normally, each application should work inside its own database so that it doesn't interfere with other applications. Therefore, our next step is to create a database for the Brew Crew project.

TIP Each application should use its own database.

To create the database, launch pgAdmin (if it's not already running), expand the PostgreSQL branch, right-click Databases, expand the Create submenu, and select Database, as shown in Figure 18.4.

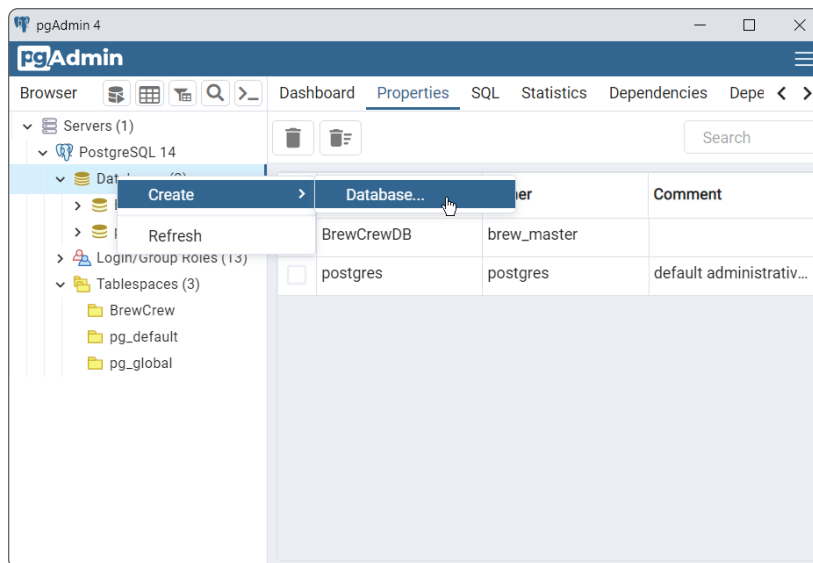


FIGURE 18.4

Set the database's name to **BrewCrewDB** and set its owner to **brew_master**, as shown in Figure 18.5.

If you switch to the SQL tab, you'll see that pgAdmin uses the following SQL statement to create this database:

```
CREATE DATABASE "BrewCrewDB"
WITH
OWNER = brew_master
ENCODING = 'UTF8'
CONNECTION LIMIT = -1;
```

After you've finished viewing the SQL code, click Save to create the database.

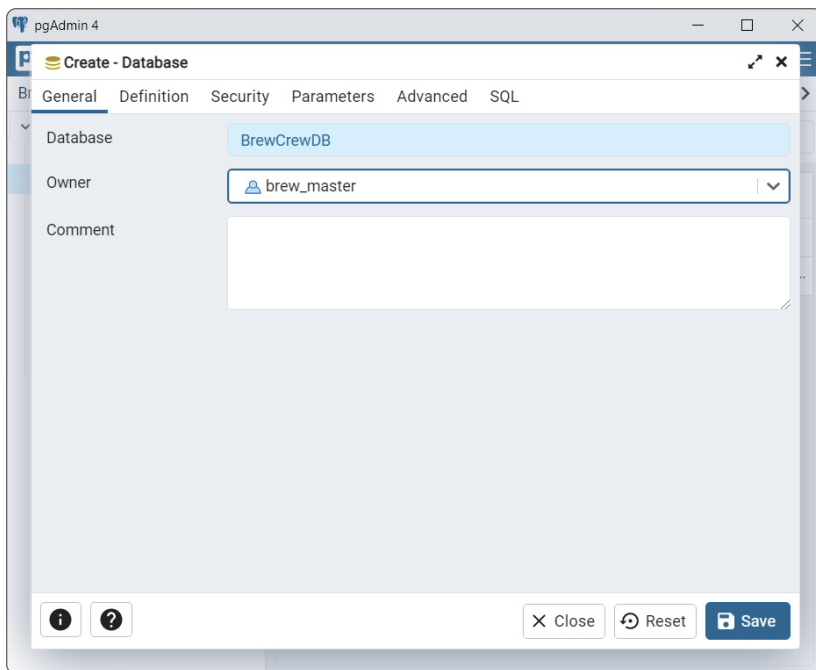


FIGURE 18.5

Define the Tables

You can create the tables in SQL code if you like in the same way we did in Chapter 16, but for this example we're going to use pgAdmin to define the tables interactively. If pgAdmin isn't already running, start it now.

Define the customers Table

Expand the Databases branch. Then expand BrewCrewDB, its Schemas branch, and the Public branch inside that. Next, right-click the Tables entry, open the Create submenu, and select Table, as shown in Figure 18.6.

On the General tab, set the table's name to **customers** and set its owner to **brew_master**.

Now, move to the Columns tab. Click the plus sign (+) to add a new row to the columns list. Set the column's name to **name** (this will hold the customer's name), set its data type to **text**, and select Not NULL.

Repeat those steps to create the `customer_id` field. Give it the type `integer` and select both Not NULL and Primary Key.

To make the database automatically generate `customer_id` values, click the pencil to the left of that column. In the editing area that appears, go to the Constraints tab and select the Identity option, as shown in Figure 18.7.

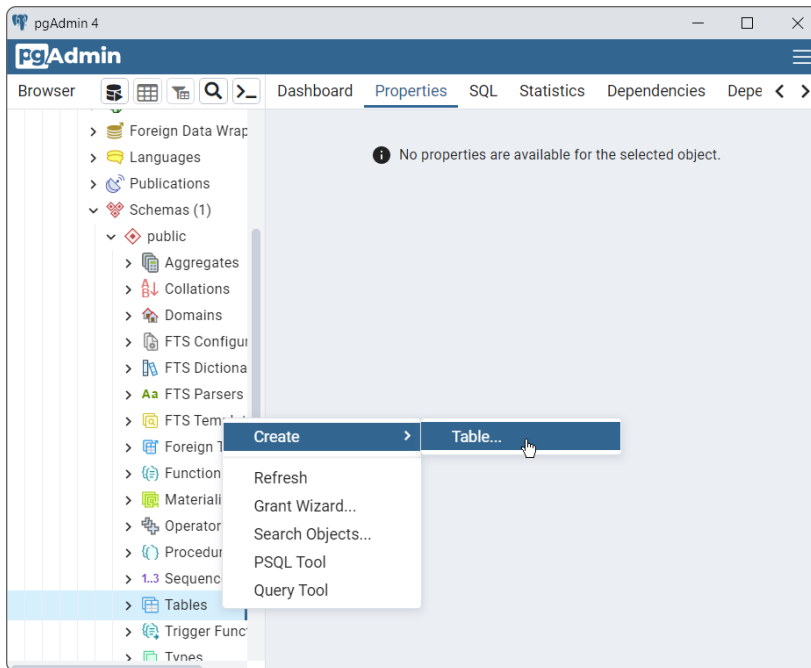


FIGURE 18.6

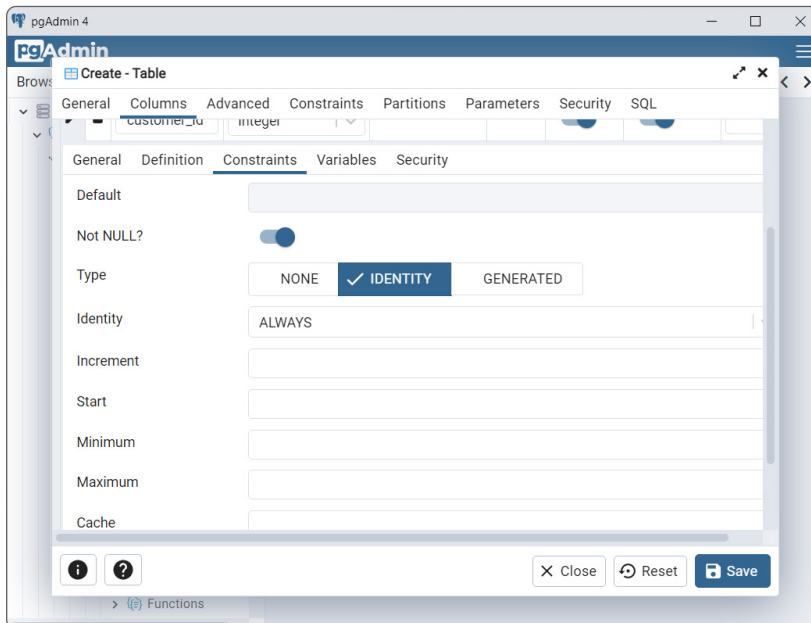


FIGURE 18.7

If you like, you can use the Increment, Start, Minimum, and Maximum fields to control how the IDs are generated. For example, you could have IDs start at 1337 and increment by 17 each time one is generated. (I don't know why you would do that, but who am I to judge?)

If you switch to the SQL tab, you'll see that pgAdmin uses the following SQL statements to create this table:

```
CREATE TABLE public.customers
(
    name text NOT NULL,
    customer_id integer NOT NULL GENERATED ALWAYS AS IDENTITY,
    PRIMARY KEY (customer_id)
);

ALTER TABLE IF EXISTS public.customers
    OWNER to brew_master;
```

After you've finished viewing the SQL code, click Save to create the table.

WHAT'S WITH THE LOWERCASE?

You've probably noticed that the names I'm using for the tables and columns in this chapter are all lowercase. This is due to the way Postgres attempts to be case-neutral. Whenever the program interacts with the database, Postgres converts table and column names into lowercase. For example, it converts the statement `SELECT * FROM Customers` into `SELECT * FROM customers`.

That's fine as long as the table and column names *really are* lowercase. For example, if you use pgAdmin to define the Customers table, then later your program will have conniptions saying that the customers table does not exist.

You can fix that by including quotes around the table and column names and using their correct capitalization, but it's all a lot easier if we just define table and column names in lowercase in the beginning. Then you can use whatever capitalization you like in your queries, Postgres will convert the names to lowercase, and everything is copacetic.

Define the orders Table

Repeat the preceding steps to create the orders table using the design shown in Figure 18.2 as a guide. This should be mostly straightforward, but here are some hints:

- Be sure to make the table's owner `brew_master`.
- Make `customer_id` an integer with Not NULL.
- Make `order_id` an integer with Not NULL, Primary Key, and the IDENTITY.
- Give the date column the `date` data type and make it Not NULL.

After you define the columns, you need to define the foreign key constraint that matches `customers.customer_id` with `orders.customer_id`. To do that, click on the `orders` table's Constraints tab, and then below that select the Foreign Key tab.

Now, click the plus sign (+) to create a new row in the table. Name the key `fk_customer_id` and click the pencil to the left to edit it.

In the editing area, click the Columns tab. In the Local Column field, select `customer_id`. In the References field, select the foreign table `public.customers`. Finally, in the Referencing field, select the foreign table's field `customer_id`.

After you've defined the fields, click Add to add the constraint to the table. Note that you could add other columns if the constraint needed to constrain multiple columns.

You can look at the other tabs if you like. For example, the Action tab lets you specify the On Update and On Delete actions for the foreign key.

If you switch to the SQL tab, you'll see that pgAdmin uses the following SQL statements to create this table:

```
CREATE TABLE public.orders
(
    customer_id integer NOT NULL,
    order_id integer NOT NULL GENERATED ALWAYS AS IDENTITY,
    date date NOT NULL,
    PRIMARY KEY (order_id),
    CONSTRAINT fk_customer_id FOREIGN KEY (customer_id)
        REFERENCES public.customers (customer_id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
        NOT VALID
);

ALTER TABLE IF EXISTS public.orders
    OWNER to brew_master;
```

After you've finished viewing the SQL code, click Save to create the table.

Define the `order_items` Table

Hang in there, we're almost done defining the tables. Repeat the preceding steps to create the `order_items` table, again using the design shown in Figure 18.2 as a guide. As before, this should be mostly straightforward, but here are some hints:

- Be sure to make the table's owner `brew_master`.
- Make all columns Not NULL.
- Set Primary Key for both of the columns `order_id` and `item_number` to make both columns part of the primary key. Making them the primary key means the values in those columns must be unique within the table. For example, there can be several records with `order_id` 1096, but the `item_number` values must be different. (Typically, the `item_number` values will be 1, 2, 3, and so forth. That differentiates them and gives each a position in the order.)
- Give the `price_each` column the `numeric` data type.

The `price_each` column should really have the `money` data type because that type uses special rules for rounding currency values. Unfortunately that type is deprecated in Postgres, so we use the `numeric` type instead. Some database engines support the `money` type and some do not.

After you define the columns, make a foreign key constraint matching `order_items.order_id` with `orders.order_id`.

If you switch to the SQL tab, you'll see that pgAdmin uses the following SQL statements to create this table:

```
CREATE TABLE public.order_items
(
    order_id integer NOT NULL,
    item_number integer NOT NULL,
    item_name text NOT NULL,
    quantity integer NOT NULL,
    price_each money NOT NULL,
    PRIMARY KEY (item_number, order_id),
    CONSTRAINT fk_order_id FOREIGN KEY (order_id)
        REFERENCES public.orders (order_id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
        NOT VALID
);

ALTER TABLE IF EXISTS public.order_items
    OWNER to brew_master;
```

After you've finished viewing the SQL code, click Save to create the table.

At this point, the database is created and ready to go, so it's time to write our Python program to add and fetch data. Before we can do that, however, we need to install the Psycopg database adapter.

CREATE THE PROGRAM

To create a Python program to work with the Brew Crew database, create a new Jupyter Notebook, and then add the code described in the following sections.

Unlike the program described in Chapter 16, this example's cells do not each connect to the database and perform a stand-alone action. In this example, the first cell (after installing Psycopg) creates a database connection, and then the following cells use that connection to insert data into the database. After the data is inserted, the program closes the connection. The final cell then opens a new connection to fetch the data and verify that it was saved correctly.

If you like, you can join most of the cells together into one long cell. I've used several cells to make it easier to understand and to debug the pieces separately.

Install Psycopg

Psycopg is a PostgreSQL database adapter for Python. For general information about Psycopg, see www.psycopg.org/docs.

If you're experienced with pip, then you can install Psycopg with the following command:

```
$ pip install psycopg2-binary
```

If you're not familiar with pip, then you can install Psycopg by executing the following command in a Jupyter Notebook cell:

```
!pip install psycopg2-binary
```

That's all there is to it!

A PSYCHO BY ANY OTHER NAME

It's kind of hard to remember the name Psycopg, mostly because it's hard to pronounce. So, where did it come from? The "pg" stands for Postgres and the rest was supposed to be "psycho." Federico Di Gregorio, who wrote the first version, says:

I wanted to call it psychopg (a reference to their psychotic driver) but I typed the name wrong.

You can see the longer version of the story at www.postgresql.org/message-id/36cffb61-3912-915c-4933-3bcd9cac063a%40dndg.it. (Or you can use the shortened link to the same article courtesy of TinyURL: <https://tinyurl.com/bd hx5eat>.)

Now that you know the story, you can pronounce it *psycho-pee-gee* and it's a lot easier to remember.

Now that the database is ready and we've installed Psycopg, it's time to work on the code that adds and fetches data.

Connect to the Database

The first cell that's actually part of the program (after the one that installs Psycopg) uses the following code to connect to the database:

```
# Prepare to use psycopg2.
import psycopg2

# Connect to the database.
conn = psycopg2.connect(
    database="BrewCrewDB",
```

```
user = "brew_master",
password = "brew_password",
host = "127.0.0.1", port = "5432")
```

This code simply imports the `psycopg2` library and then opens the connection. The program uses the connection to control the database.

Delete Old Data

The next cell executes the following code to delete any existing data in the database:

```
# Delete any existing data.
cursor = conn.cursor()
cursor.execute("DELETE FROM order_items")
cursor.execute("DELETE FROM orders")
cursor.execute("DELETE FROM customers")
conn.commit()
```

This code first uses the connection object to create a cursor. It uses the cursor's `execute` method to run SQL `DELETE` commands that remove all data from the tables.

DANGEROUS DELETIONS, REDUX

The SQL statement `DELETE FROM <table_name>` deletes all the records in the table. Usually deletions are more targeted. For example, you might use the statement `DELETE FROM order_items WHERE OrdId=1337` to delete the records associated with a particular order.

If you accidentally forget the `WHERE` clause, the database won't warn you before you delete every record in table. Before you execute any `DELETE` statement, double-check to see if it has the correct `WHERE` clause.

After it deletes the records, the code calls the connection object's `commit` method to make the deletions permanent.

Create Customer Data

The next cell uses the following code to create some records in the customers table:

```
# Create some customers records.
cmd = """INSERT INTO customers (name)
VALUES ('Bob\'\'\'s Beans')
RETURNING customer_id"""
print(cmd)
cursor.execute(cmd)
bobs_id = cursor.fetchone()[0]
```



```

print(f"Bob's ID: {bobs_id}\n")

cmd = """INSERT INTO customers (name)
VALUES (%s)
RETURNING customer_id"""
print(cmd)
cursor.execute(cmd, ("Claire's Coffee",))
claires_id = cursor.fetchone()[0]
print(f"Claire's ID: {claires_id}\n")

conn.commit()

```

This code demonstrates two methods for creating new records. First, it composes a string that contains the following SQL `INSERT` statement:

```

INSERT INTO customers (name)
VALUES ('Bob\'\'\'s Beans')
RETURNING customer_id

```

Notice the awkward syntax used to put an apostrophe inside the text Bob's Beans. Each `\'` adds an apostrophe to the string. Because the apostrophe is inside part of the string that's surrounded by single quotes, SQL requires two apostrophes so that it knows this isn't the end of the quoted section. (Yes, it's confusing. The code that creates the record for Claire's Coffee is simpler.)

The statement's last clause is `RETURNING customer_id`. The `customers` table's `customer_id` field is autogenerated, so this clause tells the database to return the newly generated value.

After it composes the statement, the code uses the cursor to execute it. It then calls the cursor's `fetchone` method to retrieve the value that the command returns. That result is a tuple, so the code uses the index `[0]` to retrieve the first value, which is the newly generated `customer_id`. The code saves that value in the variable `bobs_id` for later use.

After it creates the Bob's Beans record, the code uses a slightly different method to create the entry for Claire's Coffee. The code composes an `INSERT` statement as before, but this time it does not include the text that should be inserted into the `name` field. Instead, it replaces that value with the placeholder `%s`.

When the code calls the cursor's `execute` method, it passes in the SQL statement and a tuple of parameters that should replace any placeholders. The `name` parameter is surrounded by double quotes, so you can include an apostrophe without using the strange syntax used for Bob's Beans.

There is one weirdness here, however. If you create a tuple containing a single item in Python, the program ignores the parentheses and just uses the single value as it is. To create a tuple containing one item, you need to follow that item with a comma. It's a little strange, but not as bad as the `\'\'` required to create the Bob's Beans record.

The code fetches the new record's autogenerated `customer_id` and saves it in the variable `claires_id` for later use.

The cell finishes by committing the insertions.

Create Order Data

The next cell uses the following code to demonstrate a third record-creation technique, this time to add records to the `orders` table:

```
import datetime

# Create some orders records.
# Prepare the INSERT command.
cmd = """PREPARE insert_order_plan AS
        INSERT INTO orders (customer_id, date)
        VALUES ($1, $2)
        RETURNING order_id"""
cursor.execute(cmd)

# Execute the INSERT command.
cmd = "EXECUTE insert_order_plan (%s, %s)"
bobs_order_ids = []
cursor.execute(cmd, (bobs_id, datetime.datetime(2025, 4, 1)))
bobs_order_ids.append(cursor.fetchone()[0])

cursor.execute(cmd, (bobs_id, datetime.datetime(2025, 4, 1)))
bobs_order_ids.append(cursor.fetchone()[0])

cursor.execute(cmd, (bobs_id, datetime.datetime(2025, 4, 17)))
bobs_order_ids.append(cursor.fetchone()[0])

cursor.execute(cmd, (bobs_id, datetime.datetime(2025, 5, 1)))
bobs_order_ids.append(cursor.fetchone()[0])

claires_order_ids = []
cursor.execute(cmd, (claires_id, datetime.datetime(2025, 4, 10)))
claires_order_ids.append(cursor.fetchone()[0])

cursor.execute(cmd, (claires_id, datetime.datetime(2025, 4, 20)))
claires_order_ids.append(cursor.fetchone()[0])

conn.commit()

print("Orders created")
```

This code creates a string holding a `PREPARE` statement. That statement prepares a plan for an `INSERT` statement that will create records in the `orders` table. This time the statement replaces its values with the placeholders `$1` and `$2` and returns the autogenerated `order_id` value.

After composing the `PREPARE` statement, the code executes it to make the database create an execution plan for the `INSERT` statement. Now the program can execute that statement more quickly than it could if it created a new `INSERT` statement for each record.

Next, the code creates an `EXECUTE` statement that executes the prepared plan. This time it replaces the values that the plan expects with the `%s` placeholder.

When the code executes the `EXECUTE` statement, it passes in the values that should be inserted in place of the placeholders. The `orders` table has three fields: `customer_id`, `date`, and `order_id`. The last of those is autogenerated, so the code only passes the `customer_id` and `date` to the `execute` method. Those values are inserted into the `EXECUTE` statement, which then inserts the values into the `INSERT` plan.

After it creates each record, the code uses the command object's `fetchone` method as before to save the newly generated `order_id` values in the lists `bobs_order_ids` and `claires_order_ids`.

The code uses this technique (which I admit takes longer to explain than to use) to create four orders for Bob's Beans and two records for Claire's Coffee. Notice how the code uses the customer IDs that were generated earlier rather than hardwiring the ID values into the code.

After it has created the order records, the code commits the insertions.

Create Order Item Data

The following code in the next cell uses one final technique for inserting records, this time in the `order_items` table. Actually this is really the third technique that uses a `PREPARE` statement again, but this time the program executes that statement in a loop:

```
# Create some order_items records.

# Define the data.
item_data = [
    (bobs_order_ids[0], 1, "Coffee cup", 10, 1.99),
    (bobs_order_ids[0], 2, "Sulawesi", 5, 8.95),
    (bobs_order_ids[1], 1, "Tarrazu", 2, 19.95),
    (bobs_order_ids[1], 2, "Coffee filters", 10, 4.95),
    (bobs_order_ids[1], 3, "Napkins", 10, 4.99),
    (bobs_order_ids[1], 4, "Yirgacheffe", 5, 15.99),
    (bobs_order_ids[2], 1, "Digital scale", 1, 74.99),
    (bobs_order_ids[3], 1, "Espresso machine", 1, 2999.00),
    (claires_order_ids[0], 1, "Grinder", 1, 299.00),
    (claires_order_ids[1], 1, "Thermometer", 2, 15.49),
    (claires_order_ids[1], 2, "Foamer", 1, 9.95),
]

# Prepare the INSERT command.
cmd = """PREPARE insert_order_item_plan AS
INSERT INTO order_items
    (order_id, item_number, item_name, quantity, price_each)
VALUES ($1, $2, $3, $4, $5)"""

cursor.execute(cmd)

# Insert orders data.
```

```
cmd = "EXECUTE insert_order_item_plan (%s, %s, %s, %s, %s)"

for order_data in item_data:
    print(order_data)
    cursor.execute(cmd, order_data)

conn.commit()
print("Order items created")
```

This code first creates a list of tuples named `item_data` holding the values for each record. It uses the order IDs saved in the lists `bobs_order_ids` and `claires_order_ids` for the order ID values.

Next, the code composes a `PREPARE` statement much as before and executes it. It creates an `EXECUTE` statement and then loops through the `item_data` list executing the `EXECUTE` statement for each of its tuples.

After it finishes inserting the new records, the code commits the changes.

Close the Connection

To prove that it created the records, the next cell uses the following code to close the cursor and database connection:

```
# Close the database connection.
cursor.close()
conn.close()
```

Perform Queries

The program's final cell uses the following code to fetch data from the database:

```
# Reconnect to the database.
conn = psycopg2.connect(
    database="BrewCrewDB",
    user = "brew_master",
    password = "brew_password",
    host = "127.0.0.1", port = "5432")
cursor = conn.cursor()

# Fetch data
cmd = """SELECT date, name, orders.order_id, item_number, item_name,
    quantity, price_each, quantity * price_each
FROM customers, orders, order_items
WHERE
    customers.customer_id = orders.customer_id AND
    orders.order_id = order_items.order_id
ORDER BY date, order_id, item_number"""
cursor.execute(cmd)
rows = cursor.fetchall()

last_order_id = -1
order_total = 0
```

```

for row in rows:
    if row[2] != last_order_id:
        if order_total > 0:
            print(f"    Total: ${order_total:8.2f}\n")
            order_total = 0

        print(f"{row[0]}, {row[1]}, Order ID: {row[2]}")
        last_order_id = row[2]

    item = f"    {row[3]}. {row[4]:20} {row[5]:3} @ "
    item = item + f"${row[6]:8.2f} = ${row[7]:8.2f}"
    print(item)
    order_total += row[7]

if order_total > 0:
    print(f"    Total: ${order_total:8.2f}\n")

cursor.close()
conn.close()

```

This code connects to the database and creates a cursor. It then composes a query to fetch information from the three tables using the `customer_id` and `order_id` values to link related records. It orders the results by date, order ID, and item number. Notice that the last field selected is a calculated value that holds the item's quantity times the item's price each.

After it composes the query, the code executes it and calls the cursor's `fetchall` method to retrieve the results. This result is a list of tuples where each tuple holds one row of results.

Before it processes the results, the code defines two variables. It uses the variable `last_order_id` to track the current order's ID. It initially sets that variable to `-1` so the first row will have a different ID.

The code uses the variable `order_total` to keep track of the current order's total cost. It initially sets this value to `0`.

Having initialized the `last_order_id` and `order_total` variables, the code loops through the returned list.

If the new row's order ID is different from the one stored in `last_order_id`, then this row starts a new order. If the order total is greater than `0`, the program prints it and resets `order_total` to `0` so that it can add up the costs of the items in the next order.

If we are starting a new order, the code also prints the new order's date, customer name, and order ID. It then saves the new order ID in the variable `last_order_id` so we'll know when we start the next one.

Next, whether this is the start of a new order or not, the code prints the row's item number, description, quantity, price each, and total price. It adds the total price to `order_total`.

The program continues looping through the returned rows until it has processed them all. When it has finished displaying all the rows, the program displays the order total for the last order.

The code finishes by closing the cursor and database connection.

The following shows this cell's output:

```
2027-04-01, Bob's Beans, Order ID: 645
  1. Coffee cup          10 @ $    1.99 = $   19.90
  2. Sulawesi            5 @ $    8.95 = $   44.75
  Total: $   64.65

2027-04-01, Bob's Beans, Order ID: 646
  1. Tarrazu             2 @ $   19.95 = $   39.90
  2. Coffee filters     10 @ $    4.95 = $   49.50
  3. Napkins            10 @ $    4.99 = $   49.90
  4. Yirgacheffe        5 @ $   15.99 = $   79.95
  Total: $  219.25

2027-04-10, Claire's Coffee, Order ID: 649
  1. Grinder             1 @ $  299.00 = $  299.00
  Total: $  299.00

2027-04-17, Bob's Beans, Order ID: 647
  1. Digital scale       1 @ $   74.99 = $   74.99
  Total: $   74.99

2027-04-20, Claire's Coffee, Order ID: 650
  1. Thermometer         2 @ $   15.49 = $   30.98
  2. Foamer              1 @ $    9.95 = $    9.95
  Total: $   40.93

2027-05-01, Bob's Beans, Order ID: 648
  1. Espresso machine    1 @ $ 2999.00 = $ 2999.00
  Total: $ 2999.00
```

SUMMARY

PostgreSQL is a popular relational database system that in many ways is similar to MariaDB. The main differences between this example and the MariaDB example are due to the ways that the two programs' database adapters work.

For this example, you used pgAdmin to create an orders database. The example program then adds some records to the database's tables and fetches the data, linking related records together to reproduce the customers' orders. In a real application, you would want to add other features to let the user interactively work with the database to create, read, and edit orders.

The next chapter describes a similar example that uses C# to build and manipulate a PostgreSQL database. Before you move on to Chapter 19, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

EXERCISES

1. Name the database adapter that this chapter uses. How was it supposed to be spelled?

2. Add a new notebook cell that allows the user to create a new order for one of the existing customers. It should do the following:
 - a. Prompt the user for a customer ID. (Be sure to enter a valid ID when you're testing.)
 - b. Prompt the user for an order date.
 - c. Enter an infinite loop that prompts for the order item information: item name, quantity, and price of each. It should keep track of the item number as the user enters items. When the user enters a blank line, break out of the loop.
 - d. Create the new records.
 - i. Connect to the database and create a cursor.
 - ii. Create the `orders` record and save the autogenerated order ID in a variable.
 - iii. Create the order items records.
 - e. Close the cursor and database connection.
 - f. Rerun the cell that fetches data to verify that the new order was created correctly.

Here are some hints:

-
- Don't worry about error handling. Assume that you'll enter valid values while testing.
 - Compose the SQL statements as strings containing the values that you're adding to the database.
 - It may help with debugging to print the SQL commands before executing them.
 - Put quotes around the `date` and `item_name` values in the SQL statements.
-
3. What happens if you enter **crème brûlée** as an item name? (If you can't type those characters, search online for **creme brulee**, and then copy and paste the words with the correct accents into the program's input box.)

 4. What happens if you enter **Amy's cookies** as an item name?

 5. Rewrite the loop that creates the `order_items` records so that it uses placeholders instead of putting the field values right inside the SQL statement. Test the revised code with an item named **Amy's cookies**. Hint: Build a tuple containing all of the record's values, including the order ID and item number. Then pass that tuple into the cursor's `execute` method.

 6. What happens if you enter the order date in the format `d/m/yyyy`? What happens if you enter the specific value `20/10/2027`?

 7. What kinds of database errors could the new order code encounter? (Ignore formatting errors such as the user entering "ten" for the customer ID or entering an invalid order date like `20/40/2100`.)
-

19

PostgreSQL in C#

This example demonstrates the PostgreSQL database in a C# application. If you skipped Chapter 18, which built a similar example in Python, go to that chapter and read the beginning and the first two main sections, which are described in the following list.

- “Install PostgreSQL” explains how to install the PostgreSQL database.
- “Run pgAdmin” tells you how to use the pgAdmin tool to build the database that this example uses.

When you reach the section “Create the Program” in Chapter 18, come back here and read the following sections.

CREATE THE PROGRAM

To create a C# program to work with the Brew Crew database, create a new C# Console App (.NET Framework) and then add the code described in the following sections.

Like the program described in Chapter 17, “MariaDB in C#,” this example’s `main` method calls other methods to do the actual work. This time the `main` method creates the database connection and passes it to the other methods so that they don’t need to open their own connections.

Install Npgsql

This example uses the Npgsql data provider to allow your program to communicate with the Postgres engine. To add Npgsql to your project, follow these steps.

1. Open the Project menu and select Manage NuGet Packages.
2. Click the Browse tab and in the search box enter **Npgsql**. This will bring up around 200 packages.
3. Find the package named Npgsql, click it, and then use the Install button on the right to install it. This shouldn’t take too long, but you might still want a fast Internet connection.

If you later need to remove or update a package, open the NuGet Package Manager again and look at the Installed tab. After you select an installed package, you can use the Uninstall or Update button on the right.

Connect to the Database

The following code shows the example program's main method:

```
static void Main(string[] args)
{
    string connectString =
        @"Host=localhost;Username=brew_master;
        Password=brew_password;Database=BrewCrewDB";

    using (NpgsqlConnection conn = new NpgsqlConnection(connectString))
    {
        conn.Open();

        // Delete any existing records.
        DeleteRecords(conn);

        // Create customer records.
        int bobsId, clairesId;
        CreateCustomerRecords(conn, out bobsId, out clairesId);

        // Create order records.
        List<int> bobsOrderIds, clairesOrderIds;
        CreateOrderRecords(conn,
            bobsId, clairesId,
            out bobsOrderIds, out clairesOrderIds);

        // Create order_items records.
        CreateOrderItemRecords(conn, bobsOrderIds, clairesOrderIds);

        // Display some records.
        DisplayOrders(conn);

        // Let the user create an order.
        LetUserCreateOrder(conn);

        // Display some records.
        DisplayOrders(conn);
    }

    Console.WriteLine("Press Enter to quit ");
    Console.ReadLine();
}
```

This code creates a connect string to identify the database's server computer, username, password, and database. The main method uses that string to create a database connection, opens the connection, and then passes it to other methods to do the real work.

LOUDY WITH A CHANCE OF DATA

This example uses a connect string where the host is localhost, which represents your local system. If the database's server is located on a different computer, you would specify the server's computer as either an IP address (as in 8.8.8.8, which is the primary DNS server for Google DNS) or an Internet domain (as in `whitehouse.gov`).

This lets the program access the database over a network, so it's sort of "cloudy," but it's not really the cloud unless the server is hosted on a cloud provider.

Delete Old Data

The `DeleteRecords` method uses the following code to delete any old records that are gathering dust in the database:

```
// Delete the records to start from scratch.
static private void DeleteRecords(NpgsqlConnection conn)
{
    string deleteStatement = "DELETE FROM order_items";
    using (NpgsqlCommand deleteCmd = new NpgsqlCommand(deleteStatement, conn))
    {
        deleteCmd.ExecuteNonQuery();
    }

    deleteStatement = "DELETE FROM orders";
    using (NpgsqlCommand deleteCmd = new NpgsqlCommand(deleteStatement, conn))
    {
        deleteCmd.ExecuteNonQuery();
    }

    deleteStatement = "DELETE FROM customers";
    using (NpgsqlCommand deleteCmd = new NpgsqlCommand(deleteStatement, conn))
    {
        deleteCmd.ExecuteNonQuery();
    }
}
```

This code composes a SQL `DELETE` statement that removes the records from the `order_items` table. It then uses that statement and the database connection to create an `NpgsqlCommand` object. It then calls the command object's `ExecuteNonQuery` method to execute the statement. As I'm sure you can guess, it uses `ExecuteNonQuery` because this SQL statement is not a query.

The code repeats those steps to create two other command objects that delete records from the `orders` and `customers` tables.

DANGEROUS DELETIONS, REDUX

The SQL statement `DELETE FROM <table_name>` deletes all the records in the table. Usually deletions are more targeted. For example, you might use the statement `DELETE FROM order_items WHERE OrderId=1337` to delete the records associated with a particular order.

If you accidentally forget the `WHERE` clause, the database won't warn you before you delete every record in the table. Before you execute any `DELETE` statement, double-check to see if it has the correct `WHERE` clause.

Create Customer Data

The `CreateCustomerRecords` method uses the following code to add two records to the customers table:

```
static private void CreateCustomerRecords(NpgsqlConnection conn,
    out int bobsId, out int clairesId)
{
    using (NpgsqlTransaction transaction = conn.BeginTransaction())
    {
        string createCustomerStatement =
            @"INSERT INTO Customers (name)
            VALUES (@name) RETURNING customer_id";
        using (NpgsqlCommand insertCmd =
            new NpgsqlCommand(createCustomerStatement, conn))
        {
            insertCmd.Transaction = transaction;

            // Bob's Beans.
            Npgsql.NpgsqlParameter nameParameter =
                insertCmd.Parameters.AddWithValue("name", "Bob's Beans");
            bobsId = (int)insertCmd.ExecuteScalar();
            Console.WriteLine($"Bob's ID: {bobsId}");

            // Claire's Coffee.
            nameParameter.Value = "Claire's Coffee";
            clairesId = (int)insertCmd.ExecuteScalar();
            Console.WriteLine($"Claire's ID: {clairesId}");
        }

        transaction.Commit();
    }
}
```

This code first uses the connection object to create a transaction so it can commit or roll back actions.

It then composes a SQL `INSERT` statement that uses the placeholder `@name` for the customer's name value. It uses the database connection and the SQL statement to create a new command object and then sets the command's `Transaction` property to the transaction object that it created earlier. That allows the transaction to commit or roll back actions performed by this command object.

Next, the code creates an `NpgsqlParameter` object to represent a parameter for the SQL statement. It sets the parameter's name to `name` and sets its value to `Bob's Beans`.

The code then calls the command object's `ExecuteScalar` method. That method executes a SQL statement that returns a single value. (In the database context, *scalar* means in a single value.)

In this example, the statement's `RETURNING` clause makes the command return the newly created customer ID value that is automatically generated by the database when the record is created. The code casts the returned result into an integer and stores it in the variable `bobsId` for later use.

Next, the program changes the parameter object's `Value` property to `Claire's Coffee`, executes the command, and saves the result in variable `claireId`.

After it finishes creating the `customers` records, the code calls the transaction object's `Commit` method to make the changes permanent. If you call the `Rollback` method or if you forget to call `Commit` before the transaction object goes out of scope and is destroyed, then the actions are undone.

TRANSITORY TRANSACTIONS

If you don't use transactions, then every command takes immediate effect with no undo option. The `CreateCustomerRecords` method uses a transaction so that you can see how it's done, but the rest of the program doesn't bother. There's no point because it won't ever roll back a transaction.

Transactions are more important for sets of actions that should either all happen or all not happen. For example, if the program transfers money from one bank account to another, you don't want the program to crash halfway through after taking the money from one account but before putting it in the other.

For another example (which foreshadows Exercise 19.2), suppose you're creating a new order record that has associated order items. If you create the order record and then the order item records fail, you'll have an orphaned order record sitting around with no items in it. To prevent that, you don't want to create any of those records unless you successfully create them all.

Create Order Data

The `CreateOrderRecords` method uses the following code to create order records:

```
static private void CreateOrderRecords(NpgsqlConnection conn,
    int bobsId, int clairesId,
    out List<int> bobsOrderIds, out List<int> clairesOrderIds)
{
    // Make lists to hold new order IDs.
    bobsOrderIds = new List<int>();
    clairesOrderIds = new List<int>();

    // Make the command.
    string createOrderStatement =
```

```

        @"INSERT INTO Orders (customer_id, date)
          VALUES (@customer_id, @date) RETURNING order_id";

using (NpgsqlCommand insertCmd =
    new NpgsqlCommand(createOrderStatement, conn))
{
    // Add the parameters.
    NpgsqlParameter customerIdParameter =
        insertCmd.Parameters.Add("customer_id",
            NpgsqlTypes.NpgsqlDbType.Integer);
    NpgsqlParameter dateParameter =
        insertCmd.Parameters.Add("date", NpgsqlTypes.NpgsqlDbType.Date);

    // Prepare the command.
    insertCmd.Prepare();

    // Create Bob's records.
    customerIdParameter.Value = bobsId;
    dateParameter.Value = new DateTime(2027, 4, 1);
    bobsOrderIds.Add((int)insertCmd.ExecuteScalar());

    dateParameter.Value = new DateTime(2027, 4, 1);
    bobsOrderIds.Add((int)insertCmd.ExecuteScalar());

    dateParameter.Value = new DateTime(2027, 4, 17);
    bobsOrderIds.Add((int)insertCmd.ExecuteScalar());

    dateParameter.Value = new DateTime(2027, 5, 1);
    bobsOrderIds.Add((int)insertCmd.ExecuteScalar());

    // Create Claire's records.
    customerIdParameter.Value = clairesId;
    dateParameter.Value = new DateTime(2027, 4, 10);
    clairesOrderIds.Add((int)insertCmd.ExecuteScalar());

    dateParameter.Value = new DateTime(2027, 4, 20);
    clairesOrderIds.Add((int)insertCmd.ExecuteScalar());
}
}

```

This method creates two lists of integers (named `bobsOrderIds` and `claireOrderIds`) to hold the newly created order IDs.

It then composes a SQL `INSERT` statement that adds a record to the orders table. The statement represents the values with the placeholders `@customer_id` and `@date`. The `RETURNING` clause indicates that the statement should return the newly created order ID.

Next, the code uses the database connection and the SQL statement to create a command object. It then creates two parameter objects for the command to hold the `customer_id` and `date` values.

The program then calls the command object's `Prepare` method to ready the command for action.

The code then sets the `customer_id` and `date` parameter values for the first order. It then calls the command's `ExecuteScalar` method, converts the returned value (which is the newly minted order ID) into an integer, and saves the result in the `bobsOrderIds` list.

Next, the code sets the date parameter to a new date and inserts another record, again saving the order ID in the `bobsOrderIds` list. It repeats those steps two more times to create two more orders for Bob's Beans.

The method then sets the command's customer ID parameter to Claire's Coffee's ID. It then uses the same steps it used earlier to create two orders for Claire's Coffee.

Create Order Item Data

The `CreateOrderItemRecords` method uses the following code to add order items to the orders that the `CreateOrderRecords` method created:

```
static private void CreateOrderItemRecords(NpgsqlConnection conn,
    List<int> bobsOrderIds, List<int> clairesOrderIds)
{
    // Define the data.
    object[] orderItemsData =
    {
        new object[] { bobsOrderIds[0], 1, "Coffee cup", 10, 1.99 },
        new object[] { bobsOrderIds[0], 2, "Sulawesi", 5, 8.95},
        new object[] { bobsOrderIds[1], 1, "Tarrazu", 2, 19.95},
        new object[] { bobsOrderIds[1], 2, "Coffee filters", 10, 4.95},
        new object[] { bobsOrderIds[1], 3, "Napkins", 10, 4.99},
        new object[] { bobsOrderIds[1], 4, "Yirgacheffe", 5, 15.99},
        new object[] { bobsOrderIds[2], 1, "Digital scale", 1, 74.99},
        new object[] { bobsOrderIds[3], 1, "Espresso machine", 1, 2999.00},
        new object[] { clairesOrderIds[0], 1, "Grinder", 1, 299.00},
        new object[] { clairesOrderIds[1], 1, "Thermometer", 2, 15.49},
        new object[] { clairesOrderIds[1], 2, "Foamer", 1, 9.95},
    };

    // Make the command.
    string createItemStatement =
        @"INSERT INTO order_items
        (order_id, item_number, item_name, quantity, price_each)
        VALUES
        (@order_id, @item_number, @item_name, @quantity, @price_each)";
    using (NpgsqlCommand insertCmd =
        new NpgsqlCommand(createItemStatement, conn))
    {
        // Add the parameters.
        NpgsqlParameter orderIdParameter =
            insertCmd.Parameters.Add("order_id",
                NpgsqlTypes.NpgsqlDbType.Integer);
        NpgsqlParameter itemNumberParameter =
            insertCmd.Parameters.Add("item_number",
                NpgsqlTypes.NpgsqlDbType.Integer);
        NpgsqlParameter itemNameParameter =
            insertCmd.Parameters.Add("item_name",
                NpgsqlTypes.NpgsqlDbType.Text);
        NpgsqlParameter quantityParameter =
            insertCmd.Parameters.Add("quantity",
                NpgsqlTypes.NpgsqlDbType.Integer);
    }
}
```

```

        NpgsqlParameter priceEachParameter =
            insertCmd.Parameters.Add("price_each",
                NpgsqlTypes.NpgsqlDbType.Numeric);

        // Prepare the command.
        insertCmd.Prepare();

        // Create the records.
        foreach (object[] row in orderItemsData)
        {
            // Create this row.
            orderIdParameter.Value = row[0];
            itemNumberParameter.Value = row[1];
            itemNameParameter.Value = row[2];
            quantityParameter.Value = row[3];
            priceEachParameter.Value = row[4];
            insertCmd.ExecuteNonQuery();
        }
    }
}

```

This method uses the same basic approach used by the `CreateOrderRecords` method except it uses a loop to add records with data stored in an array.

The method first creates an array of objects, each of which is an array holding one record's values. Having defined the records' data, the code method composes a SQL `INSERT` statement with placeholders for its values as in earlier pieces of code. It uses the statement to create a command object, adds parameters to that object, and calls its `Prepare` method all as before.

Now the code loops through the record data array. For each row in the array, the program copies the row's field values into the appropriate parameters and then calls the command's `ExecuteNonQuery` method.

Logically, this approach is similar to the one used by the `CreateCustomerRecords` and `CreateOrderRecords` methods. They all create a command, give it some parameters, prepare it, fill in parameter values, and execute the command. Storing the data in an array and looping may make the code easier to read, particularly if you need to create many records.

Display Orders

The `DisplayOrders` method uses the following code to display the orders and their items:

```

// Display the orders.
static private void DisplayOrders(NpgsqlConnection conn)
{
    string customersQuery =
        @"SELECT date, name, orders.order_id, item_number, item_name,
           quantity, price_each, quantity * price_each
        FROM customers, orders, order_items
        WHERE
           customers.customer_id = orders.customer_id AND
           orders.order_id = order_items.order_id
        ORDER BY date, order_id, item_number";
}

```



```

using (NpgsqlCommand cmd =
    new NpgsqlCommand(customersQuery, conn))
{
    int lastOrderId = -1;
    double orderTotal = 0;
    NpgsqlDataReader reader = cmd.ExecuteReader();
    while (reader.Read())
    {
        if (reader.GetInt32(2) != lastOrderId)
        {
            if (orderTotal > 0)
            {
                Console.WriteLine("    Total: {0:C}\n", orderTotal);
                orderTotal = 0;
            }

            Console.WriteLine("{0,10}  {1,-15} Order ID: {2}",
                reader.GetDateTime(0).ToShortDateString(),
                reader.GetString(1),
                reader.GetInt32(2));
            lastOrderId = reader.GetInt32(2);
        }

        Console.WriteLine("    {0}. {1,-20} {2,3} @ {3,9:C} = {4,9:C}",
            reader.GetInt32(3),
            reader.GetString(4),
            reader.GetInt32(5),
            reader.GetDouble(6),
            reader.GetDouble(7));
        orderTotal += reader.GetDouble(7);
    }
    if (orderTotal > 0)
    {
        Console.WriteLine("    Total: {0:C}\n", orderTotal);
        orderTotal = 0;
    }

    // Close the reader. Important!
    reader.Close();
}
}

```

This method creates a query that selects fields from the customers, orders, and order_items tables. It also selects the calculated value `quantity * price_each`.

The `WHERE` clause uses the records' IDs to link associated records. The `ORDER BY` clause sorts the results by date, order ID (if multiple orders have the same date), and item number. The final sort by item number makes the items in an order appear in their proper order. After it has created the query, the program uses it to create a command object.

The code then defines two variables. It uses the variable `lastOrderId` to track the current order's ID. It initially sets that variable to `-1` so the first row will have a different ID.

The code uses the variable `orderTotal` to keep track of the current order's total cost. It initially sets this value to 0.

Having initialized the `lastOrderId` and `orderTotal` variables, the code calls the command's `ExecuteReader` method. That method performs the query and returns an `NpgsqlDataReader` object that can loop through the returned results.

The code then enters a `while` loop that executes as long as the reader's `Read` method returns `true`. That method advances to the next returned result row. (Initially, it points before the first row, so the first call to `Read` advances to the first row.) The `Read` method returns `true` if it successfully moved to the next row and `false` if it dropped off the end of the results.

Inside the loop, the program calls the reader's `GetInt32` method to get the value in position 2 (numbered from 0) as an integer. (If you look at the query, you'll see that the returned value in position 2 is the order ID.) If the new order ID is different from the current one stored in the variable `lastOrderId`, then the program displays the previous order's total value.

If this is a new order, the code displays the new order date, customer name, and order ID. It then saves the new order's ID in the `lastOrderId` variable so that it will know when it sees the next order's records.

After it has dealt with the case of starting a new record, the program displays the new order item's data and adds its total cost to the `orderTotal` variable.

After it has finished looping through all the returned data, the program displays the final order's total cost.

The method then closes the data reader. If you don't do that, then the next time you try to use the connection to execute a command you'll get the following error:

```
System.InvalidOperationException: 'Connection is busy'
```

Instead of explicitly closing the reader, you can use it in a `using` block so that it closes automatically.

The following text shows this method's output:

```
4/1/2027 Bob's Beans      Order ID: 633
  1. Coffee cup          10 @    $1.99 =    $19.90
  2. Sulawesi            5 @    $8.95 =    $44.75
  Total: $64.65

4/1/2027 Bob's Beans      Order ID: 634
  1. Tarrazu             2 @   $19.95 =   $39.90
  2. Coffee filters      10 @    $4.95 =   $49.50
  3. Napkins             10 @    $4.99 =   $49.90
  4. Yirgacheffe         5 @   $15.99 =   $79.95
  Total: $219.25

4/10/2027 Claire's Coffee Order ID: 637
  1. Grinder             1 @   $299.00 =  $299.00
```

```

Total: $299.00

4/17/2027 Bob's Beans      Order ID: 635
1. Digital scale          1 @    $74.99 =    $74.99
Total: $74.99

4/20/2027 Claire's Coffee Order ID: 638
1. Thermometer           2 @    $15.49 =    $30.98
2. Foamer                 1 @    $9.95 =    $9.95
Total: $40.93

5/1/2027 Bob's Beans      Order ID: 636
1. Espresso machine       1 @ $2,999.00 = $2,999.00
Total: $2,999.00

```

These results are similar to those produced by the Python version of the program.

SUMMARY

PostgreSQL is a popular relational database system that in many ways is similar to MariaDB. The main differences between this example and the MariaDB example are due to the ways that the two programs' database adapters work.

For this example, you used pgAdmin to create an orders database. The example program then adds some records to the database's tables and fetches the data, linking related records together to reproduce the customers' orders. In a real application, you would want to add other features to let the user interactively work with the database to create, read, and edit orders.

The next chapter moves away from relational databases to demonstrate a NoSQL graph database. Before you move on to Chapter 20, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

EXERCISES

1. The `CreateCustomerRecords` and `CreateOrderRecords` methods call a command object's `ExecuteScalar` method, but the `CreateOrderItemRecords` method calls `ExecuteNonQuery`. Why the difference?
2. Add a new `LetUserCreateOrder` method to the example program that allows the user to create a new order for one of the existing customers. It should do the following:
 - a. Prompt the user for a customer ID. (Be sure to enter a valid ID when you're testing.)
 - b. Prompt the user for an order date.
 - c. Enter an infinite loop that prompts for the order item information: item name, quantity, and price of each. It should keep track of the item number as the user enters items. When the user enters a blank line, break out of the loop.
 - d. Create the new records.

- i. Create a transaction.
 - ii. Create the `orders` record and save the autogenerated order ID in a variable. Make the command use the transaction.
 - iii. Create the order items records, making their commands use the transaction.
- e. Rerun the `DisplayOrders` method to verify that the new order was created correctly.

Here are some hints:

- Don't worry about error handling. Assume that you'll enter valid values while testing.
- It may help with debugging to print the SQL commands before executing them.

-
3. Set a breakpoint in the `LetUserCreateOrder` method on the statement that commits the transaction. Run the program and create a new order. To make recognizing the order easier, give it the distinctive date **1/2/2300**. Give it at least one item with the distinctive item name **Aardvark**.

When it reaches the breakpoint, right-click the following statement and select Set New Statement so that the program skips the `Commit` statement. Then let the program finish.

Now use pgAdmin to verify that the transaction prevented the program from creating the new order. To do that, follow these steps:

- a. Start pgAdmin.
 - b. In the tree view on the left, drill down to Servers ⇄ PostgreSQL ⇄ Databases ⇄ BrewCrewDB ⇄ Schemas ⇄ Public ⇄ Tables.
 - c. In the tree view, click the `orders` table. On the toolbar, click the View Data tool, which looks like a little spreadsheet. Look at the data and verify that there is no order with the date 1/2/2300.
 - d. In the tree view, click the `order_items` table. Click the View Data tool and verify that there is no order item named Aardvark.

 4. What happens if you enter **crème brûlée** as an item name? (If you can't type those characters, search online for **creme brulee** and then copy and paste the words with the correct accents into the program's input box.)

 5. What happens if you enter **Amy's cookies** as an item name?

 6. What happens if you enter the order date in the format d/m/yyyy? What happens if you enter the specific value **20/10/2027**?

 7. What kinds of database errors could the new order code encounter? (Ignore formatting errors such as the user entering "ten" for the customer ID or entering an invalid order date like 20/40/2100.)
-

20

Neo4j AuraDB in Python

This chapter's example uses Python to build a NoSQL graph database in the cloud. It uses the Neo4j AuraDB database to build and perform queries on the org chart shown in Figure 20.1.

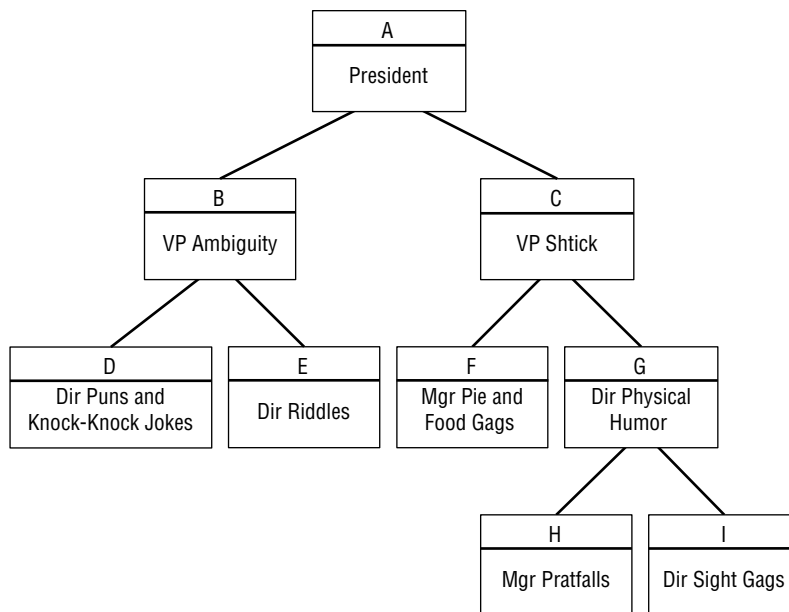


FIGURE 20.1

This example uses an adapter called `neo4j`, which is the official database adapter provided by Neo4j.

The following section explains how to install the Neo4j AuraDB database engine in the cloud. The section after that explains how to install the `neo4j` database adapter. The rest of the chapter describes the example program.

INSTALL NEO4J AURADB

Neo4j AuraDB is a fully managed graph database service in the cloud. Neo4j provides many other services that can do more than AuraDB, but this one is simple and, best of all, free.

To install AuraDB, go to <http://neo4j.com/cloud/platform/aura-graph-database>, click the Start Free button, fill out the registration form, and create a free account.

WARNING *If your ad/spam blocker software is sufficiently paranoid, it may block the Neo4j website. (I didn't have trouble but technical editor, John, did.) You may need to whitelist the site to enter.*

You should not need to enter credit card information to get a free account (unless they change the way this works later).

Also note that you only get one database instance for free. Feel free to experiment, but if you stray too far from the instructions in this chapter, you may need to delete the database and start over to get the example to work.

To create a database instance, click New Instance. On the “Get started by picking a dataset or an empty instance” page, hover over Empty Instance and click Create.

When AuraDB creates the database, it automatically generates credentials that you’ll need to use later to connect to the database. It then displays a password, as shown in Figure 20.2. Click Download to save the credentials somewhere safe.

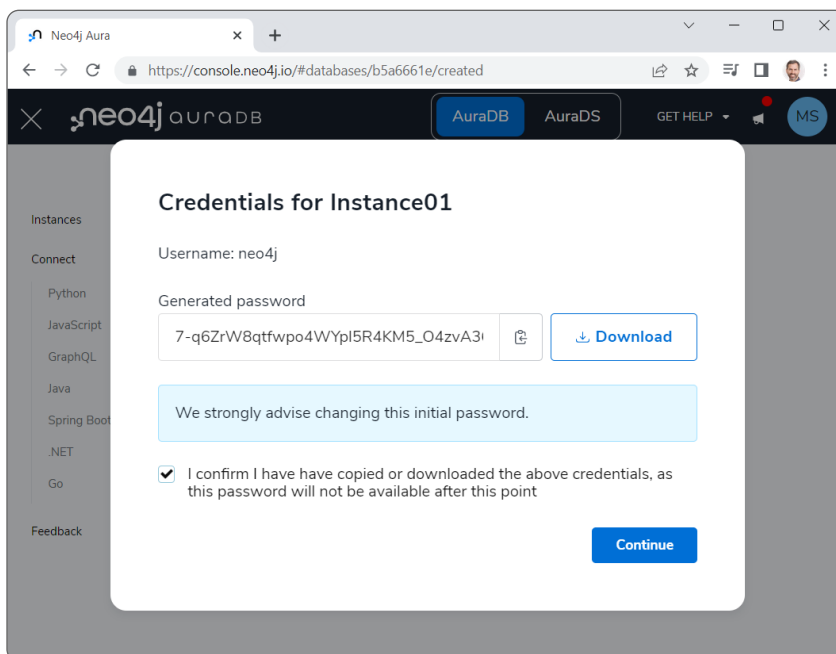


FIGURE 20.2

Check the box that indicates you'll take the blame if you lose your password, and click Continue to see a screen similar to Figure 20.3.

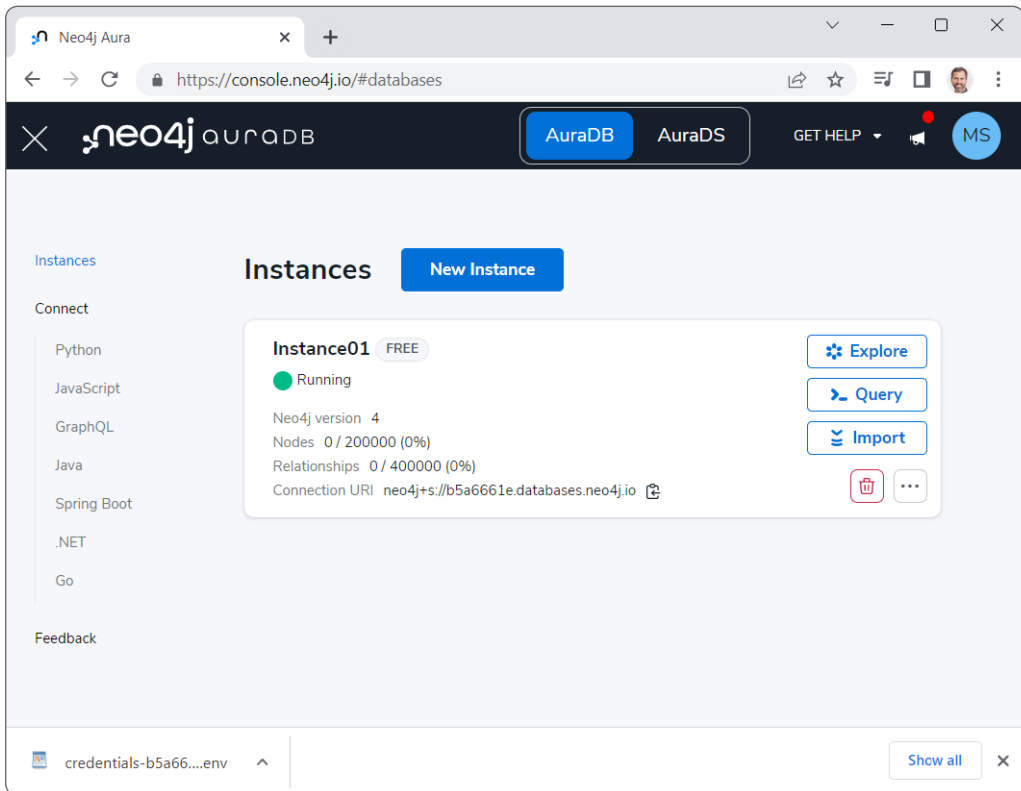


FIGURE 20.3

You can use the New Instance button to create another new database instance, but you only get one for free, so you'll either need to delete this one first or pay for the second.

NOTE During the few weeks while I experimented with AuraDB, the steps for creating a new database instance changed. If you see screens different from those described here, just do your best. They probably won't be too hard to figure out.

If you click the instance shown in Figure 20.3, you can view an example program. Tabs let you pick one of these programming languages: Python, JavaScript, GraphQL, Java, Spring Boot, .NET, or Go. You can look through that code now if you like. We'll use some of it later in this chapter and in the next chapter.

NODES AND RELATIONSHIPS

Before we start looking at code, you should know exactly what kinds of information a graph database stores.

An AuraDB database holds two types of objects: nodes and relationships. A *node* represents some sort of item such as a person, movie, location, or bank account. A *relationship* represents a relationship between two nodes.

For example, an example on the Neo4j website uses a database that represents the relationships between movies, actors, and directors. The `ACTED_IN` relationship represents the fact that a particular person acted in a movie. Similarly, the `DIRECTED` relationship means a person directed a movie. It's all fairly intuitive and easy to understand—at least until you start looking at complicated queries.

All relationships in AuraDB are *directed*, which means they point from one node to another. For example, Harrison Ford `ACTED_IN` the movie *Blade Runner*, but *Blade Runner* did not `ACTED_IN` Harrison Ford. That should be blindingly obvious, but more symmetrical relationships can be a bit more confusing.

For example, consider the relationship `IS_A_SIBLING_OF`. Just because Bart Simpson `IS_A_SIBLING_OF` Lisa Simpson, that doesn't mean that Lisa Simpson `IS_A_SIBLING_OF` Bart Simpson as far as AuraDB is concerned. If you want the relationship to go both ways, then you need to define it in both directions.

However, relationships do *implicitly* define a reversed relationship. For example, suppose Homer Simpson `IS_PARENT_TO` Maggie Simpson. Then you can search the database to find nodes X where either of the following is true:

- Homer Simpson `IS_PARENT_TO` X.
- X `IS_PARENT_TO` Maggie Simpson.

Even though you have only defined the relationship in one direction, you can search for nodes that satisfy the reverse if you want to do so.

If this is confusing, it may be a little clearer when you see some examples of database queries.

CYPHER

Cypher is the query language supported by AuraDB. Despite its cryptographic-sounding name (and somewhat cryptic syntax), Cypher is a graph query language that was developed by Neo4j. It was inspired by SQL, so it has a vaguely SQL-like flavor but with some changes to support graph queries. You can learn more about the language as implemented by AuraDB at <https://neo4j.com/developer/cypher>.

The language is now open source, and you can get information about openCypher at <http://opencypher.org>. The plan is to turn openCypher into a standard called Graph Query Language (GQL). It's not there yet, but it's likely that the final GQL will have a lot in common with Cypher.

WARNING Your ad/spam blocker software may also balk at the `openCypher.org` website. You may need to whitelist it to learn more about `openCypher`.

You'll learn more about Cypher when you see examples in the program.

CREATE THE PROGRAM

Now that the database is waiting for you in the cloud, you need to install a database driver for it. Then you can start writing code.

To create a Python program to work with the AuraDB org chart database, create a new Jupyter Notebook and then add the code described in the following sections.

Install the Neo4j Database Adapter

To install the `neo4j` database adapter, simply use the following `pip` command:

```
$ pip install neo4j
```

If you're not familiar with `pip`, you can execute the following command in a Jupyter Notebook cell instead:

```
!pip install neo4j
```

That's all there is to it!

The `neo4j` database adapter uses a somewhat unusual approach for executing commands. The following steps show the general approach:

1. Create a database session.
2. Call the session object's `write_transaction` and `read_transaction` methods, passing them an action method that performs the desired action. That method should take a transaction object as a parameter and should use the transaction object's `run` method to do the work.

To get started, we need to write the action methods. Then we can write more code to launch those methods. (Again, if this is confusing, just go with the flow until you see some example code.)

The following sections describe the lower-level action methods. The sections after those describe higher-level methods that build the org chart. Finally, the last sections in this part of the chapter describe the main program that uses the other methods to build and query the org chart.

Action Methods

The following sections describe the lower-level action methods. Each takes a transaction object as its first parameter. They then use that object to run a database command or query.

They all follow the same pattern, so you'll probably get the hang of them quickly. The more interesting differences are the query commands that they send to the database. Those queries are vaguely reminiscent of SQL, but they're not the same because they need to work with a graph database instead of a relational database.

Table 20.1 lists the action methods and gives their purposes.

TABLE 20.1: Action methods and their purposes

METHOD	PURPOSE
<code>delete_all_nodes</code>	Delete all nodes and their links.
<code>make_node</code>	Create a new node.
<code>make_link</code>	Create a relationship between two nodes.
<code>execute_node_query</code>	Execute a query that returns nodes.
<code>find_path</code>	Find a path between two nodes.

delete_all_nodes

The following code shows how the `delete_all_nodes` method deletes all of the nodes in the database:

```
# Delete all nodes from the database.
from neo4j import GraphDatabase
import logging
from neo4j.exceptions import ServiceUnavailable

# Delete all previous nodes and links.
def delete_all_nodes(tx):
    statement = "MATCH (n) DETACH DELETE n"
    tx.run(statement)
```

This code starts with some `import` statements. It then defines the `delete_all_nodes` action method. The method takes a transaction object `tx` as a parameter and passes the string `MATCH (n) DETACH DELETE n` into the transaction's `run` method to make the database execute that command. Easy, right?

This is the basic pattern that all the action methods follow. They pass a database command into the transaction object's `run` method. The part that differentiates this action method from others is the database command.

The `MATCH` command is similar to a SQL query. It uses a pattern to match objects inside the database. Later parts of the command do things with any objects that match.

In this case, the pattern `(n)` matches any node. In general, expressions inside parentheses `()` match nodes, and expressions surrounded by square brackets `[]` match relationships.

Note that you cannot delete a node if it is involved in any relationships. (This is kind of like a graph database version of a foreign key constraint.) You can either delete the relationships first or include the `DETACH` keyword to tell the database to delete those relationships automatically before deleting the node.

The last part of the command is `DELETE n`, which deletes the node `n`. Here `n` is a node that was matched by the `MATCH (n)` part of the command.

To summarize, this statement says, “Match all nodes, detach their relationships, and then delete them.” (Now that you have the hang of it, the other action methods will go faster.)

make_node

The following code shows how the `make_node` method creates a new node:

```
# Use parameters to make a node.
def make_node(tx, id, title):
    statement = "CREATE (n:OrgNode { ID:$id, Title:$title })"
    parameters = {
        "id": id,
        "title": title
    }
    tx.run(statement, parameters)
```

This method creates a statement that uses the `CREATE` command to add a node to the database. The `n:OrgNode` part tells the database that this node’s type is `OrgNode`. That’s just a name that I made up for this kind of node. You could create `EmployeeNode`, `ProductNode`, `CandyCaneNode`, or any node types that make sense for your application. You also don’t need to include the word `Node`; I just decided that it would make the queries easier to understand.

The part of the statement that’s inside the curly brackets tells the database to give the node two properties, `ID` and `Title`. The parts with the dollar signs (shown in bold) are placeholders. We’ll provide values for the placeholders shortly.

Note that property names are case-sensitive in AuraDB, so if you create a `Title` property and later search for a `title` value, you won’t get any matches. (And you’ll waste a lot of time wondering why not.)

The code then makes a dictionary that uses the placeholder names as keys and supplies values for them. The first dictionary entry associates the `$id` placeholder with the parameter `id` that was passed into `make_node`. The second entry associates the `$title` placeholder with the method’s `title` parameter.

The code finishes by calling the transaction object’s `run` method, passing it the command string and the `parameters` dictionary.

make_link

The following code shows how the `make_link` method creates a relationship between two nodes:

```
# Use parameters to create a link between an OrgNode and its parent.
def make_link(tx, id, boss):
```

```
statement = (
    "MATCH"
    "    (a:OrgNode), "
    "    (b:OrgNode) "
    "WHERE"
    "    a.ID = $id AND "
    "    b.ID = $boss "
    "CREATE (a)-[r:REPORTS_TO { Name:a.ID + '->' + b.ID }]->(b) "
)
parameters = {
    "id": id,
    "boss": boss
}
tx.run(statement, parameters)
```

This method creates a `REPORTS_TO` link for the org chart. The `MATCH` statement looks for two `OrgNode` objects, `a` and `b`, where `a.ID` matches the `id` parameter passed into the method and `b.ID` matches the `boss` parameter.

After the `MATCH` statement finds those two nodes, the `CREATE` command makes a new `REPORTS_TO` relationship leading from node `a` to node `b`. It gives the relationship a `Name` property that contains the two IDs separated by `->`. For example, if the nodes' IDs are `A` and `B`, then this relationship's `Name` is `A->B`.

execute_node_query

The following code shows the `execute_node_query` method, which executes a query that matches nodes that are named `n` in the query. This method returns a list of the query's results.

```
# Execute a query that returns zero or more nodes identified by the name "n".
# Return the nodes in a list of strings with the format "ID: Title".
def execute_node_query(tx, query):
    result = []
    for record in tx.run(query):
        node = record["n"]
        result.append(f"{node['ID']}: {node['Title']}")
    return result
```

The method first creates an empty list to hold results. It then runs the query and loops through the records that it returns.

Inside the loop, it uses `record["n"]` to get the result named `n` from the current record. That result will be a node selected by the `MATCH` statement. (You'll see that kind of `MATCH` statement a bit later.) The code copies the node's `ID` and `Title` properties into a string and adds the string to the result list.

After it finishes looping through the records, the method returns the list of results.

find_path

The following code shows how the `find_path` method finds a path between two nodes in the org chart:

```
# Execute a query that returns a path from node id1 to node id2.
# Return the nodes in a list of strings with the format "ID: Title".
def find_path(tx, id1, id2):
    statement = (
        "MATCH"
        "  (start:OrgNode { ID:$id1 } ),"
        "  (end:OrgNode { ID:$id2 } ),"
        "  p = shortestPath((start)-[:REPORTS_TO *]-end) "
        "RETURN p"
    )
    parameters = {
        "id1": id1,
        "id2": id2
    }

    record = tx.run(statement, parameters).single()
    path = record["p"]
    result = []
    for node in path.nodes:
        result.append(f"{node['ID']}: {node['Title']}")

    return result
```

This code looks for two nodes that have the IDs `id1` and `id2` that were passed into the method as parameters. It names the matched nodes `start` and `end`.

It then calls the database's `shortestPath` method to find a shortest path from `start` to `end` following `REPORTS_TO` relationships. The `*` means the path can have any length. The statement saves the path that it found in a variable named `p`.

Note that the `shortestPath` method only counts the *number* of relationships that it crosses; it doesn't consider costs or weights on the relationships. In other words, it looks for a path with the fewest *steps*, not necessarily the *shortest total cost* as you would like in a street network, for example. Some AuraDB databases can perform the least total cost calculation and execute other graph algorithms, but the free version cannot.

After it composes the database command, the method executes it, passing in the necessary parameters. It calls the result's `single` method to get the first returned result.

It then looks at that result's `p` property, which holds the path. (Remember that the statement saved the path with the name `p`.)

The code loops through the path's nodes and adds each one's `ID` and `Title` values to a result list. The method finishes by returning that list.

Org Chart Methods

That's the end of the action methods. They do the following:

- Delete all nodes.
- Make a node.
- Make a link.
- Execute a query that returns nodes.
- Find a path between two nodes.

The following sections describe the two methods that use those tools to build and query the org chart. The earlier action methods make these two relatively straightforward.

build_org_chart

The following code shows how the `build_org_chart` method builds the org chart:

```
# Build the org chart.
def build_org_chart(tx):
    # Make the nodes.
    make_node(tx, "A", "President")
    make_node(tx, "B", "VP Ambiguity")
    make_node(tx, "C", "VP Shtick")
    make_node(tx, "D", "Dir Puns and Knock-Knock Jokes")
    make_node(tx, "E", "Dir Riddles")
    make_node(tx, "F", "Mgr Pie and Food Gags")
    make_node(tx, "G", "Dir Physical Humor")
    make_node(tx, "H", "Mgr Pratfalls")
    make_node(tx, "I", "Dir Sight Gags")

    # Make the links.
    make_link(tx, "B", "A")
    make_link(tx, "C", "A")
    make_link(tx, "D", "B")
    make_link(tx, "E", "B")
    make_link(tx, "F", "C")
    make_link(tx, "G", "C")
    make_link(tx, "H", "G")
    make_link(tx, "I", "G")
```

This method calls the `make_node` method repeatedly to make the org chart's nodes. It then calls the `make_link` method several times to make the org chart's relationships.

Notice that each call to `make_node` and `make_link` includes the transaction object that `build_org_chart` received as a parameter.

query_org_chart

The following code shows how the `query_org_chart` method performs some queries on the finished org chart:

```
# Perform some queries on the org chart.
def query_org_chart(tx):
    # Get F.
    print("F:")
    result = execute_node_query(tx,
        "MATCH (n:OrgNode { ID:'F' }) " +
        "return n")
    print(f"    {result[0]}")

    # Who reports directly to B.
    print("\nReports directly to B:")
    result = execute_node_query(tx,
        "MATCH " +
        "    (n:OrgNode) -[:REPORTS_TO]->(a:OrgNode { ID:'B' }) " +
        "return n " +
        "ORDER BY n.ID")
    for s in result:
        print("    " + s)

    # Chain of command for H.
    print("\nChain of command for H:")
    result = find_path(tx, "H", "A")
    for s in result:
        print("    " + s)

    # All reports for C.
    print("\nAll reports for C:")
    result = execute_node_query(tx,
        "MATCH " +
        "    (n:OrgNode) -[:REPORTS_TO *]->(a:OrgNode { ID:'C' }) " +
        "return n " +
        "ORDER BY n.ID")
    for s in result:
        print("    " + s)
```

This method first calls the `execute_node_query` method to execute the following query.

```
MATCH (n:OrgNode { ID:'F' }) return n
```

This query finds the node with ID equal to F. The code prints the result.

Next, the method looks for nodes that have the `REPORTS_TO` relationship ending with node B. That returns all of the nodes that report directly to node B. The code loops through the results, displaying them.

The method then uses the `find_path` method to find a path from node H to node A. Node A is at the top of the org chart, so this includes all of the nodes in the chain of command from node H to the top.

The last query the method performs matches the following:

```
(n:OrgNode) -[:REPORTS_TO *]->(a:OrgNode { ID:'C' })
```

This finds nodes `n` that are related via any number (`*`) of `REPORTS_TO` relationships to node `C`. That includes all the nodes that report directly or indirectly to node `C`. Graphically those are the nodes that lie below node `C` in the org chart.

Main Program

The previous methods make working with the org chart fairly simple. All we need to do now is get them started. The following code shows the how main program does that:

```
# Replace the following with your database URI, username, and password.
uri = "neo4j+s://386baeab.databases.neo4j.io"
user = "neo4j"
password = "InsertYourReallyLongAndSecurePasswordHere"

driver = GraphDatabase.driver(uri, auth=(user, password))

with driver.session() as session:
    # Delete any previous nodes and links.
    print("Deleting old data...")
    session.write_transaction(delete_all_nodes)

    # Build the org chart.
    print("Building org chart...")
    session.write_transaction(build_org_chart)

    # Query the org chart.
    print("Querying org chart...")
    session.read_transaction(query_org_chart)

# Close the driver when finished with it.
driver.close()
```

This code first defines the uniform resource identifier (URI) where the database is located, the username, and the password. You can find these in the credential file that you downloaded when you created the database instance. (I hope you saved that file! If you didn't, then this might be a good time to delete the database instance and start over.)

Next, the code uses the URI, username, and password to create a graph database driver. It then creates a new session on the driver. It does that inside a `with` statement to ensure that the session is cleaned up properly when the program is finished with it.

The program then uses the session's `write_transaction` method to run `delete_all_nodes`. This is the step that starts `delete_all_nodes` and passes it the transaction object. It uses `write_transaction` to run `delete_all_nodes` inside a write transaction because `delete_all_nodes` modifies data.

The code runs the `build_org_chart` method similarly.

The program then calls the session's `read_transaction` method to run `query_org_chart`. It uses `read_transaction` this time because `query_org_chart` only reads data.

After it has finished displaying its results, the program closes the database driver.

The following shows the program's output:

```

Deleting old data...
Building org chart...
Querying org chart...
F:
    F: Mgr Pie and Food Gags

Reports directly to B:
    D: Dir Puns and Knock-Knock Jokes
    E: Dir Riddles

Chain of command for H:
    H: Mgr Pratfalls
    G: Dir Physical Humor
    C: VP Shtick
    A: President

All reports for C:
    F: Mgr Pie and Food Gags
    G: Dir Physical Humor
    H: Mgr Pratfalls
    I: Dir Sight Gags

```

Figure 20.4 shows the same org chart in Figure 20.1 so you can look at it to see that the output is correct.

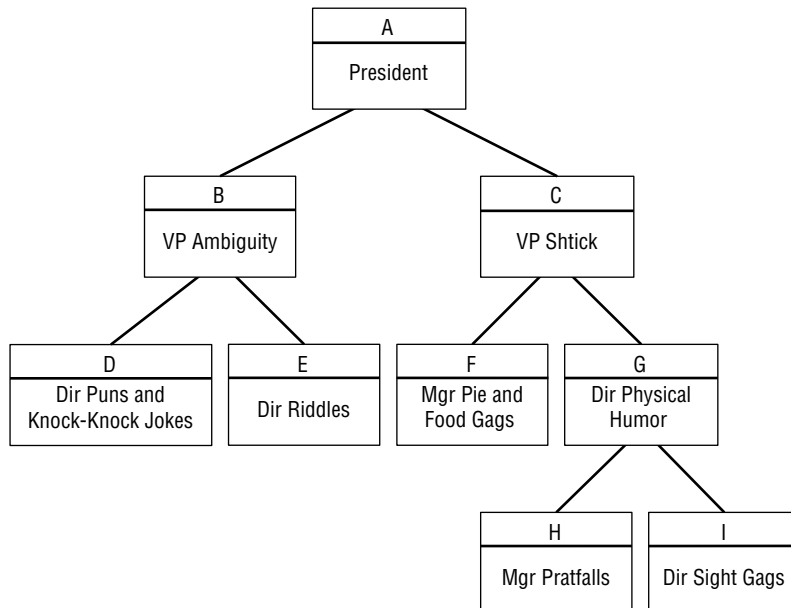


FIGURE 20.4

SUMMARY

This chapter shows how you can use Python and a NoSQL graph database to build and explore an org chart. You can use similar techniques to work with other trees and, more generally, graphs that are not trees.

As you work with this example, you might notice that operations are relatively slow, particularly if you have a slow network connection. This is generally true of cloud applications. Network communications tend to be slower than local calculations.

The pattern that this example used was to:

1. Create a database session.
2. Call the session object's `write_transaction` and `read_transaction` methods, passing them an action method.
3. Use an action method that takes a transaction object as a parameter and uses it to do the work.

The next chapter shows how to build a similar example program in C#. Because it uses a different database adapter, it also uses a different pattern.

Before you move on to that example, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

EXERCISES

1. What issue would you face if you wanted to give two quidditch players the `IS_TEAMMATES_WITH` relationship in an AuraDB graph database?
2. The program uses the following statement to find the people who report to person C:

```
MATCH
  (n:OrgNode) - [:REPORTS_TO *] -> (a:OrgNode { ID: 'C' })
return n
ORDER BY n.ID
```

- a. What would happen if you changed this statement to the following? (The changes are bold.)

```
MATCH
  (n:OrgNode) <- [:REPORTS_TO *] - (a:OrgNode { ID: 'H' })
return n
ORDER BY n.ID
```

- b. What would happen if you removed the asterisk from the original statement to get the following?

```
MATCH
  (n:OrgNode) - [:REPORTS_TO] -> (a:OrgNode { ID:'C' })
return n
ORDER BY n.ID
```

3. Are paths unique from one node to another in a simple org chart?
-

4. What does the following statement return?

```
MATCH
  (n:OrgNode) <- [:REPORTS_TO] - (:OrgNode)
  <- [:REPORTS_TO] - (:OrgNode { ID:'I' })
return n
ORDER BY n.ID
```

5. Do the following two MATCH statements return different results?

```
MATCH
  (n:OrgNode) - [:REPORTS_TO *] -> (a:OrgNode { ID:'C' })
return n
ORDER BY n.ID
```

```
MATCH
  (:OrgNode { ID:'C' }) <- [:REPORTS_TO *] - (n:OrgNode)
return n
ORDER BY n.ID
```

21

Neo4j AuraDB in C#

This chapter's example uses C# to build a NoSQL graph database in the cloud. It uses the Neo4j AuraDB database to build and perform queries on the org chart shown in Figure 21.1.

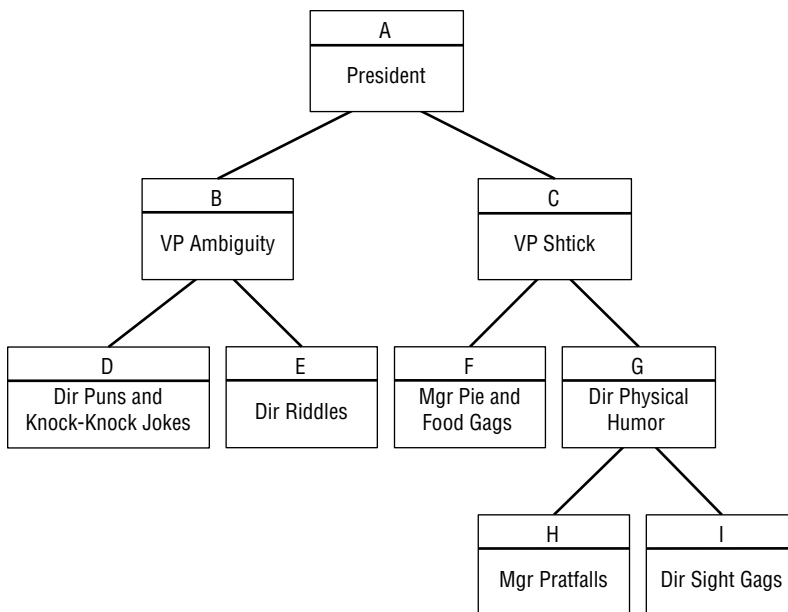


FIGURE 21.1

If you skipped Chapter 20, “Neo4j AuraDB in Python,” which built a similar example in Python, go to that chapter and read the beginning and the first three sections, which are described in the following list.

- “Install Neo4j AuraDB” explains how to install the Neo4j AuraDB graph database.

- “Nodes and Relationships” explains some basic graph database concepts that you’ll need to understand to use AuraDB.
- “Cypher” briefly introduces the Cypher graph query language that the example uses to interact with the database.

When you reach the section “Create the Program” in Chapter 20, return to this chapter and read the following sections.

CREATE THE PROGRAM

To create a C# program to work with the AuraDB org chart database, create a new C# Console App (.NET Framework) and then add the code described in the following sections.

Like the example described in Chapter 20, this example builds an assortment of helper methods that do things like create nodes, build relationships, and execute queries on the data. It then uses those methods to create two higher-level methods that build and query the org chart. Finally, the main program connects to the database and calls those two methods.

Install the Neo4j Driver

This example uses the Neo4j driver to allow your program to communicate with the AuraDB database sitting in the cloud. To add the driver to your project, follow these steps:

1. Open the Project menu and select Manage NuGet Packages.
2. Click the Browse tab and enter `Neo4j.Driver.Simple` in the search box. Click that entry in the list on the left and then click the Install button on the right. This shouldn’t take too long, but it’s still nice to have a fast Internet connection.

This example uses a pattern that is slightly different from the one used in Chapter 20. The previous example used action methods that took a transaction object as a parameter and then used that object’s methods to do the work.

In this example, the helper methods take a database driver as a parameter. Those methods use the driver’s `Session` method to get a session object that they then use to run database commands.

Session objects can start transactions that you can commit or roll back. This example simply calls the session’s `Run` method to execute database commands, and the `Run` method uses an auto-commit transaction by default so that we don’t need to worry about committing the result.

All of these features are differences in the two database adapters, but the underlying database engine is the same. You will often find multiple database adapters available for a given database engine, so you might want to shop around to find one that provides features that you like.

The session objects used in this example are “simple” sessions that work synchronously. Neo4j also provides asynchronous session objects that may be useful for some applications, particularly if your network is slow. We’re using the “simple” sessions because they are, if not exactly simple, at least simpler than the asynchronous version.

The following sections divide the program's code into three parts: action methods that build basic objects such as nodes and relationships, org chart methods that build and explore the org chart, and the main program.

Action Methods

The following sections describe the lower-level action methods. Each takes a driver object as its first parameter. They use the driver to create a session object, and then use that object to run a database command or query.

They follow the same pattern, so you'll probably get the hang of them quickly. The more interesting differences are the query commands that they send to the database. Those queries are vaguely reminiscent of SQL, but they're not the same because they need to work with a graph database instead of a relational database.

Table 21.1 lists the action methods and gives their purposes.

TABLE 21.1: Action methods and their purposes

METHOD	PURPOSE
DeleteAllNodes	Delete all nodes and their links.
MakeNode	Create a new node.
MakeLink	Create a relationship between two nodes.
ExecuteNodeQuery	Execute a query that returns nodes.
FindPath	Find a path between two nodes.

DeleteAllNodes

The following code shows how the `DeleteAllNodes` method deletes all the nodes in the database:

```
using Neo4j.Driver;
...
// Delete all previous nodes and links.
static void DeleteAllNodes(IDriver driver)
{
    using (ISession session = driver.Session())
    {
        string statement = "MATCH (n) DETACH DELETE n";
        session.Run(statement);
    }
}
```

The module includes the statement using `Neo4j.Driver` to make it easier to use the driver.

The `DeleteAllNodes` method takes a driver object as a parameter, uses it to create a session object, and then uses the session object's `Run` method to execute the database command `MATCH (n) DETACH DELETE n`.

Notice that the session object is created in a `using` block, so it is automatically disposed of when we're done with it. The other methods in this example use a similar approach. Alternatively, you could create a single session and pass it around to all the methods.

The `MATCH` command is similar to a SQL query. It uses a pattern to match objects inside the database. Then later parts of the command do something with any objects that match.

In this case, the pattern `(n)` matches any node. In general, expressions inside parentheses `()` match nodes and expressions surrounded by square brackets `[]` match relationships.

Note that you cannot delete a node if it is involved in any relationships. (This is kind of like a graph database version of a foreign key constraint.) You can either delete the relationships first or include the `DETACH` keyword to tell the database to delete those relationships automatically with the node.

The last part of the command is `DELETE n`, which deletes the node `n`. Here, `n` is a node that was matched by the `MATCH (n)` part of the command.

To summarize, this statement says, "Match all nodes, detach their relationships, and then delete them."

MakeNode

The following code shows the `MakeNode` method:

```
// Use parameters to make a node.
static void MakeNode(IDriver driver, string id, string title)
{
    using (ISession session = driver.Session())
    {
        string statement = "CREATE (n:OrgNode { ID:$id, Title:$title })";
        Dictionary<string, object> parameters =
            new Dictionary<string, object>()
            {
                {"id", id},
                {"title", title},
            };
        session.Run(statement, parameters);
    }
}
```

This method creates a session and then composes a `CREATE` command to add a node to the database. The `n:OrgNode` part tells the database that this node's type is `OrgNode`. That's just a name that I made up for this kind of node. You could create `CustomerNode`, `RecipeNode`, `SaberToothDuckNode`, or any node types that make sense for your application. You also don't need to include the word `Node`; I just decided that it would make the queries easier to understand.

The part of the statement that's inside the curly brackets tells the database to give the node two properties, `ID` and `Title`. We'll provide values for the placeholders with the dollar signs shortly.

Note that property names are case-sensitive, so if you create a `Title` property and later search for a `title` value, you won't get any matches. (And you'll waste a lot of time wondering why not.)

The code then makes a dictionary that uses the placeholder names as keys and supplies values for them. The first dictionary entry associates the `$id` placeholder with the parameter `id` that was passed into `MakeNode`. The second entry associates the `$title` placeholder with the method's `title` parameter.

The code finishes by calling the `session` object's `Run` method, passing it the command string and the `parameters` dictionary.

MakeLink

The following code shows how the `MakeLink` method creates a relationship between two nodes:

```
// Use parameters to create a link between an OrgNode and its parent.
static void MakeLink(IDriver driver, string id, string boss)
{
    using (ISession session = driver.Session())
    {
        string statement =
            "MATCH" +
            "    (a:OrgNode)," +
            "    (b:OrgNode) " +
            "WHERE" +
            "    a.ID = $id AND " +
            "    b.ID = $boss " +
            "CREATE (a)-[r:REPORTS_TO { Name:a.ID + '-' + b.ID }]->(b) ";
        Dictionary<string, object> parameters =
            new Dictionary<string, object>()
            {
                {"id", id},
                {"boss", boss},
            };
        session.Run(statement, parameters);
    }
}
```

This method creates a `REPORTS_TO` link for the org chart. The `MATCH` statement looks for two `OrgNode` objects, `a` and `b`, where `a.ID` matches the `id` parameter passed into the method and `b.ID` matches the `boss` parameter.

After the `MATCH` statement finds those two nodes, the `CREATE` command makes a new `REPORTS_TO` relationship leading from node `a` to node `b`. It gives the relationship a `Name` property that contains the two IDs separated by `->`. For example, if the nodes' IDs are `A` and `B`, then this relationship's `Name` is `A->B`.

ExecuteNodeQuery

The following code shows the `ExecuteNodeQuery` method, which executes a query that matches nodes that are named `n` in the query. This method returns a list of the query's results:

```
// Execute a query that returns zero or more nodes
// identified by the name "n".
// Return the nodes in the format "ID: Title".
static List<string> ExecuteNodeQuery(IDriver driver, string query)
{
    List<string> result = new List<string>();
    using (ISession session = driver.Session())
    {
        foreach (IRecord record in session.Run(query))
        {
            INode node = (INode)record.Values["n"];
            result.Add($"{node["ID"]}: {node["Title"]}");
        }
    }
    return result;
}
```

The method first creates a list to hold results. It then runs the query and loops through the records that it returns.

Inside the loop, it uses `record.Values["n"]` to get the result named `n` from the current record. That result will be a node selected by the `MATCH` statement. (You'll see that kind of `MATCH` statement a bit later.) The code copies the node's `ID` and `Title` properties into a string and adds the string to the result list.

After it finishes looping through the records, the method returns the list.

FindPath

The following code shows how the `FindPath` method finds a path between two nodes in the org chart:

```
// Execute a query that returns a path from node id1 to node id2.
// Return the nodes in the format "ID: Title".
static List<string> FindPath(IDriver driver, string id1, string id2)
{
    List<string> result = new List<string>();
    using (ISession session = driver.Session())
    {
        string statement =
            "MATCH" +
            " (start:OrgNode { ID:$id1 } )," +
            " (end:OrgNode { ID:$id2 } )," +
            " p = shortestPath((start)-[:REPORTS_TO *]-(end))" +
            " RETURN p";
        Dictionary<string, object> parameters =
            new Dictionary<string, object>()
    }
```

```

        {
            {"id1", id1},
            {"id2", id2},
        };

        // Get one path.
        IRecord record = session.Run(statement, parameters).Single();
        IPath path = (IPath)record.Values["p"];

        foreach (INode node in path.Nodes)
        {
            result.Add($"{node["ID"]}: {node["Title"]}");
        }
    }
    return result;
}

```

This code looks for two nodes that have the IDs passed into the method as parameters. It names the matched nodes `start` and `end`.

It then calls the database's `shortestPath` method to find a shortest path from `start` to `end` following `REPORTS_TO` relationships. The `*` means the path can have any length. The statement saves the path that it found with the name `p`.

Note that the `shortestPath` method only counts the *number* of relationships that it crosses; it doesn't consider costs or weights on the relationships. In other words, it looks for a path with the fewest *steps*, not necessarily the *shortest total cost* as you would like in a street network, for example. Some AuraDB databases can perform the least total cost calculation and execute other graph algorithms, but the free version cannot.

After it composes the database command, the method executes it, passing in the necessary parameters. It calls the result's `single` method to get the first returned result.

It then looks at that result's `p` property, which holds the path. (Remember that the statement saved the path with the name `p`.)

The code loops through the path's nodes and adds each one's `ID` and `Title` values to a result list. The method finishes by returning that list.

Org Chart Methods

That's the end of the action methods. They do the following:

- Delete all nodes.
- Make a node.
- Make a link.
- Execute a query that returns nodes.
- Find a path between two nodes.

The following sections describe the two methods that use those tools to build and query the org chart. The earlier action methods make these two relatively straightforward.

BuildOrgChart

The following code shows how the `BuildOrgChart` method builds the org chart:

```
// Build the org chart.
static void BuildOrgChart(IDriver driver)
{
    // Make the nodes.
    MakeNode(driver, "A", "President");
    MakeNode(driver, "B", "VP Ambiguity");
    MakeNode(driver, "C", "VP Shtick");
    MakeNode(driver, "D", "Dir Puns and Knock-Knock Jokes");
    MakeNode(driver, "E", "Dir Riddles");
    MakeNode(driver, "F", "Mgr Pie and Food Gags");
    MakeNode(driver, "G", "Dir Physical Humor");
    MakeNode(driver, "H", "Mgr Pratfalls");
    MakeNode(driver, "I", "Dir Sight Gags");

    // Make the links.
    MakeLink(driver, "B", "A");
    MakeLink(driver, "C", "A");
    MakeLink(driver, "D", "B");
    MakeLink(driver, "E", "B");
    MakeLink(driver, "F", "C");
    MakeLink(driver, "G", "C");
    MakeLink(driver, "H", "G");
    MakeLink(driver, "I", "G");
}
```

This method calls the `MakeNode` method repeatedly to make the org chart's nodes. It then calls the `MakeLink` method several times to make the org chart's relationships.

Notice that each call to `MakeNode` and `MakeLink` includes the transaction object that `BuildOrgChart` received as a parameter.

QueryOrgChart

The following code shows how the `QueryOrgChart` method performs some queries on the finished org chart:

```
// Perform some queries on the org chart.
static void QueryOrgChart(IDriver driver)
{
    List<string> result;

    // Get F.
    Console.WriteLine("F:");
```

```

result = ExecuteNodeQuery(driver,
    "MATCH (n:OrgNode { ID:'F' }) " +
    "return n");
Console.WriteLine($"    {result[0]}");

// Who reports directly to B.
Console.WriteLine("\nReports directly to B:");
result = ExecuteNodeQuery(driver,
    "MATCH " +
    "    (n:OrgNode)-[:REPORTS_TO]->(a:OrgNode { ID:'B' }) " +
    "return n " +
    "ORDER BY n.ID");
foreach (string s in result)
    Console.WriteLine("    " + s);

// Chain of command for H.
Console.WriteLine("\nChain of command for H:");
result = FindPath(driver, "H", "A");
foreach (string s in result)
    Console.WriteLine("    " + s);

// All reports for C.
Console.WriteLine("\nAll reports for C:");
result = ExecuteNodeQuery(driver,
    "MATCH " +
    "    (n:OrgNode)-[:REPORTS_TO *]->(a:OrgNode { ID:'C' }) " +
    "return n " +
    "ORDER BY n.ID");
foreach (string s in result)
    Console.WriteLine("    " + s);
}

```

This method first calls the `ExecuteNodeQuery` method to execute the following query:

```
MATCH (n:OrgNode { ID:'F' }) return n
```

This simply finds the node with `ID` equal to `F`. The code prints it.

Next, the method looks for nodes that have the `REPORTS_TO` relationship ending with node `B`. That returns all of the nodes that report directly to node `B`. The code loops through the results displaying them.

The method then uses the `FindPath` method to find a path from node `H` to node `A`. Node `A` is at the top of the org chart, so this includes all the nodes in the chain of command from node `H` to the top.

The last query the method performs matches the following:

```
(n:OrgNode)-[:REPORTS_TO *]->(a:OrgNode { ID:'C' })
```

This finds nodes `n` that are related via any number (`*`) of `REPORTS_TO` relationships to node `C`. That includes all the nodes that report directly or indirectly to node `C`. Graphically, those are the nodes that lie below node `C` in the org chart.

Main

The previous methods make working with the org chart fairly simple. All we need to do now is get them started.

The following code shows the main program:

```
static void Main(string[] args)
{
    // Replace the following with your database URI, username, and password.
    string uri = "neo4j+s://386baeab.databases.neo4j.io";
    string user = "neo4j";
    string password = "InsertYourReallyLongAndSecurePasswordHere";

    // Create the driver.
    using (IDriver driver = GraphDatabase.Driver(uri,
        AuthTokens.Basic(user, password)))
    {
        // Delete any previous nodes and links.
        DeleteAllNodes(driver);

        // Build the org chart.
        BuildOrgChart(driver);

        // Query the org chart.
        QueryOrgChart(driver);
    }

    Console.WriteLine("\nPress Enter to quit");
    Console.ReadLine();
}
```

This code first defines the uniform resource identifier (URI) where the database is located, the username, and the password. You can find these in the credential file that you downloaded when you created the database instance. (I hope you saved that file! If you didn't, then this might be a good time to delete the database instance and start over.)

Next, the code uses the URI, username, and password to create a graph database driver. Notice that the program creates the driver in a `using` block so that it is automatically disposed of when the program is done with it.

The program then calls the `DeleteAllNodes`, `BuildOrgChart`, and `QueryOrgChart` methods to do all the interesting work.

For its grand finale, the program displays a message and then waits for you to press `Enter` so that the output doesn't flash past and disappear before you can read it.

The following shows the program's output:

```
Deleting old data...
Building org chart...
Querying org chart...
F:
```

```
F: Mgr Pie and Food Gags

Reports directly to B:
  D: Dir Puns and Knock-Knock Jokes
  E: Dir Riddles

Chain of command for H:
  H: Mgr Pratfalls
  G: Dir Physical Humor
  C: VP Shtick
  A: President

All reports for C:
  F: Mgr Pie and Food Gags
  G: Dir Physical Humor
  H: Mgr Pratfalls
  I: Dir Sight Gags

Press Enter to quit
```

Figure 21.2 shows the same org chart in Figure 21.1, so you can look at it to see that the output is correct.

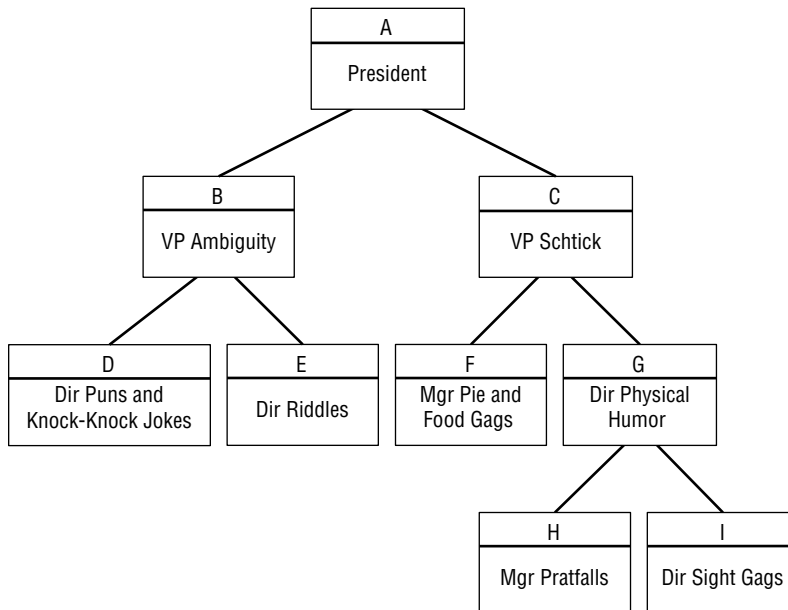


FIGURE 21.2

SUMMARY

This chapter showed how you can use C# and a NoSQL graph database to build and explore an org chart. You can use similar techniques to work with other trees and, more generally, graphs that are not trees.

As you work with this example, you might notice that operations are relatively slow, particularly if you have a slow network connection. This is generally true of cloud applications. Network communications tend to be slower than local calculations.

The pattern that this example used was to:

1. Create a database driver object.
2. Use the driver to create a session object.
3. Use the session object's `Run` method to send Cypher commands to the database engine.

The next two chapters show how to use a NoSQL document database in the cloud. Chapter 22 shows a Python example and Chapter 23 shows a similar example in C#. Before you move on to those chapters, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

EXERCISES

1. In general (not specifically in the example org chart), what does the following statement match in a tree?

```
MATCH
  (a:OrgNode { ID: 'E' }) - [:REPORTS_TO] -> (b:OrgNode)
  <- [:REPORTS_TO] - (n:OrgNode)
return n
ORDER BY n.ID
```

2. What nodes does the statement in question 1 return for the org chart shown in Figure 21.2?
3. In general, what does the following query return?

```
MATCH
  (a:OrgNode) - [:REPORTS_TO] -> (b:OrgNode) <- [:REPORTS_TO] - (c:OrgNode)
return a, c
```

4. What values does the statement in Question 3 return for the org chart shown in Figure 21.2?

-
5. In general, what does the following `MATCH` statement return?

```
MATCH
  (:OrgNode) - [r1:REPORTS_TO] -> (:OrgNode)
  - [r2: REPORTS_TO] -> (:OrgNode)
  - [r3: REPORTS_TO] -> (:OrgNode)
return r1, r2, r3
```

6. What values does the statement in Question 5 return for the org chart shown in Figure 21.2?
-
7. Write a `MATCH` statement that returns node A's grandchildren.
-

22

MongoDB Atlas in Python

This chapter's example uses Python to build a NoSQL document database in the cloud. It uses the MongoDB Atlas database to save and query the following data about assignments for East Los Angeles Space Academy graduates, as shown in Table 22.1.

TABLE 22.1: Graduate assignments

FIRSTNAME	LASTNAME	POSITION	RANK	SHIP
Joshua	Ash	Fuse Tender	6th Class	Frieda's Glory
Sally	Barker	Pilot		Scrat
Sally	Barker	Arms Master		Scrat
Bil	Cilantro	Cook's Mate		Scrat
Al	Farnsworth	Diplomat	Hall Monitor	Frieda's Glory
Al	Farnsworth	Interpreter		Frieda's Glory
Major	Major	Cook's Mate	Major	Athena Ascendant
Bud	Pickover	Captain	Captain	Athena Ascendant

The next few sections talk a bit about this data, the way document databases store information, and the MongoDB Atlas database that we'll use for this example. The rest of the chapter describes the example program.

NOT NORMAL BUT NOT ABNORMAL

If you remember the relational database normalization rules, then you know that the data in the preceding table is not normalized because some non-key fields do not depend on all the key fields. Here, the key fields are `FirstName`, `LastName`, and `Position`, because we need all three to identify a record. However, the fields `Rank` and `Ship` depend on `FirstName` and `LastName` but not on `Position`.

You can also see the problem by looking for anomalies. For example, if you change one of Sally Barker's `Ship` values, then her ship assignments will be inconsistent (an update anomaly). In fact, Al Farnsworth's `Rank` assignments already don't match.

For another example, if you delete the last record to remove the information about Bud Pickover, then you also lose the fact that the ship *Athena Ascendant* exists (a deletion anomaly).

This data won't fit in a relational database without some revisions, but it's no problem for a document database. That kind of database stores each document separately, so it doesn't matter if the documents don't hold exactly the same types of data. That does mean the data may be subject to some anomalies, but that's up to the way you (and your program) manage the data.

XML, JSON, AND BSON

Document databases do not have table definitions, so they need some other way to keep track of what "fields" are in a document. They do that by using formats that include both the field names and their values.

For example, the following shows how we might store the first record in an *Extensible Markup Language (XML)* format document:

```
<posting>
  <FirstName>Joshua</FirstName>
  <LastName>Ash</LastName>
  <Position>Fuse Tender</Position>
  <Rank>6th Class</Rank>
  <Ship>Frieda's Glory</Ship>
</posting>
```

Each of the values inside the open and close tokens is an element of whatever object contains it. For example, `Joshua` is the value of this posting's `FirstName` element.

XML also supports attributes that you can use to save some space. Here's the same data in attribute form:

```
<posting
  FirstName = "Joshua"
  LastName = "Ash"
  Position = "Fuse Tender"
  Rank = "6th Class"
  Ship = "Frieda's Glory"/>
```

This works, but the first version allows you to create repeated elements and this version doesn't. For example, Sally Barker has two positions. You can include those as separate `<Position>` elements in the first version, but you cannot give the `posting` element two `Position` properties in the second version.

JavaScript Object Notation (JSON) is another format for storing documents that's currently more popular with document databases.

The following text shows a JSON version of Sally Barker's data:

```
{
  "FirstName": "Sally",
  "LastName": "Barker",
  "Position": "Pilot",
  "Position": "Arms Master",
  "Ship": "Scrat"
}
```

This version is slightly more compact than the first XML version because it doesn't require closing elements such as `</FirstName>`. Notice that it contains multiple `Position` elements. (Just to be clear, the first XML version can also contain repeated elements.)

Notice also that Sally's data doesn't include a `Rank` because it wasn't included in the original data. That's a feature of document databases: each document sets its own rules.

Some databases let you define a database schema that defines rules about what a document must contain. For example, you could require that `posting` documents include a `Rank` value, but you aren't required to do that.

Behind the scenes MongoDB stores data in a binary version of JSON called *Binary JavaScript Object Notation (BSON)* to save space, to better encode some data types such as dates and binary values, and because it's a bit faster for some operations.

PERFECT PRONUNCIATION

Douglas Crockford, who named JSON, says it should be pronounced like the name "Jason," and the international standards ECMA-404 and ISO/IEC 21778:2017 agree. Still, many developers pronounce it "jay-sahn" for some unknowable reason.

Wikipedia says that BSON is pronounced "bee-son," but some developers think "bison" sounds better. I think "bosun" or "boson" would be more amusing.

It doesn't really matter, but I suspect there are people who will fight to the death to defend their pronunciation.

MongoDB stores documents in BSON format, but your database adapter will probably convert it to JSON when you display it. However, it may not observe niceties such as adding line breaks, indentation, and other formatting. For example, the following code shows how the Python statement `print(record)` displays Joshua Ash's data:

```
{'_id': ObjectId('62e2d70c359cf3cfda8deb21'), 'FirstName': 'Joshua',  
  'LastName': 'Ash', 'Position': 'Fuse Tender', 'Rank': '6th Class',  
  'Ship': "Frieda's Glory"}
```

This is pretty hard to read, so you may want to write some code to pull out the record's fields and display them in a more reasonable format.

INSTALL MongoDB ATLAS

MongoDB Atlas is a cloud database. To get started, go to www.mongodb.com/cloud/atlas/register and create a free user account. When you create the account, you will receive an email containing a button that you must click to verify your email address (possibly so bots can't create hordes of free Atlas accounts, or possibly just so MongoDB can send you advertising later).

WARNING *Paranoid ad/spam blocker software may balk at the `MongoDB.com` website. If yours does, you may need to whitelist that site.*

NOTE *During the few weeks while I experimented with Atlas, the steps for creating a new database instance changed. If you see screens different from those described here, just do your best. They probably won't be too hard to figure out.*

After you successfully log in, select the Database category on the left and click the big Build a Database button shown in Figure 22.1.

The next page gives you three options for the kind of service that you want:

- **Serverless**—With this option, you pay per transaction, currently starting at \$0.10 per million reads. This is a good choice if your workload varies throughout the day.
- **Dedicated**—This option gives you more advanced configuration options and a set cost, currently \$0.08 per hour regardless of the number of transactions. It's designed for large production systems.
- **Shared**—This is the free “try it out” version. It has a limit on the number of connections that you can use, uses shared memory, has limited storage, and is generally perfect for this example.

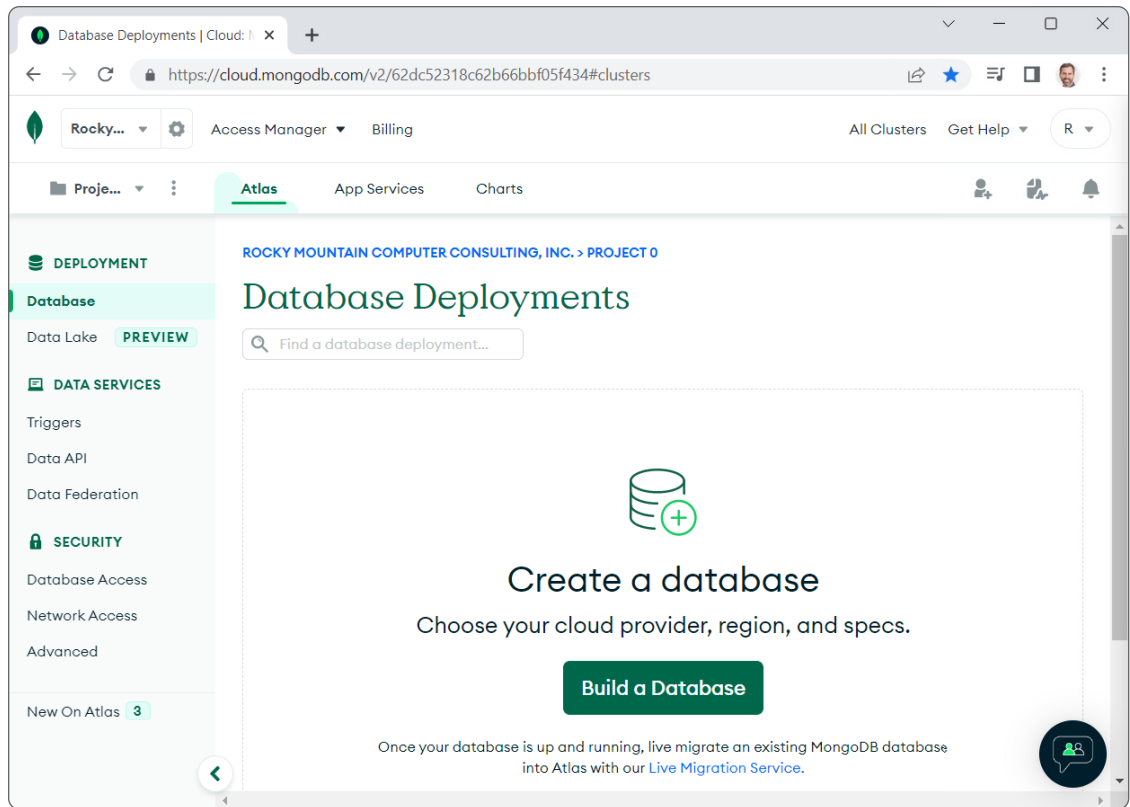


FIGURE 22.1

After you click the Shared option's Create button, you'll see a page where you can pick a cloud provider to host the database and a region. I'm using Amazon Web Services (AWS) as the provider for this example, so click the big AWS button.

Below the provider selection buttons is a list of regions that you can select to locate your server. It may be tempting to set your host location to an exotic place like Bahrain or Tokyo, but the server won't send you postcards or anything, so just pick the one that's physically closest to you. In a real application, you may be required to pick a particular location to satisfy data sovereignty rules (mentioned way back in Chapter 1, "Database Design Goals"), such as storing personal customer data within your country's borders.

If you scroll to the bottom, you can expand the Cluster Name area and enter a name for your cluster. If you don't, you'll get the accurate but uninspired name Cluster0 and you can't change the name later. I'm going to use the name PostingsDB.

After you pick your provider and optionally set the cluster name, click Create Cluster.

Next, you decide how to connect with the database. For this example, select the Username and Password options. Enter a username and password in the indicated boxes. You can use the Autogenerate Secure Password button to make the system pick a good password for you. If you do that, click Copy to copy the password to the clipboard and paste it somewhere safe. After you've set the username and password, click Create User.

Scroll down to the next section, where you indicate how you would like to access the database. The choices are My Local Environment (run the database on your computer) and Cloud Environment (run the database in the cloud). Pick Cloud Environment.

When you're finished defining the database, click Finish and Close. In the congratulatory dialog box, click Go to Databases.

TIP *If you haven't already done so, this would be a good time to bookmark the web page so you can easily get back to your cluster information.*

FIND THE CONNECTION CODE

Connecting to most databases isn't hard, but it can be annoyingly difficult to find out exactly what syntax and connection data to use. For example, how do you specify the database's location, a username, and the password? I'm glad you asked!

MongoDB's database deployments page gives you a handy tool for figuring out how to do that. If you look at Figure 22.2, you'll see a row of buttons for the cluster named PostingsDB. Click Connect to display the dialog box shown in Figure 22.3. Now, click the second choice, Connect Your Application, to see the page shown in Figure 22.4.

For this example, select Python from the left drop-down list. (For Chapter 23, "MongoDB Atlas in C#," select C#/NET.) Then from the right drop-down list, select the appropriate PyMongo driver version. If you haven't installed PyMongo yet (it would be surprising if you have), then you'll probably want the most recent version.

If you're not sure which version to use, install the database adapter (I'll show you instructions shortly), and then run the following code in a Jupyter Notebook cell:

```
# Get the pymongo version.
import pymongo
pymongo.version
```

When I ran this code, Jupyter Notebook told me that I had PyMongo version 4.2.0 installed, so I selected the newest version from the drop-down list, which was version 3.12 or later (as you can see in Figure 22.4).

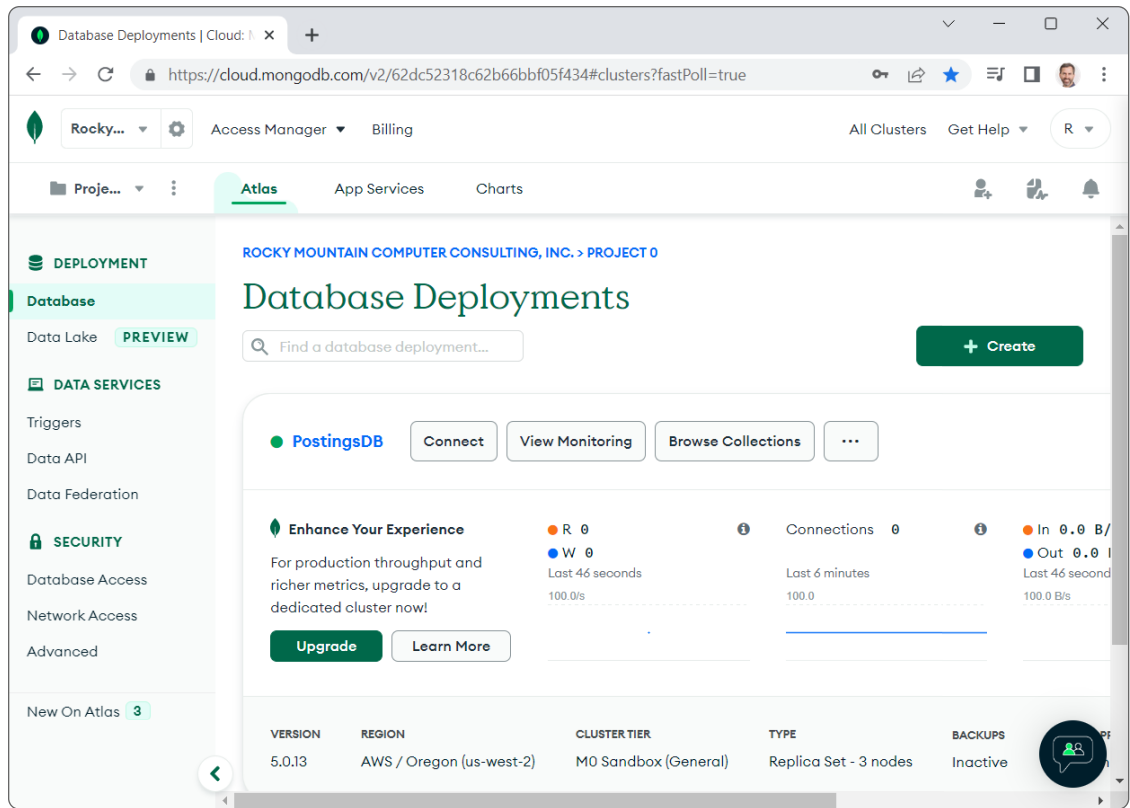


FIGURE 22.2

When you pick the driver version, the page shows a connect string that you can use to connect your program to the database. If you select Include Full Driver Code Example, as shown in Figure 22.4, then you'll see a complete Python code snippet that you can copy and paste into your program. There's even a button to the right that copies the code to the clipboard for you. Copy that value and paste it somewhere safe so that you can use it later.

CRAZY CONNECTIONS

When I first tested it, the example program sometimes timed out on a domain name system (DNS) operation while trying to connect to the database. The program threw a `LifetimeTimeout` exception with the additional detail "The DNS operation timed out" (plus a lot of other gibberish that was less useful).

I was using a slower network at the time, so that might have been the problem. After much searching on the Internet, I found a workaround that fixed the problem for me.

continues

(continued)

Instead of using the latest connect string for data adapter version 3.12 and later, I used an older format intended to work with data adapter version 3.4 and later. The following text shows the annoyingly long version 3.4 connect string:

```
mongodb://Rod:KENDmaHmdhK23Kn3@
ac-amdtfbf-shard-00-00.b1bprz5.mongodb.net:27017,
ac-amdtfbf-shard-00-01.b1bprz5.mongodb.net:27017,
ac-amdtfbf-shard-00-02.b1bprz5.mongodb.net:27017/
?ssl=true&replicaSet=atlas-uugor6-shard-0&authSource=admin
&retryWrites=true&w=majority
```

This might not work for you, but if you have trouble with DNS timeouts, you might give it a try or try some other, older connect string format.

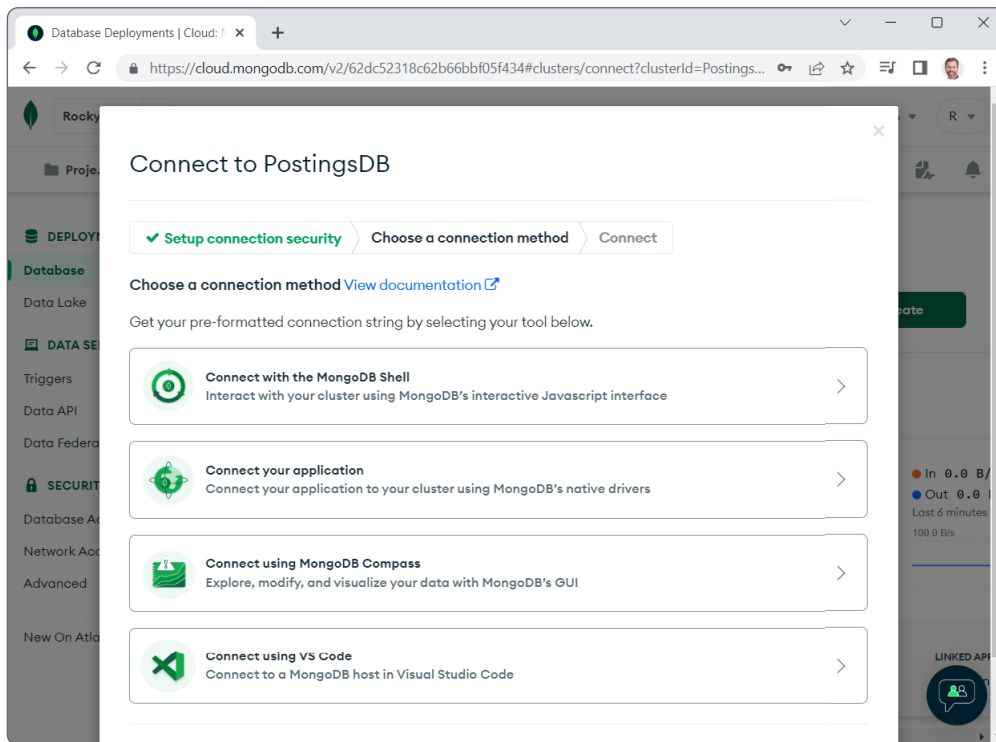


FIGURE 22.3

Notice that the snippet includes the password as `<password>`. You need to replace that (including the brackets) with the actual password. For this example we'll be lazy and include the username and password in the program, but it's always best to not include real credentials in your program's code. Ask the user for a username and password and insert them into the connection string at runtime.

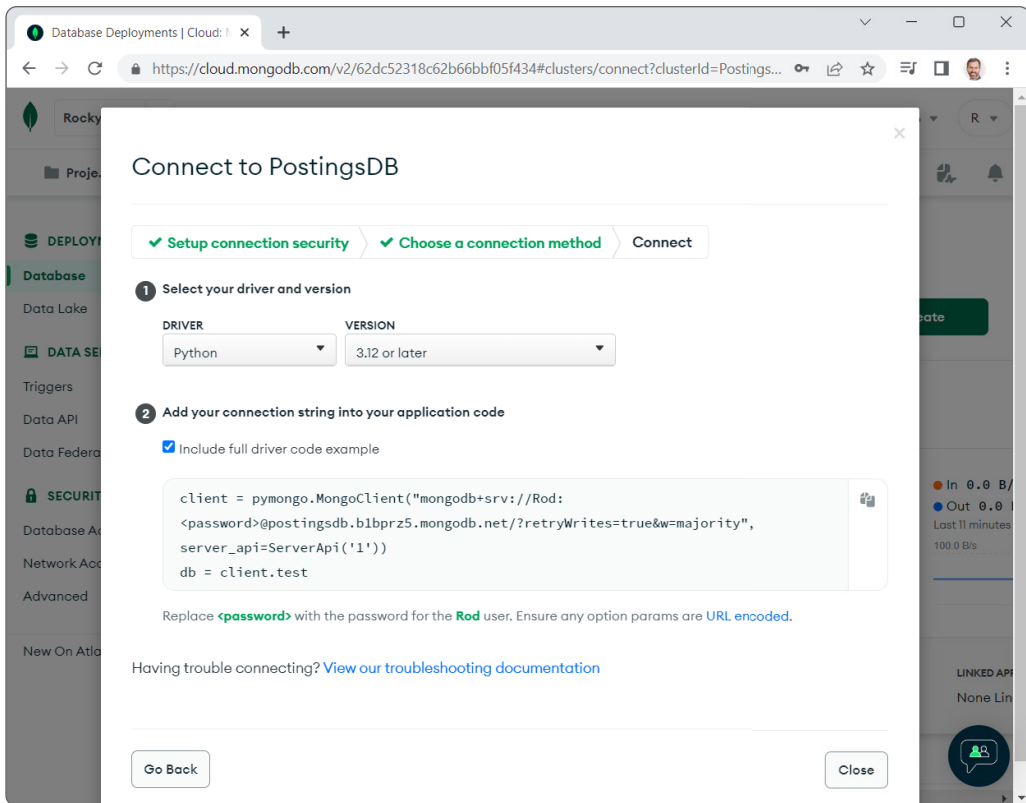


FIGURE 22.4

CREATE THE PROGRAM

Now that the database is waiting for you in the cloud, you need to install a database driver for it. Then you can start writing code.

To create a Python program to work with the Atlas ship assignment database, create a new Jupyter Notebook, and then add the code described in the following sections.

Install the PyMongo Database Adapter

The PyMongo driver is the official MongoDB driver for synchronous Atlas applications in Python. To install this driver, simply use the following `pip` command:

```
$ pip install pymongo[srv]
```

If you're not familiar with `pip`, you can execute the following command in a Jupyter Notebook cell instead:

```
!pip install pymongo[srv]
```

If you use a Notebook cell to install PyMongo, then you'll need to restart the kernel to use it. Use the Kernel menu's Restart command or click the circular Restart button. That's all there is to it!

To build the example program, create a new Jupyter Notebook and enter the following code into the first cell:

```
# Prepare to use PyMongo.
import pymongo
from pymongo import MongoClient
from pymongo.server_api import ServerApi
```

This just gets you ready to use the MongoDB client tools and the server API.

The following sections describe some helper methods that the program will use. The last section in this part of the chapter describes the main program.

Helper Methods

These methods do all of the program's real work. The main program just connects to the database and then calls them.

Table 22.2 lists the helper methods and gives their purposes.

TABLE 22.2: Helper methods and their purposes

METHOD	PURPOSE
<code>person_string</code>	Formats a string for a document
<code>connect_to_db</code>	Connects to the database
<code>delete_old_data</code>	Deletes any old data from the database
<code>create_data</code>	Adds documents to the database
<code>query_data</code>	Displays a header and the records in the postings collection

The following sections describe these helper methods.

person_string

The following `person_string` method returns a string holding the first name, last name, ship, rank, and position for a person's record formatted so that it will line up nicely in the final output:

```
# Return a string for a person's record.
Def person_string(record):
    # These should always be present.
    Name = record["FirstName"] + " " + record["LastName"]
```

```

ship = record["Ship"]

# This might be a single value or a list.
Position = record["Position"]
if isinstance(position, list):
    position = ", ".join(record["Position"])

# Rank may be missing.
If "Rank" in record:
    rank = record["Rank"]
else:
    rank = "---"

return f"{name:16}{ship:19}{rank:15}{position}"

```

The code first gets the `FirstName` and `LastName` fields from the record and concatenates them. It also gets the `Ship` value.

Because this kind of database doesn't have a fixed format for its records the way a relational database does, a field can hold more than one kind of information. In this example, Joshua Ash has one `Position` value (Fuse Tender), but Sally Barker has two (Pilot and Arms Master).

The database doesn't care what values are in a document; your program is responsible for dealing with this issue. That's why the method gets the `Position` value and then uses Python's `isinstance` function to see if that value is a list. If it is a list, then the code joins the position values, separating them with commas, and saves the resulting string in the `position` variable.

If the `Position` value is not a list, then the `position` variable already holds the value, so the program leaves it alone.

Just as the database driver doesn't care if different documents store different values in a field, it also doesn't care if a field is missing. If the code tries to access a value that isn't there, it will throw a `KeyError`.

To protect against that, the code uses the statement `if 'Rank' in record` to see if the `Rank` value is present in this document. If the value is there, then the method gets it. If the value is missing, then the method uses three dashes in its place. (The program assumes that the `FirstName`, `LastName`, `Ship` and `Position` fields are present in all records. If this isn't the case, then you need to use `if ... in record` to see if they are present.)

Having gathered all the values that it needs into variables, the method composes the person's string result and returns it. For example, the following shows the result for Sally Barker:

```
Sally Barker      Scrat                ---                Pilot, Arms Master
```

connect_to_db

The following code shows how the `connect_to_db` method connects to the database:

```

# Connect to the database.
def connect_to_db(user, password, url):

```

```
global client, db

connect_string = \
    f"mongodb+srv://{user}:{password}@{url}?retryWrites=true&w=majority"
client = MongoClient(connect_string, server_api=ServerApi('1'))
db = client.personnel
```

The method begins with the following `global` statement:

```
global client, db
```

This allows pieces of code outside of this method to use the variables `client` and `db` later so that they can manipulate the database.

The method takes a username, password, and URL as parameters. It uses them to compose a connect string (using the format provided by the web page), and uses that string to create a `MongoClient` that connects to the cluster.

The code then uses the statement `db = client.personnel` to set `MongoClient` to connect to the `personnel` database. If that database does not exist, this creates it. You can also access a collection with the syntax `db = client["personnel"]`.

delete_old_data

The following code shows the `delete_old_data` method:

```
# Delete any old data.
def delete_old_data(db):
    db.postings.delete_many({})

    print("Deleted old data\n")
```

The code `db.postings` accesses the `postings` collection in the `db` database.

In MongoDB, a collection is somewhat analogous to a table in a relational database except that it holds documents instead of records. This example uses the `postings` collection.

The code adds a call to `delete_many` to delete multiple documents from the collection. The empty curly braces `{}` pass the method an empty dictionary of things to match, so `delete_many` matches every document in the collection and deletes them all.

The method finishes by displaying a message just so you know that it did something.

create_data

The following code shows how the `create_data` method adds documents to the `postings` collection:

```
# Create sample data.
def create_data(db):
    josh = {
        "FirstName" : "Joshua",
        "LastName" : "Ash",
        "Position" : "Fuse Tender",
```

```

        "Rank" : "6th Class",
        "Ship" : "Frieda's Glory"
    }
    sally = {
        "FirstName" : "Sally",
        "LastName" : "Barker",
        "Position" : [ "Pilot", "Arms Master"],
        "Ship" : "Scrat"
    }
    bil = {
        "FirstName" : "Bil",
        "LastName" : "Cumin",
        "Position" : "Cook's Mate",
        "Ship" : "Scrat",
        "Rank" : "Hall Monitor"
    }
    al = {
        "FirstName" : "Al",
        "LastName" : "Farnsworth",
        "Position" : [ "Diplomat", "Interpreter" ],
        "Ship" : "Frieda's Glory"
    }
    major = {
        "FirstName" : "Major",
        "LastName" : "Major",
        "Position" : "Cook's Mate",
        "Rank" : "Major",
        "Ship" : "Athena Ascendant"
    }
    bud = {
        "FirstName" : "Bud",
        "LastName" : "Pickover",
        "Position" : "Captain",
        "Rank" : "Captain",
        "Ship" : "Athena Ascendant"
    }

# Insert josh individually.
result = db.postings.insert_one(josh)

# Insert the others as a group.
others = [sally, bil, al, major, bud]
result = db.postings.insert_many(others)

print("Created data\n")

```

The method first creates several dictionaries to hold data for the people that it will create. Notice that each of these defines its data exactly the same way a JSON file does.

The `Position` value is a string in Joshua's data, but it's a list of strings in Sally's data. Sally's data also does not include a `Rank`. You can look through the code if you like to verify that it defines the data shown in the table at the beginning of the chapter.

After it defines the dictionaries, the code executes the following code to insert a single document into the collection:

```
# Insert josh individually.
result = db.postings.insert_one(josh)
```

It then executes the following code to show how you can insert multiple documents all at once:

```
# Insert the others as a group.
others = [sally, bil, al, major, bud]
result = db.postings.insert_many(others)
```

In general, performing many operations with fewer commands saves network bandwidth, which can be important in a cloud application.

The method finishes by displaying a message so you know it did something.

query_data

The `query_data` method displays a header and then repeats the same pattern several times. It uses the `find` method to find documents that match a pattern, and then loops through the results to display information about the documents that were returned.

The following shows the pieces of the method one operation at a time. I'll show you the whole method in one piece at the end of this section.

Here's the first piece of the method:

```
# Query the data.
def query_data(db):
    print(f"{'Name':16}{'Ship':19}{'Rank':15}{'Position'}")
    print("-----")

    # List everyone.
    print("    *** Everyone ***")
    cursor = db.postings.find({})
    for doc in cursor:
        print(person_string(doc))
```

The method first calls `find`, passing it the empty dictionary `{}`. That gives `find` no values to match, so it returns every document in the `postings` collection. The code loops through the results and calls `person_string` to print each record's data.

Here's the second operation:

```
# People posted to Scrat.
print("\n    *** Assigned to Scrat ***")
cursor = db.postings.find({"Ship": "Scrat"})
for doc in cursor:
    print(person_string(doc))
```


This code passes the dictionary `{"Ship": "Scrat"}` to the `find` method. This makes `find` match documents that have the `Ship` property equal to `Scrat`. Again the code loops through the result to display the data for those assigned to the `Scrat`.

Next, the method uses the following code to find documents that have a `Rank` value:

```
# People with Rank values.
print("\n    *** Has a Rank ***")
cursor = db.postings.find({"Rank": {"$exists": True}})
for doc in cursor:
    print(person_string(doc))
```

The code then uses the dictionary `{"Rank": {"$exists": True}}` to find documents that have a `Rank` property. In this dictionary, the value of the `Rank` property is another dictionary. For each record, the inner dictionary invokes the database function `$exists` on the `Rank` property. If that value is `True`, then the document matches and it is added to the results.

The program then repeats that query, except this time it looks for documents that do not have a `Rank` property:

```
# People with no Rank.
print("\n    *** Has no Rank ***")
cursor = db.postings.find({"Rank": {"$exists": False}})
for doc in cursor:
    print(person_string(doc))
```

The next piece of code shows how you can match multiple criteria:

```
# Cook's Mates or people on Frieda's Glory.
print("\n    *** Cook's Mates or on Frieda's Glory ***")
cursor = db.postings.find(
    {"$or": [
        {"Position": "Cook's Mate"},
        {"Ship": "Frieda's Glory"}
    ]})
for doc in cursor:
    print(person_string(doc))
```

This time the query dictionary's single entry has a key that invokes the database's `$or` operator. That operator takes as a value a list of dictionaries that define conditions. If a document matches any of those conditions, then it is returned in the result. This part of the example searches for documents where the `Position` value is `Cook's Mate` or the `Ship` value is `Frieda's Glory`.

The database also understands the logical operators `$and`, `$not`, and `$nor`. The `$nor` operator is a combination of `$not` and `$or`, so it returns documents that do *not* match any of the conditions in the following array.

The next piece of code repeats the previous selection and then sorts the result:

```
# Cook's Mates or on Frieda's Glory, sorted by Ship.
print("\n    *** Cook's Mates or on Frieda's Glory, sorted ***")
```

```
cursor = db.postings.find(
    {"$or": [
        {"Position": "Cook's Mate"},
        {"Ship": "Frieda's Glory"}
    ]}).sort("Ship", pymongo.ASCENDING)
for doc in cursor:
    print(person_string(doc))
```

This code selects documents where the `Position` value is `Cook's Mate` or the `Ship` value is `Frieda's Glory`. It then calls the cursor's `sort` method to sort the results by the `Ship` value. The `sort` method sorts the data in the original cursor and returns a new cursor holding the sorted data.

The last piece of code repeats this query and then sorts by `Ship` and `FirstName`:

```
# Cook's Mates or on Frieda's Glory, sorted by Ship then FirstName.
print("\n    *** Cook's Mates or on Frieda's Glory, sorted ***")
cursor = db.postings.find(
    {"$or": [
        {"Position": "Cook's Mate"},
        {"Ship": "Frieda's Glory"}
    ]}).sort([
        ("Ship", pymongo.ASCENDING),
        ("FirstName", pymongo.ASCENDING)
    ])
for doc in cursor:
    print(person_string(doc))
```

This code matches documents where the `Position` value is `Cook's Mate` or the `Ship` value is `Frieda's Glory` as before. It then sorts the results first by `Ship` and then by `FirstName`. In this data set, Al Farnsworth and Joshua Ash are both assigned to `Frieda's Glory`, and this sort puts Al's document first.

The following code shows the whole method in one piece:

```
# Query the data.
def query_data(db):
    print(f"{'Name':16}{'Ship':19}{'Rank':15}{'Position'}")
    print("-----")

# List everyone.
print("    *** Everyone ***")
cursor = db.postings.find({})
for doc in cursor:
    print(person_string(doc))

# People posted to Scrat.
print("\n    *** Assigned to Scrat ***")
cursor = db.postings.find({"Ship": "Scrat"})
```

```

for doc in cursor:
    print(person_string(doc))

# People with Rank values.
print("\n    *** Has a Rank ***")
cursor = db.postings.find({"Rank": {"$exists": True}})
for doc in cursor:
    print(person_string(doc))

# People with no Rank.
print("\n    *** Has no Rank ***")
cursor = db.postings.find({"Rank": {"$exists": False}})
for doc in cursor:
    print(person_string(doc))

# Cook's Mates or people on Frieda's Glory.
print("\n    *** Cook's Mates or on Frieda's Glory ***")
cursor = db.postings.find(
    {"$or": [
        {"Position": "Cook's Mate"},
        {"Ship": "Frieda's Glory"}
    ]})
for doc in cursor:
    print(person_string(doc))

# Cook's Mates or on Frieda's Glory, sorted by Ship.
print("\n    *** Cook's Mates or on Frieda's Glory, sorted ***")
cursor = db.postings.find(
    {"$or": [
        {"Position": "Cook's Mate"},
        {"Ship": "Frieda's Glory"}
    ]}).sort("Ship", pymongo.ASCENDING)
for doc in cursor:
    print(person_string(doc))

# Cook's Mates or on Frieda's Glory, sorted by Ship then FirstName.
print("\n    *** Cook's Mates or on Frieda's Glory, sorted ***")
cursor = db.postings.find(
    {"$or": [
        {"Position": "Cook's Mate"},
        {"Ship": "Frieda's Glory"}
    ]}).sort([
        ("Ship", pymongo.ASCENDING),
        ("FirstName", pymongo.ASCENDING)
    ])
for doc in cursor:
    print(person_string(doc))

```

As a reminder, Table 22.3 contains the data inserted by the `create_data` method.

TABLE 22.3: Data inserted by the `create_data` method

FIRSTNAME	LASTNAME	POSITION	RANK	SHIP
Joshua	Ash	Fuse Tender	6th Class	Frieda's Glory
Sally	Barker	Pilot		Scrat
Sally	Barker	Arms Master		Scrat
Bil	Cilantro	Cook's Mate		Scrat
Al	Farnsworth	Diplomat	Hall Monitor	Frieda's Glory
Al	Farnsworth	Interpreter		Frieda's Glory
Major	Major	Cook's Mate	Major	Athena Ascendant
Bud	Pickover	Captain	Captain	Athena Ascendant

Finally, here's the program's output:

```
Deleted old data
```

```
Created data
```

```
Name           Ship           Rank           Position
-----
*** Everyone ***
Joshua Ash     Frieda's Glory 6th Class     Fuse Tender
Sally Barker   Scrat          ---           Pilot, Arms Master
Bil Cilantro   Scrat          Hall Monitor  Cook's Mate
Al Farnsworth  Frieda's Glory ---           Diplomat, Interpreter
Major Major    Athena Ascendant Major          Cook's Mate
Bud Pickover   Athena Ascendant Captain        Captain

*** Assigned to Scrat ***
Sally Barker   Scrat          ---           Pilot, Arms Master
Bil Cilantro   Scrat          Hall Monitor  Cook's Mate

*** Has a Rank ***
Joshua Ash     Frieda's Glory 6th Class     Fuse Tender
Bil Cilantro   Scrat          Hall Monitor  Cook's Mate
Major Major    Athena Ascendant Major          Cook's Mate
Bud Pickover   Athena Ascendant Captain        Captain

*** Has no Rank ***
Sally Barker   Scrat          ---           Pilot, Arms Master
```

```

Al Farnsworth   Frieda's Glory   ---           Diplomat, Interpreter

    *** Cook's Mates or on Frieda's Glory ***
Joshua Ash     Frieda's Glory   6th Class    Fuse Tender
Bil Cilantro   Scrat           Hall Monitor  Cook's Mate
Al Farnsworth  Frieda's Glory   ---           Diplomat, Interpreter
Major Major    Athena Ascendant Major         Cook's Mate

    *** Cook's Mates or on Frieda's Glory, sorted ***
Major Major    Athena Ascendant Major         Cook's Mate
Joshua Ash     Frieda's Glory   6th Class    Fuse Tender
Al Farnsworth  Frieda's Glory   ---           Diplomat, Interpreter
Bil Cilantro   Scrat           Hall Monitor  Cook's Mate

    *** Cook's Mates or on Frieda's Glory, sorted ***
Major Major    Athena Ascendant Major         Cook's Mate
Al Farnsworth  Frieda's Glory   ---           Diplomat, Interpreter
Joshua Ash     Frieda's Glory   6th Class    Fuse Tender
Bil Cilantro   Scrat           Hall Monitor  Cook's Mate

```

You can look through the output to see how the different `find` statements worked. The first statement used an empty dictionary, so `find` returned every document. Notice how the `person_string` method displayed three dashes for missing `Rank` values and how it concatenated values when a document had multiple `Position` values.

The next `find` picked out the documents where `Ship` is `Scrat`.

The two after that found documents that had a `Rank` value and that did not have a `Rank` value, respectively.

The first statement that used `$or` found information about Cook's Mates and those assigned to Frieda's Glory. (Admittedly, that's a combination that you might never find useful in a real program.)

The next example sorts those results by `Ship`. The final `find` sorts by `Ship` first and then `FirstName`, so Al Farnsworth comes before Joshua Ash.

Main Program

Compared to the previous methods, the main program is relatively simple:

```

# Main program.

# Connect to the database.
user = "Rod"
password = "KENDmaHmdhK23Kn"
url = "postingsdb.b1bprz5.mongodb.net"
connect_to_db(user, password, url)

# Delete old data.
delete_old_data(db)

```

```
# Create new data.
create_data(db)

# Query the data.
query_data(db)

# Close the database connection.
client.close()
```

This code stores the username, password, and database URL in variables. You should replace those values with the ones you got in the “Find the Connection Code” section earlier in this chapter.

The code calls the `connect_to_db` method to connect to the database and initialize the `db` variable. It then calls the `delete_old_data`, `create_data`, and `query_data` methods to do the interesting work. Finally, it closes the database connection.

SUMMARY

This chapter shows how you can use Python and a NoSQL document database to store and retrieve JBON documents.

As you work with this example, you might notice that operations are relatively slow, particularly if you have a slow network connection. This is generally true of cloud applications. Network communications tend to be slower than local calculations.

The PyMongo database driver uses dictionary objects to define data. You build a dictionary containing field names as their values. Then you can use the database collection’s methods such as `insert_one` and `insert_many` to add documents to the database. Later, you can use the `find` method to find those documents.

The next chapter shows how to build a similar example program in C#. That example uses a very different method for finding documents.

Before you move on to that example, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

EXERCISES

Note that the syntax for some of these is very picky and if you omit quotes here or add the wrong brackets there, then they won’t work. These exercises use some syntax that I haven’t covered in the previous sections, so don’t get too frustrated if you can’t get them to work. Try a few times and, if you get stuck, search online for more information or check the answers in Appendix A.

1. Write a `find` statement that selects documents for people posted to the Scrat who are not Cook’s Mates. Which documents does this select?

Hint: Use `$and` and `$ne` (not equals). The somewhat awkward syntax for `$ne` is:

```
{"FieldName": {"$ne": "ValueToNotMatch"}}
```

-
2. Suppose you add some new documents to the `postings` collection and they have a `MiddleInitial` value. Now assume that the program is using a variable of type `BsonDocument` and named `doc` to iterate through the results.
-
- What issue must the program handle so that it doesn't crash when processing the old documents that don't have a `MiddleInitial` value?
 - What method would the program use to determine whether the issue is present in the document `doc`?
 - What code should the program use to save the `MiddleInitial` value (or a space) into the variable `initial`?
-
3. The first parameter to the `find` method is a dictionary specifying what documents to select. An optional second parameter indicates what fields to return from the selected documents. This should be a dictionary where the keys are field names and the values are `True` or `False` to indicate whether the field should be included. Values that are not explicitly included are excluded, and vice versa.
-

Note that you cannot use both `True` and `False` in the same query, unless one of the fields is the special field `_id`, which is the internal ID value assigned by the database. For example, you can explicitly exclude `_id` and include other values.

For this exercise, write a `find` statement that selects documents for Frieda's Glory and returns only the `FirstName` and `Position` values. Instead of using the `person_string` method to display the results, just print the returned items.

4. Operators such as `$or` and `$and` let you perform logical operations on matching fields. You can also use operators to perform comparisons. For example, you can use `$eq` to see if a field equals a value, `$gt` to see if it is greater than a value, and `$in` to see if a field is in a list. (See www.mongodb.com/docs/manual/reference/operator/query-comparison for details about these operators. Use the menu on the left to learn about other operators.)
-

The `$in` operator has this syntax:

```
{ "Field": { "$in": ["value1", "value2", ... "valueN" ] } }
```

For this exercise, write a `find` statement that uses `$in` to select documents where the person is a Pilot or a Captain. Use `person_string` to display the results.

23

MongoDB Atlas in C#

This chapter's example uses C# to build a NoSQL document database in the cloud. It uses the MongoDB Atlas database to save and query the data shown in Table 23.1 about assignments for East Los Angeles Space Academy graduates.

TABLE 23.1: Assignment data

FIRSTNAME	LASTNAME	POSITION	RANK	SHIP
Joshua	Ash	Fuse Tender	6th Class	Frieda's Glory
Sally	Barker	Pilot		Scrat
Sally	Barker	Arms Master		Scrat
Bill	Cilantro	Cook's Mate		Scrat
Al	Farnsworth	Diplomat	Hall Monitor	Frieda's Glory
Al	Farnsworth	Interpreter		Frieda's Glory
Major	Major	Cook's Mate	Major	Athena Ascendant
Bud	Pickover	Captain	Captain	Athena Ascendant

If you skipped Chapter 22, “MongoDB Atlas in Python,” which built a similar example in Python, return to that chapter and read the beginning and the first four sections, which are described in the following list:

- “Not Normal but Not Abnormal” talks a bit about data that doesn't fit well in a relational database but that does fit in a document database.

- “XML, JSON, and BSON” gives a few details about the XML, JSON, and BSON formats that are commonly used by document databases.
- “Install MongoDB Atlas” explains how to create a MongoDB Atlas database in the cloud.
- “Find the Connection Code” tells how you can find the connection code that allows your program to connect to the database.

When you reach “Create the Program” in Chapter 22, return here and start on the following sections.

CREATE THE PROGRAM

To create a C# program to work with the Atlas ship assignment database, create a new C# Console App (.NET Framework) and then add the code described in the following sections.

Now that the database is waiting for you in the cloud, you need to install a database driver for it. Then you can start writing code.

Install the MongoDB Database Adapter

This example uses the MongoDB database adapter to allow your program to communicate with the MongoDB Atlas database sitting in the cloud. To add the adapter to your project, follow these steps.

1. Open the Project menu and select Manage NuGet Packages.
2. Click the Browse tab and enter **MongoDB.Driver** in the search box. Click that entry in the list on the left.
3. Click the Install button on the right. This shouldn't take too long, but it's still nice to have a fast Internet connection.

To make using the database driver easier, add the following code to the program just below the other using statements:

```
using MongoDB.Bson;
using MongoDB.Driver;
```

The following sections describe some helper methods that the program will use. The last section in this part of the chapter describes the main program.

Helper Methods

These methods do all of the program's real work. The main program just connects to the database and then calls them.

Table 23.2 lists the helper methods and gives their purposes.

TABLE 23.2: Helper methods and their purposes

METHOD	PURPOSE
PersonString	Formats a string for a document
DeleteOldData	Deletes any old data from the database
CreateDdata	Adds documents to the database
QueryData	Displays a header and the records in the postings collection

The following sections describe these helper methods.

PersonString

The following `PersonString` method returns a string holding the first name, last name, ship, rank, and position for a person's record all formatted nicely in the final output.

```
static private string PersonString(BsonDocument doc)
{
    // These should always be present.
    string name = doc["FirstName"].AsString + " " + doc["LastName"].AsString;
    string ship = doc["Ship"].AsString;

    // This might be a single value or a list.
    string position = "";
    if (doc["Position"].IsBsonArray)
    {
        position = string.Join(", ", doc["Position"].AsBsonArray);
    }
    else
    {
        position = doc["Position"].AsString;
    }

    // Rank may be missing.
    string rank = "";
    if (doc.Contains("Rank"))
        rank = doc["Rank"].AsString;
    else
        rank = "---";

    return $"{name,-16}{ship,-19}{rank,-15}{position}";
}
```

The code first gets the `FirstName` and `LastName` fields from the record and concatenates them. Notice how it uses the `AsString` property to convert the value, which starts out as an object, into a string. That property throws an `InvalidCastException` if the value is something other than a string,

such as a number or date. Your program is responsible for knowing what kinds of values the document holds and handling them appropriately.

The code also gets the `Ship` value, again as a string.

Because this kind of database doesn't have a fixed format for its records the way a relational database does, a field can hold more than one kind of information. In this example, Joshua Ash has one `Position` value (Fuse Tender), but Sally Barker has two (Pilot and Arms Master).

The database doesn't care what values are in a document, so your program must handle the issue. That's why the method gets the `Position` value and then uses the `IsBsonArray` function to see if that value is an array. If it is an array, then the code uses `string.Join` to join the position values, separating them with commas.

If the `Position` value isn't an array, then the code just saves its value in the `position` variable.

Just as the database driver doesn't care if different documents store different values in a field, it also doesn't care if a field is missing. If the code tries to access a value that isn't there, the data adapter panics and throws a `KeyNotFoundException`.

In this example, some of the documents do not have a `Rank` value, so to protect against the `KeyNotFoundException`, the code uses the statement `if (doc.Contains("Rank"))` to see if the `Rank` value is present. If the value is there, then the method gets it. If the value is missing, the method uses three dashes in its place.

Having gathered all of the values that it needs into variables, the method composes the person's result string and returns it.

The following shows the result for Sally Barker:

```
Sally Barker   Scrat           ---           Pilot, Arms Master
```

DeleteOldData

The following code shows the `DeleteOldData` method:

```
static private void DeleteOldData(IMongoCollection<BsonDocument> collection)
{
    FilterDefinition<BsonDocument>
        deleteFilter = Builders<BsonDocument>.Filter.Empty;
    collection.DeleteMany(deleteFilter);
    Console.WriteLine("Deleted old data\n");
}
```

This method takes a collection as a parameter. In MongoDB, a collection is somewhat analogous to a table in a relational database except that it holds documents instead of records. This example uses the `postings` collection.

The code gets a special predefined filter that is empty. It passes that filter to the collection's `DeleteMany` method to delete multiple documents. Passing the empty filter into the method makes it match every document, so they are all deleted.

The method finishes by displaying a message to show that it did something.

CreateData

The following code shows how the `CreateData` method adds documents to the `postings` collection:

```
// Create sample data
static private void CreateData(IMongoCollection<BsonDocument> collection)
{
    BsonDocument josh = new BsonDocument
    {
        { "FirstName" , "Joshua" },
        { "LastName" , "Ash" },
        { "Position" , "Fuse Tender" },
        { "Rank" , "6th Class" },
        { "Ship" , "Frieda's Glory" }
    };
    BsonDocument sally = new BsonDocument
    {
        { "FirstName" , "Sally" },
        { "LastName" , "Barker" },
        { "Position" , new BsonArray { "Pilot", "Arms Master" } },
        { "Ship" , "Scrat" }
    };
    BsonDocument bil = new BsonDocument
    {
        { "FirstName" , "Bil" },
        { "LastName" , "Cumin" },
        { "Position" , "Cook's Mate" },
        { "Ship" , "Scrat" }
    };
    BsonDocument al = new BsonDocument
    {
        { "FirstName" , "Al" },
        { "LastName" , "Farnsworth" },
        { "Position" , new BsonArray { "Diplomat", "Interpreter" } },
        { "Rank" , "Hall Monitor" },
        { "Ship" , "Frieda's Glory" }
    };
    BsonDocument major = new BsonDocument
    {
        { "FirstName" , "Major" },
        { "LastName" , "Major" },
        { "Position" , "Cook's Mate" },
        { "Rank" , "Major" },
        { "Ship" , "Athena Ascendant" }
    };
    BsonDocument bud = new BsonDocument
    {
        { "FirstName" , "Bud" },
        { "LastName" , "Pickover" },
        { "Position" , "Captain" },
        { "Rank" , "Captain" },
        { "Ship" , "Athena Ascendant" }
    }
}
```

```

};

// Insert josh individually.
collection.InsertOne(josh);

// Insert the others as a group.
BsonDocument[] others = { sally, bil, al, major, bud };
collection.InsertMany(others);

Console.WriteLine("Created data\n");
}

```

The method first creates several `BsonDocument` objects to hold data for the people we will create. The initializer is similar to the one that you can use to initialize a dictionary in C#. It's also similar (although less so) to a JSON file or the way a Python program defines dictionaries. (Of course, the C# version adds some curly brackets. It's not for nothing that C# is called one of the “curly bracket languages”!)

Notice that the `Position` value is a string in Joshua's data but it's a new `BsonArray` in Sally's data. Notice also that Sally's data does not include a `Rank`.

After it defines the people documents, the code executes the following code to insert a single document into the collection:

```

// Insert josh individually.
collection.InsertOne(josh);

```

It then executes the following code to show how you can insert multiple documents all at once:

```

// Insert the others as a group.
BsonDocument[] others = { sally, bil, al, major, bud };
collection.InsertMany(others);

```

In general, performing many operations with fewer commands saves network bandwidth, which is particularly important in a cloud application.

The method finishes by displaying a message so that you know it did something.

QueryData

The `QueryData` method displays a header and then repeats the same pattern several times. It uses a Language-Integrated Query (LINQ, which is pronounced “link”) query to find documents that match a pattern and then loops through the results to display information about the documents that were returned. The following code shows how the method begins. You'll see the rest of the method shortly.

```

static private void QueryData(IMongoCollection<BsonDocument> collection)
{
    Console.WriteLine("{0,-16}{1,-19}{2,-15}{3}",
        "Name", "Ship", "Rank", "Position");
    Console.WriteLine(

```

```

    "-----");

// List everyone.
Console.WriteLine("  *** Everyone ***");
var selectAll =
    from e in collection.AsQueryable<BsonDocument>()
    select e;
foreach (BsonDocument doc in selectAll)
{
    Console.WriteLine(PersonString(doc));
}

```

LINQ is a set of tools that let you embed SQL-like statements in .NET programs. It takes a little getting used to, but it's not too bad once you get the hang of it.

UNLINQUED

Note that LINQ queries are not the only way that you can use the MongoDB driver. Other techniques include passing filter objects into the `DeleteMany` method or passing lambda expressions into a collection's `Find` method. This example mostly works with LINQ because it's relatively simple, but you can use other techniques if you prefer.

The LINQ statement in the preceding code starts with the clause `from e in collection.AsQueryable<BsonDocument>()`. This makes variable `e` represent documents chosen from the collection. There's no `where` clause (you'll see those shortly), so this matches all of the documents in the collection.

The `select e` piece of the statement tells LINQ that the statement should select the entire document `e`.

When you use a LINQ statement, you save the result in a variable, in this case called `selectAll`. LINQ doesn't actually evaluate the statement until you do something with it, such as convert it into an array or iterate over it. The `QueryData` method uses a `foreach` loop to iterate through the returned documents and display them.

The rest of the method repeats those steps but with different LINQ statements. In the next few paragraphs, I'll describe those statements without the extra code that displays their headers and the results. After I finished covering those, I'll show you the complete method.

Here's the second LINQ statement:

```

var selectScrat =
    from e in collection.AsQueryable<BsonDocument>()
    where e["Ship"] == "Scrat"
    select e;

```

The clause `where e["Ship"] == "Scrat"` makes LINQ match documents that have a `Ship` value equal to `Scrat`. This syntax is simpler than the dictionaries used by the Python example described in Chapter 22.

Here's the next `find` statement:

```
var selectHasRank =
    from e in collection.AsQueryable<BsonDocument>()
    where e["Rank"] != BsonNull.Value
    select e;
```

This statement makes a document match if its `Rank` value is not null, so it returns people who have a rank.

The following statement matches documents where the `Rank` value is null, so it returns people who have no rank:

```
var selectHasNoRank =
    from e in collection.AsQueryable<BsonDocument>()
    where e["Rank"] == BsonNull.Value
    select e;
```

So far these matches have been fairly simple, but LINQ supports more complex queries that use the logical operators `&&` (and) and `||` (or). The following statement matches documents where `Position` is `Cook's Mate` or `Ship` is `Frieda's Glory`:

```
var selectMateOrFrieda =
    from e in collection.AsQueryable<BsonDocument>()
    where e["Position"] == "Cook's Mate"
        || e["Ship"] == "Frieda's Glory"
    select e;
```

This is much easier to understand than the corresponding `find` statement used by the Python example described in Chapter 22.

The last two matches in the method repeat the preceding matches and sort the results. Here's the next one:

```
var selectMateOrFriedaSorted =
    from e in collection.AsQueryable<BsonDocument>()
    where e["Position"] == "Cook's Mate"
        || e["Ship"] == "Frieda's Glory"
    orderby e["Ship"] ascending
    select e;
```

This is similar to the previous LINQ statement but with an `orderby` clause tacked on at the end to order the selected documents sorted by their `Ship` values. Optionally, you can add the keyword `ascending` (the default) or `descending`.

The following code shows the `QueryData` method's final query:

```
var selectMateOrFriedaSorted2 =
    from e in collection.AsQueryable<BsonDocument>()
    where e["Position"] == "Cook's Mate"
        || e["Ship"] == "Frieda's Glory"
    orderby e["Ship"] ascending, e["FirstName"] ascending
    select e;
```


This time the orderby clause sorts first by Ship and then by FirstName (if two documents have the same Ship).

The following code shows the whole method in one piece:

```
static private void QueryData(IMongoCollection<BsonDocument> collection)
{
    Console.WriteLine("{0,-16}{1,-19}{2,-15}{3}",
        "Name", "Ship", "Rank", "Position");
    Console.WriteLine(
        "-----");

    // List everyone.
    Console.WriteLine("    *** Everyone ***");
    var selectAll =
        from e in collection.AsQueryable<BsonDocument>()
        select e;
    foreach (BsonDocument doc in selectAll)
    {
        Console.WriteLine(PersonString(doc));
    }

    // People posted to Scrat.
    Console.WriteLine("\n    *** Assigned to Scrat ***");
    var selectScrat =
        from e in collection.AsQueryable<BsonDocument>()
        where e["Ship"] == "Scrat"
        select e;
    foreach (BsonDocument doc in selectScrat)
    {
        Console.WriteLine(PersonString(doc));
    }

    // People with Rank values.
    Console.WriteLine("\n    *** Has a Rank ***");
    var selectHasRank =
        from e in collection.AsQueryable<BsonDocument>()
        where e["Rank"] != BsonNull.Value
        select e;
    foreach (BsonDocument doc in selectHasRank)
    {
        Console.WriteLine(PersonString(doc));
    }

    // People with no Rank.
    Console.WriteLine("\n    *** Has no Rank ***");
    var selectHasNoRank =
        from e in collection.AsQueryable<BsonDocument>()
        where e["Rank"] == BsonNull.Value
        select e;
    foreach (BsonDocument doc in selectHasNoRank)
    {
        Console.WriteLine(PersonString(doc));
    }
}
```

```

    }

    // Cook's Mates or people on Frieda's Glory.
    Console.WriteLine("\n    *** Cook's Mates or on Frieda's Glory ***");
    var selectMateOrFrieda =
        from e in collection.AsQueryable<BsonDocument>()
        where e["Position"] == "Cook's Mate"
            || e["Ship"] == "Frieda's Glory"
        select e;
    foreach (BsonDocument doc in selectMateOrFrieda)
    {
        Console.WriteLine(PersonString(doc));
    }

    // Cook's Mates or people on Frieda's Glory, sorted by Ship.
    Console.WriteLine(
        "\n    *** Cook's Mates or on Frieda's Glory, sorted ***");
    var selectMateOrFriedaSorted =
        from e in collection.AsQueryable<BsonDocument>()
        where e["Position"] == "Cook's Mate"
            || e["Ship"] == "Frieda's Glory"
        orderby e["Ship"] ascending
        select e;
    foreach (BsonDocument doc in selectMateOrFriedaSorted)
    {
        Console.WriteLine(PersonString(doc));
    }

    // Cook's Mates or people on Frieda's Glory, sorted by Ship and FirstName.
    Console.WriteLine(
        "\n    *** Cook's Mates or on Frieda's Glory, sorted ***");
    var selectMateOrFriedaSorted2 =
        from e in collection.AsQueryable<BsonDocument>()
        where e["Position"] == "Cook's Mate"
            || e["Ship"] == "Frieda's Glory"
        orderby e["Ship"] ascending, e["FirstName"] ascending
        select e;
    foreach (BsonDocument doc in selectMateOrFriedaSorted2)
    {
        Console.WriteLine(PersonString(doc));
    }
}

```

Main Program

Compared to the `QueryData` method, the main program is relatively simple:

```

Static void Main(string[] args)
{
    // Connect to MongoDB.
    String user = "Rod";
    string password = "EnterYourSecretPasswordHere";

```

```

string url = "postingsdb.b1bprz5.mongodb.net";
string connectionString =
    $"mongodb+srv://{user}:{password}@{url}?retryWrites=true&w=majority";

MongoClient client = new MongoClient(connectionString);
IMongoDatabase db = client.GetDatabase("personnel");
IMongoCollection<BsonDocument> collection =
    db.GetCollection<BsonDocument>("postings");

// Delete any existing documents.
DeleteOldData(collection);

// Create new data.
CreateData(collection);

// Query the data.
QueryData(collection);

// Don't close the database connection.
// MongoDB uses a connection pool so it will be reused as needed.

Console.WriteLine("\nPress Enter to quit.");
Console.ReadLine();
}

```

This code stores the username, password, and database URL in variables. You should replace the username and password with the values that you used when you set up the database.

The code then uses them to compose the database connect string. In this example that string is:

```

mongodb+srv://Rod:EnterYourSecretPasswordHere@postingsdb.b1bprz5.mongodb.net/
?retryWrites=true&w=majority

```

The program uses the connect string to create a new `MongoClient`, uses the client to get the `personnel` database, and uses that database to get the `postings` collection that will contain the documents.

Next, the program calls the `DeleteOldData`, `CreateData`, and `QueryData` methods to do the interesting work.

This database adapter keeps database connections in a pool so that it can reuse them when they are needed. To make the pool more efficient, the program does not close its connection; that way, it can be reused.

As a reminder, Table 23.3 contains the data inserted by the `CreateData` method.

TABLE 23.3 Data inserted by the `CreateData` method

FIRSTNAME	LASTNAME	POSITION	RANK	SHIP
Joshua	Ash	Fuse Tender	6th Class	Frieda's Glory
Sally	Barker	Pilot		Scrat
Sally	Barker	Arms Master		Scrat
Bil	Cumin	Cook's Mate		Scrat
Al	Farnsworth	Diplomat	Hall Monitor	Frieda's Glory
Al	Farnsworth	Interpreter		Frieda's Glory
Major	Major	Cook's Mate	Major	Athena Ascendant
Bud	Pickover	Captain	Captain	Athena Ascendant

Finally, here's the program's output:

```
Deleted old data
```

```
Created data
```

```
Name           Ship           Rank           Position
-----
*** Everyone ***
Joshua Ash     Frieda's Glory 6th Class     Fuse Tender
Sally Barker   Scrat          ---           Pilot, Arms Master
Bil Cumin     Scrat          ---           Cook's Mate
Al Farnsworth Frieda's Glory Hall Monitor   Diplomat, Interpreter
Major Major    Athena Ascendant Major          Cook's Mate
Bud Pickover   Athena Ascendant Captain        Captain

*** Assigned to Scrat ***
Sally Barker   Scrat          ---           Pilot, Arms Master
Bil Cumin     Scrat          ---           Cook's Mate

*** Has a Rank ***
Joshua Ash     Frieda's Glory 6th Class     Fuse Tender
Al Farnsworth Frieda's Glory Hall Monitor   Diplomat, Interpreter
Major Major    Athena Ascendant Major          Cook's Mate
Bud Pickover   Athena Ascendant Captain        Captain

*** Has no Rank ***
Sally Barker   Scrat          ---           Pilot, Arms Master
```

```

Bil Cumin      Scrat      ---      Cook's Mate

    *** Cook's Mates or on Frieda's Glory ***
Joshua Ash    Frieda's Glory  6th Class  Fuse Tender
Bil Cumin     Scrat      ---      Cook's Mate
Al Farnsworth  Frieda's Glory  Hall Monitor  Diplomat, Interpreter
Major Major   Athena Ascendant  Major      Cook's Mate

    *** Cook's Mates or on Frieda's Glory, sorted ***
Major Major   Athena Ascendant  Major      Cook's Mate
Joshua Ash    Frieda's Glory  6th Class  Fuse Tender
Al Farnsworth  Frieda's Glory  Hall Monitor  Diplomat, Interpreter
Bil Cumin     Scrat      ---      Cook's Mate

    *** Cook's Mates or on Frieda's Glory, sorted ***
Major Major   Athena Ascendant  Major      Cook's Mate
Al Farnsworth  Frieda's Glory  Hall Monitor  Diplomat, Interpreter
Joshua Ash    Frieda's Glory  6th Class  Fuse Tender
Bil Cumin     Scrat      ---      Cook's Mate

Press Enter to quit.

```

You can look through the output to see how the different LINQ queries worked. The first did not use a where clause, so it returned every document. Notice how the `PersonString` method displayed three dashes for missing Rank values and how it concatenated values when a document had multiple Position values.

The next query picked out the documents where Ship is Scrat.

The two after that found documents that had a Rank value and that did not have a Rank value, respectively.

The first statement that used `||` found information about Cook's Mates and those assigned to Frieda's Glory (admittedly a combination that you might never find useful).

The next example sorts those results by Ship. The final query sorts by Ship first and then FirstName, so Al Farnsworth comes before Joshua Ash.

SUMMARY

This chapter shows how you can use C# and a NoSQL document database to store and retrieve BSON documents. As you work with the example, you might notice that some operations are relatively slow, particularly if you have a slow network connection. This is generally true of cloud applications. Network communications tend to be slower than local calculations.

The example uses `BsonDocument` objects to define its data. It then uses collection methods such as `InsertOne` and `InsertMany` to add the data to the database. Later, it uses LINQ queries to find documents.

The next two chapters show how to use a NoSQL key-value database in the cloud. Chapter 24 shows a Python example and Chapter 25 shows a similar example in C#. Before you move on to those chapters, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

EXERCISES

1. Write a LINQ query that finds documents for people posted to the Scrat who are not Cook's Mates. Which documents does this select?

2. Suppose you add some new documents to the `postings` collection and they have a `MiddleInitial` value. Now assume that the program is using a `BsonDocument` object named `doc` to iterate through the results.
 - a. What issue must the program handle so it doesn't crash when processing the old documents that don't have `MiddleInitial`?
 - b. What method would the program use to determine whether the issue is present in the document `doc`?
 - c. What code should the program use to save the `MiddleInitial` value (or a space) into the variable `initial`?

3. Instead of using LINQ to select documents, you can pass a function into a collection's `Find` method. That function should take a `BsonDocument` as a parameter and it should return `true` if that document should be selected. For simple filters, you can use an expression lambda.

For this exercise, write a piece of code that uses this technique to select documents representing people posted to the Scrat. Add a call to the `ToList` method to the end of the result to turn it into something that you can iterate. Then loop through the list and display the selected documents.

4. The example passes a filter definition into the collection's `DeleteMany` method. An alternative is to pass a selection function similar to the functions described in Exercise 3.

For this exercise, write a piece of code that uses this technique to delete the documents that have no `Rank` value.

24

Apache Ignite in Python

This chapter's example uses Python to build a NoSQL key-value database. The example runs on the local computer, but it shouldn't be hard to adapt it to run in the cloud.

This example is much less complicated than the examples described in the previous chapters, largely because a key-value database has fewer features. Relational, graph, and document databases all provide sophisticated querying features. In contrast, a key-value database mostly just stores values and later retrieves them.

Apache Ignite can actually function as a relational database, but Chapters 16 through 19 demonstrated that kind of database, so this chapter uses only Ignite's key-value features.

A key-value database lets you associate a piece of data with a key and then later use the key to retrieve the data.

You can compare a key-value database to your computer's environment variables. You can set an environment variable and give it a name. Later, you can get that variable by its name.

For example, to set the environment variable named `GREETING` in Windows, you can open a console window and enter the following command.

```
SET GREETING=Howdy Pard!
```

Later, you can use the following code to show the saved value:

```
ECHO %GREETING%
```

A program can use code to do something similar. For example, the following code shows how a Python program might display the variable that you set in the command window:

```
import os
print(os.getenv('GREETING'))
```

The following code shows the C# version:

```
Console.WriteLine(Environment.GetEnvironmentVariable("GREETING"));
```

By default, Ignite stores data in RAM, so the data is lost when your program ends. This chapter explains how to use volatile or nonvolatile data. The following section explains how to install Apache Ignite. The rest of the chapter describes the example program.

INSTALL APACHE IGNITE

Apache Ignite is an open source database, and you can find free and non-free versions. You can find an overview at https://en.wikipedia.org/wiki/Apache_Ignite or visit the Ignite website <https://ignite.apache.org>.

You can also find a quick-start guide for Python that includes installation instructions at <https://ignite.apache.org/docs/latest/quick-start/python>.

Here's my abbreviated list of installation instructions.

1. Go to <https://ignite.apache.org/download.cgi#binaries>. Click the Binary Releases link at the top of that page to go to the Binary Releases section and download the latest zip file. (You can also download the documentation if you like.)
2. Unzip the files into a convenient directory. I put mine in `C:\users\rod\apache-ignite-2.14.0-bin`. If you just unzip the files, you'll get a directory named `apache-ignite-2.14.0-bin` inside the `apache-ignite-2.14.0-bin` directory, so either move into the zip file and drag the subdirectory to a good location or rearrange the files to suit your preferences after you unzip.

The download file is fairly large (version 2.14 is about 252 MB), so use a fast network connection for the download. It will also expand when you unzip it (version 2.14 expands to around 383 MB), so you'll need a fair amount of disk space.

The Apache Ignite community is quite active, so it's likely that a newer (and bigger) version will be available by the time you read this. There's already been a new version since I started writing this chapter.

START A NODE

Before you can use Ignite in your program, you need to start an Ignite node, and that node should remain running as long as you want to work with it.

To start a node, you need to run the batch file named `ignite.bat`, passing it the location of a configuration file to tell Ignite how to start the node. That configuration determines, among other things, whether the node should store data persistently.

NOTE *These are Windows batch files. If you're using Linux, then you'll need to modify them. The quick-start web page can help.*

Running the batch file isn't hard, but to make the process even easier, I created two batch files to start the node with or without persistence.

Without Persistence

If you start a node without persistence, then the data is stored in memory and is lost when the node stops. To start the node without persistence, use the following batch file, which I named

`start_node.bat`:

```
cd c:\users\rod\apache-ignite-2.14.0-bin\bin\  
ignite.bat  
pause
```

This file uses the `cd` command to set the current directory to the location of the `ignite.bat` file. It then runs `ignite.bat` to start the node. When the node starts, you should see a command window similar to the one shown in Figure 24.1.

```
C:\WINDOWS\system32\cmd.exe
[08:43:53] Configured failure handler: [hnd=StopNodeOrHaltFailureHandler [tryStop=false, timeout=0, super=AbstractFailureHandler [ignoredFailureTypes=UnmodifiableSet [SYSTEM_WORKER_BLOCKED, SYSTEM_CRITICAL_OPERATION_TIMEOUT]]]]
[08:43:54] Message queue limit is set to 0 which may lead to potential OOMs when running cache operations in FULL_ASYNC or PRIMARY_SYNC modes due to message queues growth on sender and receiver sides.
[08:43:56] Data Regions Started: 4
[08:43:56] ^-- sysMemPlc region [type=internal, persistence=false, lazyAlloc=false,
[08:43:56] ... initCfg=40MB, maxCfg=100MB, usedRam=0MB, freeRam=100%, allocRam=40MB]
[08:43:56] ^-- default region [type=default, persistence=false, lazyAlloc=true,
[08:43:56] ... initCfg=256MB, maxCfg=3224MB, usedRam=0MB, freeRam=100%, allocRam=0MB]
[08:43:56] ^-- TxLog region [type=internal, persistence=false, lazyAlloc=false,
[08:43:56] ... initCfg=40MB, maxCfg=100MB, usedRam=0MB, freeRam=100%, allocRam=40MB]
[08:43:56] ^-- volatileDsMemPlc region [type=user, persistence=false, lazyAlloc=true,
[08:43:56] ... initCfg=40MB, maxCfg=100MB, usedRam=0MB, freeRam=100%, allocRam=0MB]
[08:43:56] Security status [authentication=off, sandbox=off, tls/ssl=off]
[08:43:56] Performance suggestions for grid (fix if possible)
[08:43:56] To disable, set -DIGNITE_PERFORMANCE_SUGGESTIONS_DISABLED=true
[08:43:56] ^-- Switch to the most recent 11 JVM version
[08:43:56] ^-- Set max direct memory size if getting 'OOM: Direct buffer memory' (add '-XX:MaxDirectMemorySize=<size>[g|M|k|K]' to JVM options)
[08:43:56] Refer to this page for more performance suggestions: https://ignite.apache.org/docs/latest/perf-and-troubleshooting/memory-tuning
[08:43:56] To start Console Management & Monitoring run ignitevisorcmd.{sh|bat}
[08:43:56] Ignite node started OK (id=fa7672b1)
[08:43:56] Topology snapshot [ver=1, locNode=fa7672b1, servers=1, clients=0, state=ACTIVE, CPUs=8, offheap=3.1GB, heap=1.0GB]
[08:43:56] ^-- Baseline [id=0, size=1, online=1, offline=0]
```

FIGURE 24.1

That window will remain running as long as the node is running.

The batch file's final `pause` command is only there in case there's a problem starting `ignite.bat`. For example, if `ignite.bat` isn't in the directory passed to the `cd` command, the `pause` command makes the output pause so that you can read the error message. Without that command, the window would vanish too quickly for you to see the error message.

You can start as many nodes as you like, and they should find each other automatically.

With Persistence

If you start a node with persistence enabled, then as much data as possible is stored in memory and it is all shadowed to hard disk. If you stop the node and restart it, the node reloads the saved data.

To start the node with persistence, use the following batch file, which I named `start_node_persistent.bat`:

```
cd c:\users\rod\apache-ignite-2.14.0-bin\bin\  
ignite.bat ..\examples\config\persistentstore\example-persistent-store.xml  
pause
```

This file uses the same `cd` command and runs `ignite.bat` as before. This time it passes the location of a configuration file to `ignite.bat`. You can look at that configuration file to see what it does. The most important part for our purposes is that it sets the value `persistenceEnabled` to `true`.

If you look closely at the avalanche of information in the command window, you'll see a couple of differences with the window shown in Figure 24.1. First, you'll find the following information:

```
Ignite node started OK (id=36174a6d)  
[15:24:56] >>> Ignite cluster is in INACTIVE state (limited functionality  
available). Use control.(sh|bat) script or IgniteCluster.state(ClusterState  
.ACTIVE) to change the state.  
[15:24:56] Topology snapshot [ver=1, locNode=36174a6d, servers=1, clients=0,  
state=INACTIVE, CPUs=8, offheap=3.1GB, heap=1.0GB]
```

This says the node is in an inactive state. For some reason, that's the default if you start the node with persistence enabled. If you try to access the node while it is inactive, you get the following message:

```
CacheCreationError: Cannot perform the operation because the cluster is  
inactive. Note, that the cluster is considered inactive by default if Ignite  
Persistent Store is used to let all the nodes join the cluster. To activate the  
cluster call Ignite.active(true).
```

You can use the Ignite control script `control.bat` (or `control.sh` if you have a Linux accent to change the state, but I found it easy enough to make the program do it.

To make it easier to test the database with and without persistence, I've broken the example program into sections that define a class, write some data, and read some data. The following sections describe those pieces. Later in the chapter, I'll summarize how to use the pieces to demonstrate persistence. In a real application, you might want to combine the pieces or call them from a main program.

CREATE THE PROGRAM

Now that you know how to start the node with and without persistence, you need to install a database adapter for it. Then you can start writing code.

Install the pyignite Database Adapter

To install the pyignite driver, simply use the following `pip` command:

```
$ pip install pyignite
```

If you're not familiar with `pip`, you can execute the following command in a Jupyter Notebook cell instead:

```
!pip install pyignite
```

That's all there is to it!

Define the Building Class

The example program saves a few pieces of information into the database. One of those pieces is a `Building` class. To allow Ignite to understand the data saved in the object's fields, you need to define it properly.

The following code shows how the program defines the `Building` class:

```
# Cell 1.
# Define the Building class with a schema
# so the database can understand its fields.
from pyignite import *
from pyignite.datatypes import String, IntObject

class Building(metaclass=GenericObjectMeta, schema={
    'name': String,
    'city': String,
    'height': IntObject,
}):

    def __init__(self, name, city, height):
        self.name = name
        self.city = city
        self.height = height
```

This code uses the `GenericObjectMeta` class to define the `Building` class. The interesting part here is the schema, which is a dictionary that defines the class's fields and their data types. Ignite uses the data types to understand how it should save and retrieve data for `Building` objects. In addition to defining the class's fields, this cell creates a constructor to make initializing objects easier.

Save Data

The following cell saves data into the database.

```
# Create some data.
from pyignite import *
from pyignite.datatypes import String, IntObject
```

```
from pyignite.cluster import *

client = Client()
with client.connect('localhost', 10800):
    my_cluster = cluster.Cluster(client)
    my_cluster.set_state(ClusterState.ACTIVE)

    misc_data_cache = client.get_or_create_cache('misc_data')

    building = Building('Burj Khalifa', 'Dubai', 2717)
    misc_data_cache.put(100, building)

    misc_data_cache.put('fish', 'humuhumunukunukuapua\'a')
    misc_data_cache.put(3.14, 'pi')

print("Created data")
```

After the `imports` statements, the code creates a client. This example assumes that the node is running on the local computer, so it connects to `localhost`. You can change that value to an IP address if the node is running on another computer. You can also replace the parameters with a list of IP addresses and port numbers, as in the following code:

```
nodes = [
    ('127.0.0.1', 10800),
    ('127.0.0.1', 10801),
    ('127.0.0.1', 10802),
]

with client.connect(nodes):
    ...
```

After it has connected to the node, the code gets a `Cluster` object representing the cluster that it is using and calls its `set_state` method to activate the cluster.

If you started the node without persistence, then it is already active and this does nothing. If you started the node with persistence, then it is initially inactive and this activates it.

Next, the code calls the client's `get_or_create_cache` method to retrieve a cache named `misc_data`. As the method's name implies, this gets the cache if it already exists, and it creates an empty cache if it does not exist.

The next statements add data to the cache using an assortment of data types for both keys and values.

The program creates a new `Building` object representing Burj Khalifa, the tallest building in the world. It then saves that object in the database, giving it the integer key 100.

Next, the cell saves the string value “humuhumunukunukuapua’a” in the cache with the string key “fish.” (It’s the official state fish of Hawaii. It’s also called the reef triggerfish, but humuhumunukunukuapua’a is more fun to say.)

Finally, the code saves the string value “pi” in the cache with the floating point key 3.14.

This cell finishes by displaying a message to show that it finished.

Read Data

The following cell reads data back from the node:

```
# Cell 3.
# Read and display the data.
from pyignite import *
from pyignite.datatypes import String, IntObject
from pyignite.cluster import *

client = Client()
with client.connect('localhost', 10800):
    my_cluster = cluster.Cluster(client)
    my_cluster.set_state(ClusterState.ACTIVE)

    misc_data_cache = client.get_or_create_cache('misc_data')

    building = misc_data_cache.get(100)
    print(building)

    text = misc_data_cache.get('fish')
    print(text)

    number = misc_data_cache.get(3.14)
    print(number)
```

The code first creates a client, connects to the database, and activates the cluster in case it is not already active. It then uses the client's `get_or_create_cache` method to get the cache.

The program then calls the cache's `get` method, passing it the keys that the previous cell used to save the data. It uses the key 100 to retrieve the `Building` object and prints that object. Next, it uses the string key "fish" to fetch the fish's name and displays that value. Finally, it uses the floating point key 3.14 to fetch the associated string and prints that string. Here's the result:

```
Building(name='Burj Khalifa', city='Dubai', height=2717, version=1)
humuhumunukunukuapua'a
pi
```

If you would rather not create a `Building` class with a schema, you can store and retrieve building data in another format such as a JSON document or delimited string.

Demonstrate Volatile Data

To demonstrate the program using volatile data, use the following steps:

1. Run `start_node.bat` to start the node without persistence.
2. Run Cell 1 to define the `Building` class.
3. Run Cell 2 to create some data.
4. Run Cell 3 to read and display the data. It should look as expected.

5. Close the node's command window and rerun `start_node.bat` to restart the node, again without persistence.
6. Run Cell 3 to read and display the data. Notice that all of the values are displayed as "none."

Demonstrate Persistent Data

To demonstrate the program using volatile data, use the following steps:

1. Run `start_node_persistent.bat` to start the node without persistence.
2. Run Cell 1 to define the `Building` class.
3. Run Cell 2 to create some data.
4. Run Cell 3 to read and display the data. It should look as expected.
5. Close the node's command window and rerun `start_node_persistent.bat` to restart the node, again with persistence.
6. Run Cell 3 to read and display the data. This time all the values should be displayed normally.

SUMMARY

This chapter shows how you can use Python and a key-value database to store and retrieve values in a cache. You can use a configuration file to enable persistence if you want to save data when you stop and restart the node. If persistence is enabled, the node starts in an inactive state, so you'll need to activate it either by using the control script `control.bat` (or `control.sh` in Linux), or you can make the program do it.

If you like, you can use the database to pass information between the C# program described in Chapter 25 and the Python program described in this chapter. For example, you can use the C# program to save data into the node and then use the Python program to read the data. That should work for simple data types such as integers and strings, but it may not work with objects. For example, the Python and C# programs represent the `Building` class in slightly different ways, so they can't reliably pass `Building` objects through the database. If you really need to pass an object back and forth, you could serialize the object in a string, pass that to the other program, and then deserialize it on the other end.

The next chapter shows how to build a similar example program in C#. Before you move on to that example, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

EXERCISES

If you don't know how to do something in the following exercises, try the obvious thing and see if it works (or peek at Appendix A).

1. Suppose you have a program used by everyone in your department and you want to display a daily message such as your company's stock price or a motivational and/or confusing quote

like "Seek success, but prepare for vegetables." (You can generate your own quotes at InspiroBot, <https://inspirobot.me>.) What code would you use to set the quote each day? What code would the users' program use to read the quote?

2. What code would you use to write and then read the following lists of values?

- apple, banana, cherry
 - 1, 1, 2, 3, 5, 8, 13
 - 3.14159, 2.71828, 1.61803, 6.0221515e23
-

3. Can you use a list as a key? What code would you use to save and retrieve values?

4. The `Cache` class's `get_all` method takes as a parameter a list or tuple of keys. It returns a dictionary holding the keys and their values.

a. Write some code to add the following values to the database individually.

NAME	PHONE NUMBER
Glenn	Pennsylvania 6-5000
Jenny	867-5309
John	36 24 36
Marvin	Beechwood 4-5789
Tina	6060-842

- b. Write some code to extract those values into a dictionary and print the dictionary. For bonus points, display the dictionary nicely with each name/phone number pair on a different line.
- c. Write some code that uses the dictionary to print Glenn's and Jenny's phone numbers.
-

25

Apache Ignite in C#

This chapter's example uses C# to build a NoSQL key-value database. The example runs on the local computer, but it shouldn't be hard to adapt it to run in the cloud.

This example is much less complicated than the examples described in the previous chapters, largely because a key-value database has fewer features. Relational, graph, and document databases all provide sophisticated querying features. In contrast, a key-value database mostly just stores values and later retrieves them.

Apache Ignite can actually function as a relational database, but Chapters 16 through 19 demonstrated that kind of database, so this chapter uses only Ignite's key-value features.

A key-value database lets you associate a piece of data with a key and then later use the key to retrieve the data. By default, Ignite stores data in RAM, so the data is lost when your program ends. This chapter explains how to use volatile or nonvolatile data.

If you skipped Chapter 24, which built a similar example in Python, go to that chapter and read the beginning and the first two sections, which explain how to install Ignite and how to start a database node with or without persistence.

When you reach the section "Create the Program" in Chapter 24, return here and start on the following sections.

CREATE THE PROGRAM

Before you can make a C# program use the Ignite database, you need to install a database adapter for it. Then you can start writing code.

Install the Ignite Database Adapter

This example uses the Apache.Ignite database adapter to allow your program to communicate with the Ignite database. To add the adapter to your project, follow these steps:

1. Open the Project menu and select Manage NuGet Packages.
2. Click the Browse tab and enter **Apache.Ignite** in the search box to see the list shown in Figure 25.1. Click the Apache.Ignite entry in the list on the left.
3. Click the Install button on the right. This shouldn't take too long, but it's still nice to have a fast Internet connection.

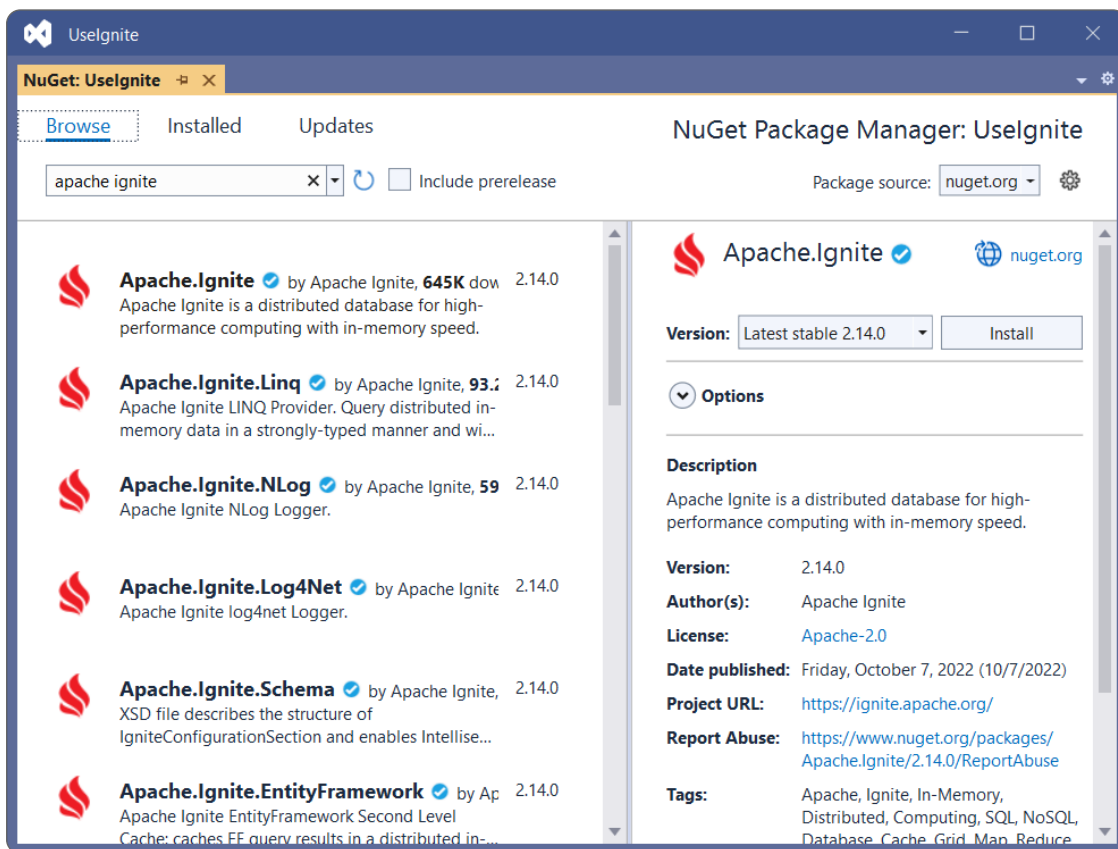


FIGURE 25.1

To make using the database driver easier, add the following code to the program just below the other using statements:

```
using Apache.Ignite.Core;
using Apache.Ignite.Core.Client;
using Apache.Ignite.Core.Client.Cache;
```

The following section describes the program's main method. The sections after that describe the helper methods that the main method calls.

The Main Program

To make testing the database with and without persistence easier, the main method lets you write or read data separately. To do that, it enters an infinite loop where you can pick the operation that you want to run.

The following code shows the main method:

```
static void Main(string[] args)
{
    for ( ; ; )
    {
        Console.WriteLine("0. Quit");
        Console.WriteLine("1. Write data");
        Console.WriteLine("2. Read data");
        Console.WriteLine();
        Console.Write("Choice: ");

        string choice = Console.ReadLine();
        if (choice == "0")
        {
            break;
        }
        else if (choice == "1")
        {
            WriteData();
        }
        else if (choice == "2")
        {
            ReadData();
        }
        else
        {
            Console.WriteLine($"Unknown choice '{choice}'");
        }
        Console.WriteLine();
    }
}
```

The code enters an infinite loop that lets you enter choices 0, 1, or 2 to trigger one of the following actions:

- 0—Exit the infinite loop so the program ends.
- 1—Call the `WriteData` method.
- 2—Call the `ReadData` method.

The example writes several kinds of data into the database including a `Building` object. The following section describes the `Building` class. The two sections after that describe the `WriteData` and `ReadData` methods.

The Building Class

The following code shows the `Building` class:

```
// Define the Building class.
internal class Building
{
    internal String Name, City;
    internal int Height;
    internal Building(string name, string city, int height)
    {
        Name = name;
        City = city;
        Height = height;
    }

    public override string ToString()
    {
        return $"{Name}, {City}, {Height}";
    }
}
```

This class is relatively straightforward. It defines fields to hold a building's name, city, and height. It also defines a constructor that makes initializing the fields easy, and it overrides the `ToString` method to make it easy to print a `Building` object's values.

The WriteData Method

The following code shows how the `WriteData` method saves data in the database:

```
// Create some data.
private static void WriteData()
{
    try
    {
        IgniteClientConfiguration cfg =
            new IgniteClientConfiguration
            {
                Endpoints = new[] { "localhost:10800" }
            };

        using (IIgniteClient client = Ignition.StartClient(cfg))
        {
            // Activate the cluster if needed.
            IClientCluster cluster = client.GetCluster();
        }
    }
}
```

```

        cluster.SetActive(true);

        // Create the cache.
        ICacheClient<object, object> cache =
            client.GetOrCreateCache<object, object>("misc_data");

        // Add some data.
        Building building = new Building("Burj Khalifa", "Dubai", 2717);
        cache.Put(100, building);
        cache.Put("fish", "humuhumunukunukuapua'a");
        cache.Put(3.14, "pi");
    }
}
catch (Exception ex)
{
    Console.WriteLine(ex.ToString());
}
Console.WriteLine("Created data");
}

```

This method creates an `IgniteClientConfiguration` object, setting its `Endpoints` property to an array of IP addresses and port numbers. This example uses the address `localhost` to represent the local computer, but you could add other addresses here.

The code then uses the configuration object to create an Ignite client. It creates the client in a `using` statement so it is automatically disposed of when the program no longer needs it.

Next, the method uses the client to get a cluster object and activates it in case it was started with persistence and is currently inactive. (If you know that the database is active, then you can skip this, and you don't need to create the cluster object.)

The program then uses the client's appropriately named `GetOrCreateCache` method to get or create the cache named `misc_data`.

The `ICacheClient` interface uses generic parameters to let you indicate the data types of the cache's keys and values. For example, if you're using integer keys to store string values, then you would use the generic types `<int, string>`. That's a good practice because it prevents you from using the cache incorrectly. For example, it would stop you from accidentally saving a `KnockKnockJoke` object in a cache filled with `EarningsReports`. For this example, though, I wanted to use different data types for both the keys and values, so I declared this cache using `<object, object>`.

At this point, adding data to the cache is simple. The code first creates a `Building` object and passes it into the cache with key 100.

Next, the method saves the string value "humuhumunukunukuapua'a" in the cache with the string key "fish." (It is the official state fish of Hawaii. It's also called the reef triggerfish, but humuhumunukunukuapua'a is more fun to say.)

Finally, the code saves the string value "pi" in the cache with the floating point key 3.14.

This method finishes by displaying a message to show that it finished.

The *ReadData* Method

The following code shows how the `ReadData` method reads and displays data from the node:

```
// Read and display the data.
private static void ReadData()
{
    try
    {
        IgniteClientConfiguration cfg =
            new IgniteClientConfiguration
            {
                Endpoints = new[] { "localhost:10800" }
            };

        using (IIgniteClient client = Ignition.StartClient(cfg))
        {
            // Activate the cluster if needed.
            IClientCluster cluster = client.GetCluster();
            cluster.SetActive(true);

            // Create the cache.
            ICacheClient<object, object> cache =
                client.GetOrCreateCache<object, object>("misc_data");

            // Fetch some data.
            if (cache.ContainsKey(100))
                Console.WriteLine(cache.Get(100));
            else
                Console.WriteLine("Value 100 missing");

            if (cache.ContainsKey("fish"))
                Console.WriteLine(cache.Get("fish"));
            else
                Console.WriteLine("Value 'fish' missing");

            if (cache.ContainsKey(3.14))
                Console.WriteLine(cache.Get(3.14));
            else
                Console.WriteLine("Value 3.14 missing");
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.ToString());
    }
}
```

The code first creates a configuration object, uses it to create a client, activates the cluster in case it is not already active, and gets or creates the cache as before.

The method then fetches the values that were added by the `WriteData` method. For each value, it first uses the cache's `ContainsKey` method to see if the key is present. The Python data adapter quietly

returns `None` if you try to access a missing key, but the C# adapter ruthlessly throws a `KeyNotFoundException` exception. You can catch the exception, but it's better to avoid it in the first place by calling `ContainsKey`.

The method fetches and displays the three values saved by the `WriteData` method. The following shows the result:

```
Burj Khalifa, Dubai, 2717
humuhumunukunukuapua'a
pi
```

Demonstrate Volatile Data

To demonstrate the program using volatile data, use the following steps:

1. Run `start_node.bat` to start the node without persistence.
2. Run the program and select option 1 to create some data.
3. Select option 2 to fetch and display the data. It should look as expected.
4. Close the node's command window and rerun `start_node.bat` to restart the node, again without persistence.
5. Keep the program running or restart it and select option 2 to read and display the data. Notice that the program displays missing messages for all the values.

Demonstrate Persistent Data

To demonstrate the program using volatile data, use the following steps:

1. Run `start_node_persistent.bat` to start the node without persistence.
2. Run the program and select option 1 to create some data.
3. Select option 2 to fetch and display the data. It should look as expected.
4. Close the node's command window and rerun `start_node_persistent.bat` to restart the node, again with persistence.
5. Keep the program running or restart it and select option 2 to read and display the data. Notice that the program displays the expected values.

SUMMARY

This chapter showed how you can use C# and a key-value database to store and retrieve values in a cache. You can use a configuration file to enable persistence if you want to save data when you stop and restart the node. If persistence is enabled, the node starts in an inactive state, so you'll need to activate it either by using the control script `control.bat` (or `control.sh` in Linux) or by making the program do it.

If you like, you can use the database to pass information between the Python program described in Chapter 24 and the C# program described in this chapter. For example, you can use the Python program's Cell 2 to save data into the node and then use the C# program to read the data. That should work for simple data types such as integers and strings, but it may not work with objects. For example, the Python and C# programs represent the `Building` class in slightly different ways, so they can't reliably pass `Building` objects through the database. If you really need to pass an object back and forth, you could serialize the object in a string, pass that to the other program, and then deserialize it on the other end.

This chapter marks the end of the database demonstration programs. None of them are a complete customer-ready application, but they demonstrate many of the basic techniques that you need to get started. Using the techniques described up to this point in the book, you can design and implement flexible, robust relational databases, and use NoSQL databases.

Database programming is an enormous topic, however, and there's much more to study. The next part of the book explains some more advanced database design and development topics.

Some of the earlier chapters used SQL to define and manipulate relational databases. The following chapter provides a lot more detail about the SQL language.

Before you move on to Chapter 26, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to selected exercises in Appendix A.

EXERCISES

If you don't know how to do something in the following exercises, try the obvious thing and see if it works (or peek at Appendix A).

1. Before it tries to get a value, the example program calls `ContainsKey` and displays a "missing" message if the key is not in the database. Write a `GetString` extension method to make that easier by checking `ContainsKey` and returning either the value converted to a string or "missing" if the key is not present.
2. Suppose you have a program used by everyone in your department and you want to display a daily message such as your company's stock price or a motivational and/or confusing quote like "Seek success, but prepare for vegetables." (You can generate your own quotes at [InspiroBot](https://inspirobot.me), <https://inspirobot.me>.) What code would you use to set the quote each day? What code would the users' program use to read the quote? (You can use the `GetString` extension method if you like.)
3. What code would you use to write the following arrays of values into the database?
 - `apple, banana, cherry`
 - `1, 1, 2, 3, 5, 8, 13`
 - `3.14159, 2.71828, 1.61803, 6.0221515e23`

What code would you use to fetch the arrays and display them as in the following? (Don't worry about calling `ContainsKey` for this. Assume the values are present.)

```
{ apple, banana, cherry }  
{ 1, 2, 3, 4, 5 }  
{ 3.14159, 2.71828, 1.61803, 6.0221515E+23 }
```

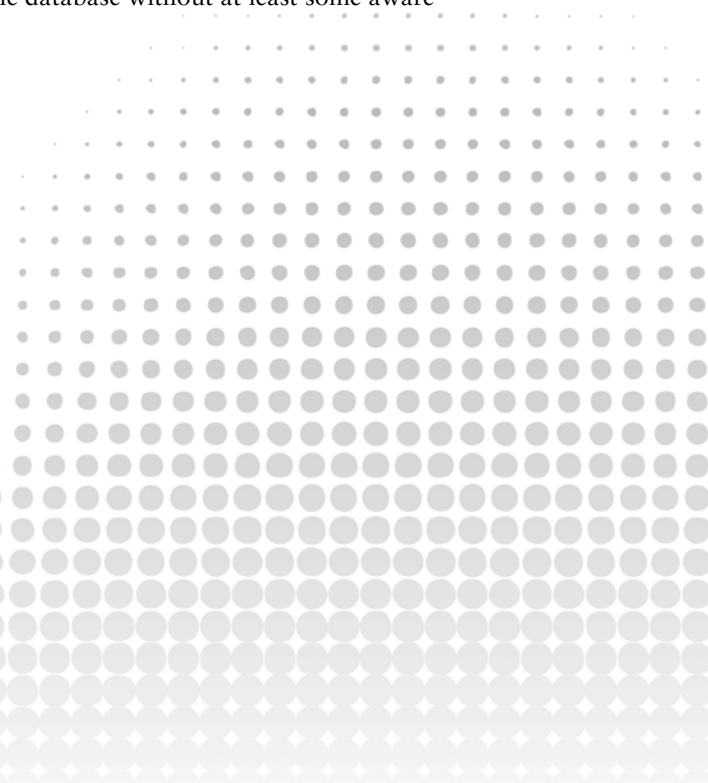
4. Can you use an array as a key? What code would you use to save and retrieve values?
-

PART 5

Advanced Topics

- **Chapter 26:** Introduction to SQL
- **Chapter 27:** Building Databases with SQL Scripts
- **Chapter 28:** Database Maintenance
- **Chapter 29:** Database Security

The chapters in the final part of this book deal with advanced topics. Though a good grasp of these topics is not strictly necessary for designing a database, it is very important for database practitioners. You can design a database without understanding these topics, but it's unlikely that you will do a good job implementing the database without at least some awareness of them.



Chapter 26 provides an introduction to SQL (Structured Query Language). Chapter 27 explains how to use SQL to make scripts that create, populate, and otherwise manipulate databases. Chapter 28 discusses database maintenance issues, and Chapter 29 describes database security.

Although these chapters don't cover every last detail of these topics, they should provide you with enough background to let you build a database competently.

26

Introduction to SQL

Eventually, you (or someone else) must build the database that you've designed. Also at some point, someone will probably want to actually use the database you've spent so much time designing.

Structured Query Language (SQL, sometimes pronounced “sequel”) is a database language that includes commands that let you build, modify, and manipulate relational databases. Many database tools use SQL either directly or behind the scenes to create and manipulate the database. For example, Figure 26.1 shows how the pgAdmin tool uses SQL to define the `order_items` table used in Chapters 18 and 19, which cover PostgreSQL.

Because SQL is standardized, many database management tools use it to control relational databases. They use buttons, drop-down menus, check boxes, and other user interface controls to let you design a database interactively, and then they send SQL code to the database engine to do the work.

In addition to using database tools to generate your SQL code, you can write that code yourself, which lets you create and modify the database as needed. It also lets you add data to put the database in a known state for testing. In fact, SQL is so useful for building databases that it's the topic of the next chapter.

NoSQL MAY HAVE SQL

SQL was designed to work with relational databases, so you might think that NoSQL databases don't support it. Some don't, but others do. Whether a NoSQL database supports SQL depends on the type of database and the specific product.

For example, a column-oriented database can act as a relational database that uses a particular storage strategy, so it may be able to execute standard SQL statements.

continues

(continued)

A document database may also be able to execute SQL on the fields defined inside its documents, although you may need some extra condition checking and error handling to deal with documents that contain different fields.

A graph database, such as Neo4j AuraDB, may support graph query language (GQL), but its only real similarity to SQL is the fact that they both end with QL.

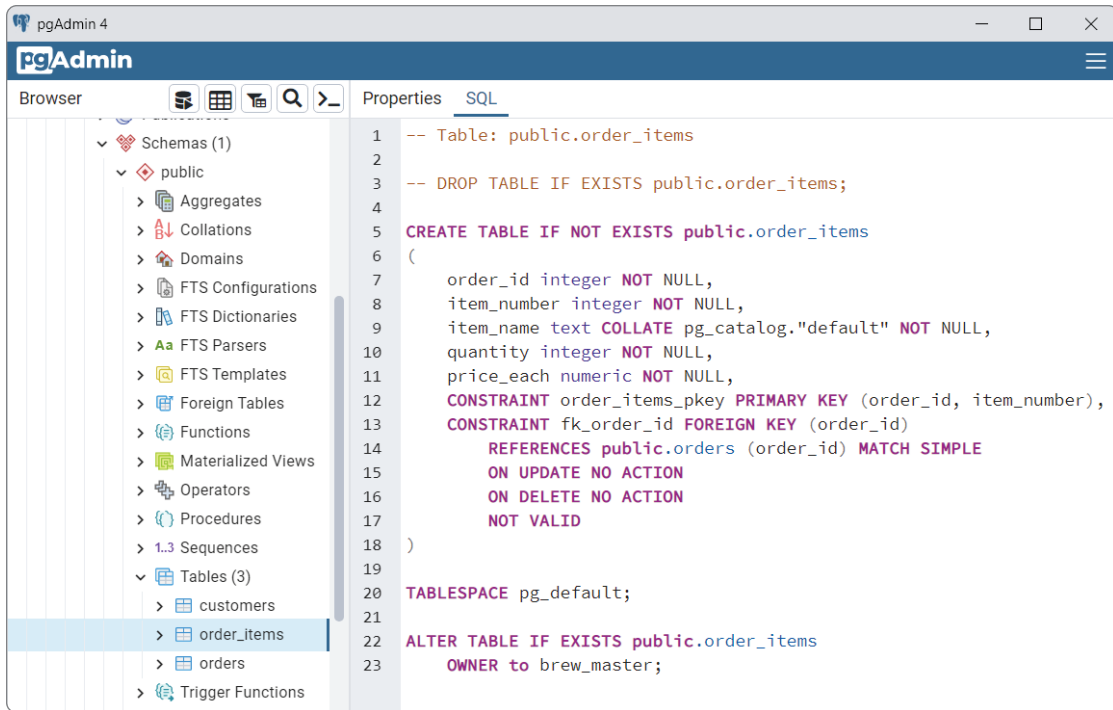


FIGURE 26.1

SQL is such an important part of database development that your education as a database designer is sadly lacking if you don't at least understand the basics. (The other developers will rightfully mock you if you don't chuckle when you see a T-shirt that says, "SELECT * FROM People WHERE NOT Clue IS null.")

In this chapter, you'll learn how to use SQL to do the following:

- Create and delete tables.
- Insert data into tables.
- Select data from the database using various criteria and sort the results.
- Modify data in the database.
- Delete records.

BACKGROUND

SQL was developed by IBM in the mid-1970s. It is an English-like command language for building and manipulating relational databases.

From a small but ambitious beginning, SQL has grown into a large language containing about 70 commands with hundreds of variations. Because SQL has grown so large, I cannot possibly cover every nook and cranny. Instead, this chapter gives a brief introduction to SQL and then describes some of the most useful SQL commands in greater detail. Even then, this chapter doesn't have room to cover most of the more interesting commands completely. The `SELECT` statement alone includes so many variations that you could practically write a book about just that one command.

FINDING MORE INFORMATION

SQL is intuitive enough that, once you master the basics, you should be able to get pretty far on your own. The Internet is practically clogged with websites that are chock-full of SQL goodness in the form of tutorials, references, FAQs, question and answer forums, and discussion groups.

In fact, I'll start you off with a small list of websites right now. The following list shows a few websites that provide SQL tutorials:

- <https://sqlcourse.com>
- <https://sql.org>
- <https://sqltutorial.net>
- <https://sqltutorial.org>
- <https://tutorialspoint.com/sql/index.htm>
- <https://w3schools.com/sql>

For help on specific issues, you should find a few SQL forums where you can ask questions. An enormous number of developers work with databases and SQL on a daily basis, so there are lots of forums out there. Just search for **SQL forum** to find more forums than you can shake a `SELECT` statement at.

Tutorials can help you get started using some of the more common SQL statements, but they aren't designed as references. If you need more information about a particular command, you should look for a SQL reference. At the following sites you'll find references for different versions of SQL:

- **MariaDB**—<https://mariadb.com/kb/en/sql-statements>
- **Microsoft Transact-SQL**—<https://learn.microsoft.com/en-us/sql/t-sql/language-reference>
- **MySQL**—<https://dev.mysql.com/doc/refman/8.0/en>
- **Oracle SQL**—www.adp-gmbh.ch/ora/sql
- **PostgreSQL**—<https://postgresql.org/docs/8.1/sql-commands.html>

Each of these pages provides simple navigation to look up specific SQL commands.

Note, however, that these web pages deal with specific versions of SQL. Though the basics of any version of SQL are fairly standard, there are some differences between the various flavors.

STANDARDS

Although all relational databases support SQL, different databases may provide slightly different implementations. This is such an important point that I'll say it again.

WARNING *Different databases provide slightly different implementations of SQL.*

Both the International Organization for Standardization (ISO) and the American National Standards Institute (ANSI) have standards for the SQL language, and most database products follow those standards pretty faithfully. However, different database products also add extra features to make certain chores easier. Those extras can be useful but only if you are aware of which features are standard and which are not, and you use the extra features with caution.

NOTE *I don't know why the International Organization for Standardization is abbreviated ISO instead of IOS. It's probably a French thing. Or perhaps that's just the way it is, like when NIST, formerly the National Institute of Standards and Technology, decided that NIST no longer stands for anything and is just a name like Alice or Trevor.*

For example, in the Transact-SQL language used by SQL Server, the special values @@TOTAL_ERRORS, @@TOTAL_READS, and @@TOTAL_WRITES return the total number of disk write errors, disk reads, and disk writes, respectively, since SQL Server was last started. Other relational databases don't provide those, although they may have their own special values that return similar statistics.

If you use these kinds of special features haphazardly, it may be very hard to rebuild your database or the applications that use it if you are forced to move to a new kind of database. For that matter, extra features can make it hard for you to reuse tools and techniques that you develop in your next database project.

NOTE *In one project I worked on, we didn't know which database engine we would be using in the final application for almost a year. Management had two database vendors bidding against each other, so they kept changing their minds to keep the pressure on the vendors. They didn't finally pick one until development was almost complete, so it was critical that we stuck to standard SQL with no extra database-dependent features.*

Fortunately, most flavors of SQL are 90 percent identical. You can guard against troublesome changes in the future by keeping the nonstandard features in a single code module or library as much as possible.

Usually the places where SQL implementations differ the most is in system-level chores such as database management and searching metadata. For example, different databases might provide different tools for searching through lists of tables, creating new databases, learning about the number of reads and writes that the database performed, examining errors, and optimizing queries.

NOTE *The SQL code shown in this chapter works for MariaDB but may need some fine-tuning for other databases. It should work, but I make no promises!*

MULTISTATEMENT COMMANDS

Many (but not all) databases allow you to execute multiple commands at once if you separate them with semicolons. Even then, some databases disable that capability by default. In that case, you need to take extra steps to enable this feature.

For example, the following code shows how a Python program can execute multistatement commands in MariaDB:

```
# Semi-colons.

# Create the tables.
import pymysql
from pymysql.connections import CLIENT

# Connect to the database server.
conn = pymysql.connect(
    host="localhost",
    user="root",
    password="G4eJuHFihFOa13m",
    database="Chapter26Data",
    client_flag=CLIENT.MULTI_STATEMENTS)

# Create a cursor.
cur = conn.cursor()

cur.execute("""
DROP TABLE IF EXISTS Crewmembers;
CREATE TABLE Crewmembers (
    FirstName    VARCHAR(45)    NOT NULL,
    LastName     VARCHAR(45)    NOT NULL
);
""")

cur.execute("""
INSERT INTO Crewmembers VALUES ("Malcolm", "Reynolds");
INSERT INTO Crewmembers VALUES ('Hoban "Wash"', "Washburne");
```

```
INSERT INTO Crewmembers VALUES ("Zoë", "Washburne");
INSERT INTO Crewmembers VALUES ("Kaylee", "Frye");
INSERT INTO Crewmembers VALUES ("Jayne", "Cobb");
SELECT * FROM Crewmembers
"""

cur.execute("SELECT * FROM Crewmembers")
rows = cur.fetchall()
for row in rows:
    print(row[0] + " " + row[1])

# Close the connection.
conn.commit()
conn.close()
```

This code first imports the `pymysql` and `pymysql.connections.CLIENT` modules. It then connects to the database. The code sets the connect method's `client_flag` parameter to `CLIENT.MULTI_STATEMENTS` to enable multistatement commands. The setup code finishes by creating a cursor to manipulate the database.

The program then executes a `DROP TABLE` statement and a `CREATE TABLE` statement in a single command. Notice how the commands are separated by semicolons. MariaDB is uncooperative if you leave out the semicolons. You can omit the semicolon after the final command if you like, but I've included it for consistency. (It also makes it easier to add more commands at the end of the sequence without forgetting to add new semicolons.)

Next, the code uses a multistatement command to execute five `INSERT` statements.

In both `execute` calls, the Python code uses the multiline string delimiter `"""` so the statements can include single or double quotes. The second `INSERT` statement uses single quotes around the first name value so that it can include embedded double quotes for Wash's nickname.

The opening and closing triple quotes are on the lines before and after the SQL code, so the SQL code is exactly as it would appear in a script.

After it inserts the records, the code executes a `SELECT` statement and displays the new records. The code finishes by committing the changes and closing the database connection.

The following shows the program's output:

```
Malcolm Reynolds
Hoban "Wash" Washburne
Zoë Washburne
Kaylee Frye
Jayne Cobb
```

If you include a `SELECT` statement inside a multistatement command the `fetchall` method returns an empty result, so you should execute `SELECT` statements separately, at least in MariaDB.

BASIC SYNTAX

As is mentioned earlier in this chapter, SQL is an English-like language. It includes command words such as `CREATE`, `INSERT`, `UPDATE`, and `DELETE`.

SQL is case-insensitive. That means it doesn't care whether you spell the `DELETE` keyword as `DELETE`, `delete`, `Delete`, or `DeLeTe`.

SQL also doesn't care about the capitalization of database objects such as table and field names. If a database has a table containing people from the Administrative Data Organization, SQL doesn't care whether you spell the table's name `ADOPEOPLE`, `AdoPeople`, or `aDOPEople`. (Although the last one looks like "a dope ope," which may make pronunciation tricky and might annoy your admins.)

To make the code easier to read, however, most developers write SQL command words in ALL CAPS and they write database object names using whatever capitalization they used when building the database. For example, I prefer `MixedCase` for table and field names.

Unfortunately, not all database products handle variations in capitalization in exactly the same way. For example, as you learned in Chapter 18, "PostgreSQL in Python," PostgreSQL "helpfully" converts table and column names to lowercase. Run some tests to see whether you and your database have the same capitalization fashion sense, and then be prepared to do some rewriting if you're forced to switch databases.

A final SQL feature that makes commands easier to read is that SQL ignores whitespace. That means it ignores spaces, tabs, line feeds, and other "spacing" characters, so you can use those characters to break long commands across multiple lines or to align related statements in columns.

For example, the following code shows a typical `SELECT` command as I would write it:

```
SELECT FirstName, LastName, Clue,  
       Street, City, State, Zip  
FROM People  
WHERE NOT Clue IS NULL  
ORDER BY Clue, LastName, FirstName
```

This command selects name and address information from the records in the `People` table where the record's `Clue` field has a value that is not null. (Basically, it selects people who have a clue, normally a pretty small data set.) It sorts the results by `Clue`, `LastName`, and `FirstName`.

This command places the different SQL clauses on separate lines to make them easier to read. It also places the person's name and address on separate lines and indents the address fields so they line up with the `FirstName` field. That makes it easier to pick out the `SELECT`, `FROM`, `WHERE`, and `ORDER BY` clauses.

COMMAND OVERVIEW

SQL commands are typically grouped into four categories: Data Definition Language (DDL), Data Manipulation Language (DML), Data Control Language (DCL), and Transaction Control Language (TCL).

Note that some commands have several variations. For example, in Transact-SQL the `ALTER` command has the versions `ALTER DATABASE`, `ALTER FUNCTION`, `ALTER PROCEDURE`, `ALTER TABLE`, `ALTER TRIGGER`, and `ALTER VIEW`. The following tables just provide an overview of the main function (`ALTER`) so you know where to look if you need one of these.

NO DO-OVERS

SQL statements are immediate and unundoable. The database doesn't care if you have your fingers crossed, these statements can drop tables, delete records, destroy users, and generally wreak havoc with no warnings or backsies. For example, the database doesn't give you any warning or make you confirm a `DROP TABLE` statement. It instantly destroys the table and all of the data it contains. You cannot undo this command, so be really, really, *really* sure you want to do it before you execute a `DROP TABLE` command.

If you execute these commands interactively, then you're responsible for the outcome. If your program executes these commands, then it must be certain that the user really wants to live with the consequences.

The only mitigating tool at your disposal is the transaction, which lets you commit or roll back one or more transactions as a group. For example, you can execute a series of statements and, if any of them fails, you can roll back the transaction to cancel them all. You must do that, however; the database won't do it for you.

It's also a good idea to make sure your backups are up-to-date before you start any task that may damage the database such as dropping, adding, or modifying tables. (Chapter 28, "Database Maintenance," talks about backups.)

DDL commands define the database's structure. Table 26.1 briefly describes the most commonly used DDL commands.

TABLE 26.1: DDL commands

COMMAND	PURPOSE
ALTER	Modifies a database object such as a table, stored procedure, or view. The most important variation is <code>ALTER TABLE</code> , which lets you change a column definition or table constraint.
CREATE	Creates objects such as tables, indexes, views, stored procedures, and triggers. In some versions of SQL, this also creates databases, users, and other high-level database objects. Two of the most important variations are <code>CREATE TABLE</code> and <code>CREATE INDEX</code> .
DROP	Deletes database objects such as tables, functions, procedures, triggers, and views. Two of the most important variations are <code>DROP TABLE</code> and <code>DROP INDEX</code> .

DML commands manipulate data. They let you perform the CRUD operations: Create, Read, Update, and Delete. (Where athletes “talk smack,” database developers “talk CRUD.”) Table 26.2 summarizes the most common DML commands. Generally developers think of only the CRUD commands `INSERT`, `SELECT`, `UPDATE`, and `DELETE` as DML commands, but this table includes cursor commands because they are used to select records.

TABLE 26.2: DML commands

COMMAND	PURPOSE
<code>CLOSE</code>	Closes a cursor.
<code>DECLARE</code>	Declares a cursor that a program can use to fetch and process records a few at a time instead of all at once.
<code>DELETE</code>	Deletes records from a table.
<code>FETCH</code>	Uses a cursor to fetch rows.
<code>INSERT</code>	Inserts new rows into a table. A variation lets you insert the result of a query into a table.
<code>SELECT</code>	Selects data from the database, possibly saving the result into a table.
<code>TRUNCATE</code>	Deletes all of the records from a table as a group without logging individual record deletions. It also removes empty space from the table while <code>DELETE</code> may leave empty space to hold data later. (Some consider <code>TRUNCATE</code> a DDL command, possibly because it removes empty space from the table. Or possibly just to be contrary. Or perhaps I’m the contrary one.)
<code>UPDATE</code>	Changes the values in a record.

DCL commands allow you control access to data. Depending on the database, you may be able to control user privileges at the database, table, view, or field level. Table 26.3 summarizes the two most common DCL commands.

TABLE 26.3: DCL commands

COMMAND	PURPOSE
<code>GRANT</code>	Grants privileges to a user
<code>REVOKE</code>	Revokes privileges from a user

TCL commands let you use transactions. A transaction is a set of commands that should be executed atomically as a single unit, so either every command is executed or none of the commands is executed.

For example, suppose you want to transfer money from one bank account to another. It would be bad if the system crashed after you subtracted money from the first account but before you added it to the other. If you put the two commands in a transaction, the database guarantees that either both happen or neither happens.

Table 26.4 summarizes the most common TCL commands.

TABLE 26.4: TCL commands

COMMAND	PURPOSE
BEGIN	Starts a transaction. Operations performed before the next COMMIT or ROLLBACK statement are part of the transaction.
COMMIT	Closes a transaction, accepting its results.
ROLLBACK	Rewinds a transaction's commands back to the beginning of the transaction or to a savepoint defined within the transaction.
SAVE	Creates a savepoint within a transaction. (Transact-SQL calls this command SAVE TRANSACTION whereas PostgreSQL calls it SAVEPOINT.)

The following sections describe the most commonly used commands in greater detail.

CREATE TABLE

The CREATE TABLE statement builds a database table. The basic syntax for creating a table is:

```
CREATE TABLE table_name (parameters)
```

Here *table_name* is the name you want to give to the new table and *parameters* is a series of statements that define the table's columns. Optionally, *parameters* can include column-level and table-level constraints.

A column definition includes the column's name, its data type, and optional extras such as a default value or the keywords NULL or NOT NULL to indicate whether the column should allow null values.

A particularly useful option that you can add to the CREATE TABLE statement is IF NOT EXISTS. This clause makes the statement create the new table only if it doesn't already exist.

For example, the following statement creates a Students table with three fields. Notice how the code uses whitespace to make the field data types and NOT NULL clauses align so they are easier to read:

```
CREATE TABLE IF NOT EXISTS Students (
    StudentId    INT           NOT NULL    AUTO_INCREMENT,
    FirstName    VARCHAR(45)   NOT NULL,
    LastName     VARCHAR(45)   NOT NULL,
    PRIMARY KEY (StudentId)
)
```

The StudentId field is an integer (INT) that is required (NOT NULL). The database automatically generates values for this field by adding 1 to the value it last generated (AUTO_INCREMENT).

The FirstName and LastName fields are required variable-length strings up to 45 characters long.

The table's primary key is the StudentId field.

A key part of a column's definitions is the data type. The following list summarizes the most common SQL data types:

- **BLOB**—A Binary Large Object. This is any chunk of binary data such as a JPEG file, audio file, video file, or JSON document. The database knows nothing about the internal structure of this data so, for example, if the BLOB contains a JSON document the database cannot search its contents.
- **BOOLEAN**—A true or false value.
- **CHAR**—A fixed-length string. Use this for strings that always have the same length such as two-letter state abbreviations or five-digit ZIP Codes.
- **DATE**—A month, date, and year such as August 29, 1997.
- **DATETIME**—A date and time such as 2:14 a.m., August 29, 1997.
- **DECIMAL (p, s)**—A fixed-point number where *p* (precision) gives the total number of digits and *s* (scale) gives the number of digits to the right of the decimal. For example, DECIMAL(6, 2) holds numbers of the form 1234.56.
- **INT**—An integer value.
- **NUMBER**—A floating-point number.
- **TIME**—A time without a date such as 2:14 a.m..
- **TIMESTAMP**—A date and time.
- **VARCHAR**—A variable-length string. Use this for strings of unknown lengths such as names and street addresses.

Specific database products often provide extra data types and aliases for these types. They also sometimes use these names for different purposes. For example, in different databases the INT data type might use 32 or 64 bits, and the database may provide other data types such as SMALLINT, TINYINT, BIGINT, and so forth to hold integers of different sizes.

Most databases can handle the basic data types but before you make specific assumptions (for example, that INT means 32-bit integer), check the documentation for your database product.

The following code builds a frankly hacked together table with the sole purpose of demonstrating most of the common data types. Notice that this command uses the combined FirstName and LastName fields as the table's primary key:

```
CREATE TABLE IF NOT EXISTS MishmashTable (
  FirstName          VARCHAR(40)  NOT NULL,
  LastName           VARCHAR(40)  NOT NULL,
  Age               INT          NULL,
```

```

    Birthdate           DATE           NULL,
    AppointmentDateTime DATETIME        NULL,
    PreferredTime       TIME             NULL,
    TimeAppointmentCreated TIMESTAMP      NULL,
    Salary              DECIMAL(8,2)     NULL,
    IncludeInvoice      BOOLEAN          NULL,
    Street              VARCHAR(40)      NULL,
    City                VARCHAR(40)      NULL,
    State               CHAR(2)         NULL,
    Zip                 CHAR(5)         NULL,
    PRIMARY KEY (FirstName, LastName)
)

```

BUILD A TABLE

The `CREATE TABLE` statement can be quite complicated. You need to specify the fields' names, data types, default values, and whether they allow null values. You can also specify the table's primary key and indexes, and foreign key constraints.

Try writing a `CREATE TABLE` statement to make an inventory items table:

1. Create the InventoryItems table.

Give the new table the following fields:

NAME	TYPE	REQUIRED?
UPC	String up to 40 characters	Yes
Description	String up to 45 characters	Yes
BuyPrice	Number of the form 12345678.90	No
SellPrice	Number of the form 12345678.90	Yes
QuantityinStock	Integer	Yes
ExpirationDate	Date	No
StockLocation	String up to 40 characters	No
ShelfLocation	String up to 40 characters	No
ReorderWhen	Integer	No
ReorderAmount	Integer	No
VendorId	Integer	No

2. Make the table's primary key the UPC field.

- Some of the fields are used in foreign key constraints. Make indexes for those fields. Set their `DELETE` and `UPDATE` actions to `NO ACTION`. The fields are as follows:

LOCAL FIELD	FOREIGN TABLE	FOREIGN FIELD
VendorId	Vendors	VendorId
StockLocation	StockLocations	StockLocation
ShelfLocation	ShelfLocations	ShelfLocation

- Make foreign key constraints for the fields listed in step 3. Set their `DELETE` and `UPDATE` actions to `NO ACTION`. (That prevents the database from deleting or modifying a record in the foreign table if the value is needed by one of these constraints.)

How It Works

- Create the `InventoryItems` table.

The main part of the `CREATE TABLE` statement defines the fields, their data types, default values if any, and whether they are required. The following code shows the basic statement:

```
CREATE TABLE IF NOT EXISTS InventoryItems (
  UPC          VARCHAR(40)    NOT NULL,
  Description  VARCHAR(45)    NOT NULL,
  BuyPrice    DECIMAL(10,2)   NULL,
  SellPrice   DECIMAL(10,2)   NOT NULL,
  QuantityInStock INT        NOT NULL,
  ExpirationDate DATE        NULL,
  StockLocation VARCHAR(40)   NULL,
  ShelfLocation VARCHAR(40)   NULL,
  ReorderWhen INT            NULL,
  ReorderAmount INT          NULL,
  VendorId    INT            NULL
)
```

- Make the table's primary key the `UPC` field.

To define the primary key, you simply add a `PRIMARY KEY` clause to the statement like this:

```
PRIMARY KEY (UPC),
```

- Make indexes for fields used in foreign key constraints.

Creating indexes for fields used in foreign key constraints is not mandatory (at least in many databases), but it makes matching up the related values in the two tables faster. Some databases, such as MySQL Workbench, create indexes for fields used in foreign keys by default.

For this example, use the `INDEX` clause to define these indexes. The following lines of code create the indexes:

```
INDEX Index_VendorId (VendorId ASC),
```

continues

(continued)

```
INDEX Index_StockLocation (StockLocation ASC),
INDEX Index_ShelfLocation (ShelfLocation ASC),
```

4. Make foreign key constraints for the fields listed in step 3.

Use the `CONSTRAINT` clause to define the constraints. Give the constraints names that indicate the table and fields, and the fact that these are foreign key constraints. The following code defines these constraints:

```
CONSTRAINT FK_InventoryItems_VendorId
  FOREIGN KEY (VendorId)
  REFERENCES Vendors (VendorId)
  ON DELETE NO ACTION
  ON UPDATE NO ACTION,

CONSTRAINT FK_InventoryItems_StockLocation
  FOREIGN KEY (StockLocation)
  REFERENCES StockLocations (StockLocation)
  ON DELETE NO ACTION
  ON UPDATE NO ACTION,

CONSTRAINT FK_InventoryItems_ShelfLocation
  FOREIGN KEY (ShelfLocation)
  REFERENCES ShelfLocations (ShelfLocation)
  ON DELETE NO ACTION
  ON UPDATE NO ACTION
```

Note that the foreign key fields must be indexes in the foreign table, at least in MariaDB.

The following code shows the complete `CREATE TABLE` statement:

```
CREATE TABLE IF NOT EXISTS InventoryItems (
  UPC          VARCHAR(40)    NOT NULL,
  Description  VARCHAR(45)    NOT NULL,
  BuyPrice    DECIMAL(10,2)   NULL,
  SellPrice   DECIMAL(10,2)   NOT NULL,
  QuantityInStock INT        NOT NULL,
  ExpirationDate DATE        NULL,
  StockLocation VARCHAR(40)   NULL,
  ShelfLocation VARCHAR(40)   NULL,
  ReorderWhen INT            NULL,
  ReorderAmount INT          NULL,
  VendorId    INT            NULL,

  PRIMARY KEY (UPC),

  INDEX Index_VendorId (VendorId ASC),
  INDEX Index_StockLocation (StockLocation ASC),
  INDEX Index_ShelfLocation (ShelfLocation ASC),

  CONSTRAINT FK_InventoryItems_VendorId
    FOREIGN KEY (VendorId)
```

```

REFERENCES Vendors (VendorId)
ON DELETE NO ACTION
ON UPDATE NO ACTION,

CONSTRAINT FK_InventoryItems_StockLocation
FOREIGN KEY (StockLocation)
REFERENCES StockLocations (StockLocation)
ON DELETE NO ACTION
ON UPDATE NO ACTION,

CONSTRAINT FK_InventoryItems_ShelfLocation
FOREIGN KEY (ShelfLocation)
REFERENCES ShelfLocations (ShelfLocation)
ON DELETE NO ACTION
ON UPDATE NO ACTION
)

```

The following chapter has more to say about using the `CREATE TABLE` statement to build databases.

CREATE INDEX

The previous `CREATE TABLE` example uses `INDEX` clauses to define indexes for the table as it is being created. The `CREATE INDEX` statement adds an index to a table after the table has been created.

For example, the following statement adds an index named `IDX_People_Names` to the `People` table. This index makes it easier to search the table by using the records' combined `FirstName/LastName` fields:

```
CREATE INDEX IDX_People_Names ON People (FirstName, LastName)
```

Some databases, such as PostgreSQL, don't let you create indexes inside a `CREATE TABLE` statement. In that case, you must use `CREATE INDEX` if you want the table to have an index. You can also use `CREATE INDEX` to add an index to a table if you didn't realize you would need one or you just forgot to do it earlier.

There's also sometimes a strategic reason to add the index after you create the table. Relational databases use complicated self-balancing trees to provide indexes. When you add or delete a record, the database must perform a nontrivial amount of work to update its index structures. If you add the records in sorted order, as is often the case when you first populate the database, this can mean even more work than usual because the tree structures tend to have trouble with sorted values so they perform a lot more work.

When you create an index after the table is populated, the database must perform a fair amount of work to build the index tree, but it has a big advantage that it doesn't when indexing records one at a time: it knows how many records are in the table. Instead of resizing the index tree as it is built one record at a time, the database can build a big empty tree, and then fill it with data.

Some databases may not use this information effectively, but some can fill the table and then add an index more quickly than they can fill the table if the index is created first. Note that the difference is

small, so you probably shouldn't worry about creating the indexes separately unless you are loading a *lot* of records.

DROP

The `DROP` statement removes an object from the database. For example, the following statement removes the index named `IDX_People_Names` from the `People` table in a MariaDB database:

```
DROP INDEX IDX_People_Names ON People
```

The following statement shows the Transact-SQL language version of the previous command:

```
DROP INDEX People.IDX_People_Names
```

The following shows the PostgreSQL version:

```
DROP INDEX IDX_People_Names
```

The other most useful `DROP` statement is `DROP TABLE`. You can add the `IF EXISTS` clause to make the database ignore the command if the table does not already exist. That makes it easier to write scripts that drop tables before creating them. (If you don't add `IF EXISTS` and you try to drop a table that doesn't exist, the script will crash.)

The following command removes the `People` table from the database:

```
DROP TABLE IF EXISTS People
```

INSERT

The `INSERT` statement adds data to a table in the database. This command has several variations. For the following examples, assume the `People` table was created with the following command:

```
CREATE TABLE IF NOT EXISTS People (  
    PersonId      INT          NOT NULL      AUTO_INCREMENT,  
    FirstName     VARCHAR(45)  NOT NULL      DEFAULT '<missing>',  
    LastName      VARCHAR(45)  NOT NULL      DEFAULT '<none>',  
    State         VARCHAR(10)  NULL,  
    PRIMARY KEY (PersonId)  
)
```

The simplest form of the `INSERT` statement lists the values to be inserted in the new record after the `VALUES` keyword. The values must have the correct data types and must be listed in the same order as the fields in the table.

The following command inserts a new record in the `People` table:

```
INSERT INTO People VALUES (1, "Rod", "Stephens", "CO")
```

QUALITY QUOTES

SQL doesn't care whether you use single or double quotes. That can be useful when you need to compose a SQL statement in a particular programming language. For example, the following code builds the earlier `INSERT` statement in C#:

```
string statement = "INSERT INTO People
VALUES (1, 'Rod', 'Stephens', 'CO')"
```

This code uses double quotes to delimit the string, so it's convenient to use single quotes to delimit the values inside the statement.

Some databases will not let you specify values for `AUTO INCREMENT` fields such as `PersonId` in this example. If you specify the value `null` for such a field, the database automatically generates a value for you. (However, some databases won't even let you specify `null`. In that case, you must use a more complicated version of `INSERT` that lists the fields so you can omit the `AUTO INCREMENT` field.)

If you replace a value with the keyword `DEFAULT`, the database uses that field's default value if it has one.

When it executes the following command, the database automatically generates a `PersonId` value, the `FirstName` value defaults to `<missing>`, the `LastName` value is set to `Markup`, and the `State` value is set to `null`:

```
INSERT INTO People VALUES (null, DEFAULT, "Markup", null)
```

NOTE *Pop quiz: How does the database know that it should not set the `FirstName` field to the value `DEFAULT` to make this person's name be "DEFAULT Markup"? As you can probably guess, the database knows because `DEFAULT` is not in quotes. There's a difference between "DEFAULT" and `DEFAULT`.*

The next form of the `INSERT` statement explicitly lists the fields that it will initialize. The values in the `VALUES` clause must match the order of those listed earlier. Listing the fields that you are going to enter lets you omit some fields or change the order in which they are given.

The following statement creates a new `People` record. It explicitly sets the `FirstName` field to `Snortimer` and the `State` field to `Confusion`. The database automatically generates a new `PersonId` value and the `LastName` value gets its default value `<none>`:

```
INSERT INTO People (FirstName, State) VALUES ("Snortimer", "Confusion")
```

The final version of `INSERT INTO` described here gets the values that it will insert from a `SELECT` statement (described in the next section) that pulls values from a table.

The following example inserts values into the `SmartPeople` table's `LastName` and `FirstName` fields. It gets the values from a query that selects `FirstName` and `LastName` values from the `People` table where the corresponding record's `State` value is not `Confusion`:

```
INSERT INTO SmartPeople (LastName, FirstName)
  SELECT LastName, FirstName
  FROM People
  WHERE State <> "Confusion"
```

Unlike the previous `INSERT` statements, this version may insert many records in the table if the query returns a lot of data.

SELECT

The `SELECT` command retrieves data from the database. This is one of the most often used and complex SQL commands. The basic syntax is as follows:

```
SELECT select_clause
FROM from_clause
[ WHERE where_clause ]
[ GROUP BY group_by_clause ]
[ ORDER BY order_by_clause [ ASC | DESC ] ]
```

The parts in square brackets are optional and the vertical bar between `ASC` and `DESC` means you can include one or the other of those keywords.

The following sections describe the main clauses in more detail.

SELECT Clause

The `SELECT` clause specifies the fields that you want the query to return.

If a field's name is unambiguous given the tables selected by the `FROM` clause (described in the next section), you can simply list the field's name as in `FirstName`.

If multiple tables listed in the `FROM` clause have a field with the same name, you must put the table's name in front of the field's name as in `People.FirstName`.

The special value `*` tells the database that you want to select all of the available fields. If the query includes more than one table in the `FROM` clause and you want all of the fields from a specific table, you can include the table's name before the asterisk, as in `People.*`.

For example, the following query returns all the fields for all the records in the `People` table:

```
SELECT * FROM People
```

Optionally, you can give a field an alias by following it with the keyword `AS` and the alias that you want it to have. When the query returns, it acts as if that field's name is whatever you used as an alias. This is useful for such things as differentiating among fields with the same name in different tables, for creating a new field name that a program can use for a nicer display (for example, changing the `CustName` field to `Customer Name`), or for creating a name for a calculated column.

A particularly useful option you can add to the `SELECT` clause is `DISTINCT`. This makes the database return only one copy of each set of values that are selected.

For example, suppose the `Orders` table contains customer first and last names. The following MySQL query selects the `FirstName` and `LastName` values from the table, concatenates them into a single field with a space in between, and gives that calculated field the alias `Name`. The `DISTINCT` keyword means the query will only return one of each `Name` result even if a single customer has many records in the table.

```
SELECT DISTINCT CONCAT(FirstName, " ", LastName) AS Name FROM Orders
```

The following code shows the Transact-SQL equivalent of this statement:

```
SELECT DISTINCT FirstName + " " + LastName AS Name FROM Orders
```

FROM Clause

The `FROM` clause lists the tables from which the database should pull data. Normally if the query pulls data from more than one table, it either uses a `JOIN` or a `WHERE` clause to indicate how the records in the tables are related.

For example, the following statement selects information from the `Orders` and `OrderItems` tables. It matches records from the two using a `WHERE` clause that tells the database to associate `Orders` and `OrderItems` records that have the same `OrderId` value.

```
SELECT * FROM Orders, OrderItems
WHERE Orders.OrderId = OrderItems.OrderId
```

Several different kinds of `JOIN` clauses perform roughly the same function as the previous `WHERE` clause. They differ in how the database handles records in one table that have no corresponding records in the other table.

For example, suppose the `Courses` table contains the names of college courses and holds the values shown in Table 26.5.

TABLE 26.5: Courses records

COURSEID	COURSENAME
CS 120	Database Design
CS 245	The Customer: A Necessary Evil
D? = h@p	Introduction to Cryptography

Also suppose the `Enrollments` table contains the information in Table 26.6 about students taking classes.

TABLE 26.6: Enrollments records

FIRSTNAME	LASTNAME	COURSEID
Guinevere	Conkle	CS 120
Guinevere	Conkle	CS 101
Heron	Stroh	CS 120
Heron	Stroh	CS 245
Maxene	Quinn	CS 245

Now, consider the following query:

```
SELECT * FROM Enrollments, Courses
WHERE Courses.CourseId = Enrollments.CourseId
```

This may seem like a simple enough query that selects enrollment information plus each student's class name. For example, one of the records returned would be as follows:

FIRSTNAME	LASTNAME	COURSEID	COURSEID	COURSENAME
Guinevere	Conkle	CS 120	CS 120	Database Design

Note that the result contains two CourseId values, one from each table.

The way in which the kinds of JOIN clause differ is in the way they handle missing values. If you look again at the tables, you'll see that no students are currently enrolled in Introduction to Cryptography. You'll also find that Guinevere Conkle is enrolled in CS 101, which has no record in the Courses table.

The following query that uses a WHERE clause discards any records in one table that have no corresponding records in the second table:

```
SELECT * FROM Enrollments, Courses
WHERE Courses.CourseId = Enrollments.CourseId
```

The following statement uses the INNER JOIN clause to produce the same result:

```
SELECT * FROM Enrollments INNER JOIN Courses
ON (Courses.CourseId = Enrollments.CourseId)
```

Table 26.7 shows the results of these two queries.

TABLE 26.7: Courses records

FIRSTNAME	LASTNAME	COURSEID	COURSEID	COURSENAME
Guinevere	Conkle	CS 120	CS 120	Database Design
Heron	Stroh	CS 120	CS 120	Database Design
Maxene	Quinn	CS 245	CS 245	The Customer: A Necessary Evil
Heron	Stroh	CS 245	CS 245	The Customer: A Necessary Evil

The following statement selects the same records except it uses the `LEFT JOIN` clause to favor the table listed to the left of the clause in the query (`Orders`). If a record appears in that table, it is listed in the result even if there is no corresponding record in the other table.

```
SELECT * FROM Orders LEFT JOIN OrderItems
ON (Orders.OrderId = OrderItems.OrderId)
```

Table 26.8 shows the result of this query. Notice that the results include a record for Guinevere Conkle's CS 101 enrollment even though CS 101 is not listed in the `Courses` table. In that record, the fields that should have come from the `Courses` table have null values.

TABLE 26.8: Courses records

FIRSTNAME	LASTNAME	COURSEID	COURSEID	COURSENAME
Guinevere	Conkle	CS 120	CS 120	Database Design
Heron	Stroh	CS 120	CS 120	Database Design
Maxene	Quinn	CS 245	CS 245	The Customer: A Necessary Evil
Guinevere	Conkle	CS 101	NULL	NULL
Heron	Stroh	CS 245	CS 245	The Customer: A Necessary Evil

Conversely the `RIGHT JOIN` clause makes the query favor the table to the right of the clause, so it includes all the records in that table even if there are no corresponding records in the other table. The following query demonstrates the `RIGHT JOIN` clause:

```
SELECT * FROM Orders RIGHT JOIN OrderItems
ON (Orders.OrderId = OrderItems.OrderId)
```

Table 26.9 shows the result of this query. This time there is a record for the Introduction to Cryptography course even though no student is enrolled in it.

TABLE 26.9: Courses records

FIRSTNAME	LASTNAME	COURSEID	COURSEID	COURSENAME
Guinevere	Conkle	CS 120	CS 120	Database Design
Heron	Stroh	CS 120	CS 120	Database Design
Maxene	Quinn	CS 245	CS 245	The Customer: A Necessary Evil
Heron	Stroh	CS 245	CS 245	The Customer: A Necessary Evil
NULL	NULL	NULL	D? = h@p	Introduction to Cryptography

Both the left and right joins are called *outer joins* because they include records that are outside of the “natural” records that include values from both tables.

Many databases, including MySQL and Access, don’t provide a join to select all records from both tables like a combined left and right join. You can achieve a similar result by using the UNION keyword to combine the results of a left and right join. The following query uses the UNION clause:

```
SELECT * FROM Courses LEFT JOIN Enrollments
  ON Courses.CourseId=Enrollments.CourseId
UNION
SELECT * FROM Courses RIGHT JOIN Enrollments
  ON Courses.CourseId=Enrollments.CourseId
```

Table 26.10 shows the results.

TABLE 26.10: Courses records

FIRSTNAME	LASTNAME	COURSEID	COURSEID	COURSENAME
Guinevere	Conkle	CS 120	CS 120	Database Design
Heron	Stroh	CS 120	CS 120	Database Design
Maxene	Quinn	CS 245	CS 245	The Customer: A Necessary Evil

FIRSTNAME	LASTNAME	COURSEID	COURSEID	COURSENAME
Guinevere	Conkle	CS 101	NULL	NULL
Heron	Stroh	CS 245	CS 245	The Customer: A Necessary Evil
NULL	NULL	NULL	D? = h@p	Introduction to Cryptography

WHERE Clause

The `WHERE` clause provides a filter to select only certain records in the tables. It can compare the values in the tables to constants, expressions, or other values in the tables. You can use parentheses and logical operators such as `AND`, `NOT`, and `OR` to build complicated selection expressions.

For example, the following query selects records from the `Enrollments` and `Courses` tables where the `CourseId` values match and the `CourseId` is alphabetically less than `CS 200` (upper division classes begin with `CS 200`):

```
SELECT * FROM Enrollments, Courses
WHERE Enrollments.CourseId = Courses.CourseId
      AND Courses.CourseId < 'CS 200'
```

Table 26.11 shows the result.

TABLE 26.11: Courses records

FIRSTNAME	LASTNAME	COURSEID	COURSEID	COURSENAME
Guinevere	Conkle	CS 120	CS 120	Database Design
Heron	Stroh	CS 120	CS 120	Database Design

GROUP BY Clause

If you include an aggregate function such as `AVERAGE` or `SUM` in the `SELECT` clause, the `GROUP BY` clause tells the database which fields to look at to determine whether values should be combined.

For example, the following query selects the `CustomerId` field from the `CreditsAndDebits` table. It also selects the sum of the `Amount` field values. The `GROUP BY` clause makes the query combine values that have matching `CustomerId` values for calculating the sums. The result is a list of every `CustomerId` and the corresponding current total balance calculated by adding up all of that customer's credits and debits.

```
SELECT CustomerId, SUM(Amount) AS Balance
FROM CreditsAndDebits
GROUP BY CustomerId
```

ORDER BY Clause

The `ORDER BY` clause specifies a list of fields that the database should use to sort the results. The optional keyword `DESC` after a field makes the database sort that field's values in descending order. (The default order is ascending. You can explicitly include the `ASC` keyword if you want to make the order obvious.)

The following query selects the `CustomerId` field and the total of the `Amount` values for each `CustomerId` from the `CreditsAndDebits` table. It sorts the results in descending order of the total amount so that you can see who has the largest balance first.

```
SELECT CustomerId, SUM(Amount) AS Balance
FROM CreditsAndDebits
GROUP BY CustomerId
ORDER BY Balance DESC
```

The following query selects the distinct first and last name combinations from the `Enrollments` table and orders the results by `LastName` and then by `FirstName`. (For example, if two students have the same last name `Zappa`, then `Dweezil` comes before `Moon Unit`.)

```
SELECT DISTINCT LastName, FirstName
FROM Enrollments
ORDER BY LastName, FirstName
```

BE SELECTIVE

Though the `SELECT` statement has many variations, the basic ideas are reasonably intuitive so a little practice will go a long way toward learning how to write `SELECT` statements.

Suppose the `Authors` table has fields `AuthorId`, `FirstName`, and `LastName`. Suppose also that the `Books` table has fields `AuthorId`, `Title`, `ISBN` (International Standard Book Number), `MSRP` (Manufacturer's Suggested Retail Price), `Year`, and `Pages`. Write a query to select book titles, prices, and author names. Concatenate the authors' first and last names and give the result the alias `Author`. Select only books where `MSRP` is less than \$10.00 (which these days may not return many results). Sort the results by price in ascending order.

1. Write the `SELECT` clause.
2. Write a `FROM` clause to select an inner join using the `Authors` and `Books` tables.
3. Write a `WHERE` clause to select records where `MSRP < $10.00`.
4. Write an `ORDER BY` clause to sort the results by `MSRP` in ascending order.

How It Works

1. Write the `SELECT` clause.

The basic `SELECT` clause includes the `Books` table's `Title` and `MSRP` fields. It also concatenates the `Authors` table's `FirstName` and `LastName` fields with a space in between. The following code shows the `SELECT` clause:

```
SELECT MSRP, Title, CONCAT(FirstName, " ", LastName) AS Author
```

2. Write a `FROM` clause to select an inner join using the Authors and Books tables.

The `FROM` clause selects an inner join using the Authors and Books tables. This query uses the tables' `AuthorId` fields to match corresponding records. The following code shows the `FROM` clause:

```
FROM Books INNER JOIN Authors
ON (Books.AuthorId = Authors.AuthorId)
```

3. Write a `WHERE` clause to select records where `MSRP < $10.00`.

The `WHERE` clause adds a further condition on the selected records, requiring that the `MSRP` value be less than 10. The following code shows the `WHERE` clause:

```
WHERE MSRP < 10
```

4. Write an `ORDER BY` clause to sort the results by `MSRP` in ascending order.

The `ORDER BY` clause sorts the results by `MSRP`. The default order for `ORDER BY` clauses is ascending, so the statement doesn't need to explicitly include the `ASC` keyword. The following code shows the `ORDER BY` clause:

```
ORDER BY MSRP
```

The following code shows the complete query.

```
SELECT MSRP, Title, CONCAT(FirstName, " ", LastName) AS Author
FROM Books INNER JOIN Authors
ON (Books.AuthorId = Authors.AuthorId)
WHERE MSRP < 10
ORDER BY MSRP
```

UPDATE

The `UPDATE` statement changes the values in one or more records' fields. The basic syntax is:

```
UPDATE table SET field = new_value
WHERE where_clause
```

For example, the following statement fixes a typo in the Books table. It changes the `Title` field's value to "The Portable Door" in any records that currently have the incorrect `Title` "The Potable Door."

```
UPDATE Books SET Title = "The Portable Door"
WHERE Title = "The Potable Door"
```

The `WHERE` clause is extremely important in an `UPDATE` statement. If you forget the `WHERE` clause, the update affects every record in the table! (See the earlier warning about do-overs, crossed fingers, and backsies.) If you forget the `WHERE` clause in the previous example, the statement would change the title of *every* book to "The Portable Door," which is probably not what you intended.

The effects of the `UPDATE` statement are immediate and irreversible so forgetting the `WHERE` clause can be disastrous. (In fact, some developers have suggested that an `UPDATE` statement without a `WHERE` clause should generate an error unless you take special action to say "yes, I'm really, really sure.")

UPDATES

Suppose the Sales table includes the fields EmployeeId, Year, Month (which holds three-letter month abbreviations), TotalSales (the number of light sabers sold), and Salary. Write an update statement that gives a \$100 bonus to employees who made their sales quota of 10 light sabers sold during the month of August 2025.

1. Write the UPDATE statement, including the table name and the SET clause.
2. Write the WHERE clause to select the records that should be updated.

How It Works

1. Write the UPDATE statement, including the table name and the SET clause.

The UPDATE clause will affect the Sales table. You need to add \$100 to the Salary field for certain records. To do that, the clause sets the Salary value to current Salary value plus \$100. The following code shows the UPDATE clause:

```
UPDATE Sales
SET Salary = Salary + 100
```

2. Write the WHERE clause to select the records that should be updated.

The WHERE clause has three parts that require the number of light sabers sold to be at least 10, the month to be AUG, and the year to be 2025. The following code shows the WHERE clause:

```
WHERE TotalSales >= 10
AND Month = "AUG"
AND Year = 2025
```

The following code shows the complete UPDATE statement:

```
UPDATE Sales
SET Salary = Salary + 100
WHERE TotalSales >= 10
AND Month = "AUG"
AND Year = 2025
```

DELETE

The DELETE statement removes records from a table. The basic syntax is:

```
DELETE FROM table
WHERE where_clause
```

For example, the following statement removes all records from the Books table where the AuthorId is 7:

```
DELETE FROM Books
WHERE AuthorId = 7
```

As is the case with `UPDATE`, the `WHERE` clause is very important in a `DELETE` statement. If you forget the `WHERE` clause, the `DELETE` statement removes every record from the table mercilessly and without remorse.

SUMMARY

SQL is a powerful tool. The SQL commands described in this chapter let you perform basic database operations such as determining the database's structure and contents. This chapter explained how to do the following:

- Use the `CREATE TABLE` statement to create a table with a primary key, indexes, and foreign key constraints.
- Use `INSERT` statements to add data to a table.
- Use `SELECT` statements to select data from one or more tables, satisfying specific conditions, and sort the result.
- Use the `UPDATE` statement to modify the data in a table.
- Use the `DELETE` statement to remove records from a table.

SQL statements let you perform simple tasks with a database such as creating a new table or inserting a record. By combining many SQL statements into a script, you can perform elaborate procedures such as creating and initializing a database from scratch. The next chapter explains this topic in greater detail. It describes the benefits of using scripts to create databases and discusses some of the issues that you should understand before writing those scripts.

Before you move on to Chapter 27, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

EXERCISES

1. Write SQL statements to create the three tables shown in Figure 26.2. Include the primary keys, foreign key constraints, and indexes on the fields used in those constraints.

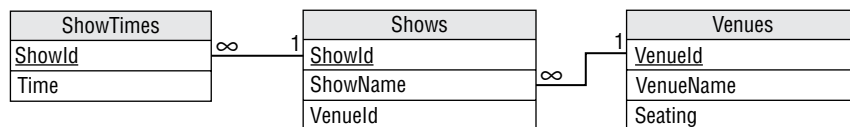


FIGURE 26.2

2. Write a series of SQL statements to insert the data shown in Figure 26.3.

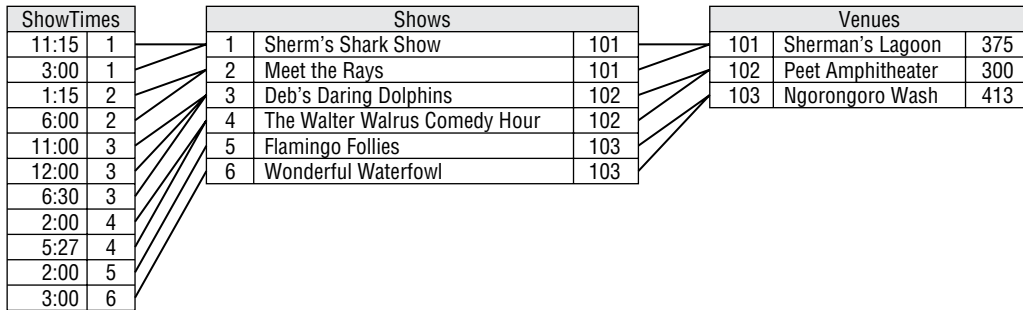


FIGURE 26.3

Hint: Use 24-hour clock times as in 14:00 for 2:00pm.

- Management has decided that no two shows should start fewer than 15 minutes apart. Write SQL statements to change the 2:00 Flamingo Follies show to 2:15 and the 3:00 Sherm's Shark Show to 3:15.

Hint: Include both the ShowTimes and Shows tables in the `UPDATE` clause. Then use a `WHERE` clause to select the correct record by Time and ShowName.

- Write a SQL statement to select data from the tables and produce the following result.

SHOW	TIME	LOCATION
Deb's Daring Dolphins	11:00am	Peet Amphitheater
Sherm's Shark Show	11:15am	Sherman's Lagoon
Deb's Daring Dolphins	12:00pm	Peet Amphitheater
Meet the Rays	1:15pm	Sherman's Lagoon
The Walter Walrus Comedy Hour	2:00pm	Peet Amphitheater
Flamingo Follies	2:15pm	Ngorongoro Wash
Wonderful Waterfowl	3:00pm	Ngorongoro Wash
Sherm's Shark Show	3:15pm	Sherman's Lagoon
The Walter Walrus Comedy Hour	5:27pm	Peet Amphitheater
Meet the Rays	6:00pm	Sherman's Lagoon
Deb's Daring Dolphins	6:30pm	Peet Amphitheater

Hints:

- Sort the results by Show and then Time.
- In MySQL at least, `SHOW` is a keyword, which means you cannot simply use `AS Show` to give the ShowName field the alias Show because that would confuse the database. Instead put quotes around the word Show wherever you need it as in `AS "Show"` and `ORDER BY "Show"`.
- To format the times as in 6:30 PM in MySQL, use the `DATE_FORMAT` function. To make the times line up nicely on the right, use the `LPAD` function to pad them on the left with spaces. The following code shows how:

```
LPAD (DATE_FORMAT (Time, "%l:%i %p"), 8, " ")
```

- Unfortunately when you pad the times, the `ORDER BY` statement treats the result as a string rather than a time. That means, for example, " 3:00 PM" comes alphabetically before "11:00 AM" because " 3:00 AM" begins with a space. To fix this, use the `TIME` function to convert the times as strings back into times in the `ORDER BY` clause. For example, if you use the alias Time for the result of this field, then the `ORDER BY` clause should contain the following:

```
TIME (Time)
```

5. Write a SQL statement to select data from the tables and produce the following result.

TIME	SHOW	LOCATION
11:00am	Deb's Daring Dolphins	Peet Amphitheater
11:15am	Sherm's Shark Show	Sherman's Lagoon
12:00pm	Deb's Daring Dolphins	Peet Amphitheater
1:15pm	Meet the Rays	Sherman's Lagoon
2:00pm	The Walter Walrus Comedy Hour	Peet Amphitheater
2:15pm	Flamingo Follies	Ngorongoro Wash
3:00pm	Wonderful Waterfowl	Ngorongoro Wash
3:15pm	Sherm's Shark Show	Sherman's Lagoon
5:27pm	The Walter Walrus Comedy Hour	Peet Amphitheater
6:00pm	Meet the Rays	Sherman's Lagoon
6:30pm	Deb's Daring Dolphins	Peet Amphitheater

See Exercise 4 for hints.

27

Building Databases with SQL Scripts

The previous chapter provided an introduction to using SQL to create and manage databases. That chapter also hinted at techniques for using SQL scripts to make database maintenance easier.

This chapter goes a little further. It discusses some of the details that you need to take into account when you use scripts to manage a database.

In this chapter, you learn how to:

- Know when scripts can be useful.
- Build tables in a valid order.
- Insert data into tables in a valid order.
- Drop tables in a valid order.

WHY BOTHER WITH SCRIPTS?

SQL statements let you create, populate, modify, and delete the tables in a database. In many database products, SQL statements even let you create and destroy the database itself. For example, MySQL's `CREATE DATABASE` and `DROP DATABASE` statements create and destroy databases.

If you put these SQL commands in a script, you can rerun that script whenever it's necessary. You can easily rebuild the database if it gets corrupted, make copies of the database on other computers, fill the tables with data to use when running tests, and reinitialize the data after the tests are finished.

Being able to reinitialize the data to a known state can also be very helpful in tracking down bugs. It's extremely hard to find a bug if it just pops up occasionally and then disappears again. If you can reinitialize the database and then make a bug happen by following a series of predictable steps, it's much easier to find and fix the problem.

SCRIPT CATEGORIES

Scripts that are useful for managing databases fall into at least four categories, described in the following subsections.

Database Creation Scripts

Database creation scripts build the database's structure. They build the tables, primary keys, indexes, foreign key constraints, field and table check constraints, and all of the other structure that doesn't change as the data is modified.

Basic Initialization Scripts

Basic initialization scripts initialize basic data that is needed before the database can be useful. This includes system parameter tables, lookup tables, and other tables that hold data that changes only rarely when the database is in use.

For example, you might use one of these scripts to initialize a list of allowed states or regions, physical constants (the speed of light, Avogadro's number, Finagle's Variable Constant), or define data type conversion constants (how many centimeters in an inch, how many millimeters in an attoparsec, how many seconds in a microfortnight).

Data Initialization Scripts

These scripts place data in tables. These range from small scripts that initialize a few values to huge monster scripts that insert thousands of records into the database.

Often, it's useful to have a separate subcategory for test scripts that fill the tables with data for use in specific tests. You would run a script to prepare the data, and then run the test. If the test can be executed by SQL statements, the script might perform the test too. Sometimes, it may be useful to have a separate test initialization script for every use case defined by your requirements documents.

It's also often useful to have separate scripts to initialize different parts of the database. For example, you might have a script that creates users, another that creates orders, and a third that creates invoice and payment data.

You can build scripts that invoke smaller scripts to perform larger tasks. For example, you might make a test initialization script that calls the standard user initialization script, and then inserts or updates specific records in other tables to prepare for the test that you are about to perform.

For example, the following MySQL script invokes three others. It creates a database, selects it, and then calls three other scripts that create a table, insert some data, and select the data. It then drops the database.

```
CREATE DATABASE MultipleScripts;  
  
USE MultipleScripts;  
  
SOURCE C:\Rod\DB Design\MultiScript1.sql  
SOURCE C:\Rod\DB Design\MultiScript2.sql  
SOURCE C:\Rod\DB Design\MultiScript3.sql  
  
DROP DATABASE MultipleScripts;
```

It may not always be necessary to break the database scripts into little pieces, but on some projects it's even useful to have two separate scripts to create and initialize each table. Then if you change a table, it's easy to find the creation and initialization code for it. Higher-level scripts can then call those scripts to build and initialize the database.

Similarly, if you have a bug in one table (perhaps it's missing a field or foreign key constraint), you can fix its creation code and rerun the scripts.

Cleanup Scripts

Often it's easier to simply drop the database and re-create it than it is to clean up the mess left by a test, but occasionally it's useful to truncate or drop only some of the tables. For example, if the database contains millions of records, it may be easier and faster to repair changes made by tests than to rebuild the whole thing from scratch.

It's not always easy to undo changes made by a complex series of tests, particularly if you later make changes to the tests. In fact, it's often hard to even tell if you've successfully undone the changes. For those reasons, I usually prefer to rebuild the database from scratch when possible.

Saving Scripts

Just as any other piece of software does, scripts change during development, testing, and use. Also as is the case for other types of software, it's often useful to look back at previous versions of scripts. To ensure that those versions are available, always keep the old versions of scripts. Later if you discover a problem, you can compare the current and older versions to see what's changed.

One way to keep old scripts is to use version control software. Programs such as Git (see <https://github.com>), CVS (Concurrent Versions System, see www.nongnu.org/cvs) and Apache Subversion (abbreviated SVN and found at <https://subversion.apache.org>) keep track of different versions of files. You can store your scripts in one of those systems, and then update the files whenever you create a new version. Then you can always go back and see the older versions if you have a reason. (Search online for **version control software** to see many other tools.)

GITTYUP

Git is a version control system that lets you manage documents. GitHub is a company that lets you manage Git repositories in the cloud. GitHub is a for-profit company, but its free edition is good enough for basic needs.

If you don't feel like using a formal version control system, you can invent your own in one of several ways. For example, you can put a version number in the script filenames. You might make a script named `MakeUsers.sql` that fills the `Users` table. The file `MakeUsers.sql` would always contain the most current version and `MakeUsers001.sql`, `MakeUsers002.sql`, and so forth would contain older versions.

Another approach is to email scripts to yourself when you revise them. Later, you can search through the emails sorted by date to see the older versions. To keep your normal email account uncluttered so you can easily find those offers for mortgage debt elimination, jobs as a rebate processor, and pleas for help in getting \$10 million out of Nigeria (you get to keep \$3 million for your trouble), create a separate email account to hold the scripts.

ORDERING SQL COMMANDS

One issue that you should consider when building scripts is that some commands must be executed in a particular order. For example, suppose the `Races` table for your cheese-rolling database (search online or see www.cheeseprofessor.com/blog/british-cheese-rolling) has a `WinnerId` field that refers to the `Racers` table's `RacerId` field as a foreign key. In that case, you must create the `Racers` table before you create the `Races` table. Clearly, the `Races` table cannot refer to a field in a table that doesn't yet exist.

Usually, you can create the tables in some order so none refers to another table that doesn't yet exist. (Tip: build lookup tables first.) If for some bizarre reason there is no such ordering, you can use an `ALTER TABLE ADD FOREIGN KEY` statement (or a similar statement in whatever version of SQL you are using) to create the foreign key constraints after you build all of the tables.

You may also be able to tell the database to turn off constraint checking while you build the tables. For example, the following MySQL script builds three tables that are mutually dependent. `TableA` refers to `TableB`, `TableB` refers to `TableC`, and `TableC` refers to `TableA`.

```
SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL';

CREATE DATABASE CycleDb;
USE CycleDb;

CREATE TABLE TableC (
  CType    VARCHAR(10)    NOT NULL,
  AType    VARCHAR(10)    NULL,

  PRIMARY KEY (CType),

  INDEX FK_CrefA (AType ASC),

  CONSTRAINT FK_CrefA
    FOREIGN KEY (AType)
    REFERENCES TableA (AType)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION
```

```

);

CREATE TABLE TableB (
    BType    VARCHAR(10)    NOT NULL,
    CType    VARCHAR(10)    NULL,

    PRIMARY KEY (BType),

    INDEX FK_BrefC (CType ASC),

    CONSTRAINT FK_BrefC
        FOREIGN KEY (CType)
        REFERENCES TableC (CType)
        ON DELETE NO ACTION
        ON UPDATE NO ACTION
);

CREATE TABLE TableA (
    AType    VARCHAR(10)    NOT NULL,
    BType    VARCHAR(10)    NULL,

    PRIMARY KEY (AType),

    INDEX FK_ArefB (BType ASC),

    CONSTRAINT FK_ArefB
        FOREIGN KEY (BType)
        REFERENCES TableB (BType)
        ON DELETE NO ACTION
        ON UPDATE NO ACTION
);

INSERT INTO TableA VALUES("A value", "B value");
INSERT INTO TableB VALUES("B value", "C value");
INSERT INTO TableC VALUES("C value", "A value");

SET SQL_MODE=@OLD_SQL_MODE;
SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS;
SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS;

SELECT * FROM TableA;
SELECT * FROM TableB;
SELECT * FROM TableC;

DROP DATABASE CycleDb;

```

The first three statements tell the database not to check for unique key values, not to check foreign key constraints, and to use traditional SQL behavior (in short, give errors instead of warnings).

The script then creates the three interdependent tables and inserts some values in them. It then restores the original values for the SQL mode, foreign key checking, and unique key checking.

The script finishes by performing some queries and dropping the database.

Just as you may need to create tables in a particular order, you may need to insert values into the tables in a particular order. For example, you'll need to create the Racers record for Abby Lampe before you create the Races record for the 2022 women's race because Abby won that race.

If you cannot find a legal ordering for the `INSERT` statements, you may be able to disable the database's checks just as you can while creating tables. The preceding script inserts records that depend on each other, so there is no valid way to enter those values without disabling the error checking.

Finally, if you delete records or entire tables, you may need to do so in a particular order. After you've built your cheese-rolling database, you cannot remove Abby Lampe's record from the Racers table before you remove the Races record that refers to her.

ORDERING TABLES

I haven't actually seen a real database with mutually dependent tables so I have always been able to put them in a valid order. To hone your ordering skills, make a list showing a valid order for creating the tables in the design shown in Figure 27.1:

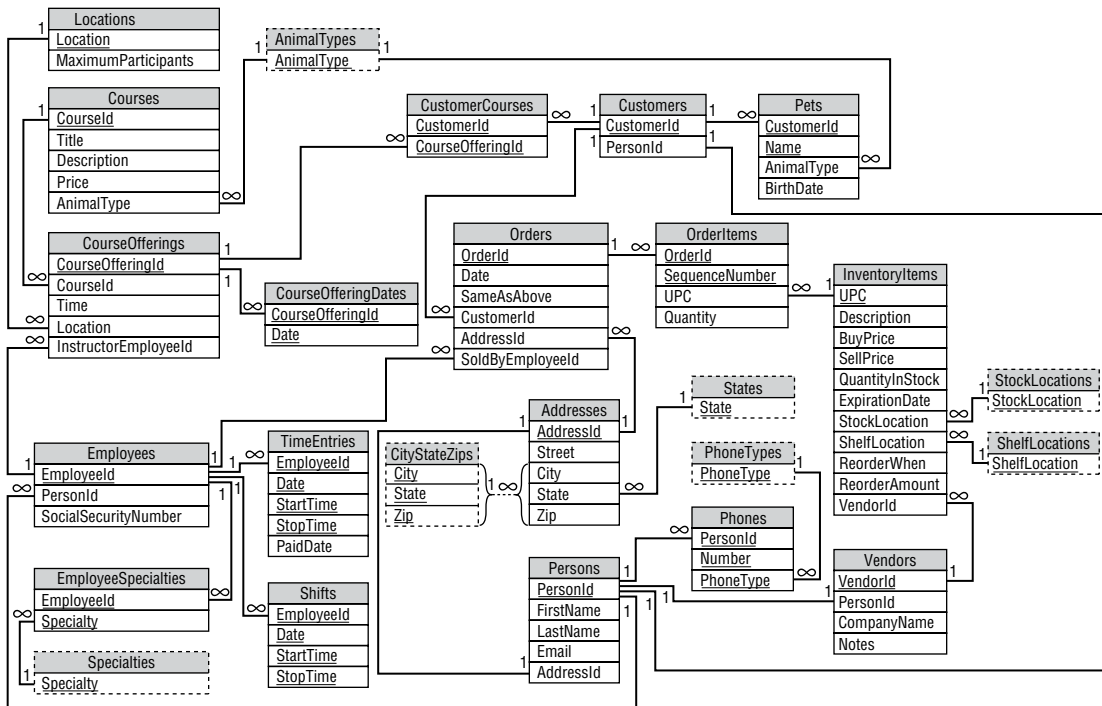


FIGURE 27.1

A list where some of the objects must come before others but not all of the relationships are known is called a *partial ordering*. Creating a full ordering that satisfies a partial ordering is called *extending the partial ordering*.

1. Define a partial ordering by making a list of the tables. Next to each table, make a *predecessor list* showing all the other tables that must be defined *before* that one.
2. Use this list to build an output list giving the tables in a valid order.

For this example, you *might* be able to intuit a valid order, but it's a complicated design so it could be tricky. Fortunately, there's a relatively simple algorithm that lets you extend partial orderings.

 - a. Look through the list you created in step 1 and find any tables that have no predecessors.
 - b. Put those tables in the output list.
 - c. Remove (or cross out) those tables from any other table's predecessor list.
 - d. Remove (or cross out) the outputted tables' rows from the list.
 - e. Repeat this process until every table is in the output list or you find a group of mutually dependent tables.

How It Works

1. Define a partial ordering by making a list of the tables.

If two tables are related by a one-to-many relationship, the table on the “many” side of the relationship depends on the table on the “one” side. For example, a CourseOfferings record refers to a Locations record (each course offering occurs at some location), so you must create the Locations table before you can create the CourseOfferings table.

If two tables are related by a one-to-one relationship, you need to think a bit harder about which depends on the other. Normally, such a relationship involves the primary key of only one of the tables. In that case, the non-primary key table depends on the primary key table. For example, the Addresses and Persons tables in Figure 27.1 have a one-to-one relationship. The relationship connects the Addresses table's primary key AddressId with the Persons table's non-primary key AddressId field, so the Persons table (non-primary key) depends on the Addresses table (primary key).

Another way to think of that is to ask, “Which table provides detail for the other table?” In this case, you would normally look up a Persons record, and then search for that person's address. You probably wouldn't look up an address, and then search for the person who lives there.

Table 27.1 shows the database's tables and their predecessors.

continues

*(continued)***TABLE 27.1:** Initial predecessors

TABLE	PREDECESSORS
Addresses	CityStateZips, States
AnimalTypes	
CityStateZips	
CourseOfferingDates	CourseOfferings
CourseOfferings	Locations, Courses, Employees
Courses	AnimalTypes
CustomerCourses	CourseOfferings, Customers
Customers	Persons
Employees	Persons
EmployeeSpecialties	Specialties, Employees
InventoryItems	StockLocations, ShelfLocations, Vendors
Locations	
OrderItems	Orders, InventoryItems
Orders	Customers, Employees, Addresses
Persons	Addresses
Pets	Customers, AnimalTypes
Phones	PhoneTypes, Persons
PhoneTypes	
ShelfLocations	
Shifts	Employees

TABLE	PREDECESSORS
Specialties	
States	
StockLocations	
TimeEntries	Employees
Vendors	Persons

(Give yourself bonus points if you cringed a bit at this table and said to yourself, “Hey, that’s not in 1NF because the Predecessors column doesn’t hold a single value!”)

- Use this list to build an output list giving the tables in a valid order.

During the first pass through this list, the AnimalTypes, CityStateZips, Locations, PhoneTypes, ShelfLocations, Specialties, States, and StockLocations tables have no predecessors. You can immediately output those tables (so you can build them first in the database creation script) and remove them from the list. (You may have been able to guess that just by looking at Figure 27.1 because these are all lookup tables. Lookup tables generally don’t rely on other tables.)

Table 27.2 shows the revised list with the lookup tables removed.

TABLE 27.2: Table predecessors after one round

TABLE	PREDECESSORS
Addresses	
CourseOfferingDates	CourseOfferings
CourseOfferings	Courses, Employees
Courses	
CustomerCourses	CourseOfferings, Customers
Customers	Persons
Employees	Persons
EmployeeSpecialties	Employees

continues

TABLE 27.2 (continued)

TABLE	PREDECESSORS
InventoryItems	Vendors
OrderItems	Orders, InventoryItems
Orders	Customers, Employees, Addresses
Persons	Addresses
Pets	Customers
Phones	Persons
Shifts	Employees
TimeEntries	Employees
Vendors	Persons

After removing the lookup tables, the revised list has two tables with no predecessors: Addresses and Courses. Output them and build them next in the database creation script. Table 27.3 shows the revised list.

TABLE 27.3: Table predecessors after two rounds

TABLE	PREDECESSORS
CourseOfferingDates	CourseOfferings
CourseOfferings	Employees
CustomerCourses	CourseOfferings, Customers
Customers	Persons
Employees	Persons
EmployeeSpecialties	Employees
InventoryItems	Vendors
OrderItems	Orders, InventoryItems

TABLE	PREDECESSORS
Orders	Customers, Employees
Persons	
Pets	Customers
Phones	Persons
Shifts	Employees
TimeEntries	Employees
Vendors	Persons

Now the list has only one table with no predecessors, Persons, and you might wonder if we'll soon reach a state where we're stuck with no tables without predecessors. Cross your fingers, output the Persons table, and update the list so it looks like Table 27.4.

TABLE 27.4: Table predecessors after three rounds

TABLE	PREDECESSORS
CourseOfferingDates	CourseOfferings
CourseOfferings	Employees
CustomerCourses	CourseOfferings, Customers
Customers	
Employees	
EmployeeSpecialties	Employees
InventoryItems	Vendors
OrderItems	Orders, InventoryItems
Orders	Customers, Employees
Pets	Customers

continues

TABLE 27.4 (continued)

TABLE	PREDECESSORS
Phones	
Shifts	Employees
TimeEntries	Employees
Vendors	

You can breathe a sigh of relief when you see that the new list contains several tables without predecessors: Customers, Employees, Phones, and Vendors. Output them and update the list to get Table 27.5.

TABLE 27.5: Table predecessors after four rounds

TABLE	PREDECESSORS
CourseOfferingDates	CourseOfferings
CourseOfferings	
CustomerCourses	CourseOfferings
EmployeeSpecialties	
InventoryItems	
OrderItems	Orders, InventoryItems
Orders	
Pets	
Shifts	
TimeEntries	

At this point, most of the tables have no predecessors, so we may be home free. Output the CourseOfferings, EmployeeSpecialties, Orders, Pets, Shifts, and TimeEntries tables and update the table to get the list in Table 27.6.

TABLE 27.6 Table predecessors after five rounds

TABLE	PREDECESSORS
CourseOfferingDates	
CustomerCourses	
OrderItems	

Now, the three remaining tables have no predecessors: CourseOfferingDates, CustomerCourses, and OrderItems. You can build those tables last when you create the database.

A complete ordering of the tables is: AnimalTypes, CityStateZips, Locations, PhoneTypes, ShelfLocations, Specialties, States, StockLocations, Addresses, Courses, Persons, Customers, Employees, Phones, Vendors, CourseOfferings, EmployeeSpecialties, InventoryItems, Orders, Pets, Shifts, TimeEntries, CourseOfferingDates, CustomerCourses, OrderItems.

Note that this is not the only possible complete ordering for these tables. Each time a group of tables had no predecessors, you could have created them in any order.

In some cases you can also mix tables from different “rounds.” For example, in Table 27.1, the AnimalTypes table has no predecessors and the Courses table depends only on that table. After you output the AnimalTypes table, you can output the Courses table without waiting for all of the other lookup tables.

TIP For more information on this and other interesting algorithms, see my book *Essential Algorithms: A Practical Approach to Computer Algorithms Using Python and C#, Second Edition (Wiley, 2019)*.

SUMMARY

SQL scripts can make building and maintaining a database much easier than working manually with database tools such as MySQL or Access. They are particularly useful for repeatedly performing tasks such as initializing the database before performing a test.

This chapter explained:

- Why scripts are useful
- Different categories of useful scripts such as database creation, basic initialization, data initialization, and cleanup
- How to save different versions of scripts
- How to create tables, insert data, remove data, and delete tables in a valid order

Scripts are useful for maintaining databases. The following chapter discusses some of the typical maintenance chores that you should perform to keep a database in working order. Before you move on to Chapter 28, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

EXERCISES

1. Consider the movie database design shown in Figure 27.2. A movie can have many actors and producers but only one director. Actors, producers, and directors are all persons and any person can hold any of those positions, sometimes simultaneously. (For example, in *Star Trek IV: The Voyage Home*, Leonard Nimoy is the director, an actor, and a writer. In *The Nutty Professor*, Eddie Murphy plays practically everyone.)

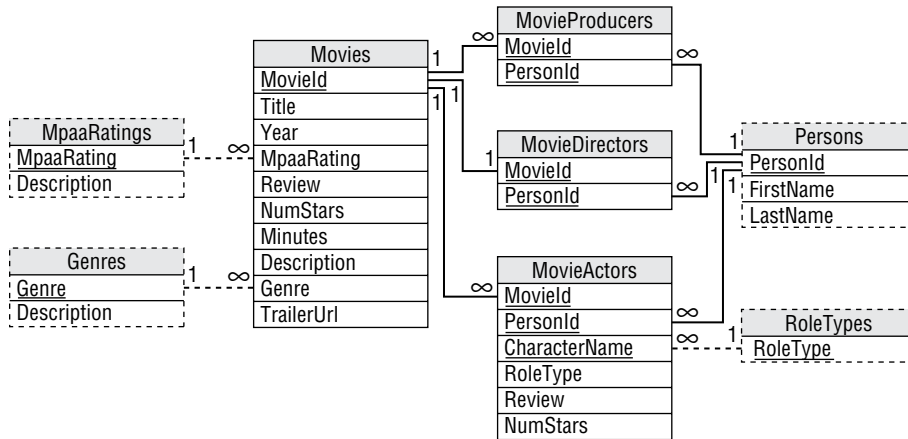


FIGURE 27.2

Find an order in which you can create the database's tables so no table is created before another that depends on it.

2. Write a SQL script to build the movie database shown in Figure 27.2.

28

Database Maintenance

At this point, you've learned how to identify customer needs, design a database, refine the design, and implement the database interactively or by using scripts. Even after you start using the database, however, the work isn't done. You (or someone) must perform regular maintenance to keep the database healthy and efficient.

Like a high-performance sports car, the database needs regular maintenance to keep it running at peak efficiency. Just as the best engineering and construction in the world won't save your engine if you drive 100,000 miles without an oil change, your database design won't give you optimal performance if you don't give it regular tune-ups. (At least the database doesn't need collision insurance.)

This chapter describes some of the maintenance chores that must be performed to keep the database working smoothly. Unfortunately, the details of performing these chores differ greatly in different databases, so the exact steps you need to perform are not included here. Instead, this chapter describes the issues that you should keep in mind when you design the database's maintenance schedule. You should consult the documentation for your particular database product to flesh out the details.

In this chapter, you will learn the following:

- What tasks are necessary to keep a database functional
- How to schedule backups to safeguard data
- What you can do to keep a database working efficiently

BACKUPS

Backups are one of the most important parts of database maintenance. Unfortunately, they are also sometimes the most neglected part of a larger software effort. No database design can protect against system crashes, power failures, and the user accidentally deleting critical information. Without good backups, you could lose significant chunks of data.

NOTE *In one project, a ne'er-do-well tried to delete our entire code database! Fortunately, he was stopped in time. We had backups anyway, but we would probably have wasted an hour or two recovering the lost files. Unfortunately, corporate policy didn't allow burying him up to his neck in an ant hill.*

There are two main kinds of backups that many databases support: full and incremental.

A full backup makes a copy of everything in the database. Depending on the size of the database, this might take a lot of time and disk space. For a reasonably small database that is only used during business hours, you might have the computer automatically back up the database at midnight. Even if the backup takes several hours, the computer has little else to do in the middle of the night. It might slow down your SETI@home program (<https://setiathome.berkeley.edu>) so it may take a little longer to contact Starfleet, but a full backup is the easiest to restore.

An *incremental backup* records only data that has changed since the last backup. To restore an incremental backup, you first restore the most recent full backup, and then reapply the changes saved in the incremental backups that you made since then.

Making an incremental backup is faster than making a full backup but restoring the data is harder. Because they are faster, incremental backups are useful for really big databases where it would take too long to make a full backup.

For example, suppose your database stores data from your company's Internet of things (IoT) devices, including smart lamps, thermostats, noise monitors, security cameras, door locks, vending machines, the cafeteria's salt shakers, and the company mascot's litter box. (Believe it or not, there are IoT devices that do all of those things.) Suppose that you need the database to be running at full speed 20 hours a day on weekdays, but a full backup takes 12 hours. Then, on Saturday mornings, you might make a full backup, and on other days at midnight you would make an incremental backup to save the changes for the past 24 hours.

Now suppose the database crashes and burns on a Thursday. To restore the database, you would restore the previous Saturday's full backup, and then apply the incremental backups for Sunday, Monday, Tuesday, and Wednesday in order. That could take quite a while.

You can restore the database faster if you use *differential backups*. These are similar to incremental backups, but they store changes since the previous full backup rather than the previous backup of any kind. For example, you could create a full backup on Saturday as before, and then make a differential backup on each of the following days until the next Saturday. Now if the database goes down in flames on Thursday, you only need to restore the full backup and then Wednesday's differential backup. This takes fewer steps than restoring from incremental backups, but it takes more time to make differential backups because they hold more changes.

You can even use a hybrid strategy. For example, you could make a full backup on Saturday, incremental backups on Sunday, Monday, and Tuesday, a differential backup on Wednesday, and incremental backups on Thursday and Friday. This is much more complicated, so I don't think I would recommend it unless you have very special needs.

In fact, unless your database crashes frequently, it may be best to just perform fast incremental backups, and then suffer through the slow restore process on the rare occasions when things go awry. If your database does crash often, you may be better off spending your time trying to fix that issue instead of optimizing your restore speed.

Some databases allow you to perform backups while the database is in use. This is critical for databases that must be available most or all of the time. The backup will slow the database down so you still need to schedule backups for off-peak periods such as weekends or the middle of the night, but at least the database can keep running.

For example, my local grocery store's cash registers perform downloads, uploads, and backups in the middle of the night. If you stop in around midnight, the self-checkout machines usually run much slower than they do during the day. (I was there in the spirit of scientific inquiry, not because I was debugging software at midnight and needed a donut. Honest!)

One final note about backups. Backups are intended to protect you against unexpected damage to the database. That includes normal damage caused by local disasters such as power glitches, the CIH virus (see https://en.wikipedia.org/wiki/CIH_virus), spilled soda, and EBCAK (Error Between Chair and Keyboard) problems, but it also includes larger physical calamities such as fire, tornado, volcanic eruption, and invasion by space aliens. Your backups do you no good if they're stored next to the database's computer and you are hit by one of these. A full backup won't do you any good if the flash drive or DVD that holds it is sitting on top of the computer and a meteor reduces the whole thing to a pile of steel and plastic splinters.

To avoid this kind of problem, think about taking backups off-site. Of course, that creates a potential security issue if your data is sensitive (for example, credit card numbers, medical records, or salary information).

3-2-1 BACKUP!

You can use the 3-2-1 backup strategy for greater resilience in the face of cyberattacks, hurricanes, and good old human error. Here, you keep three copies of your data (two backups plus the copy that's in use) on two types of media (like a hard drive and a flash drive) with one copy off-site.

There are also other variations such as 3-2-2 where you keep both backup copies off-site. (The cloud is a reasonable choice for one of the off-site locations.)

MAKE A BACKUP PLAN

Suppose you have a really large database. A full backup takes around 10 hours, whereas an incremental backup takes about 1 hour per day of changes that you want to include in the backup. You are not allowed to make backups during the peak usage hours of 3:00 a.m. to 11:00 p.m. weekdays, and 6:00 a.m. to 8:00 p.m. on weekends.

Figure out what types of backups to perform on which days to make restoring the database as easy as possible.

1. Figure out when you have time for a full backup.
2. Each day after the full backup, make a differential backup if you can. If you don't have time for a differential backup, make an incremental backup.

How It Works

1. Figure out when you have time for a full backup.

The following table shows the number of off-peak hours you have available for performing backups during each night of the week.

NIGHT	OFF-PEAK START	OFF-PEAK END	OFF-PEAK HOURS
Monday	11:00 p.m.	3:00 a.m.	4
Tuesday	11:00 p.m.	3:00 a.m.	4
Wednesday	11:00 p.m.	3:00 a.m.	4
Thursday	11:00 p.m.	3:00 a.m.	4
Friday	11:00 p.m.	6:00 a.m.	7
Saturday	8:00 p.m.	6:00 a.m.	10
Sunday	8:00 p.m.	3:00 a.m.	7

The only time when you have enough off-peak hours to perform a full backup is Saturday night.

2. Each day after the full backup, make a differential backup if you can. If you don't have time for a differential backup, make an incremental backup.

On Sunday, Monday, Tuesday, and Wednesday nights, you have at least four free hours, so you can make a differential backup holding up to four days' worth of changes. If you need to restore one of these, you only need to apply the previous full backup and then one differential backup.

On Thursday and Friday nights, you don't have time to go all the way back to the previous full backup. You do have time, however, for an incremental backup from the previous night. If you need to restore on Friday morning, you'll have to restore Saturday's full backup, the Wednesday night differential backup, and Thursday's incremental backup.

The following table shows the complete backup schedule.

NIGHT	BACKUP TYPE
Saturday	Full
Sunday	Differential from Saturday
Monday	Differential from Saturday
Tuesday	Differential from Saturday
Wednesday	Differential from Saturday
Thursday	Incremental from Wednesday
Friday	Incremental from Thursday

Don't forget to store copies of the backups in a secure off-site location.

DATA WAREHOUSING

Many database applications have two components: an online part that is used in day-to-day business and an offline “data warehousing” part that is used to generate reports and perform more in-depth analysis of past data.

The rules for a data warehouse are different than those for an online database. Often, a data warehouse contains duplicated data, non-normalized tables, and special data structures that make building reports easier. Warehoused data is updated much less frequently than online data, and reporting flexibility is more important than speed.

For the purposes of this chapter, it's important that you be aware of your customers' data warehousing needs so that you can plan for appropriate database maintenance. In some cases, that may be as simple as passing a copy of the most recent full backup to a data analyst. In others, it may mean writing and periodically executing special data extraction routines.

For example, as part of nightly maintenance (backups, cleaning up tables, and what have you), you might need to pull sales data into a separate table or database for later analysis.

This book isn't about data warehousing so this chapter doesn't say any more about it. For a more complete overview, see https://en.wikipedia.org/wiki/Data_warehouse. For more

in-depth coverage, see a book about data warehousing. If you search online for **data warehouse books**, you'll find plenty of candidates. For example, you might take a look at *Building the Data Warehouse 4th Edition* by W. H. Inmon (Wiley, 2005) or *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling, 3rd Edition* by Ralph Kimball (Wiley, 2013).

REPAIRING THE DATABASE

Although databases provide many safeguards to protect your data, they sometimes still become corrupted. They are particularly prone to index corruption because it can take a while to update a database's index structures. If the computer crashes while the database is in the middle of updating its index trees, the trees may contain garbage, invalid keys, and pointers leading to nowhere (similar to the Gravina Island Bridge, https://en.wikipedia.org/wiki/Gravina_Island_Bridge).

When an index becomes corrupted, the results you get from queries may become unpredictable. You may get the wrong records, records in the wrong order, or no records at all. The program using the database may even crash.

To ensure that your database works properly, you should periodically run its repair tools. That should clean up damaged records and indexes.

COMPACTING THE DATABASE

When you delete a record, many databases don't actually release the space that the record occupied. Some databases may be able to undelete the record in case you decide you want it later, but most databases do this so they can reuse the space later for new records. If you add and then remove a lot of records, however, the database can become full of unused space.

The trees that databases typically use to store indexes are self-balancing. That ensures that they never grow too tall and skinny and that ensures searches are fast, but it also means that they contain extra unused space. They use that extra space to make adding new entries in the trees more efficient but, under some circumstances, the trees can contain *a lot* of unused space.

These days disk space is relatively cheap (as little as a few pennies per gigabyte), so the "wasted" space may not be much of an issue. Just pull the sock full of money out from under your bed and buy a bigger hard drive. Having some extra unused space in the database can even make adding and updating the database faster.

In some cases, however, parts of the database may become fragmented so the database may take longer to load records that are logically adjacent but that are scattered around the disk. In that case, you may get better performance if you compact and defragment the database. Look at your database product's instructions and notes to learn about good compaction strategies.

PERFORMANCE TUNING

Normally, you don't need to worry too much about how the database executes a query. In fact, if you start fiddling around with the way queries are executed, you take on all sorts of unnecessary

responsibility. It's kind of like being an air traffic controller; when everything works, no one notices that you're doing your job, but when something goes wrong everyone knows it was your fault.

Generally, you shouldn't try to tell the database how to do its job, but often you can help it help itself. Some databases use a statistical analysis of the values in an index to help decide how to perform queries using that index. If the distribution of the values changes, you can sometimes help the database realize that the situation has changed. For example, the Transact-SQL statement `UPDATE STATISTICS` makes a SQL Server database update its statistics for a table or view, possibly leading to better performance in complex queries.

Often, you can make queries more efficient by building good indexes. If you know that the users will be looking for records with specific values in a certain field, put an index on that field. For example, if you know that you will need to search your *Magic the Gathering* card database by power and toughness, then add those fields as indexes.

If you have a lot of experience with query optimization, you may even be able to give the database a hint about how it should perform a particular query. For example, you may know that a `GROUP BY` query will work best if the database uses a hashing algorithm. In Transact-SQL, you could use the `OPTION (HASH GROUP)` clause to give the database that hint (<https://learn.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql>). Only *serious* SQL nerds (with IQs exceeding their weights in pounds) should even consider this level of meddling. If you don't know what you're doing, you may hurt performance more than you help. (Even databases don't like backseat drivers.)

Some databases provide tools such as query analyzers or execution plan viewers so you can see exactly how the database will perform an operation. That not only lets you learn more about how queries are performed so you can aspire to write your own query hints, but it also lets you look for problems in your database design. For example, an execution plan may point out troublesome `WHERE` clauses that require executing a function many times, searches on fields that are not indexed, and nested loops that you might be able to remove by rewriting a query.

More expensive database products may also be able to perform other optimizations at a more physical level. For example, database replication allows several databases to contain the same information and remain synchronized. This can be useful if you perform more queries than updates. If one database is the master and the others are used as read-only copies, the copies can take some of the query burden from the main database.

Another advanced optimization technique is partitioning. A partitioned table stores different records in different locations, possibly on different hard disks or even different computers scattered around the cloud. If your typical queries normally divide the data along partition boundaries, the separate partitions can operate more or less independently. You may even be able to back up different partitions separately, improving performance.

In a variation on partitioning, you use multiple databases to handle different parts of the data. For example, you might have different databases to handle customers in different states or time zones. Because the databases operate independently, they are smaller and faster, and you can back them up separately. They may also help you satisfy data residency requirements, which were described way back in Chapter 1, "Database Design Goals." You can extract data into a data warehouse to perform queries that involve more than one database.

THE KEYS TO SUCCESS

Not all indexes are created equal. You need to tailor a table's indexes and keys to help the database perform the queries that you expect to actually perform.

Suppose you have a Customers table that contains the usual sorts of fields: CustomerId, FirstName, LastName, Street, City, State, and Zip. It also includes some demographic information such as BirthDate, AnnualIncome, Gender, FavoriteColor, and ShoeSize.

1. Decide which fields should be indexed to support normal database queries that must join Orders and OrderItems records to Customers records.
2. Decide which fields should be indexed to support typical customer queries where a customer wants information about an order.
3. Decide which fields should be indexed to support reporting queries such as “Find all orders greater than \$100 placed by female customers earning \$30,000–\$50,000 between ages 15 and 25.”

How It Works

1. Decide which fields should be indexed to support normal database queries that must join Orders and OrderItems records to Customers records.

In a typical database, the Customers table's CustomerId field links to the Orders table. The Orders table has an OrderId field that links to the OrderItems table. CustomerId is the Customers table's primary key and OrderId is the Orders table's primary key. Relational databases automatically index the primary key (I have yet to meet one that doesn't), so you don't need to add any additional indexes to support this typical joining operation.

2. Decide which fields should be indexed to support typical customer queries where a customer wants information about an order.

Depending on your business, customers *might* know (or be able to figure out) their customer IDs. For example, I can read the account number from my bank statements, utility bill, and telephone bill. But customers who walk into your store, phone you, or send you a flaming email typically don't know their customer IDs. You could force them to go dig through their garbage looking for an old statement while you joke with the other customer service representatives (“I can hear him digging through the trash compactor! <Snigger>”), but that's not very customer-friendly.

It would be better if you could look up customers given something they actually might know such as their name, address, or phone number. Name works very well (most customers over two years of age know their names), although the user interface must be prepared to handle duplicates. (So you can ask, “Are you the Zaphod Beeblebrox on Improbable Blvd or the one on Unlikely Terrace?”) You probably also need to handle ambiguous first names for cases where it's not clear which member of the household opened the account or when someone goes by a nickname. (“The name's George but everyone calls me Dubbya.”)

You may also want to consider spelling errors. (“Is that Melllvar with three L’s?”) If you have a very large customer base, you might want to look into soundex, metaphone, and other algorithms for handling names phonetically. (See https://en.wikipedia.org/wiki/Phonetic_algorithm.)

Even considering these issues, the combination of LastName/FirstName is an excellent choice for a secondary index.

Address and phone number also make good keys. Usually, they are slightly less natural for customers but sometimes they may have special meaning that makes them more useful. For example, the phone number is critical for telephone companies, so it might make sense to look up the records for the phone number that is giving the customer problems.

3. Decide which fields should be indexed to support reporting queries such as “Find all orders greater than \$100 placed by female customers earning \$30,000–\$50,000 between ages 15 and 25.”

To really understand how this query works in all of its gruesome detail, you would probably need to look at the database’s execution plan.

Does the database search for customers in the right age group, and then look through their orders to find those greater than \$100 ordered by women within the target income range? In that case, you might improve performance by indexing the BirthDate field.

Does the database select orders placed by women with the proper income, and then look through those to find the ones with the right total prices and birth date? In that case, you might improve performance by indexing the AnnualIncome field.

The best approach depends on exactly what your data looks like. Typically, I recommend that you not try to optimize this type of query until you have tried it out with some real data and you know there is a problem. After all, there’s a chance that the query will be fast enough without adding any extra indexes.

However, not knowing what’s going on rarely prevents me from having an opinion, so let me mention two points.

First, the Gender field would make a terrible index. It can only hold one of a few values so using that field to select women doesn’t really help the query narrow down its search much. The database will still need to wade through about half of its records to figure out which ones are interesting. Using BirthDate, AnnualIncome, or total purchase price would probably narrow the search much more effectively, so if the database is stupid enough to filter using Gender first, you should probably change the query somehow to coerce it into doing something more sensible.

Second, this is a query for a data warehouse, not for an online system. This is an offline query used to study data after the fact, so it probably doesn’t need to execute in real time. It could be that your boss’s, boss’s, boss said, “I bet if we changed these hideous plaid golf shorts to pink, we could sell more to teenage girls” and you’re stuck gathering data to justify

continues

(continued)

this brilliant insight. This query probably won't take all that long to execute even without extra indexes. Adding extra indexes to a table makes inserting, updating, and deleting records in that table slower, so it's better to just grit your teeth and take a few hours to run this sort of one-time query at midnight rather than slowing down typical database operations to satisfy this one query. (During the night, management will have an epiphany and decide to focus on building a play stove painted in camouflage colors that transforms into a robot to sell more play kitchens to boys anyway and this issue will be forgotten.)

SUMMARY

Designing and building a database is one thing, but keeping it running efficiently is another. Without proper maintenance, a database can become infected with bloated tables, inefficient indexes, and even corrupted data. This chapter explained that to get the most out of a database, you must do the following:

- Perform regular full and incremental backups.
- Extract data into a data warehouse to perform offline queries.
- Repair damaged indexes.
- Build indexes to support the queries that you will actually perform.
- Optionally compact tables and index structures to remove unused space.

Unfortunately, the exact details of performing these tasks are specific to different kinds of databases, so this chapter cannot provide all of the details. The following chapter describes another topic that is database-specific: security. Though the precise details for providing security depends on the type of database you are using, the following chapter describes some of the general issues that you should take into account to keep your data secure.

Before you move on to Chapter 29, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

EXERCISES

1. Suppose your database is big enough that it takes about 4 hours to perform a full backup and 2 hours to perform an incremental backup per day of changes that you want to include in the backup. Peak hours are 4:00 a.m. to 11:00 p.m. on weekdays and 6:00 a.m. to 9:00 p.m. on weekends. Design a backup schedule for the database that doesn't require any backup during peak hours.
2. Your business flourishes (a problem we all wish we had), so the database described in Exercise 1 grows and now takes 6 hours to perform a full backup and 3 hours per day of changes for an

incremental backup. A large part of your success comes from increased sales in new time zones, so your peak hours have grown to 3:00 a.m. to 12:00 midnight on weekdays and 5:00 a.m. to 10:00 p.m. on weekends. Design a new backup schedule that doesn't require any backup during peak hours.

3. How could you partition the database to make backups easier?
-

29

Database Security

Like database maintenance, database security is an important topic with details that vary from database to database. This chapter doesn't try to cover everything there is to know about database security. Instead, it explains some of the general concepts that you should understand.

In this chapter, you learn how to:

- Pick a reasonable level of security for the database.
- Choose good passwords.
- Give users necessary privileges.
- Promote a database's physical security.

THE RIGHT LEVEL OF SECURITY

Database security can range from nonexistent to tighter than Fort Knox. You can allow any user or application to connect to the database, or you can use encryption to prevent even the database itself from looking at data that it shouldn't see.

Though many people think more security is better, that's not always the case. Some databases can encrypt the data they contain, so it's very hard for bad guys to peek at your data. Unfortunately, it takes extra time to encrypt and decrypt data as you read and write it in the database, and that slows things down. For most applications, that level of security is overkill.

Although you may not need as much security as the White House, Bank of America, or the Tokyo Stock Exchange, it does make sense to take advantage of whatever security features your database does provide. The following sections describe some of the security features that you should look for in a database product.

Rather than getting the most powerful security system money can buy, you should consider the needs of your application and the security features that are available. Then you can decide how tightly to lock things down.

PASSWORDS

Passwords are the most obvious form of security in most applications. Different databases handle passwords differently and with different levels of safety. The following sections describe some of the password issues that you should consider when you build a database application.

Single-Password Databases

Different databases provide different kinds of password protection. At the weaker end of the spectrum, some databases provide only a single password for the entire database. A database may be protected by a password or not, but those are your only options.

The single password provides access to the entire database. That means a cyber villain who learns the password can get into the database and do pretty much anything. It also means that anyone who should use the database must share that password. One consequence of that is that you cannot easily tell which user makes which changes to the data.

In practice that often means the program that provides a user interface to the database knows the password, and then it may provide its own extra layer of password protection. For example, the application might store usernames and passwords (hopefully encrypted, not in their plain-text form) in a table. When the user runs the program, it uses its hard-coded password to open the database and verifies the user's name and password in the table. It then decides whether to allow the user in (and decides what privileges the user deserves) or whether it should display a nasty message, shut itself down, send threatening email to the user's boss, and so forth.

There are a couple of reasons why this is a weak approach. First, the program must contain the database password in some form, so it can open the database. Even if you encrypt the password within the code, a determined hacker will be able to get it back out. At worst, a tenacious bit-monkey could examine the program's memory while it was executing and figure out what password the database used.

A second reason why this approach can be risky is that it relies on the correctness of the user interface. Every nontrivial program contains bugs, so there's a chance that users will find some way to bypass the homemade security system and sneak in somewhere they shouldn't be.

Individual Passwords

More sophisticated databases give each user a separate password, and that has several advantages over a single password database.

If the database logs activity, you can tell who logged into the database and when. If there are problems, the log may help you narrow down who caused the problem. If the database logs every interaction with the database (or if your application does), you can tell exactly who messed up.

Another advantage to individual passwords is that the user interface program doesn't ever need to store a password. When the program starts, the user enters a username and password, and the program tries to use them to open the database. The database either opens or not and the program doesn't need to worry about why. Even a "seriously dope uberhacker with mad ninja skillz" can't dig a password out of the application if the password isn't there.

Because the database takes care of password validation, you can focus on what the program is supposed to help the users do instead of worrying about whether you made a mistake in the password validation code.

If your database allows individual user passwords, use them. They provide a lot of benefits with relatively little extra work on your part.

Operating System Passwords

Some databases don't manage passwords very well. They may use little or no encryption, may not enforce any password standards (allowing weak passwords such as "12345" and "password"), and may even write passwords into log files where a hacker can find them relatively easily.

If your database can integrate its own security with the security provided by the operating system, make it do so. In any case, take advantage of the operating system's security. Make sure users pick good operating system passwords and don't share them. Hackers won't get a chance to attack your database if they can't even log in to the operating system.

Good Passwords

Picking good passwords is something of an art. You need to pick something obscure enough that an evil hacker (or your prankster coworkers) can't guess but that's also easy enough for you to remember. It's easy to become overloaded when you're expected to remember the database password in addition to your computer username and password, bank PIN number, voicemail password, online banking password, PayPal password, eBay password, locker combination, anniversary, and children's names.

And, you don't want to use the same password for all of these because then if someone ever steals your gym membership password, they know *all* of your passwords.

Many companies have policies that require you to use certain characters in your password (must include letters, numbers, and a special character such as \$ or #, and you need to type every other character with your left hand and your eyes crossed). They also force you to change your password so often it's pretty much guaranteed that you'll forget it. (I've never quite understood that. Do they assume that a hacker will guess your password and then say, "Whew! That was hard. I think I'll wait a month or two before I take advantage of this password and trash the database?" Okay, I know they're really worried about someone just prowling through the database unnoticed and they want to change the password to shut them out as quickly as possible, but I'm not sure which is more common, an eavesdropper or someone who wreaks havoc as soon as they break in.)

So what do users do when faced with dozens of passwords that must pass complex checks? They write their passwords down where they are easy to find. They pick sequential passwords such as Secret1, Secret2, and so forth. They use names and dates that are easy to remember and guess. (Once as a security check I attacked our own password database to see how many passwords I could guess. By throwing names, dates, and common words at the database, I was able to guess more than half of the 300 or so passwords in just a few hours.)

It's much better to give the users a little extra training, so they can figure out how to pick a really good password and then not require changes so often. For example, a series of unrelated words is a

lot better than a single word but is usually just as memorable. The password `beeR&Pizza%suckS` is pretty easy to remember, tricky to guess, and what self-respecting hacker would ever want to type that? Replacing letters in the password with other symbols can further obscure the message. Replacing “z” with “2” and “e” with “3” turns this password into `b33R&Pi22a%suckS`. (Search online for “leet” to learn about a hacker wannabe language that uses this kind of substitution to make plain and simple text practically unintelligible. Or look at congressional legislation or a legal contract for some *serious* incomprehensibility.)

A technique that is particularly useful for touch typists is to shift your fingers before typing. For example, if you type “Potatoe” (with the “optional” extra “e”) with your fingers shifted one key to the right you get “[pysypr” on a standard qwerty keyboard. Combine a few of these tricks and you can build a fairly tough password that’s still reasonably easy to remember.

There are a few “don’ts” when it comes to making good passwords. Don’t use names, dates, places, ID numbers (such as Social Security numbers or driver’s licenses), or anything else that would be easy to guess or figure out by rummaging through your email or trash. In fact, don’t use words at all, unless you do something to obscure them such as replacing letters with other symbols or keyboard shifting. A few words together, even if they’re logically incompatible (such as “Politician” and “Honest” or “Inexpensive” and “Plumber”) are easy to guess. Remember that modern computers are *really fast*, so guessing a few million or even a few billion password combinations is child’s play.

PRIVILEGES

Most relational databases allow you to restrict each user’s access to specific tables, views, and even columns within a table. Typically, you would define groups such as Clerks or Managers, and then grant permission for users in those groups to view certain data. You may also be able to grant exceptions for individual users. (You can perform similar feats of cleverness yourself in your application even if you’re using a single password database, but it’s a lot more work.)

For example, suppose your medical database contains three levels of patient data. Data such as patient name, phone number, and appointment schedule would be available to almost everyone in the hospital. A summer intern could use that data to make and confirm appointments.

Medical history, appointment notes, prescriptions, and test results would be available only to medical staff such as nurses, physician assistants, doctors, and specialists like x-ray technicians.

Insurance and billing information would be available to the billing department and the person who takes your \$150 copay when you enter the lobby.

You could get into serious trouble if some of that data were to slip out to hackers or the wrong users.

If you use the database’s security features to prevent certain users from viewing sensitive data, you don’t need to worry about the wrong people seeing the wrong data. If Ann is an appointment specialist, she will be able to view Bob’s phone number so she can call him but the database won’t let her view Bob’s medical history.

Some databases also provide row-level security that allows you to restrict access to particular rows in a table. For example, suppose a table contains government documents that are labeled with one of the security levels Public, Secret, Top Secret, and Radioactive (you get thrown in jail if anyone finds

those). When a program queries this table, it compares the user's privileges with the records' security labels and returns only those that the user should be able to see.

Other databases provide access control at a less-refined level. They may let you restrict access to a table but not to particular columns or rows within a table. Fortunately, you can provide similar behavior by using views.

A view is the result of a query. It looks a lot like a table, but it may contain only some of the columns or records in one or more tables. If the database doesn't provide column-level security, you can deny access to the table, and then create different views for the different groups of users. For the Patients table, you would create separate views that include contact data, medical data, and billing data. Now you can grant access for the views to let users see the types of data they should be able to view.

The SQL `GRANT` and `REVOKE` statements let you give and withdraw privileges. It is generally safest to give users the fewest privileges possible to do their jobs. Then if the user interface contains a bug and tries to do something stupid, such as dropping a table or showing the user sensitive information, the database won't allow it.

Rather than remembering to remove every extraneous privilege from a new user, many database administrators revoke all privileges, and then explicitly grant those that are needed. That way the administrator cannot forget to remove some critical privilege like `DROP TABLE` or `LAUNCH NUCLEAR MISSILES`.

The following three MySQL scripts demonstrate user privileges. You can execute the first and third scripts in the MySQL Command Line Client. You need to start the Command Line Client in a special way (described shortly) to use the second script properly.

The following script prepares a test database for use:

```
CREATE DATABASE UserDb;
USE UserDb;

-- Create a table.
CREATE TABLE People (
  FirstName          VARCHAR(5)      NOT NULL,
  LastName           VARCHAR(40)     NOT NULL,
  Salary             DECIMAL(10,2)   NULL,
  PRIMARY KEY (LastName, FirstName)
);

-- Create a new user with an initial password.
-- Note that this password may appear in the logs.
CREATE USER Rod IDENTIFIED BY 'secret';

-- Revoke all privileges for the user.
REVOKE ALL PRIVILEGES, GRANT OPTION FROM Rod;

-- Grant privileges that the user really needs.
--GRANT INSERT ON UserDb.People TO Rod;
GRANT INSERT (FirstName, LastName, Salary) ON UserDb.People TO Rod;
GRANT SELECT (FirstName, LastName) ON UserDb.People TO Rod;
GRANT DELETE ON UserDb.People TO Rod;
```

This script creates the database UserDB and gives it a People table. It then creates a user named Rod, giving it the password “secret.” (Yes, that is a terrible password. Don’t do something like this in your database!)

Next, the script drops all privileges, including the GRANT privilege (which would allow users to grant privileges to themselves). It then grants privileges that allow the user to insert FirstName, LastName, and Salary values into the People table, select only the FirstName and LastName values, and delete records from the table.

Before you can execute the next script, you need to start the MySQL Command Line Client as the user Rod. To do that, start a command window (in Windows, open the Start menu, type **Command Prompt**, and press Enter). At the command prompt, change to the directory that contains the MySQL Command Line Client, `mysql.exe`. After you move to that directory, start the MySQL Command Line Client by executing this command:

```
mysql -u Rod -p
```

You might need to use `mysql --u Rod --p` on Linux systems.

Note that the username is case-sensitive, so type **Rod**, not *rod* or *ROD*. When prompted, enter the password **secret**. The Command Line Client should run in the operating system command window and you should see the `mysql` prompt.

Now, you can execute the following script to test the user’s privileges:

```
USE UserDB;

-- Make some records.
INSERT INTO People VALUES('Annie', 'Lennox', 50000);
INSERT INTO People VALUES('Where', 'Waldo', 60000);
INSERT INTO People VALUES('Frank', 'Stein', 70000);

-- Select the records.
-- This fails because we don't have SELECT privilege on the Salary column.
SELECT * FROM People ORDER BY FirstName, LastName;

-- Select the records.
-- This works because we have SELECT privilege on FirstName and LastName.
SELECT FirstName, LastName FROM People ORDER BY FirstName, LastName;

-- Create a new table.
-- This fails because we don't have CREATE TABLE privileges.
CREATE TABLE MorePeople (
    FirstName    VARCHAR(5)    NOT NULL,
    LastName     VARCHAR(40)   NOT NULL,
    PRIMARY KEY (LastName, FirstName)
);

-- Delete the records.
DELETE FROM People;
```

This script sets UserDB as the default database and inserts some records into the People table. This works because the user Rod has privileges to insert FirstName, LastName, and Salary values into this table.

Next, the script tries to select all the fields in this table. That operation fails because Rod doesn't have the privilege to select the Salary field. Even if the user-interface application managing this database contains a bug and tries to select salary data, the database won't let the user see the Salary field.

The script then tries to select the FirstName and LastName values from the People table. That works because Rod does have privileges to select those fields.

Next, the script tries to create a table and fails because Rod doesn't have that privilege.

Finally, the script deletes all the records from the table. That works because Rod has that privilege.

After you test the user's privileges, you can close the Command Line Client by entering the command **exit**. You can then close the operating system window by typing **exit** again.

Back in the original MySQL Command Line Client that created the database and the user, you can execute the third script to clean up:

```
DROP USER Rod;

DROP DATABASE UserDb;
```

The technique of removing all privileges and then granting only those that are absolutely necessary is very useful for preventing mistakes. In fact, many database administrators deny even the administrator accounts all of the dangerous privileges that they don't need on a daily basis (such as `DROP TABLE` and `CREATE USER`). The account still has the `GRANT` privilege so it can grant itself more power if necessary, but that takes an extra step so it's harder to accidentally make dangerous mistakes such as dropping critical tables.

A similar technique is for administrators to log in as a less powerful "mortal" user normally and only log into an administrator account when they really need to do something special and potentially dangerous.

A PRIVILEGED FEW

Consider the database design shown in Figure 29.1.

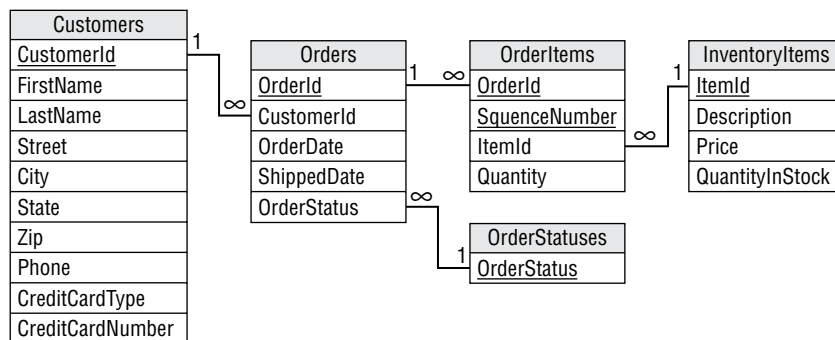


FIGURE 29.1

continues

(continued)

Many users need relatively few privileges to do their jobs. Write a SQL script that gives a shipping clerk enough privileges to fulfill orders in the database shown in Figure 29.1 by following these steps:

1. Make a permission table showing the CRUD (Create, Read, Update, and Delete) privileges that the user needs for the tables and their fields.
2. Deny all privileges.
3. Grant `SELECT` privileges for the fields in the Customers, Orders, OrderItems, and InventoryItems tables that the clerk needs to properly address a shipment.
4. Grant `UPDATE` privileges for the fields in the Customers, Orders, OrderItems, and InventoryItems tables that the clerk needs to properly record a shipment.

How It Works

1. Make a table showing the CRUD (Create, Read, Update, and Delete) privileges that the user needs for the tables and their fields.
2. Deny all privileges.

The following MySQL code creates a ShippingClerk user and revokes all privileges. (Again this is a terrible password. Don't use it.)

```
CREATE USER ShippingClerk IDENTIFIED BY 'secret';

-- Revoke all privileges for the user.
REVOKE ALL PRIVILEGES, GRANT OPTION FROM ShippingClerk;
```

3. Grant `SELECT` privileges for the fields in the Customers, Orders, OrderItems, and InventoryItems tables that the clerk needs to properly address a shipment.

To prepare and ship orders, the user must see all fields in the Orders, OrderItems, and InventoryItems tables. The clerk must also see the name and address information in the Customers table. The following MySQL statements grant privileges to select those fields:

```
GRANT SELECT ON ShippingDb.Orders TO ShippingClerk;
GRANT SELECT ON ShippingDb.OrderItems TO ShippingClerk;
GRANT SELECT ON ShippingDb.InventoryItems TO ShippingClerk;
GRANT SELECT (CustomerId, FirstName, LastName, Street, City, State,
             Zip, Phone) ON ShippingDb.Customers TO ShippingClerk;
```

Notice that the last statement grants privileges to select specific fields and doesn't let the clerk view the customer table's other fields such as CreditCardNumber.

4. Grant `UPDATE` privileges for the fields in the Customers, Orders, OrderItems, and InventoryItems tables that the clerk needs to properly record a shipment.

When shipping an order, the clerk must update the InventoryItems table's QuantityInStock field. The clerk must also update the Orders table's OrderStatus and ShippedDate fields. The following statements grant the necessary privileges:

```
GRANT UPDATE (QuantityInStock) ON ShippingDb.InventoryItems
  TO ShippingClerk;
GRANT UPDATE (OrderStatus, ShippedDate) ON ShippingDb.Orders
  TO ShippingClerk;
```

INITIAL CONFIGURATION AND PRIVILEGES

Databases (and many other software tools) often come preconfigured to make it easy for you to get started. Find out how the database is initially configured and modify the default settings to make the database more secure.

For example, databases often come with an administrator account that has a default username and password. It is amazing how many people build a database and don't bother changing those default settings. Anyone who knows the defaults cannot only open your database but can do so with administrator privileges, so they can do anything they want to your data. Hackers are very aware of these default accounts, and not surprisingly, trying to open those accounts is often the first attack a hacker tries.

TOO MUCH SECURITY

Ironically, one of the most common security problems I've seen in large applications is caused by too much security. The user interface application tries to restrict users, so they cannot do things they're not supposed to do accidentally or otherwise. When it's done properly, that type of checking is quite important, but if the system is too restrictive and too hard to change, the users will find ways to circumvent your security.

For example, suppose an application manages telephone accounts. Customer representatives can disconnect a customer for outstanding bills, answer customer questions, and reconnect service when payments are received. They can also reconnect service if the customer comes up with a really good sob story. ("My doggy Mr. Tiddles ate the bill. I sent it in anyway half chewed up, but the Post Office returned it for insufficient postage. It would have been a day late, but I was on a cruise and the ship crossed the International Date Line. I can pay now but it would be in a third-party check written in Florins from a bank in a country that no longer exists, etc.") At this point, the representative hands the customer to a shift supervisor who reconnects services for 15 days in self-defense just to shut the customer up.

Unfortunately, a lot of customers have sob stories that are more believable than this one (it's hard to imagine one less believable), so the shift supervisors waste a lot of time approving service reconnections. To save time, the supervisor writes their user ID and password in huge letters on the whiteboard at the front of the room so every representative can approve reconnections without interrupting the supervisor's online shopping.

Your expression of amusement should change to one of horror when you learn that this is actually a true story. (Not the one about Mr. Tiddles, the one about the password on the whiteboard.) I once visited a telephone billing center where anyone could log on as a supervisor at any time to approve special actions without wasting the supervisor's time.

At this point, a reasonable security feature, making supervisors approve special reconnections, has completely backfired. Not only can anyone approve special reconnections, but they could log on as a supervisor and perform all sorts of other unauthorized actions without leaving any trace of who actually did them. Fortunately, physical security at that center was tight, so a hacker couldn't just look in a window and grab the supervisor's password.

The moral is, restrict access to various database features appropriately, but make it easy for the customers to change the settings. If the supervisors could have changed the program to allow representatives to approve special reconnections, this would never have been a problem.

PHYSICAL SECURITY

Many system administrators spend a great deal of effort on software and network security and do nothing about physical security. It makes sense to focus on network attacks because an open Internet connection makes you potentially vulnerable to millions of would-be hackers and cybersnoops from around the globe.

However, focusing exclusively on software security and ignoring physical security is like installing a cat door on Fort Knox. While an unsecured Internet connection does expose you to a huge number of potential hackers, you shouldn't completely ignore local villains.

Though most employees are honest and hardworking, there have been several spectacular cases where employees have stolen data. There have also been many cases where employees and contractors have lost data through carelessness.

In one case, a former Boeing employee was accused of stealing 320,000 files using a flash drive. Boeing estimated that the files could cause \$5 billion to \$15 billion in damages if they fell into the wrong hands. (See <https://networkworld.com/article/2292651/ex-boeing-worker-accused-of-stealing-documents.html>.)

I generally prefer to assume that people are basically honest, but that doesn't mean you should make it easier for them to make bad decisions and silly mistakes.

For about \$30, you can put a 1 TB flash drive in your wallet or a 256 GB micro SD card in your phone. For a bit more, you can fit a few terabytes in your phone.

I'm not suggesting that you frisk employees before they leave for home, but if your database contains financial data, credit card numbers, and other proprietary secrets (such as numerical algorithms for picking lottery numbers), you should at least provide some supervision to discourage employees from walking out with the database.

Many powerful computers are also relatively small so, in some cases, it may be possible for someone to simply pick up your server and walk away with it. If the computer is too large to carry away, a few minutes with a screwdriver will allow just about anyone to remove the hard drive. Keeping your

database server in a locked office that's accessible by an internal network provides some extra security.

Even if you lock the network down so cyber villains can't find a seam to open, you should also consider outgoing connections. An employee can probably email data outside of your system or surf to a website that allows file uploading.

Laptop security is a particularly tricky issue lately. Laptops are designed for portability. If you didn't need that portability, you would probably buy a less expensive desktop computer, so you must assume the laptop will go off-site. Laptop theft is a huge and growing problem, so you should assume that any data you have on your laptop may be stolen. If you absolutely must store sensitive data on a laptop, encrypt it. Don't assume the laptop's operating system security will stop a thief from reading the hard disk. The web contains lots of sites with advice for preventing laptop theft, so look around and adopt whatever measures you can.

I once worked at a company that didn't allow cameras or cell phones with cameras because they were afraid someone might steal their corporate secrets (not that we had any worth stealing). However, they didn't prohibit flash drives, USB drives, laptops, MP3 players (which have drives that can hold computer files—see https://en.wikipedia.org/wiki/Pod_slurping), outgoing email, or web surfing to sites where you could upload files. They had plugged one possible channel for misdeeds but had left many others open. (My theory is that management was a bit behind the times and wasn't familiar enough with the other methods to realize that they were a potential problem.)

This all begs the question of whether the company has any data worth stealing. In the time I worked there, I saw lots of company confidential material but nothing that had any real financial or strategic value. Before you start installing security cameras and metal detectors, you should ask yourself how likely it is that someone would want to steal your data and how expensive the loss would be. Then you can take appropriate measures to reduce the likelihood of loss.

Though ignoring physical security is a mistake, obsessing over it can make you paranoid. Not everyone is an undercover agent for your competition or looking to sell credit card numbers to drug dealers. Take reasonable measures but try not to go unnecessarily overboard.

SUMMARY

Database security doesn't happen all by itself. Many databases provide sophisticated security features, but it's up to you to take advantage of them. This chapter described some of the issues you should consider to protect your data against accidental and malicious damage. It explained how to:

- Decide on a reasonable level of security for the database.
- Restrict privileges so users cannot harm the database accidentally or intentionally.
- Protect the database physically.

The chapters in this book explain how to determine customers' data needs. They explain how to build data models to study those needs, how to use the models to form a design, and how to refine the design to make it more efficient. Finally, the chapters explain how to implement the database, and how to examine the database's maintenance and security needs.

Having studied these chapters, you are ready to design and build effective databases, but there's a lot more to learn. Although I've tried to cover the most important topics of database design in some depth, database design is a huge topic, and you can always learn more. You may want to increase your knowledge by surfing the web or reading other database books that focus on different aspects of database design and development. In particular, you may want to seek out books that deal with specific issues for whichever database product you are using. You may also want to sign up for free trials of a few database products. The ever-growing cloud has made it extremely easy to try different products to see what they have to offer.

Before you leave these pages, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

EXERCISES

For these exercises, consider the database design shown in Figure 29.1.

In the "A Privileged Few" activity earlier in this chapter, you determined the privileges needed by a shipping clerk for this database. These exercises consider other roles that users will play.

1. Build a permission table showing the privileges needed by an order entry clerk to create new orders. Write SQL statements to create an order entry clerk with the appropriate privileges.

2. Build a permission table showing the privileges needed by a customer service clerk who answers questions about users' accounts and orders. This clerk should be able to modify customer and order data, and should be able to cancel an order that has not yet shipped.

3. Build a permission table showing the privileges needed by an inventory manager who orders new inventory as needed. This person also changes an order's status from Back Ordered to Ordered when there is enough inventory to fulfill the order. The inventory manager also keeps the InventoryTable up-to-date and may need to add, remove, or modify records.

A

Exercise Solutions

CHAPTER 1: DATABASE DESIGN GOALS

Exercise 1 Solution

The following list summarizes how the book provides (or doesn't) database goals:

- **CRUD**—This book doesn't let you easily *Create* information. You could write in new information, but there isn't much room for that and that's not really its purpose. The book lets you *Read* information, although it's hard for you to find a particular piece of information unless it is listed in the table of contents or the index. You can *Update* information by crossing out the old information and entering the new. You can also highlight key ideas by underlining, by using a highlighter, and by putting bookmarks on key pages. Finally, you can *Delete* data by crossing it out.
- **Retrieval**—The book's mission in life is to let you retrieve its data, although it can be hard to find specific pieces of information unless you have bookmarked them or they are in the table of contents or the index.
- **Consistency**—I've tried hard to make the book's information consistent. If you start making changes, however, it will be extremely difficult to ensure that you make related changes to other parts of the book.
- **Validity**—The book provides no data validation. If you write in new information, the book cannot validate your data. (If you write, "Normalization rocks!" the book cannot verify that it indeed rocks.)
- **Easy Error Correction**—Correcting one error is easy; simply cross out the incorrect data and write in the new data. Correcting systematic errors (for example, if I've methodically misspelled "the" as "thue" and the editors didn't catch it) would be difficult and time-consuming.

- **Speed**—The book’s structure will hopefully help you learn database design relatively efficiently, but a lot relies on your reading ability.
- **Atomic Transactions**—The book doesn’t really support transactions of any kind, much less atomic ones.
- **ACID**—Because it doesn’t support transactions, the book doesn’t provide ACID.
- **Persistence and Backups**—The book’s information is nonvolatile so you won’t lose it if the book “crashes.” If you lose the book or your dog eats it, then you can buy another copy but you’ll lose any updates that you have written into it. You can buy a second copy and back up your notes into that one, but the chances of a tornado destroying your book are low and the consequences aren’t all that earth-shattering, so I’m guessing you’ll just take your chances.
- **Low Cost and Extensibility**—Let’s face it, books are pretty expensive these days, although not as expensive as even a cheap computer. You can easily buy more copies of the book, but that isn’t really extending the amount of data. The closest thing you’ll get to extensibility is buying a different database-related book or perhaps buying a notebook to improve your note-taking.
- **Ease of Use**—This book is fairly easy to use. You’ve probably been using books for years and are familiar with the basic user interface.
- **Portability**—It’s a fairly large book, but you can reasonably carry it around. You can’t read it remotely the way you can a computerized database, but you can carry it on a bus.
- **Security**—The book isn’t password protected, but it doesn’t contain any top-secret material, so if it is lost or stolen you probably won’t be as upset by the loss of its data as by the loss of the concert tickets that you were using as a bookmark. It’ll also cost you a bit to buy a new copy if you can’t borrow someone else’s.
- **Sharing**—After you lose your copy, you could read over the shoulder of a friend (although then you need to read at their pace) or you could borrow someone else’s book. Sharing isn’t as easy as it would be for a computerized database, however, so you might just want to splurge and get a new copy.
- **Ability to Perform Complex Calculations**—Sorry, that’s only available in the more expensive artificially intelligent edition.

Overall, the book is a reasonably efficient read-only database with limited search and correction capabilities. As long as you don’t need to make too many corrections, it’s a pretty useful tool. The fact that instructional books have been around for a long time should indicate that they work pretty well.

Exercise 2 Solution

This book provides a table of contents to help you find information about general topics and an index to help you find more specific information if you know the name of the concept that you want to study. Note that the page numbers are critical for both kinds of lookup.

Features that help you find information in less obvious ways include the introductory chapter that describes each chapter’s concepts in more detail than the table of contents does, and cross-references within the text.

Exercise 3 Solution

CRUD stands for the four fundamental database operations: Create (add new data), Read (retrieve data), Update (modify data), and Delete (remove data from the database).

Exercise 4 Solution

A chalkboard provides:

- **Create**—Use chalk to write on the board.
- **Read**—Look at the board.
- **Update**—Simply erase old data and write new data.
- **Delete**—Just erase the old data.

A chalkboard has the following advantages over a book:

- **CRUD**—It’s easier to create, read, update, and delete data.
- **Retrieval**—Although a chalkboard doesn’t provide an index, it usually contains much less data than a book so it’s easier to find what you need.
- **Consistency**—Keeping the data consistent isn’t trivial but again, because there’s less data than in a book, you can find and correct any occurrences of a problem more easily.
- **Easy Error Correction**—Correcting one error is trivial; just erase it and write in the new data. Correcting systematic errors is harder, but a chalkboard contains a lot less data than a book, so fixing all of the mistakes is easier.
- **Backups**—You can easily back up a chalkboard by taking a picture of it with your cell phone. (This is actually more important than you might think in a research environment where chalkboard discussions can contain crucial data.)
- **Ease of Use**—A chalkboard is even easier to use than a book. Toddlers who can’t read can still scribble on a chalkboard.
- **Security**—It’s relatively hard to steal a chalkboard nailed to a wall, although a ne’er-do-well could take a cell-phone picture of the board and steal the data. (In some businesses and schools, cleaning staff are forbidden to erase chalkboards without approval to prevent accidental data loss.)
- **Sharing**—Usually everyone in the room can see what’s on a chalkboard at the same time. This is one of the main purposes of chalkboards.

A book has the following advantages over a chalkboard:

- **Persistence**—A chalkboard is less persistent. For example, someone brushing against the chalkboard may erase data. (I once had a professor who did that regularly and always ended the lecture with a stomach covered in chalk.)
- **Low Cost and Extensibility**—Typically, books are cheaper than chalkboards, at least large chalkboards.
- **Portability**—Books typically aren’t nailed to a wall (although the books in the Hogwarts library’s restricted section are chained to the shelves).

The following database properties are roughly equivalent for books and chalkboards:

- **Validity**—Neither provides features for validating new or modified data against other data in the database.
- **Speed**—Both are limited by your reading (and writing) speed.
- **Atomic Transactions**—Neither provides transactions.
- **ACID**—Neither provides transactions, so neither provides ACID.
- **Ability to Perform Complex Calculations**—Neither can do this (unless you have some sort of fancy interactive computerized book or chalkboard).

In the final analysis, books contain a lot of information and are intended for use by one person at a time, whereas chalkboards hold less information and are tools for group interaction. Which you should use depends on which of these features you need.

Exercise 5 Solution

A recipe card file has the following advantages over a book:

- **CRUD**—It's easier to create, read, update, and delete data in a recipe file. Updating and deleting data is also more aesthetically pleasing. In a book, these changes require you to cross out old data and optionally write in new data in a place where it probably won't fit too well. In a recipe file, you can replace the card containing the old data with a completely new card.
- **Consistency**—Recipes tend to be self-contained, so this usually isn't an issue.
- **Easy Error Correction**—Correcting one error in the recipe file is trivial; just replace the card that holds the error with one that's correct. Correcting systematic errors is harder but less likely to be a problem. (What are the odds that you'll mistakenly confuse metric and English units and mix up liters and tablespoons? But you can go to [https://gizmodo.com/five-massive-screw-ups-that-wouldn't-have-happened-if-we-1828746184](https://gizmodo.com/five-massive-screw-ups-that-wouldn-t-have-happened-if-we-1828746184) to read about five disasters that were caused by that kind of mix-up.)
- **Backups**—You could back up a recipe file fairly easily. In particular, it would be easy to make copies of any new or modified cards. I don't know if anyone (except perhaps Gordon Ramsay or Martha Stewart) does this.
- **Low Cost and Extensibility**—It's extremely cheap and easy to add a new card to a recipe file.
- **Security**—You could lose a recipe file, but it will probably stay in your kitchen most of the time, so losing it is unlikely. Someone could break into your house and steal your recipes, but you'd probably give copies to anyone who asked (except for your top-secret death-by-chocolate brownie recipe).

A book has the following advantages over a recipe file:

- **Retrieval**—A recipe file's cards are sorted, essentially giving it an index, but a book also provides a table of contents. With this kind of recipe file, it would be hard to simultaneously sort cards alphabetically and group them by type such as entrée, dessert, aperitif, or midnight snack.

- **Persistence**—The structure of a recipe file is slightly less persistent than that of a book. If you drop your card file down the stairs, the cards will be all mixed up (although that may be a useful way to pick a random recipe if you can't decide what to have for dinner).

The following database properties are roughly equivalent for books and recipe files:

- **Validity**—Neither provides features for validating new or modified data against other data in the database.
- **Speed**—Both are limited by your reading and writing speeds.
- **Atomic Transactions**—Neither provides transactions.
- **ACID**—Neither provides transactions, so neither provides ACID.
- **Ease of Use**—Many people are less experienced with using a recipe file than a book, but both are fairly simple. (Following the recipes will probably be harder than using the file, at least if you cook anything interesting.)
- **Portability**—Both books and recipe files are portable, although you may never need your recipes to leave the kitchen.
- **Sharing**—Neither is easy to share.
- **Ability to Perform Complex Calculations**—Neither can do this. (Some computerized recipe books can adjust measurements for a different number of servings, but index cards cannot.)

Instructional books usually contain tutorial information, and you are expected to read them in big chunks. A recipe file is intended for quick reference and you generally use specific recipes rather than reading many in one sitting. The recipe card file is more like a dictionary and has many of the same features.

Exercise 6 Solution

ACID stands for Atomicity, Consistency, Isolation, and Durability.

- *Atomicity* means transactions are atomic. The operations in a transaction either all happen or none of them happen.
- *Consistency* means the transaction ensures that the database is in a consistent state before and after the transaction.
- *Isolation* means the transaction isolates the details of the transaction from everyone except the person making the transaction.
- *Durability* means that once a transaction is committed, it will not disappear later.

Exercise 7 Solution

BASE stands for the distributed database features of Basically Available, Soft state, and Eventually consistent.

- *Basically Available* means that the data is available. These databases guarantee that any query will return some sort of result, even if it's a failure.
- *Soft state* means that the state of the data may change over time, so the database does not guarantee immediate consistency.

- *Eventually consistent* means these databases do eventually become consistent, just not necessarily before the next time that you read the data.

Exercise 8 Solution

The CAP theorem says that a distributed database can guarantee only two out of three of the following properties:

- **Consistency**—Every read receives the most recently written data (or an error).
- **Availability**—Every read receives a non-error response, if you don't guarantee that it receives the most recently written data.
- **Partition tolerance**—The data store continues to work even if messages are dropped or delayed by the network between the store's partitions.

This need not be a “pick two of three” situation if the database is not partitioned. In that case, you can have both consistency and availability, and partition tolerance is not an issue because there is only one partition.

Exercise 9 Solution

If transaction 1 occurs first, then Alice tries to transfer \$150 to Bob and her balance drops below \$0, which is prohibited.

If transaction 2 occurs first, then Bob tries to transfer \$150 to Cindy and his balance drops below \$0, which is prohibited.

So, transaction 3 must happen first: Cindy transfers \$25 to Alice and \$50 to Bob. Afterward Alice has \$125, Bob has \$150, and Cindy has \$25.

At this point, Alice and Bob have enough money to perform either transaction 1 or transaction 2.

If transaction 1 comes second, then Alice, Bob, and Cindy have \$0, \$275, and \$25, respectively. (If he can, Bob should walk away at this point and quit while he's ahead.) Transaction 2 follows and the three end up with \$0, \$125, and \$175, respectively.

If transaction 2 comes second, then Alice, Bob, and Cindy have \$125, \$0, and \$175, respectively. Transaction 1 follows and the three end up with \$0, \$125, and \$175, respectively.

Therefore, the allowed transaction orders are 3 – 1 – 2 and 3 – 2 – 1. Note that the final balances are the same in either case.

Exercise 10 Solution

If the data is centralized, then it does not remain on your local computer. In particular, if your laptop is lost or stolen, you don't need to worry about your customers' credit card information because it is not on your laptop.

Be sure to use good security on the database so cyber-criminals can't break into it remotely. Also don't use the application on an unsecured network (such as in a coffee shop or shopping mall) where someone can electronically eavesdrop on you.

CHAPTER 2: RELATIONAL OVERVIEW

Exercise 1 Solution

This constraint means that all salespeople must have a salary or work on commission but they cannot have both a salary and receive commissions.

Exercise 2 Solution

In Figure A.1, lines connect the corresponding database terms.

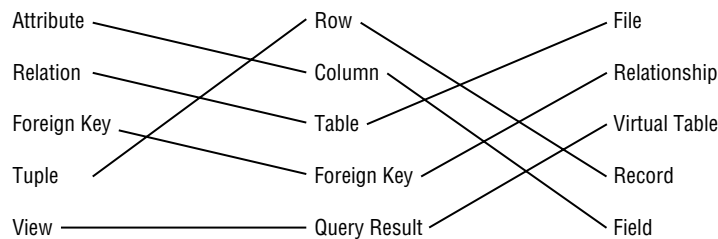


FIGURE A.1

Exercise 3 Solution

State/Abbr/Title is a superkey because no two rows in the table can have exactly the same values in those columns.

Exercise 4 Solution

Engraver/Year/Got is not a superkey because the table could hold two rows with the same values for those columns.

Exercise 5 Solution

The candidate keys are State, Abbrev, and Title. Each of these by itself guarantees uniqueness so it is a superkey. Each contains only one column so it is a minimal superkey, and therefore a candidate key.

All of the other fields contain duplicates and any combination that doesn't have duplicates in the data shown (such as Engraver/Year) is just a coincidence (someone could engrave two coins in the same year). That means any superkey must include at least one of State, Abbrev, or Title to guarantee uniqueness, so there can be no other candidate keys.

Exercise 6 Solution

The domains for the columns are:

- **State**—The names of the 50 U.S. states
- **Abbrev**—The abbreviations of the 50 U.S. states

- **Title**—Any text string that might describe a coin
- **Engraver**—People’s names
- **Year**—A four-digit year—more precisely, 1999 through 2008
- **Got**—“Yes” or “No”

Exercise 7 Solution

Room/FirstName/LastName and FirstName/LastName/Phone/CellPhone are the possible candidate keys.

CellPhone can uniquely identify a row if it is not null. If CellPhone is null, then we know that Phone is not null because all students must have either a room phone or a cell phone. But roommates share the same Phone value, so we need FirstName and LastName to decide which is which. (Basically Phone/CellPhone gets you to the Room.)

Exercise 8 Solution

In this case, FirstName/LastName is not enough to distinguish between roommates. If their room has a phone, they might not have cell phones so there’s no way to tell them apart in this table. In this case, the table has no candidate keys. That might be a good reason to add a unique column such as StudentId. (Or if the administration assigns rooms, just don’t put two John Smiths in the same room. You don’t have to tell them it’s because of your poorly designed database!)

Exercise 9 Solution

The room numbers are even so you could use `Room Is Even`. (Don’t worry about the syntax for checking that a value is even.) You could also use some simple range checks, such as `(Room >= 100) AND (Room < 300)`, depending on what room numbers are actually allowed.

You might also notice that every Phone value has the same area code and exchange 202-237, so you could check for that.

Exercise 10 Solution

Every student must have a Phone or CellPhone value, so you could check that `(Phone <> null) OR (CellPhone <> null)`.

CHAPTER 3: NOSQL OVERVIEW

Exercise 1 Solution

This data forms a tree, so you could store it in a graph database. If there is some inbreeding, then the data forms a network rather than a tree, but it will still fit easily into a graph database. (The fact that the data represents *relationships* among dogs is also a hint that you might want to use a graph database.)

A graph database would let you perform relationship-oriented queries such as finding all the ancestors on the source dog's father's side or determining the number of generations between this dog and TV celebrity Lassie. Most graph databases also allow you to query on node properties in case you want to find dogs with certain characteristics such as show winners and flyball champions.

If there is no inbreeding, then you could save the tree in an XML or JSON file, although that would reduce the kinds of queries you could perform.

You could also store the data in a relational database. That would let you search for dog characteristics but would make it harder to study the relationships in the tree or network.

Overall, the graph database will probably give you the largest assortment of query capabilities with the least effort.

Exercise 2 Solution

This may seem like a more complicated database than the one in Exercise 1, but it's still a tree with two main branches connected to the source dog: one leading to descendants and one leading to ancestors. (If there is inbreeding, then it's a network as before.)

For the same reasons described in the solution to Exercise 1, a graph database will likely give you the best query capabilities with the least work.

Exercise 3 Solution

Application settings are easy to store in almost any database.

If you use a flat file, then you need to write code to save and retrieve values.

If you use an XML or JSON file, then you may be able to use built-in programming language tools or libraries to save and retrieve values more easily. You could save the XML or JSON data in a document database.

A key-value database would allow applications to easily load and update settings as needed, but it might be overkill. If the settings are different for each user, then saving them all in a shared database will increase the total overhead somewhat.

Similarly, you could store settings in a relational database and, similarly, that might be overkill.

Any of the solution that stores settings in a centralized location (such as a document database, key-value database, or relational database) would allow administrators to fix settings if a user makes a window zero pixels wide or drags a window completely off the screen. Those approaches also mean that if you logged into the application from any computer, you found your personal settings ready and waiting for you.

Despite those advantages, I prefer the simplicity of storing settings in simple XML or JSON files stored in the program's executable directory unless a user typically logs in from several different computers.

Exercise 4 Solution

This sounds like a very simple database whose major requirement is graphing, so a spreadsheet can probably handle this. That would limit the application to tasks that spreadsheets can perform,

however, and if the users later decide that they want to store more complex data and perform sophisticated queries on it, you might wish you'd chosen a relational database.

Exercise 5 Solution

A spreadsheet can also handle this requirement, but there's the same risk that the users will later decide they need more features than a spreadsheet can easily handle.

Exercise 6 Solution

A spreadsheet will *still* work, with the same caveat. At this point, however, I would notice that the users are starting to add more and more features. I would want to explore the requirements more fully and make sure this is *really* their final request before committing to a spreadsheet. It would be better to move to a more complicated database model now than to have to rebuild everything from scratch in six months (or just as likely, have users complain about how the spreadsheet doesn't do all of the things they didn't tell you it was supposed to do).

Exercise 7 Solution

This is a fairly simple tree so it will fit easily in a graph database. It's such a small tree (relatively speaking) that it seems unlikely that you'll need to perform complex ad hoc queries, so you could store it in an XML or a JSON file.

Exercise 8 Solution

This probably needs to be some sort of relational database. They are great at handling large amounts of interconnected data and performing complex ad hoc queries.

Which type of relational database you should pick (regular, object-oriented, or some other flavor) depends largely on your development philosophy and environment.

Exercise 9 Solution

As in Exercise 8, this problem cries out for some kind of relational database. To make the boss happy, you could use an object-oriented database. In several projects, I've used an object-relational mapping approach planted on top of a relational database and it has always worked quite well.

Exercise 10 Solution

If the recipe book will be fairly small, you could just put each recipe on a separate page in a Microsoft Word document and use Word's search capability to find recipe names, part of a meal, or an ingredient. (Fooled you, didn't I? That wasn't one of the main topics covered in this chapter! However, it would be a reasonable solution for such a simple application. Remember, the goal is to provide a useful solution with the minimum amount of work.)

Of the solutions that *are* described in this chapter, you could pick a relational database. It will provide better search capabilities than a simpler flat file, spreadsheet, XML file, or JSON file.

A truly object-oriented database would probably be serious overkill for this project. (I would only pick one of them if I wanted practice with a particular new tool—for example, one that I knew was going to be used on a future project.)

You could also store each recipe as a document in a document database. You could still query on the fields inside the recipes like you can with a relational database. The document database would allow different recipes to use different formats if necessary.

This is a pretty good example where a NoSQL database might provide some useful flexibility. It would be fairly easy to build this in a relational database, but a document database would let you easily use new formats later if you run across a strange recipe that didn't fit the usual pattern.

Exercise 11 Solution

This could require some serious sorting and searching, so a relational database is probably your best bet. (You would use a separate table or two to define power decks.) Which flavor you should pick (regular, object-oriented, etc.) depends largely on your development philosophy and environment.

Alternatively, you could store each card's data in a document inside a document database. You should probably be able to figure out which fields you will need in advance, however, so the flexibility of being able to give each document a different format seems less useful than it would be for the recipe database.

Exercise 12 Solution

This application would require some serious search capabilities, so you might think “relational database.” That would work, but the different media types have different characteristics, so you would need separate tables for each and that would make queries more complicated. This would still be possible, but it would be a fair amount of work.

Alternatively, you could use a column-oriented database. It would look like a single huge table, but the different media types would have different columns. For example, movies (such as *The Pelican Brief* and *Shrek*) would list Lithgow in the Actor column while books (such as *The Remarkable Farkle McBride* and *Marsupial Sue*) would list him in the Author column.

If you later decide to add new information to the database, such as `RottenTomatoesRating` for movies and `Awards` for all media, you could simply add those columns. In fact, you could even add new media such as audiobooks with little effort. You could add those features to a relational database, but it would be more work.

It might also be interesting to build a graph database to examine the relationships among different works. (For example, that would let you dominate the game “Six Degrees of Kevin Bacon.”) That would be a lot of work, however, and wouldn't be required for the original problem statement.

Exercise 13 Solution

This database will require some serious sorting, searching, and grouping, so a relational database may be in order. That would allow you to perform complex queries linking players and their teams.

Unfortunately different sports have different statistics, league structures, numbers of players, and other basic information, so it might be hard to build a single table to hold information for them all. Later, when you add dragon boating and quidditch, you may need to restructure the database.

As was the case with the media database in Exercise 12, a document-column-oriented database would allow you to perform queries while also giving you greater flexibility for later improvements.

Exercise 14 Solution

This data is so simple that it could conveniently be stored in just about any kind of database. If the application uses a database for some other purpose, you might consider adding this information to it because the database will be there anyway.

Otherwise, you should use the simplest solution that makes sense. A plain-old text file posted on a network-accessible server would work just fine. Alternatively, you could use a document database or key-value database. You could even squeeze the message of the day into a relational, column-oriented, or graph database, although that would not be a natural fit.

CHAPTER 4: UNDERSTANDING USER NEEDS

Exercise 1 Solution

In Figure A.2, lines connect the customer roles with their corresponding descriptions.

Customer Role	Description
Convert	Someone who won't be around for long. May be helpful or may not care all that much.
Customer Champion	Answers your questions about the project.
Customer Representative	Anyone who has an interest in the project.
Devil's Advocate	Makes things generally run smoothly. Not glamorous but very useful.
Executive Champion	Provides a reality check and prevents groupthink.
Generic Bad Guy	Ranges from annoying naysayer to malicious saboteur/super villain.
Short-Timer	A user who originally was against your project that you include in the development process to bring them onto your side.
Sidekick/Gopher	The highest ranking customer driving the project. Willing to fight super villains.
Stakeholder	Thoroughly understands the customers' needs. Has the authority to make decisions that stick.

FIGURE A.2

Exercise 2 Solution

A use case can cover any part of the customers' operation, including big or little pieces of the whole process. In fact, it's easier to test a big scenario if you break it into smaller pieces. The answer that doesn't describe a use case is:

- c. It should cover the customer's entire operation from start to finish.

Exercise 3 Solution

Brainstorming sessions should include everyone interested so the correct answer is:

- d. All of the above

(Although technically Customer Representatives and the Devil's Advocate are also Stakeholders, so "c. All interested Stakeholders" is also sort of correct. Let's not quibble.)

Exercise 4 Solution

The correct answer is:

- b. Ask the customer why they think that.

You never know if the customer knows more than they're admitting and if they might have very good reasons for suggesting that kind of database. Even if they're wrong, the reasons they give will tell you more about the situation and may lead to other important insights.

Answer "d. Study the problem to see if that kind of database makes sense," almost works because it's good to see if that kind of database makes sense, but it's also important to know why the customer thinks it does.

Exercise 5 Solution

Whenever you don't understand something about the customers' operation you should ask someone, so the correct answer is:

- a. Ask someone what that's all about.

The answer you get may be as arbitrary as "that's just the way Mark likes to do it," but in this fictitious scenario the customers use the first date stamp to record when the order was received and the second to indicate that the order entry operator looked at the back of the order to check for notes and comments.

If you didn't ask, you might have incorrectly placed two date fields in the Orders table. Once the process is online, however, you won't need the second date because there is no "other side" of the order to check. (Looking at the back of your computer monitor won't tell you much.) All of the notes and comments will be in a text box at the bottom of the online form.

Exercise 6 Solution

The following table summarizes the fields' data requirements:

FIELD	REQUIRED?	DOMAIN
Address 1	Yes	Valid street addresses or street names without numbers
Address 2	No	Apartment, suite, floor, etc.
Company	No	Valid company names (which could be practically anything)
Street Address	Yes	Valid street addresses or street names without numbers
Apt/Suite/Other	No	Apartment, suite, floor, etc.
City	Yes	Valid cities
State	Yes	Valid states
ZIP Code	No	Five-digit or ZIP+4 codes as in 12345 and 12345-6789

The required fields are marked on the form with asterisks.

The fields on this form have one complex interdependency: you must include either the city and state, or the ZIP Code. (If you include the ZIP Code, then the form looks up the city and state.) This isn't obvious from those fields because none of them is marked with an asterisk, so the form includes text to explain this.

The form could use a foreign key validation for the city, checking against a table listing every city in the country. It would be a huge table and would probably contain errors, so in many applications this might not be worth the effort. However, this application needs the city to look up the ZIP Code, so if the city isn't legal the lookup will fail. (In fact, that may be the best way to validate the data: see if you can look up the ZIP Code.)

The form could also verify that the ZIP Code is valid for the city, if the user enters both. Again, the whole point is to look up a ZIP Code, so it would be easy to check it against any value that the user entered.

Exercise 7 Solution

Backup policy is a data reliability issue more than a security issue, so the correct answer is:

- c. The frequency with which you need to perform backups.

However, the two issues are often closely related. For example, in many applications backups must be stored securely so that sensitive data doesn't fall into the wrong hands. Backups are also useful if a hacker gets into your system and trashes the database.

Exercise 8 Solution

The correct answer is:

- d. It depends (you need more information).

This is probably a Priority 1 or 2 feature, depending on how serious Frank is and how soon he wants to add this feature. This doesn't sound too complicated (it would probably just require a few new fields in an inventory table or a new plant lookup table), so I would say if Frank is serious then he should make this a Priority 1 feature and add it to the database design. I would also make this data not required in case Frank doesn't have time to enter all of this information right away for every kind of plant.

Exercise 9 Solution

MOSCOW stands for Must, Should, Could, Won't.

Exercise 10 Solution

The answer to this one depends on the operating system that you're using. I'm currently sitting at a computer running Windows 11, so here's how my use case might read:

- **Goals**—Authorized users should be able to log in and unauthorized users should not.
- **Summary**—The user tries to enter a username and password or PIN. If they are correct, the user is allowed access to the system.
- **Actors:**
 - The user—tries to log in.
 - The operating system—validates the username and password and grants or denies access.
- **Pre-Conditions**—No one is currently logged in to the system. (The case when someone else is already logged in is a different use case.)
- **Post-Conditions**—If the user enters a valid username and password or PIN, the system is logged in and displays the user's desktop. If the user enters an invalid username/password combination, the system remains logged out and the user cannot see the desktop or any of the data in the computer.
- **Normal Flow**—The tester should try all the possible combinations of blank, valid, and invalid usernames, passwords, and PINs and click OK. The following table lists the combinations and their desired results. The tester should fill in the blank column with "Pass" or "Fail" to indicate whether each test gave the desired result.

USERNAME	PASSWORD	DESIRED RESULT	PASS/FAIL
Blank username	Blank password	No access	
Blank username	Valid password	No access	
Blank username	Invalid password	No access	
Blank username	Valid PIN	No access	
Blank username	Invalid PIN	No access	
Valid username	Blank password	No access	
Valid username	Valid password	Access	
Valid username	Invalid password	No access	
Valid username	Valid PIN	Access	
Valid username	Invalid PIN	No access	
Valid username	Valid password for different account	No access	
Valid username	Valid PIN for different account	No access	
Invalid username	Blank password	No access	
Invalid username	Invalid password	No access	
Invalid username	Invalid PIN	No access	

- **Alternative Flow**—Instead of clicking OK, the user could click Cancel. The system should reset the screen, clearing the username and password text boxes.
- **Notes**—In all cases that do not give the user access, the system should deny access in exactly the same way so that the user cannot learn, for example, that they have guessed a valid username but an invalid password. That would give a ne'er-do-well a valid username to attack and that would be bad.

Note that this use case specifies the user's actions with enough detail that a relatively inexperienced user could follow it.

Exercise 11 Solution

When a heavy-hitter like a vice president attacks, you need to call in your Executive Champion. Ideally, they can point to your requirements document and show that you did, in fact, consider farbulistic granilation, and that everyone agreed the allowance was sufficient. If you didn't consider this issue, then you may need to put in some extra study to give your Executive Champion ammunition to fend off the attack.

If your Executive Champion doesn't have enough clout to fight off the Supervillain, then you could be in trouble.

One project I worked on really did have Supervillains and Executive Champions at that level in a Fortune 500 company. I won't bore you with the details, but our Executive Champion and Customer Champion spent a huge amount of time fending off attacks for about two years before the project finished. (I don't think they enjoyed that part of the project.)

CHAPTER 5: TRANSLATING USER NEEDS INTO DATA MODELS

Exercise 1 Solution

Figure A.3 shows one possible solution.

- In the `STUDENT` class, `COURSE` and `PROJECT` have cardinality 0..N and 0..1, respectively. This doesn't capture the fact that at least one of these two attributes must include at least one value.
- Similarly the `INSTRUCTOR` class does not capture the fact that at least one of the `COURSE` or `PROJECT` attributes must include at least one value.

Exercise 2 Solution

Figure A.4 shows an inheritance diagram for the `Person`, `Student`, and `Instructor` entities. It also shows the relationship between the `Person` and `Phone` entities.

The `Phone` entity doesn't have a primary key because it doesn't make sense to search for just a `Phone` entity by itself. Instead, you can find the `Phone` entities corresponding to a `Person` entity. That means `Phone` is a weak entity so it is surrounded by a thick rectangle and its identifying relationship is drawn with a thick arrow.

Figure A.5 shows one possible ER diagram for the college course data.

The diagram has the following constraints:

- It doesn't make sense to look for a particular `CourseResult` so it doesn't have a primary key. Instead you can look for `CourseResults` associated with a `Student` or with a `Course`. That means `CourseResult` is a weak entity, so it is drawn with a thick rectangle and it is connected to its identifying relationships with thick arrows.
- Similarly `ProjectResult` is a weak entity.

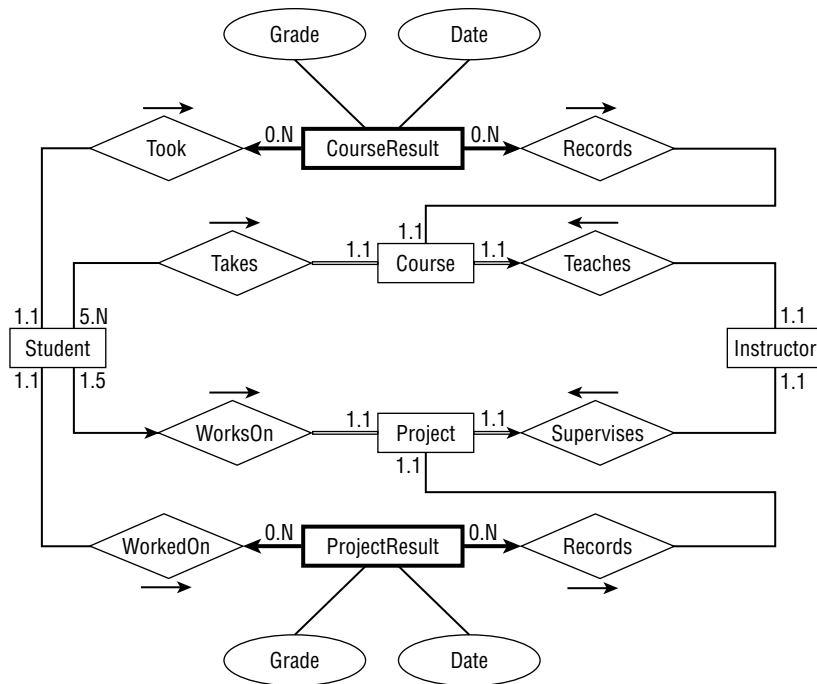


FIGURE A.5

- A Course must be involved in a relationship with a Student (or else the Course is canceled), so its line leading toward Student is double (a participation constraint).
- Similarly a Project must be involved in a relationship with a Student, so its line leading toward Student is double (a participation constraint).
- A Course must be involved in a relationship with an Instructor (someone has to teach it), so its line leading toward Instructor is double (a participation constraint). A Course can have only one Instructor, so the line is also an arrow (a key constraint).
- Similarly, a Project must be involved in a relationship with exactly one Instructor, so its line leading toward Instructor is a double arrow (participation and key constraint).
- A Student can work on at most one Project at a time, so its line leading to Project is an arrow (key constraint).

Special notes:

- The Student entity's relationships with Course and Project do not indicate that a Student must be involved with at least one Course or a Project.
- Similarly, the Instructor entity's relationships with Course and Project do not indicate that an Instructor must be involved with at least one Course or a Project.

Exercise 3 Solution

Figure A.6 shows one possible solution.

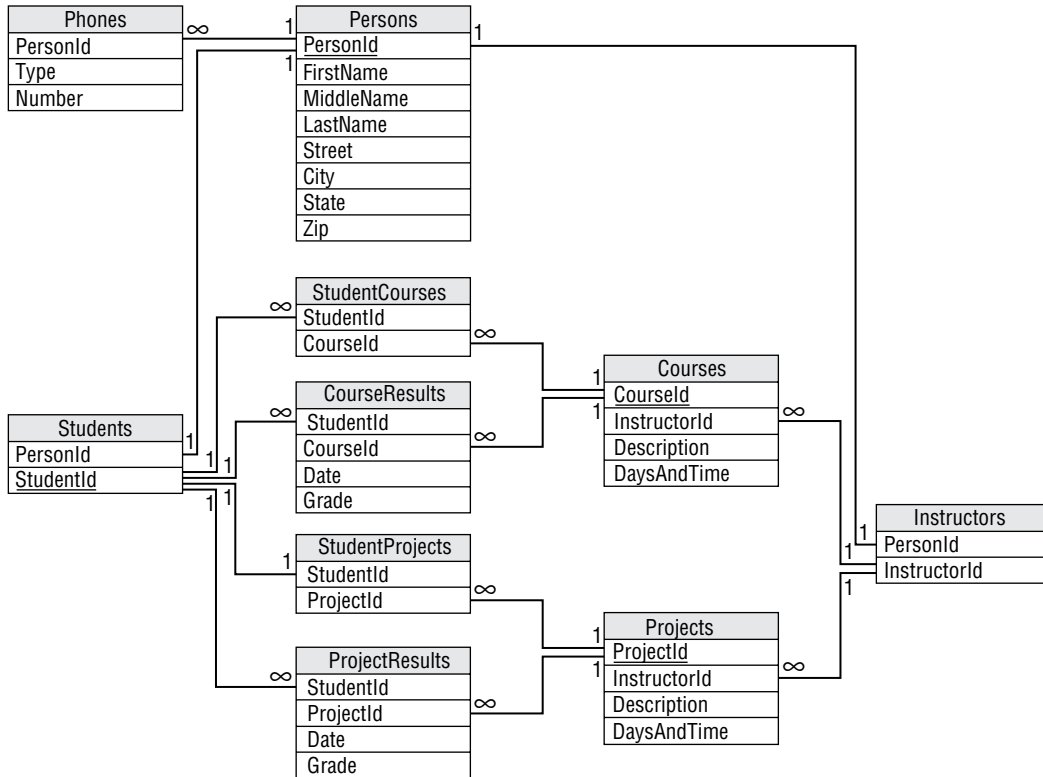


FIGURE A.6

Notice the way this model handles the fact that `Student` and `Instructor` inherit from `Person`. The `Persons` table holds the basic `Person` information and a `PersonId`. The `Students` and `Instructors` tables include `PersonId` foreign keys to link to the corresponding basic `Person` data.

Note also the different approach used for the `Student/Course` and `Instructor/Course` relationships. Because a course has exactly one instructor, the `Instructors` and `Courses` tables are connected with a simple one-to-many relationship. In contrast, a course has many students, so the relationship uses an intermediate `StudentCourses` table to connect the two to build a many-to-many relationship. (The same reasoning applies to the `Student/Project` and `Instructor/Project` relationships.)

Finally, notice the difference between the `Student/Course` and `Student/Project` relationships. A student can be enrolled in any number of courses but at most one project, so the first is a many-to-many relationship while the second is a one-to-one relationship.

Unfortunately, this solution doesn't capture every aspect of the system either. In particular, it doesn't indicate that a `Student` must be enrolled in at least one `Course` or a `Project`. Similarly, it doesn't show

that an Instructor must teach at least one Course or supervise at least one Project. The model also doesn't include data type, required, and other domain data. All of this should be noted in separate documents.

Exercise 4 Solution

Figure A.7 shows one possible solution.

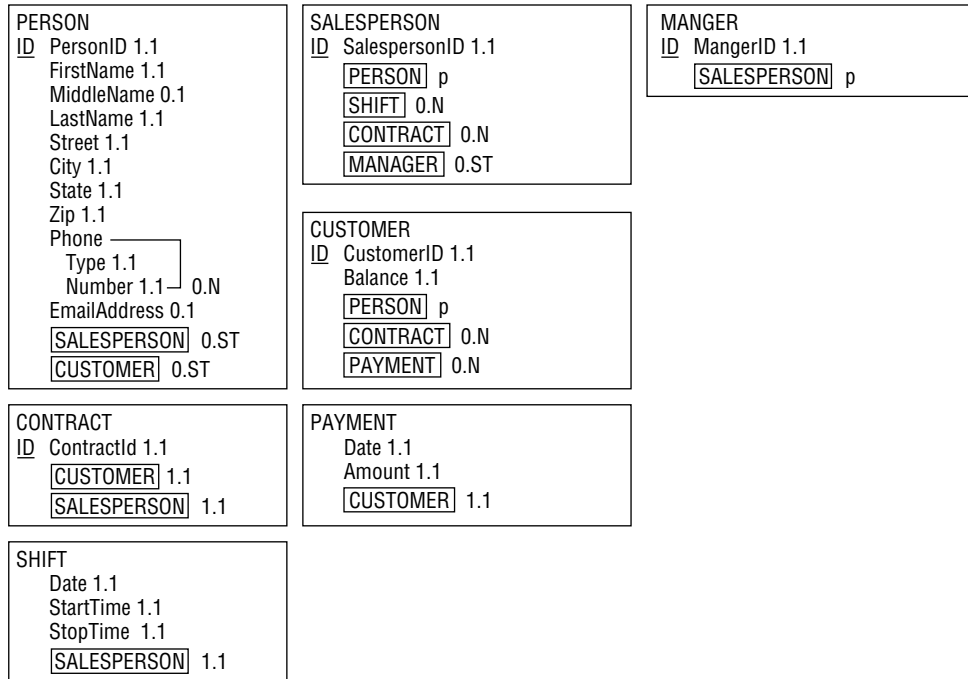


FIGURE A.7

Special notes: the semantic object model actually does a pretty good job of capturing the Mike's Trikes data. About the only item that isn't described explicitly is the manager's role. In this model, you can deduce the manager at any given time by examining the manager's shift data. If Mike needed a more explicit record of who is managing during a salesperson's shift or when a contract was sold, the model would need to be modified.

Exercise 5 Solution

Figure A.8 shows an inheritance diagram for the Person, Customer, Salesperson, and Manager entities. It also shows the relationship between the Person and Phone entities.

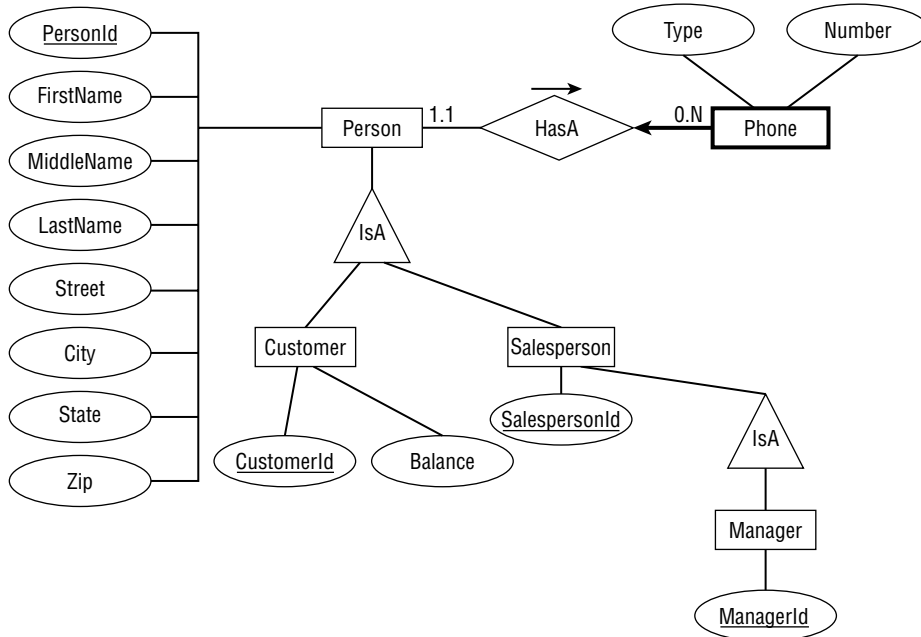


FIGURE A.8

The **Phone** entity doesn't have a primary key because it doesn't make sense to search for just a **Phone** entity by itself. Instead, you can find the **Phone** entities corresponding to a **Person** entity. That means **Phone** is a weak entity, so it is surrounded by a thick rectangle and its identifying relationship is drawn with a thick arrow.

Figure A.9 shows one possible ER diagram for Mike's Trikes.

The diagram's constraints are:

- **Payment** is a weak entity because you look up payments via the **Customer** who made them. **Payment** is drawn with a thick rectangle and a thick arrow pointing toward its identifying relationship.
- **Shift** is also a weak entity because you look up shift data via the **Salesperson** who works the shift. **Shift** is drawn with a thick rectangle and a thick arrow pointing toward its identifying relationship.
- A **Customer** must be involved in at least one **Contract** (we don't make a **Customer** record until **Customer Purchases Contract**), so its line leading toward **Contract** is double (a participation constraint).
- A **Contract** must have exactly one **Customer** and exactly one **Salesperson**, so the lines leading out of **Contract** toward those other entities are double (participation constraint) and arrows (key constraint).

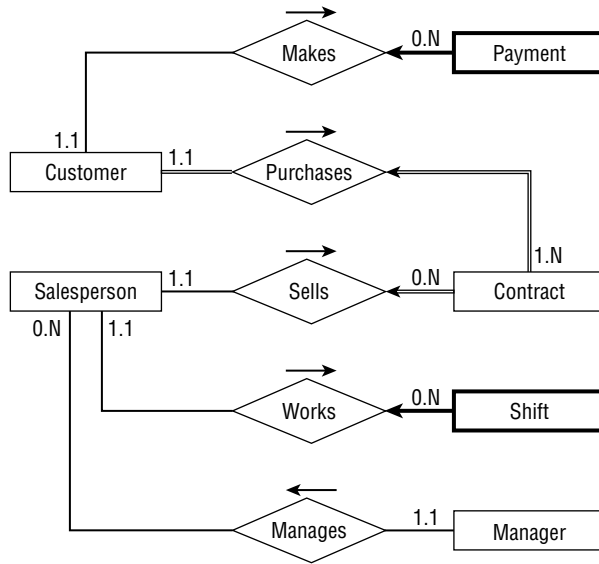


FIGURE A.9

Special notes:

- This diagram does not emphasize the fact that a `Manager` is also a `Salesperson`, so a manager could play the role of the `Salesperson` in the diagram. You could add the `Manager Works Shift` relationship but that would complicate the diagram.

Exercise 6 Solution

Figure A.10 shows one possible solution.

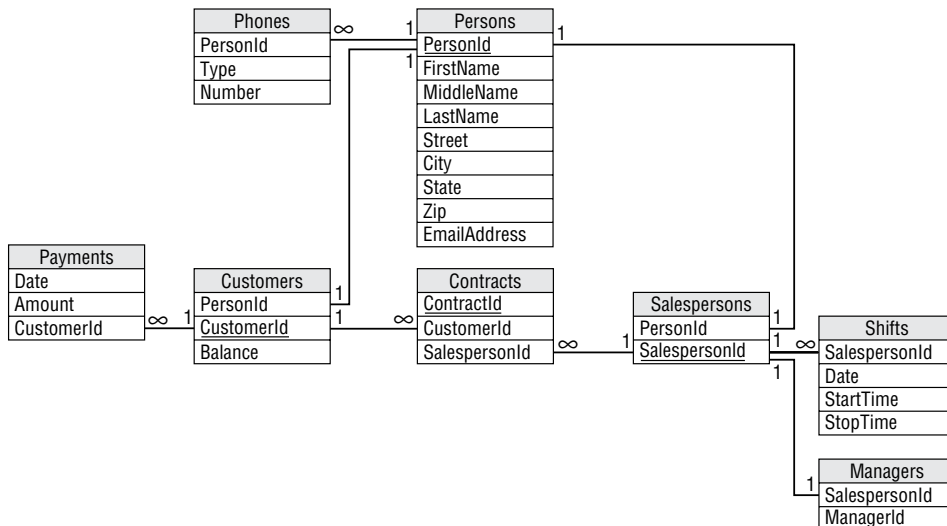


FIGURE A.10

Notice how this model builds the inheritance hierarchy. The Customers and Salespersons tables use PersonId foreign key fields to link to their corresponding Persons records. The Managers table uses a SalespersonId foreign key field to link to Salespersons records.

As usual, the model doesn't capture all of the information available about the situation. In particular, it doesn't indicate that a Customers record must be associated with at least one Contracts record. You should write down this and other facts such as field data types and domain information in separate documents.

Exercise 7 Solution

Figure A.11 shows one possible solution.

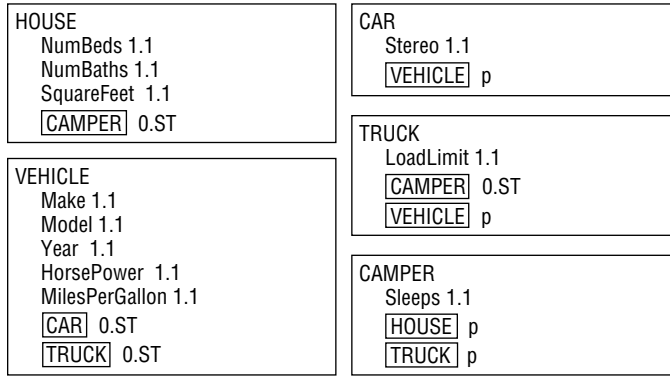


FIGURE A.11

Exercise 8 Solution

Figure A.12 shows one possible solution.

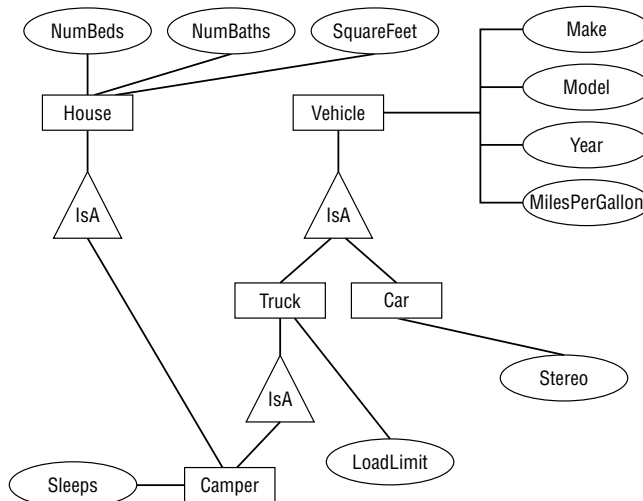


FIGURE A.12

CHAPTER 6: EXTRACTING BUSINESS RULES

Exercise 1 Solution

The following chart describes the Phones table.

FIELD	REQUIRED	DATA TYPE	DOMAIN	SANITY CHECKS
PersonId	Yes	ID	Persons.PersonId	
Type	Yes	String	List: Home, Work, Cell, Fax	
Number	Yes	String	Phone numbers	

The following chart describes the Persons table.

FIELD	REQUIRED	DATA TYPE	DOMAIN	SANITY CHECKS
PersonId	Yes	ID	Any ID	
FirstName	Yes	String	Any string	
MiddleName	No	String	Any string	
LastName	Yes	String	Any string	
Street	Yes	String	Any string	
City	Yes	String	Any string	
State	Yes	String	List: (states)	
Zip	Yes	String	ZIP or ZIP+4 format	Verify ZIP or ZIP+4 format
EmailAddress	No	String	Valid email address	Contains one @ symbol
MedicalNotes	?	String	Any string	
IceQualified?	?	Yes/No	Yes or No	
RockQualified?	?	Yes/No	Yes or No	
JumpQualified?	?	Yes/No	Yes or No	

The following chart describes the Guides table.

FIELD	REQUIRED	DATA TYPE	DOMAIN	SANITY CHECKS
PersonId	Yes	ID	Persons.PersonId	
GuideId	Yes	ID	Any ID	
IceInstructor?	Yes	Yes/No	Yes or No	
RockInstructor?	Yes	Yes/No	Yes or No	
JumpInstructor?	Yes	Yes/No	Yes or No	

The following chart describes the Explorers table.

FIELD	REQUIRED	DATA TYPE	DOMAIN	SANITY CHECKS
PersonId	Yes	ID	Persons.PersonId	
ExplorerId	Yes	ID	Any ID	

The following chart describes the Organizers table.

FIELD	REQUIRED	DATA TYPE	DOMAIN	SANITY CHECKS
PersonId	Yes	ID	Persons.PersonId	
OrganizerId	Yes	ID	Any ID	

The following chart describes the Adventures table.

FIELD	REQUIRED	DATA TYPE	DOMAIN	SANITY CHECKS
AdventureId	Yes	ID	Any ID	
ExplorerId	Yes	ID	Explorers.ExplorerId	
EmergencyContact	Yes	ID	Persons.PersonId	
OrganizerId	Yes	ID	Organizers.OrganizerId	

FIELD	REQUIRED	DATA TYPE	DOMAIN	SANITY CHECKS
TrekId	Yes	ID	Treks.TrekId	
DateSold	Yes	Date	Any date	Before the trek's start date. Between January 1, 2000 and December 31, 2050 (or some other very early and late dates).
IncludeAir?	Yes	Yes/No	Yes or No	
IncludeEquipment?	Yes	Yes/No	Yes or No	
TotalPrice	Yes	Currency	Monetary amount > \$0	Price > \$250 (or some minimum sane value).
Notes	?	Yes/No	Yes or No	

The following chart describes the Treks table.

FIELD	REQUIRED	DATA TYPE	DOMAIN	SANITY CHECKS
TrekId	Yes	ID	Any ID	
GuideId	Yes	ID	Guides. GuideId	
Description	Yes	String	Any string	Length > 100 (anything shorter couldn't say enough).
Locations	Yes	String	List of locations	Length > 5.
StartLocation	Yes	String	A location	Length > 5.
EndLocation	Yes	String	A location	Length > 5.

continues

(continued)

FIELD	REQUIRED	DATA TYPE	DOMAIN	SANITY CHECKS
StartDate	Yes	Date	Any date	StartDate is on or before EndDate. Between January 1, 2000 and December 31, 2050 (or some other very early and late dates).
EndDate	Yes	Date	Any date	EndDate is on or after StartDate. Between January 1, 2000 and December 31, 2050 (or some other very early and late dates).
Price	Yes	Currency	Monetary amount > \$0	Price > \$250 (or some minimum sane value). Price > some minimum price per day times the number of days (EndDate–StartDate).
MaxExplorers	Yes	Number	Number > 0	Number > 0. Number < 20 (or some maximum sane amount).
IceRequired?	Yes	Yes/No	Yes or No	
RockRequired?	Yes	Yes/No	Yes or No	
JumpRequired?	Yes	Yes/No	Yes or No	

Exercise 2 Solution

The following list describes business rules that can be implemented in field or table checks for the Phones table:

- **Type**—Verify that the type is one of Home, Work, Cell, or Fax. Alternatively, if you think this list might change in the future, you could put these values in a lookup table.
- **Number**—Verify that the value has a valid phone number format. In the United States, you would probably want to verify that it is a 10-digit number of the format ###-###-#### and you should allow for an extension.

The following list describes business rules that can be implemented in field or table checks for the Persons table:

- **FirstName/MiddleName/LastName**—Verify that this combination is unique. This will prevent you from adding the same person twice, perhaps as an explorer and as an emergency contact.

It would also be natural to try to validate the `EmailAddress` field in a field check. Unfortunately, valid email address formats are quite complicated so this probably doesn't belong in the simpler field and table checks.

Similarly, it might be nice to look up the explorer's `City`, `State`, and `Zip` values to make sure they are compatible. If you build a table listing all of the possible combinations, this wouldn't be a hard check, but it would be an enormous table and it's probably not worth all the extra effort. (For bonus points, though, you could probably use a web service to perform this check over the Internet. If you don't know what a web service is, don't worry about it.)

You could also look up the `State` value in a list built into a field check. Although it's unlikely that the list of allowed states will change often, this list is so long that it's easier to manage in a separate lookup table rather than in a very long field check. (And who knows, Canada may eventually be officially recognized as "The Maple Leaf State.") (Just kidding! But this does bring up a whole series of questions about non-U.S. explorers. This model ignores those issues completely. Yes, I feel guilty.)

The `Explorers`, `Organizers`, and `Guides` tables should verify that their records are unique. That means checking uniqueness for the `Explorers` table's `PersonId/ExplorerId` fields, the `Organizers` table's `PersonId/OrganizerId` fields, and the `Guides` table's `PersonId/GuideId` fields.

The following list describes business rules that can be implemented in field or table checks for the `Adventures` table:

- **Table**—Verify that the trek has room for this explorer.
- **Table**—Verify that the explorer's `IceQualified?`, `RockQualified?`, and `JumpQualified?` values include those required for this trek.
- **ExplorerId/TrekId**—Verify that this combination is unique. An explorer should not buy the same trek twice. (We're assuming that the same trip on different dates gets a different record in the `Treks` table and thus a different `TrekId`. Some people may very well want to go to the same places again.)
- **EmergencyContact**—Verify that the `EmergencyContact` is not going on the same trek listed for this `Adventures` record.
- **IncludeAir?/Notes**—If `IncludeAir?` is `Yes`, then the `Notes` field should include flight information such as the explorer's starting airport and meal preferences. The database can probably not verify that the notes make sense (who knows if the low-sodium meal is available on that flight?), but it can verify that the `Notes` entry has some minimum length if `IncludeAir?` is `Yes`.

The `Adventures` table would be a natural place to try to deal with the discounts for purchasing airline tickets or renting equipment. You would set `TotalPrice` equal to the trek's cost minus any discounts. (Note that this model doesn't have room to hold information about the equipment rented. The full model would need more order-related information along those lines.)

In any case, the discount schedule seems likely to change so it's better handled later, not in a simple field or table check.

The following list describes business rules that can be implemented in field or table checks for the `Treks` table:

- **Table**—Verify that the guide's `IceQualified?`, `RockQualified?`, `JumpQualified?`, `IceInstructor?`, `RockInstructor?`, and `JumpInstructor?` values include those required for this trek.

Exercise 3 Solution

The following list summarizes business rules that should be extracted from the database's structure:

- If you really want to validate email addresses, it would be better to do so outside of the field and table checks. You could put this code in a stored procedure, code library, or middle tier.
- If you use a lookup table to validate phone number types (Home, Work, Cell, or Fax), do so here.
- If you're going to perform a complex City/State/Zip lookup, this is where to do it. You might use a huge table or you might call a web service over the Internet.
- If you use a lookup table to validate State values, do so here.
- This is where you would calculate an adventure's TotalPrice. You would look up discount information stored in a separate table and perform the calculation. You could put this code in a stored procedure, code library, or middle-tier layer.
- The fact that one of the company's owners asked which calculation would give the customer the biggest discount if they both purchase airline tickets and rent equipment (by the way, add the two discounts and take 15 percent off to give the biggest discount) further implies that they might someday change the way they perform this calculation. That gives you more reason to extract this rule from the database so it's easier to change later.
- If the adventure's IncludeAir? value is Yes, you could try to parse the Notes field to see if the flight and meal information is present. If you really need this check, you should move the flight and meal information into separate fields so they are easier to find and examine. (I've seen several systems that make these sorts of checks on notes fields, mostly because the requirements changed after the database was built and they couldn't easily modify the tables to include new fields. Scanning notes is always hard, so mistakes are common.)

Exercise 4 Solution

The PhoneTypes table would have only one field: Type. The records would initially include Home, Work, Cell, and Fax.

The States table would have only one field: State. The records would list all of the allowed State values: AL, AK, AS, ..., WY.

The DiscountParameters table would have two fields: Type and Amount. Type would give the discount type (Air or Equipment) and Amount would be the discount amount (15 or 5 percent).

An additional Parameters table would have two fields: Name and Value. This table would hold parameters used in other calculations so that they would be easier to update than they would be if they were embedded in check constraints. The following table describes the initial values in this table.

NAME	VALUE	PURPOSE
MinimumDate	January 1, 2000	Sanity check date for DateSold, StartDate, and EndDate
MaximumDate	December 31, 2050	Sanity check date for DateSold, StartDate, and EndDate
MinimumTotalPrice	\$250	Sanity check price for an Adventure's TotalPrice
MinimumTrekPrice	\$250	Sanity check price for a Trek's Price
MinimumPricePerDay	\$100	Sanity check minimum price per day for a Trek's Price
MaximumExplorers	20	Sanity check maximum number of explorers on a trek

CHAPTER 7: NORMALIZING DATA

Exercise 1 Solution

- a. The list isn't in 1NF because it violates these 1NF rules:
1. **Each column must have a unique name.** The two Email fields have the same name.
 2. **The order of the rows and columns doesn't matter.** The order of the Email columns represents the student's preferred email address.
 3. **Each column must have a single data type.** The MajorOrSchool field holds both majors and schools.
 5. **Each column must contain a single value.** The Name field contains the student's first and last names together.

Let's take these rules one at a time.

1. **Each column must have a unique name.** The two Email fields have the same name. You could fix this problem by giving them different names. For example, you could name them Email1 and Email2. The numbers would indicate the student's preferred email address solving the problem with Rule 2. This is the approach taken by the Phone1 and Phone2 fields so it might work, right? Not really.

There's another equally important issue here. These two Email fields represent the same kind of data with only a minor difference: priority. Aside from the student's preference of which comes first, the two fields hold identical values. How do we know you won't want to add a third email address later? You've already got two, why not three or four? Simply renaming the fields solves the duplicate name issue, but locks you in to exactly two email addresses. Not only would that prevent you from adding more email addresses, but in many cases the second field would be empty.

This is also flirting with 1NF rule number 6: Columns cannot contain repeating groups. A better solution to the multiple Email field problem would be to pull those fields into a new StudentEmails table.

While we're thinking about multiple fields holding the same kind of data, let's take a closer look at the Phone1, PhoneType1, Phone2, and PhoneType2 fields. Although they have different names, they also represent the same kind of information and you're probably even more likely to want a third phone number than you are to want a third email address. Although these fields technically don't violate 1NF (aside from Rule 6), it's probably worthwhile moving them into a new StudentPhones table.

2. **The order of the rows and columns doesn't matter.** The order of the Email columns represents the student's preferred email address. The new StudentEmails table should have a Priority column to capture the student's preference. Similarly, the new StudentPhones table should have a Priority column to indicate the student's preference.
3. **Each column must have a single data type.** The MajorOrSchool field holds both majors and schools. It should be split into Major and School fields. Note that students have a school whether they have a major or not, so the School field should always contain a value while the Major field may contain null.
5. **Each column must contain a single value.** The Name field contains the student's first and last names together. Here, you need to decide whether the name value is atomic. In other words, will you *ever* need to do something with just a first name or just a last name? Chances are good that you'll want to at least be able to search for last names (so you can easily look up students), so you should split this field into FirstName and LastName fields.

b. Figure A.13 shows a relational diagram for this model.

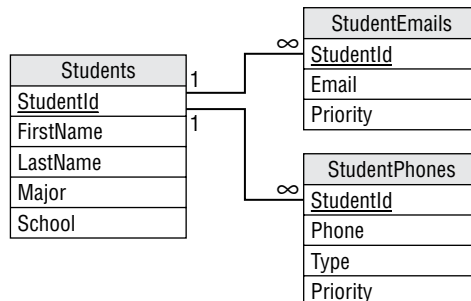


FIGURE A.13

Exercise 2 Solution

- a. The list isn't in 1NF because it violates these 1NF rules:
2. **The order of the rows and columns doesn't matter.** The order of the rows represents the rows' priorities.
 5. **Each column must contain a single value.** The Items column contains a comma-separated list of values.
- b. The following table shows one way to convert the list into 1NF.

LOCATION	ITEM	PRIORITY
Grocery store	milk	1
Grocery store	eggs	1
Grocery store	bananas	1
Office supply store	paper	2
Office supply store	pencils	2
Office supply store	divining rod	2
Post Office	stamps	3
Computer store	flash drive	4
Computer store	8" floppy disks	4

The primary key for this table is the combination Location/Item.

Exercise 3 Solution

- a. The list isn't in 2NF because it violates the 2NF rule:
2. **All of the non-key fields depend on all of the key fields.** The Priority field depends on Location but not on Item. That's why its values are repeated so many times in the table.
- b. The solution is to pull the non-key field (Priority) out into a new table and use the key field that it depends on (Location) as the link to the original data. Figure A.14 shows the new relational design.

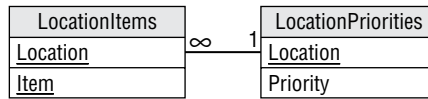


FIGURE A.14

Figure A.15 shows the new tables holding the original data.

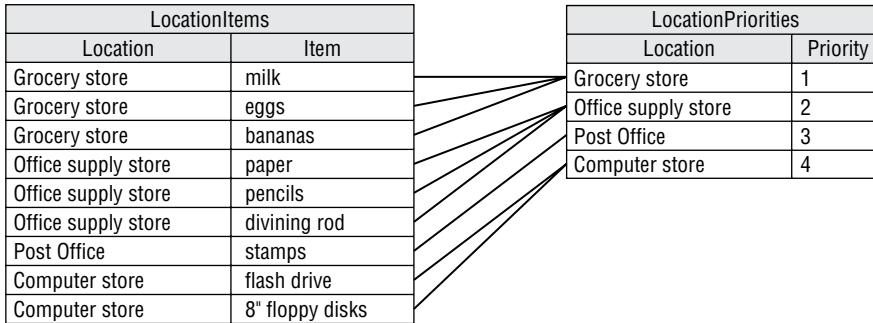


FIGURE A.15

Exercise 4 Solution

- a. The list isn't in 3NF because it violates the 3NF rule:
 - 2. **It contains no transitive dependencies.** In this table, the Department field depends on the Project. Because those fields are not key fields, this is a transitive dependency.
- b. The solution is to pull the dependent field (Department) out into a new table and use the field that it depends on (Project) as the link to the original data. Figure A.16 shows the new relational design.

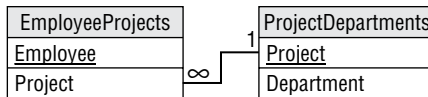


FIGURE A.16

Figure A.17 shows the new tables holding the original data.

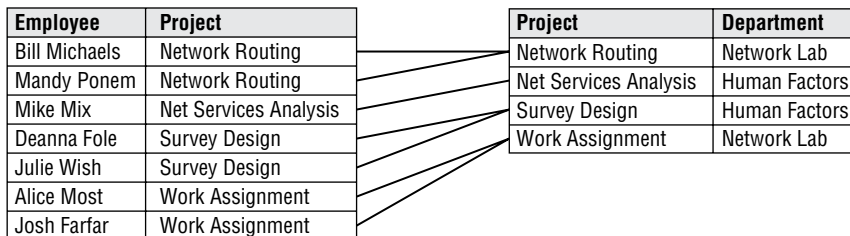


FIGURE A.17

Exercise 5 Solution

a. The table isn't in 5NF because it violates the 5NF rule:

2. **It contains no related multivalued dependencies.** In this table, Person determines Food (the type the person can make), Person determines Tools (those in the person's kitchen), and Tool partially determines Food (you can't make muffins without a muffin tin). This makes a related multivalued dependency. Figure A.18 shows an ER diagram for this model.

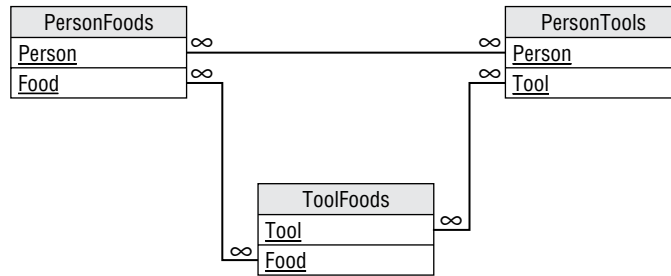


FIGURE A.18

b. The solution is to break the single table into three new tables that record the three different relationships: Person/Food, Person/Tool, and Tool/Food. Figure A.19 shows the new relational model.

PersonFoods	
Person	Food
Alice	Muffins
Alice	Omelets
Alice	Pancakes
Bob	Muffins
Bob	Omelets
Bob	Pancakes
Cyndi	Omelets

PersonTools	
Person	Tool
Alice	Muffin tin
Alice	Omelet pan
Alice	Pancake griddle
Bob	Omelet pan
Cyndi	Muffin tin
Cyndi	Pancake griddle

ToolFoods	
Tool	Food
Muffin tin	Muffins
Omelet pan	Omelets
Pancake griddle	Pancakes

FIGURE A.19

Exercise 6 Solution

Figure A.20 shows the matching between normal forms and their rules.

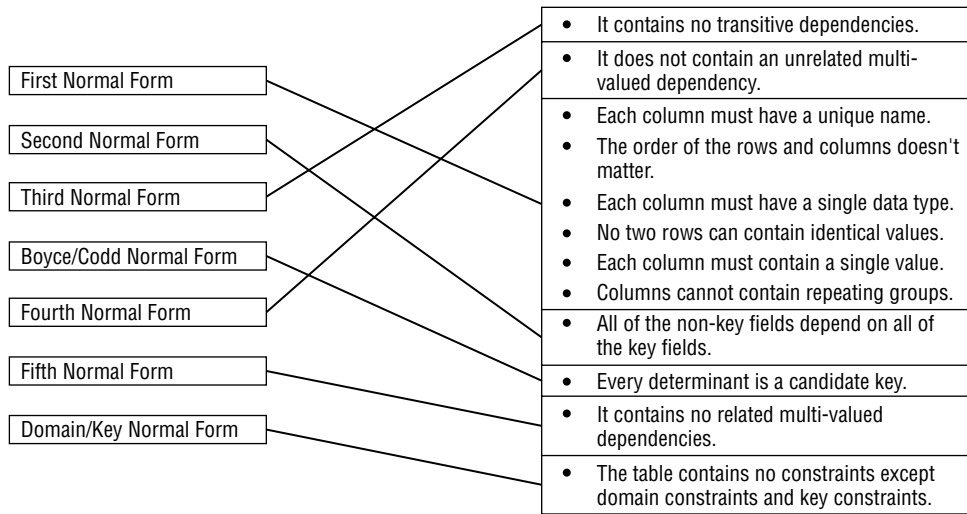


FIGURE A.20

CHAPTER 8: DESIGNING DATABASES TO SUPPORT SOFTWARE

Exercise 1 Solution

The following ShipClasses table contains the allowed combinations of Ship and Class.

SHIP	CLASS
Luxury Liner	1st Class
Luxury Liner	2nd Class
Luxury Liner	3rd Class
Luxury Liner	4th Class
Luxury Liner	5th Class
Schooner	1st Class
Schooner	2nd Class
Tuna Boat	1st Class
Barge	None

Because the validation involves two fields, this must be a two-field foreign key constraint. In the Trips table, the combination of fields Ship/Class will be a foreign key referencing the ShipClasses table's Ship/Class fields.

Exercise 2 Solution

The Students table holds information about students, so it is an object table. Similarly, the Departments table holds information about the school's departments and the Classes table holds information about classes, so they are also object tables.

The StudentClasses table links the Students and Classes tables, so it is a link table. Similarly, the DepartmentClasses table links the Departments and Classes tables, so it is also a link table.

Exercise 3 Solution

This table is trying to hold information about three different concepts: the first player, the second player, and the match they will play.

To fix it, create a Players table with fields PlayerId, Name, and Rank. Put all of the player information in this table for all of the Player1 and Player2 entries. This is an object table holding information about players.

Then create a Matches table that has fields PlayerId1, PlayerId2, and MatchTime. This is a link table that links the Players table to itself. It also holds extra information about the link: the times of the matches.

Exercise 4 Solution

The following list tells which daily values should be stored in a redundant variable and which should be calculated as needed.

- a. Average minutes late for an airline at a particular airport. This will require finding and averaging up to a few hundred values, so it should be possible to calculate as needed.
- b. Average minutes late for all airlines at a particular airport. This will require finding and averaging several hundred values. It might still be possible to perform this calculation as needed.
- c. Average minutes late for an airline across the country. This could require a huge number of calculations. If this is a common query (for example, if many people are asking for this information all over the country hundreds of times per day), then it might be better to store and update the information as planes take off and land rather than calculating it as needed.
- d. Average minutes late for all airlines across the entire country. This will require an enormous number of calculations. This could take quite a while even if the database isn't heavily used, so it might be best to store this value rather than calculating it as needed.

Of course, as long as you're going to store some of these values, you might want to just store them all so you can treat them uniformly.

CHAPTER 9: USING COMMON DESIGN PATTERNS

Exercise 1 Solution

Figure A.21 shows an ER diagram to represent Parcheesi matches.

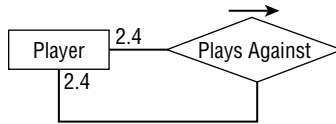


FIGURE A.21

Exercise 2 Solution

Figure A.22 shows a relational model for recording information about Parcheesi matches. PlayerId1 finished first, PlayerId2 finished second, PlayerId3 finished third, and PlayerId4 finished fourth.

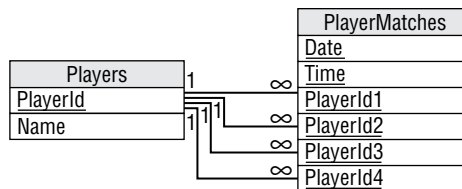


FIGURE A.22

Exercise 3 Solution

Figure A.23 shows an ER diagram that represents the relationships between Match, Move, and Ply.

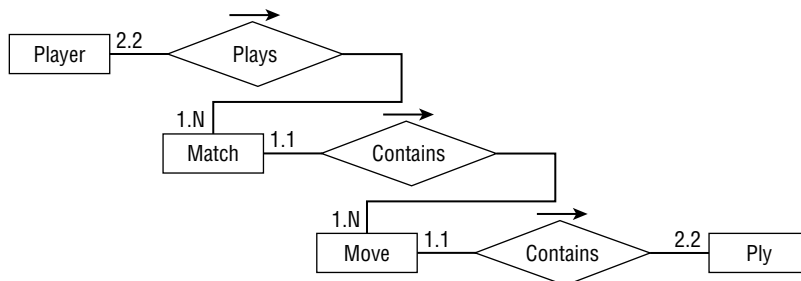


FIGURE A.23

Exercise 4 Solution

Figure A.24 shows a relational model for recording chess Match, Move, and Ply data.

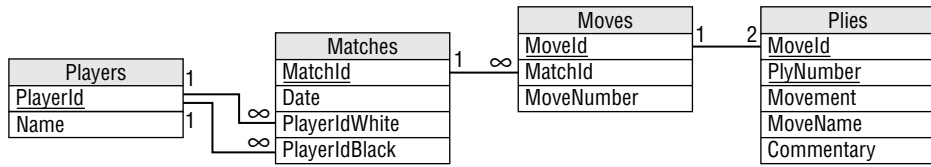


FIGURE A.24

You can model the one-to-two relationship between Moves and Plies by making the domain of the PlyNumber field include the values 1 and 2. You can implement that as a field-level check constraint on PlyNumber. There’s no need to make this a foreign key constraint because the International Chess Federation will never change a move to include more than or fewer than two plies.

Note that the fact that MoveId/PlyNumber is the Plies table’s primary key ensures that each move cannot contain two plies with the same PlyNumber.

Exercise 5 Solution

Figure A.25 shows the chess model without the Moves table.

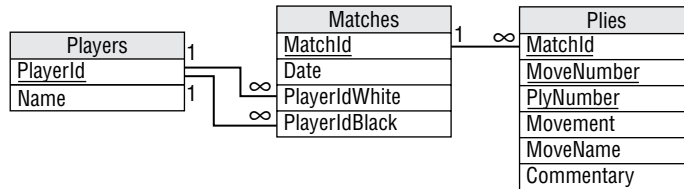


FIGURE A.25

The new diagram doesn’t explicitly show that there should be exactly two plies per move. It has converted the old one-to-two relationship into a new one-to-many relationship.

The database still needs to verify that there are only two plies per move, however. You can still use a field-level check constraint to verify that the PlyNumber is either 1 or 2. The fact that MatchId/MoveNumber/PlyNumber is the Plies table’s primary key ensures that any move in a given match cannot contain two plies with the same PlyNumber.

Exercise 6 Solution

The network pattern described in the section “Network Data” earlier in this chapter uses the two tables shown in Figure A.26. The Nodes table holds node IDs and coordinates. The Links table holds link times and the IDs of the nodes that each link connects.

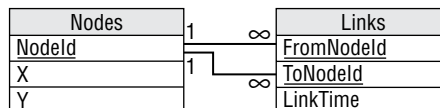


FIGURE A.26

The pipe network exercise is slightly different because it is an undirected network. In other words, each link has the same “value” no matter which direction you cross it. The solution shown in Figure A.26 isn’t perfect because the FromNodeId and ToNodeId fields imply a direction for the link. To use this design, you would either need to recognize that a Links record connecting node1 to node2 also represents a link connecting node2 with node1. Or you could insert two records for each link with the order of the node IDs switched, but that would double the number of records and all of that duplication screams out, “I’m not normalized!”

In normalization terms, FromNodeId and ToNodeId store the same kind of data. For a directed network, the two fields are not exactly the same thing, so there’s some benefit to using two fields with different names to store their data and differentiate them.

Normalization purists would say that the link’s node data should be moved into a new table with an extra field to tell you which was the “from” node and which was the “to” node. For a directed network, the extra layer of indirection seems like a lot of work for little benefit. In addition to making you follow extra links to find the data, you would also need to perform some new validations to ensure that every link corresponded to exactly two nodes.

However, this more normalized design works somewhat better for the undirected network that we have in this exercise because moving the link’s nodes into a new table removes the implication that one is the “from” node and one is the “to” node.

You still need a way to ensure that each link has two nodes, however. One way to do that is to give the new table a NodeNumber field to indicate which node this is, make the domain of NodeNumber be the numbers 1 and 2, and make LinkId/NodeNumber the primary key. That ensures that any link can have only two nodes. This design is shown in Figure A.27.

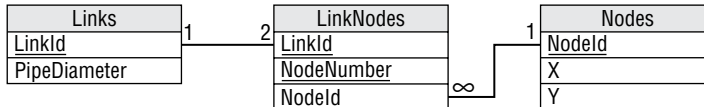


FIGURE A.27

This is the same as the normalized design for a directed network. The only difference is that in the undirected network you treat the NodeNumber field as a simple index to ensure that a link has two nodes whereas in a directed network you use that field to tell which node is “from” and which is “to.”

Exercise 7 Solution

This is fairly straightforward temporal data. Figure A.28 shows a model to hold cheese item data. A CheeseItem record would probably hold other information such as the quantity of cheese purchased, the lot number, and so forth.

CheeseItem
<u>CheeseItemId</u>
CheeseType
SellByDate

FIGURE A.28

Exercise 8 Solution

Figure A.29 shows the new model to hold cheese item data. Instead of a SellByDate, this version stores the date when the cheese was made and a link leading to the shelf life. The CheeseType record could also store other cheese data such as the type of milk used (cow, buffalo, goat, yak, horse, and so forth), the location where it was made, and a description (a firm, fruity and nutty cheese reminiscent of locusts and with a hint of lichen).

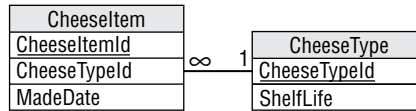


FIGURE A.29

In the model, the CheeseItem table is the same size as the model for Exercise 7 and there's a new table, so you could ask if this is an improvement. In terms of looking up expiration data for a particular cheese item, it is not. It takes more space and requires an extra lookup plus a calculation ($\text{MadeDate} + \text{ShelfLife}$) to find the cheese item's sell-by date.

However, this model provides more consistency and avoids update anomalies because it ensures that each item of a particular kind of cheese uses the same shelf life.

Exercise 9 Solution

The following table shows the cost per month for the different plans.

PLAN	STORAGE COST (PER MONTH)	RETRIEVAL COST (PER MONTH)	TOTAL COST (PER MONTH)
Standard	$\$0.0200 * 10,000 \text{ GB} = \200	$\$0.00 * 1,000 \text{ GB} = \0	\$200
Nearline	$\$0.0100 * 10,000 \text{ GB} = \100	$\$0.01 * 1,000 \text{ GB} = \10	\$110
Coldline	$\$0.0040 * 10,000 \text{ GB} = \40	$\$0.02 * 1,000 \text{ GB} = \20	\$60
Archive	$\$0.0012 * 10,000 \text{ GB} = \12	$\$0.05 * 1,000 \text{ GB} = \50	\$62

In this example, the Coldline plan is the least expensive.

CHAPTER 10: AVOIDING COMMON DESIGN PITFALLS

Exercise 1 Solution

This table has a lot of problems. Specific problems include:

- The Name field includes two logical fields, FirstName and LastName, so the table isn't even in first normal form.
- Your client plans to look up the state from the Zip value. Why doesn't he also look up the city? The table should be changed to either also look up the city or have separate City, State, and Zip fields (the second option is a lot easier, although less normalized).
- The two phone number fields are not differentiated. In other words, how do you know which number is a home phone, cellphone, or work phone? Which is the daytime number and which is the evening number? These fields should be moved into a Phones table with an additional field indicating the type of the phone number.
- Having at most two phone numbers is an arbitrary limit. Someday, a customer will probably want to leave more than two numbers. When you create the Phones table, you should not restrict a customer to two entries.
- The Address field has a bad name because Address implies that the field contains an entire address when, in fact, it only contains the street information. This field's name should be something like Street or StreetAddress.
- The Stuff field has a terrible name because "Stuff" could mean just about anything! This field's name should be changed to something like Interests.
- The freshly renamed Interests field lists more than one value. (The fact that the name is plural is a hint.) This field's data should be moved into a new CustomerInterests table. You should also make an Interests lookup table to list the allowed values so CustomerInterests can use it as a foreign key constraint.
- Planning for future changes, you might also suggest adding an Email field.

Your client's assumption that you can just build Orders and other tables implies that the plan isn't very well-thought-out. This project definitely needs a lot more planning and a complete database design before you start slapping tables together. This kind of homegrown project also rarely includes documentation of any kind, so you'll need to do a lot of documentation work early in the project. (Though this type of project may provide many hours of lucrative consulting later for debugging, it's the frustrating kind of consulting.)

Exercise 2 Solution

Because this client is opening a new store, you should wonder if they will grow even more in the next few years. Flying cars are also a brand-new technology, and if it they become as popular as *The Jetsons* cartoon indicates, demand for rentals could skyrocket.

This database will need extra testing at very high loads to verify that the database design can meet ever-increasing performance demands.

In contrast, a well-established party rental store probably won't experience explosive growth in the near future because it's been around for a while and it isn't selling new technology. You still need to thoroughly test their application, but your load testing doesn't need to run at loads quite as far beyond the current level.

Exercise 3 Solution

This table is hyper-normalized. Although you can break a street address into name, number, prefix, and so forth, there are very few applications where that is necessary. If you will only ever need to use the address information to send mail to someone, then you can combine all of this information in a single Street field. You can even include the apartment or suite number.

Similarly, you can probably combine the Zip and PlusFour fields into a single Zip field. If you're only going to use the ZIP Code to write addresses, there's no need to use separate fields.

The Floor and Neighborhood information is also probably not useful. (Although if your business is renting apartments, you might want to be able to search for ground floor apartments or apartments within a certain neighborhood. In that case, these fields might make sense.)

Here's the new list of fields:

- CustomerId
- Street
- City
- State
- Zip

So much simpler!

Exercise 4 Solution

In this model, the Phones table is fairly unconstrained because it allows a person to have any number of any type of phone number. All the fields are required. Some other validations that you could build into this table include:

FIELD	CONSTRAINT	IMPLEMENTATION
PersonId	Exists	Foreign key match to Persons.PersonId.
Type	Enumerated value	Foreign key match to new PhoneTypes table.
Number	Format	Let the database verify that the value has format ###-###-####.

In the Persons table, every field except MiddleName should be required. The table can implement the following constraints:

FIELD	CONSTRAINT	IMPLEMENTATION
State	Enumerated value	Foreign key match to new States table.
Zip	Format	Let the database verify that the value has format ##### or #####-####.

All of the fields in the Courses and Projects tables should be required, although you might want to allow a blank InstructorId and DaysAndTime so you can create a course before you're ready to schedule it. This table should also have a foreign key constraint requiring that the InstructorId exist in the Instructors table.

The Students and Instructors tables should require all fields. They should also have a foreign key constraint requiring that their PersonId fields have values that exist in the Persons table.

StudentCourses and StudentProjects are linking tables used to implement many-to-many relationships. Their fields should be required and foreign key constraints should verify that their values exist in the corresponding tables.

CourseResults and ProjectResults are also linking tables that implement many-to-many relationships. They should require that all fields and foreign key constraints should verify that their ID values exist in the corresponding tables.

CourseResults and ProjectResults should also use constraints to verify that the Grade fields contain acceptable values. If Grade is numeric, then a check constraint should verify that it is between 0 and 100 (or whatever scale the school uses). If the Grade value includes A+, A, A-, B+, and so forth, then the tables should use foreign key constraints to verify that the Grade exists in a new PossibleGrades table.

Finally, you could check that the Date fields in the CourseResults and ProjectResults tables come after the corresponding student's enrollment date.

CHAPTER 11: DEFINING USER NEEDS AND REQUIREMENTS

Exercise 1 Solution

The following table summarizes the Course entity's fields.

FIELD	REQUIRED?	DATA TYPE	DOMAIN
Title	Yes	String	Any string
Description	Yes	String	Any string
MaximumParticipants	Yes	Integer	Greater than 0

FIELD	REQUIRED?	DATA TYPE	DOMAIN
Price	Yes	Currency	Greater than 0
AnimalType	Yes	String	One of: Cat, Dog, Bird, Bat, and so on
Dates	Yes	Dates	List of dates
Time	Yes	Time	Between 8 a.m. and 11 p.m.
Location	Yes	String	One of: Room 1, Room 2, yard, arena, and so on
Trainer	No	Reference	The Employee teaching the course
Students	No	Reference	Customers table

Because the Dates and Time fields are required, we cannot create a course until it is scheduled.

A more complex validation for new records should verify that there are no other courses scheduled for the same location with overlapping dates and times.

Exercise 2 Solution

The following table summarizes the Employee entity's fields.

FIELD	REQUIRED?	DATA TYPE	DOMAIN
FirstName	Yes	String	Any first name.
LastName	Yes	String	Any last name.
Street	Yes	String	Any street name and number. Not validated.
City	Yes	String	Any city name. Not validated?
State	Yes	String	Foreign key to States table.
Zip	Yes	String	Valid ZIP Code. Not validated?

continues

(continued)

FIELD	REQUIRED?	DATA TYPE	DOMAIN
Email	No	String	Valid email address. If provided, send the customer a monthly email newsletter.
HomePhone	No	String	Valid 10-digit phone number.
CellPhone	No	String	Valid 10-digit phone number.
SocialSecurityNumber	Yes	String	Valid Social Security number.
Specialties	No	String	Zero or more of: Dog, Cat, Horse, Bird, Fish, Snail, and so on.

Exercise 3 Solution

Alicia and the Pampered Pet employees think of work shift assignments as coming in one-week batches. Alicia posts schedules one week at a time.

However, the database might not actually need to create records representing weeks of assignments. Instead, it can track individual work assignments that represent an employee working certain hours on a given day. The user interface and any work assignment reports will gather the assignments for a particular week and display the results in the familiar week-at-a-time format.

That means the Shift entity can be relatively simple:

FIELD	REQUIRED?	DATA TYPE	DOMAIN
Employee	Yes	Reference	Refers to the assignment's employee.
Date	Yes	Date	Valid dates. For new records, verify that the date is on or after today.
StartTime	Yes	Time	6 a.m. or later.
StopTime	Yes	Time	Between StartTime + 1 hour and 11 p.m.

Exercise 4 Solution

The following table summarizes the Customer entity's fields.

FIELD	REQUIRED?	DATA TYPE	DOMAIN
FirstName	Yes	String	Any first name.
LastName	Yes	String	Any last name.
Street	See notes	String	Any street name and number. Not validated.
City	See notes	String	Any city name. Not validated?
State	See notes	String	Foreign key to States table.
Zip	See notes	String	Valid ZIP Code. Not validated?
Email	See notes	String	Valid email address. If provided, send the customer a monthly email newsletter.
HomePhone	See notes	String	Valid 10-digit phone number.
CellPhone	No	String	Valid 10-digit phone number.
Pets	No	String	Pet names, ages, and types.

The system only creates customer records in one of the following circumstances:

- The customer enrolls in a course. In that case, we require either a home or cell phone number so that we can contact the customer in case there's a change in schedule or some other unexpected event occurs (for example, Sveta contracts Capgras syndrome and won't work with Charlie anymore).
- The customer wants to receive postal mailings about sales and courses. In that case, the address information is required.
- The customer wants to receive email about sales and courses. In that case, the email address is required.
- We are shipping items to the customer. In that case, the address information and at least one phone number is required.

Exercise 5 Solution

Like the Shift entity, TimeEntry is simpler than it might appear. Users typically think of timekeeping as a weekly chore, so they tend to think of a week's worth of time entries. However, individually each time entry is quite simple. The timekeeping user interface and any related reports (including printing payroll checks) will gather the assignments for a particular week and display the results appropriately.

The following table summarizes the TimeEntry entity's fields.

FIELD	REQUIRED?	DATA TYPE	DOMAIN
Employee	Yes	Reference	The employee who worked (or at least pretended to work)
Date	Yes	Date	Before now
StartTime	Yes	Time	Before now
StopTime	Yes	Time	After StartTime and before now
PaidDate	No	Date	Before now

The PaidDate field records the date on which the employee's check was printed covering this time entry.

A more complex check for new records should verify that no existing record for this employee has an overlapping date and time.

Exercise 6 Solution

The Vendor entity gives the name of a company that provides Pampered Pet products. (Peter Piper picked a peck of Pampered Pet products.) It includes information about a contact person at the company.

The following table summarizes the Vendor entity's fields.

FIELD	REQUIRED?	DATA TYPE	DOMAIN
CompanyName	Yes	String	Any company name.
ContactFirstName	Yes	String	Any first name.
ContactLastName	Yes	String	Any last name.

FIELD	REQUIRED?	DATA TYPE	DOMAIN
Street	Yes	String	Any street name and number. Not validated.
City	Yes	String	Any city name. Not validated?
State	Yes	String	Foreign key to States table.
Zip	Yes	String	Valid ZIP Code. Not validated?
ContactEmail	No	String	Valid email address.
ContactPhone	Yes	String	Valid 10-digit phone number.
Notes	No	String	Miscellaneous instructions and notes.

CHAPTER 12: BUILDING A DATA MODEL

Exercise 1 Solution

Food items could be treated like any other inventory item, although their expiration dates would probably be much shorter. Some items might not even be counted in inventory if they expire quickly. For example, the database will need an entry for coffee so you can add one to an order, but there's no point trying to update the `QuantityInStock` every time someone makes a new pot.

Exercise 2 Solution

An easy solution would be to add a new `Certifications` attribute to the `EMPLOYEE` class listing the courses that the employee can teach. This would be a foreign key field referring to `COURSE` classes. In the ER model, the `Employee` entity would have a new relationship with the `Course` entity. This would be a moderately hard change but probably doable.

Alternatively, you could create a new `Instructor` subclass that inherits from `Employee`. This would require creating a new class/entity, so it would be harder.

Exercise 3 Solution

Add a new `StoreId` attribute to the `Order` entity. That part wouldn't be too hard. At a minimum, you would also need to add a `Store` entity to look up allowed store IDs. That would be a little harder. If you also want to record real information about each store, such as an `Address` (which would require a link to the `Addresses` table), the change would be a lot harder.

Exercise 4 Solution

You could add a link between the Course entity and the Address entity. This wouldn't be too hard, but it does require a new relationship, which means it would be harder than simply adding a new attribute to the Course entity.

Exercise 5 Solution

It would be easy to store these as Course entities with a Price of \$0. The Pampered Pet could advertise them just like any other course. Probably no one would care if people attended without creating Customer entities.

Exercise 6 Solution

Adding more addresses to an order would make the Order/Address relationship many-to-many. You would need to add an intermediate table to represent the Order/Address pairs and replace the existing one-to-many relationship with two new one-to-many relationships. This would be a fairly difficult change.

Exercise 7 Solution

The easy solution would be to add a Phone attribute to the Order entity. However, Figure 12.9 shows that the design already has a Phone entity associated with the Person entity. Rather than creating a new attribute, it would be slightly more complicated but more flexible to reuse the Phone entity.

Before doing any of this, however, it would be worth asking the customers whether they will ever need to allow multiple phone numbers for an order. After all, they're adding one and there's nothing to stop them from adding another, particularly because the Person entity already allows any number of phone numbers.

Unfortunately, adding multiple phone numbers to the Order entity would create a many-to-many relationship (one order can have many phone numbers and one phone number might be used to place any number of orders, probably by the same customer). To implement this, you would need to make an OrderPhone entity and two new one-to-many relationships. That would be a much harder change than simply adding a new Phone attribute to the Order entity.

Exercise 8 Solution

The obvious solution is to add a new Department attribute to the InventoryItem entity. However, that would create a functional dependency in that entity's attributes. InventoryItem already has a ShelfLocation field that tells where the item is when it is on display in the store. That location is in some department, so adding a new Department attribute would partially duplicate that data and that could lead to inconsistent data. For example, an item could be listed as shelved in the Fish department but its Department field could be set to Reptile.

A better solution would be to make a new Departments entity that maps ShelfLocation values to departments. This requires adding a new table and a new relationship between InventoryItem and Departments, so it would be fairly difficult.

Exercise 9 Solution

This would require a couple of changes. First, you would need to add effective date attributes to the Address entity. You would also need to change the user interface significantly to let the user decide which of a customer’s addresses to use for any given operation. If the program simply uses the address that is in effect when an order is placed, that might be manageable.

Overall, however, this change seems like a lot of trouble and the need is so unclear that I would ask the customers why they wanted to do this and try to talk them out of it if they don’t have a good reason.

Exercise 10 Solution

The discount applied to an order would need to be recorded, so the simplest solution would be to add a new Discount attribute to the Order entity.

More complicated solutions could track types of discounts to ensure consistency. Then, for example, the employee entering an order would enter a coupon or discount code rather than the actual discount percentage, so entering an incorrect discount would be less common. This solution would require creating a new Discounts entity and a relationship between it and the Order entity, so it would be a more complicated solution.

Exercise 11 Solution

Figure A.30 shows one possible solution. It uses the CompetitorRobot entity to implement the Competitor/Robot relationship and it uses the RobotMatch entity to implement the Robot/Match relationship.

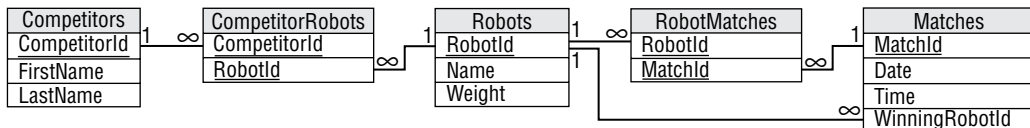


FIGURE A.30

Exercise 12 Solution

Figure A.31 shows one possible relational design.

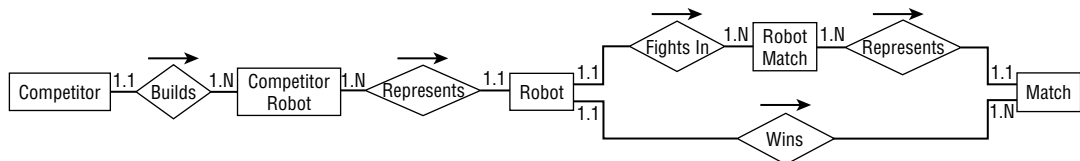


FIGURE A.31

CHAPTER 13: EXTRACTING BUSINESS RULES

Exercise 1 Solution

The following list describes the primary keys and required fields for each table:

- **Competitors**—The `CompetitorId` field is the primary key. Other required fields are `FirstName`, `LastName`, `Street`, `City`, `State`, `Zip`.
- **CompetitorRobots**—This is an intermediate table. Its primary key includes both of its fields. Both of its fields are also foreign key constraints to other tables, so they are completely constrained.
- **Robots**—The `RobotId` field is the primary key. `Name` and `Class` must be required in order to compete. The `Weight`, `MaxSpeed`, and `Chassis` fields could also be required.
- **RobotMatches**—This is an intermediate table. Its primary key includes both of its fields. Both of its fields are also foreign key constraints to other tables, so they are completely constrained.
- **Matches**—The `MatchId` field is the primary key. `Date`, `Time`, and `Location` are also required. `WinningRobotId` cannot be required because the `Matches` record will probably be created before the match occurs and at that time the winner isn't known (unless the fix is in).
- **RobotWeapons**—This table lists the weapons that are built into each robot (chainsaw, axe, grapple, laser cannon). Both of its fields are part of the primary key, which means both are required. The `RobotId` field is a foreign key constraint referring to the `Robots` table, so it is completely constrained.

Exercise 2 Solution

The following list describes sanity checks for each table:

- **Competitors**—`Zip` should have a valid ZIP Code format similar to either 12345 or 12345-6789.
- **CompetitorRobots**—None.
- **Robots**—If present, `Weight > 0` and `Weight < 10,000 lbs`. If present, `Speed >= 0` and `Speed < 30 mph`.
- **RobotMatches**—None.
- **Matches**—When created, `Date >= today`. `Time > 8 a.m.` and `Time < 11 p.m.`
- **RobotWeapons**—None.

Exercise 3 Solution

The following list describes lookup tables for each table's fields:

- **Competitors**—You could build a full `City/State/Zip` lookup table, but it would be big and hard to maintain. You could use the trick described in this chapter of using a table to validate

common City/State/Zip values but allow values not in the table. However, competitors in Robot Wars come from all over the country so there's no good list of the most likely City/State/Zip combinations. Competitors could even come from all over the world so it may not be worth trying to validate particular address formats.

- **CompetitorRobots**—Both of this table's fields are used in foreign key constraints already.
- **Robots**—Chassis should be one of 4 Wheel, 6 Wheel, Tank Tread, Hovercraft, and so forth. Class should be one of Light, Medium, Heavy, Under \$1000, and so forth. The allowed values should be moved into new Chassis and Classes lookup tables.
- **RobotMatches**—Both of this table's fields are used in foreign key constraints already.
- **Matches**—Location should be one of Arena 1, Arena 2, Pond, and so forth. These values should be added to a Locations lookup table.
- **RobotWeapons**—WeaponType should be one of Chainsaw, Axe, Rail Gun, Plasma Cannon, and so forth. Those values should be added to a WeaponTypes table.

Exercise 4 Solution

The three somewhat more complicated business rules that I thought of that really should be implemented in some manner are:

- Two matches should not be scheduled for the same place at the same time. This can be implemented as a uniqueness constraint on the Matches table's combined Date/Time/Location values. (This assumes the matches fit in nice time slots so that we don't need to worry about them overlapping.)
- A robot should not be scheduled for two matches at the same time.
- Because competitors must control their robots during a match, none of a competitor's robots should be scheduled for two matches at the same time. (If two robots share multiple co-owners, the team could split up and be in two matches at once, but that would make the database just plain ugly. If that sort of change is required, you'll be glad you provided this check in a stored procedure, a middle tier, or some other place that's reasonably easy to change.)

Some other possible rules that I thought of include:

- A competitor can have no more than one robot in each match.
- A competitor can have no more than one robot in each class.
- A robot can have no more than two weapons.
- Weight, speed, chassis, and weapons could be part of what determines class. For example, classes could include Heavy, Light & Fast, Wheeled, or Single Weapon. Those definitions would be complicated and would probably change regularly.
- How the matches are assigned could be part of a set of business rules. For example, it could be single elimination, double elimination (if a robot can be repaired), winners and losers brackets, or a giant brawl.

- The contest could be expanded to include robots from all over the world. In that case, you would need to redesign the Competitors table to handle international address formats. You might want to add a second street address field, you'll definitely need to add a County field, and forget about validating postal codes!

Exercise 5 Solution

Figure A.32 shows the new relational model with the lookup tables added.

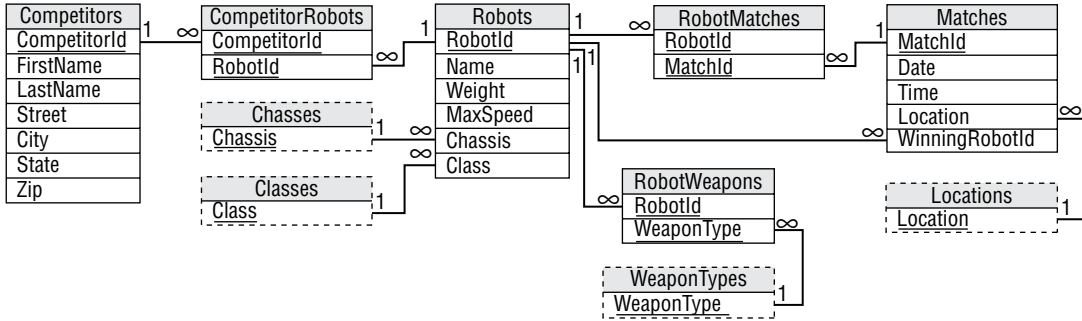


FIGURE A.32

CHAPTER 14: NORMALIZING AND REFINING

Exercise 1 Solution

This table isn't in 1NF because it contains two columns that hold multiple values. The Show column holds the names of all shows at a particular venue and the Times column holds all of the times for shows at a location.

Figure A.33 shows a relational design that stores this data in 1NF.

Shows
ShowName
Time
Venue
Seating

FIGURE A.33

The following table shows the data in this new format.

SHOWNAME	TIME	VENUE	SEATING
Sherm's Shark Show	11:15	Sherman's Lagoon	375

SHOWNAME	TIME	VENUE	SEATING
Sherm's Shark Show	3:00	Sherman's Lagoon	375
Meet the Rays	1:15	Sherman's Lagoon	375
Meet the Rays	6:00	Sherman's Lagoon	375
Deb's Daring Dolphins	11:00	Peet Amphitheater	300
Deb's Daring Dolphins	12:00	Peet Amphitheater	300
Deb's Daring Dolphins	6:30	Peet Amphitheater	300
The Walter Walrus Comedy Hour	2:00	Peet Amphitheater	300
The Walter Walrus Comedy Hour	5:27	Peet Amphitheater	300
Flamingo Follies	2:00	Ngorongoro Wash	413
Wonderful Waterfowl	3:00	Ngorongoro Wash	413

At this point, I'm sure you realize that this table contains so much redundant information that there must be something wrong with it.

Exercise 2 Solution

The solution to Exercise 1 isn't in 2NF because some non-key fields depend on only some (not all) of the primary key fields. A particular show only occurs in one location (it would be hard to move the dolphins to different amphitheaters for different shows), so the Venue and Seating fields depend only on ShowName and not on Time.

The solution is to move the Venue and Seating data into a new table connected to the original table by the ShowName. Because the original table now only holds show time information, I'm going to rename it ShowTimes and call the new table Shows. Figure A.34 shows the result.

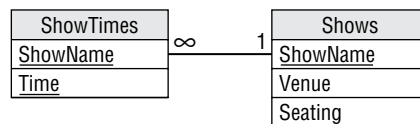


FIGURE A.34

Figure A.35 shows the new tables holding their data.

ShowTimes		Shows		
Sherm's Shark Show	11:15	Sherm's Shark Show	Sherman's Lagoon	375
Sherm's Shark Show	3:00	Meet the Rays	Sherman's Lagoon	375
Meet the Rays	1:15	Deb's Daring Dolphins	Peet Amphitheater	300
Meet the Rays	6:00	The Walter Walrus Comedy Hour	Peet Amphitheater	300
Deb's Daring Dolphins	11:00	Flamingo Follies	Ngorongoro Wash	413
Deb's Daring Dolphins	12:00	Wonderful Waterfowl	Ngorongoro Wash	413
Deb's Daring Dolphins	6:30			
The Walter Walrus Comedy Hour	2:00			
The Walter Walrus Comedy Hour	5:27			
Flamingo Follies	2:00			
Wonderful Waterfowl	3:00			

FIGURE A.35

Exercise 3 Solution

The solution to Exercise 2 isn't in 3NF because the Shows table contains a transitive dependency: the Seating field is determined by the Venue field. In the original table, the dependency isn't obvious because the same Venue and Seating values are not repeated. In Figure A.35 the problem is shown by the repeated Venue/Seating pairs in the Shows table.

The solution is to move the seating information into a new table to match venues with their capacities. The new table should use the Venue field to link back to the Shows table. Because this table describes the venues, I'll call it Venues. (Clever, huh?) Figure A.36 shows the new design.

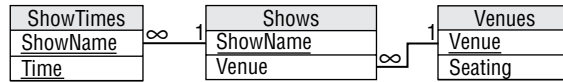


FIGURE A.36

Figure A.37 shows the data in the new tables.

ShowTimes		Shows		Venues	
Sherm's Shark Show	11:15	Sherm's Shark Show	Sherman's Lagoon	Sherman's Lagoon	375
Sherm's Shark Show	3:00	Meet the Rays	Sherman's Lagoon	Peet Amphitheater	300
Meet the Rays	1:15	Deb's Daring Dolphins	Peet Amphitheater	Ngorongoro Wash	413
Meet the Rays	6:00	The Walter Walrus Comedy Hour	Peet Amphitheater		
Deb's Daring Dolphins	11:00	Flamingo Follies	Ngorongoro Wash		
Deb's Daring Dolphins	12:00	Wonderful Waterfowl	Ngorongoro Wash		
Deb's Daring Dolphins	6:30				
The Walter Walrus Comedy Hour	2:00				
The Walter Walrus Comedy Hour	5:27				
Flamingo Follies	2:00				
Wonderful Waterfowl	3:00				

FIGURE A.37

Exercise 4 Solution

Changing show names, time, or venue names is difficult for the design shown in Figure A.36 because those fields are used as primary keys. To increase the database's flexibility, all you need to do is make artificial keys (ID numbers) for the tables. Because ShowName was only in the ShowTimes table to provide a link to the Shows table, it is no longer needed in ShowTimes. Similarly, the Venue field in the Shows table was only there to link to the Venues table, so Venue is no longer needed in the Shows table.

Figure A.38 shows the more flexible design.

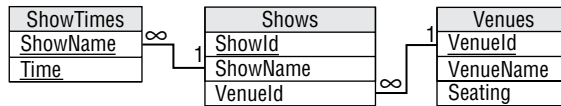


FIGURE A.38

Figure A.39 shows the data in the new tables. The ShowId values are between 1 and 6, and the VenueId values are between 101 and 103, so it's easy to see which are which.

ShowTimes		Shows			Venues		
11:15	1	1	Sherm's Shark Show	101	101	Sherman's Lagoon	375
3:00	1	2	Meet the Rays	101	102	Peet Amphitheatre	300
1:15	2	3	Deb's Daring Dolphins	102	103	Ngorongoro Wash	413
6:00	2	4	The Walter Walrus Comedy Hour	102			
11:00	3	5	Flamingo Follies	103			
12:00	3	6	Wonderful Waterfowl	103			
6:30	3						
2:00	4						
5:27	4						
2:00	5						
3:00	6						

FIGURE A.39

Notice that the tables contain no repeated data other than their ID values so that you can easily change a show's name or time, or a venue's name.

CHAPTER 15: EXAMPLE OVERVIEW

Exercise 1 Solution

If you haven't started Jupyter Notebook, do so now. In Windows, you can search the Start menu for **Jupyter Notebook**.

Create a new Jupyter Notebook, place the following code in a cell, and run the cell:

```
print("Hello World!")
```

The result should look like the following:

```
Hello World!
```

Exercise 2 Solution

If you haven't started Visual Studio, do so now. In Windows, you can search the Start menu for **Visual Studio**.

When Visual Studio starts, click **Create A New Project**. If Visual Studio is already running and not displaying the startup form, open the **File** menu, select the **New** submenu, and choose **Project**.

In the **Create A New Project** dialog box, set the three drop-down lists on the right so that they display **C#**, **Windows**, and **Console**. Select the **Console App (.NET Framework)** project type and click **Next**.

On the next page, enter a project name like **HelloWorld** and set the location to a convenient directory. Select the "Place solution and project in the same directory" option if you like. (I almost always do. It makes the project less cluttered.) Select a reasonably recent framework version and click **Create**.

When the code editor opens, add code to make the `main` method look like the following:

```
static void Main(string[] args)
{
    Console.WriteLine("Hello World!");
    Console.ReadLine();
}
```

The first line displays "Hello World!" and the second waits until you press **Enter** so that the console window doesn't immediately disappear.

Exercise 3 Solution

A database adapter is a library that provides a bridge between your program and the database. Your code invokes methods defined by the adapter, and those methods access the database.

Exercise 4 Solution

Literate programming is a style where code is interspersed with documentation and program output such as text, graphs, pictures, and other artifacts. Jupyter Notebook uses a literate programming model, and you can experiment with it if you like.

For example, create a new cell and click it. Then open the drop-down near the right end of the toolbar (it probably says **Code** initially) and select **Markdown**. Then enter the following text into the cell and run it:

```
# Level 1
## Level 2
### Level 3
```

Exercise 5 Solution

You can use pip to install packages in Python, as in the following statement in a command window:

```
$ pip install pyignite
```

You can also use pip to install packages in Jupyter Notebook, as in the following statement in a notebook cell:

```
!pip install pyignite
```

Exercise 6 Solution

You can use the NuGet package manager to install packages in Visual Studio. Open the Project menu and select Manage NuGet Packages to launch the manager.

Select the Browse tab and search for the package that you want. Click the package you want and then click Install on the right to install it.

CHAPTER 16: MARIADB IN PYTHON

Exercise 1 Solution

Michael Widenius's children are My (MySQL), Max (MaxDB), and Maria (MariaDB).

Exercise 2 Solution

This example briefly demonstrates the HeidiSQL database management tool.

Exercise 3 Solution

The main advantage to using a database management tool is that you don't need to know SQL. Of course, you need to know how to use the tool, but it's generally easier to wander around an application looking at menus to figure out how to achieve your goals than it is to suddenly remember what SQL command you need to use. Many database management tools also display the SQL code that they are using to build the database, so you can use that feature to learn a bit more about SQL.

Exercise 4 Solution

This main disadvantage to using a database management tool is that it probably cannot easily reapply all the steps that you've taken to build the database. If you later decide that you made a mistake early on, then you might need to modify or rebuild much of the database. In contrast, if you used a program to build the database, then you can probably make a few changes to the code and then rerun the program.

A second disadvantage is that this kind of tool tends to use only the SQL commands that are standard and supported by most databases. That means you cannot take advantage of any special features that the database provides. (Of course, using special features can also be a problem if you later need to switch database engines, so maybe using only standard features is an advantage.)

Exercise 5 Solution

The following code shows my solution:

```
# Let the user enter AnimalPlanets records.
import pymysql

# Connect to the database server.
conn = pymysql.connect(
    host="localhost",
    user="root",
    password="TheSecretPassword",
    database="AnimalData")

# Create a cursor.
cur = conn.cursor()

running = True
while True:
    print()
    try:
        # Get the animal ID and planet ID.
        animal_id = input("Animal ID: ")
        if animal_id == "":
            break
        animal_id = int(animal_id)
    except ValueError:
        print("Animal ID must be an integer")
        continue

    try:
        planet_id = input("Planet ID: ")
        if planet_id == "":
            break
        planet_id = int(planet_id)
    except ValueError:
        print("Planet ID must be an integer")
        continue

    try:
        # Add the new record.
        cmd = """INSERT INTO AnimalPlanets
            (AnimalId, PlanetId) VALUES """
        cmd = cmd + f"({int(animal_id)}, {int(planet_id)})"
        print(cmd)
        cur.execute(cmd)
        print("Record added")
    except Exception as e:
        print("SQL error: " + str(e))
        continue

# Commit the changes.
conn.commit()

# Close the connection.
conn.close()
```

Exercise 6 Solution

If you try to use an animal ID that is not in the Animals table (such as 100) or a planet ID that is not in the Planets table (such as -1), then the new record violates the AnimalPlanet's foreign key constraint and the database throws a tantrum.

If you try to duplicate a record that is already in the table, then the new record violates the requirement that the table's primary key values be unique and the database again throws a tantrum.

If you enter a number that is too big to fit in the INT data type used by the AnimalId and PlanetId fields, then the database raises an "Out of range value" error. (The largest value that will fit is 2,147,483,647.)

CHAPTER 17: MARIADB IN C#

Exercise 1 Solution

The `ExecuteNonQuery` method executes statements that do not return records. The `ExecuteReader` method executes a query and returns a `MySqlDataReader` object that lets you loop through the returned results.

Exercise 2 Solution

The statement `DROP TABLE IF EXISTS AnimalPlanets` will drop the table if it exists and do nothing if it does not exist.

Exercise 3 Solution

This main advantage to using a database management tool is that you don't need to know SQL. Of course, then you need to know how to use the tool, but it's generally easier to wander around an application looking at menus to figure out how to achieve your goals than it is to suddenly remember what SQL command you need to use. Many database management tools also display the SQL code that they are using to build the database, so you can use that feature to learn a bit more about SQL.

Exercise 4 Solution

This main disadvantage to using a database management tool is that it probably cannot easily reapply all the steps that you've taken to build the database. If you later decide that you made a mistake early on, then you might need to modify or rebuild much of the database. In contrast, if you used a program to build the database, then you can probably make a few changes to the code and rerun the program.

A second disadvantage is that this kind of tool tends to use only the SQL commands that are standard and supported by most databases. This means you cannot take advantage of any special features that the database provides. (Of course, using special features can also be a problem if you later need to switch database engines, so maybe using only standard features is an advantage.)


```

        $"({animalId}, {planetId})";
        Console.WriteLine(cmd.CommandText);
        cmd.ExecuteNonQuery();
        Console.WriteLine("Record added");
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
        continue;
    }
} // End for ( ; ; )
} // End using MySqlCommand cmd
} // End using MySqlConnection conn
}

```

Exercise 6 Solution

If you enter a number that is too big to fit in the `int` data type, the `int.Parse` method throws an `OverflowException`. You can avoid that by using `int.TryParse` to see if the value is either not an integer or too big.

If you try to use an animal ID that is not in the `Animals` table (such as 100) or a planet ID that is not in the `Planets` table (such as -1), then the new record violates the `AnimalPlanet`'s foreign key constraint and the database throws a tantrum.

If you try to duplicate a record that is already in the table, then the new record violates the requirement that the table's primary key values be unique and the database again throws a tantrum.

CHAPTER 18: POSTGRESQL IN PYTHON

Exercise 1 Solution

The database adapter is `Psycpg`. It should have been spelled `Psychpg`.

Exercise 2 Solution

The following code shows my solution:

```

# Exercise 18.1
# Prepare to use psycpg2.
import psycpg2

# Get the customer ID.
customer_id = input("Customer ID: ")

# Get the order date.
order_date = input("Order Date (m/d/yyyy): ")

# Get order items.
items = []

```

```
while True:
    item_name = input("    Item Name: ")
    if item_name == "":
        break
    quantity = input("    Quantity: ")
    price_each = input("    Price Each: ")

    # Add the new item to the list.
    items.append((item_name, quantity, price_each))

# Connect to the database.
conn = psycopg2.connect(
    database="BrewCrewDB",
    user = "brew_master",
    password = "brew_password",
    host = "127.0.0.1", port = "5432")
cursor = conn.cursor()

# Create the order.
cmd = "INSERT INTO orders (customer_id, date) "
cmd = cmd + f"VALUES ({customer_id}, '{order_date}')"
cmd = cmd + "RETURNING order_id"
print(cmd)
cursor.execute(cmd)
order_id = cursor.fetchone()[0]
print(f"Order ID: {order_id}\n")

# Create the order items.
item_number = 1
for item in items:
    cmd = "INSERT INTO order_items "
    cmd = cmd + "(order_id, item_number, item_name, quantity, price_each) "
    cmd = cmd + f"VALUES ({order_id}, {item_number}, "
    cmd = cmd + f"'{item[0]}', {item[1]}, {item[2]})"
    item_number = item_number + 1
    print(cmd)
    cursor.execute(cmd)

# Commit the insertions.
conn.commit()

# Close the cursor and connection.
cursor.close()
conn.close()
```

Exercise 3 Solution

This is no problem. Jupyter Notebook and PostgreSQL can handle Unicode characters so the program can save “crème brûlée” in the database.

Exercise 4 Solution

This is a problem. The following text shows the SQL statement that creates the `order_items` record:

```
INSERT INTO order_items
  (order_id, item_number, item_name, quantity, price_each)
VALUES
  (505, 1, 'Amy's cookies', 12, 1)
```

The apostrophe in `Amy's cookies` matches with the initial apostrophe that starts the item name string, so the database thinks that string has ended. Then the remaining text, `s cookies'`, confuses it and you get the following error:

```
SyntaxError: syntax error at or near "s"
LINE 1: ..._name, quantity, price_each) VALUES (505, 1, 'Amy's cookies'...
```

Exercise 5 Solution

The following code shows the way I rewrote the item insertion loop:

```
# Create the order items.
item_number = 1
for item in items:
    cmd = "INSERT INTO order_items "
    cmd = cmd + "(order_id, item_number, item_name, quantity, price_each) "
    cmd = cmd + "VALUES (%s, %s, %s, %s, %s)"
    item_number = item_number + 1
    print(cmd)
    values = (order_id, item_number) + item
    cursor.execute(cmd, values)
```

The second-to-last line creates the tuple holding all of the record's values. The last line passes the `INSERT` statement and that tuple to the cursor's `execute` method.

Exercise 6 Solution

By default, the database assumes that dates use the `m/d/y` format in SQL statements. If you enter a date in `d/m/y` format, it switches the month and date.

If you enter `20/10/2027`, the program crashes with the following error because it assumes you mean the 10th day of the 20th month:

```
DatetimeFieldOverflow: date/time field value out of range: "20/10/2027"
LINE 1: ...SERT INTO orders (customer_id, date) VALUES (193, '20/10/202...
```

HINT: Perhaps you need a different `"datestyle"` setting.

There are various ways that you can make PostgreSQL use a different date format, such as setting `datestyle` in the `postgresql.conf` file or executing the following command in the PostgreSQL console:

```
set datestyle to European
```

In any case, it may be best to include the format in the date prompt as in “Date (m/d/yyyy):” so that the user knows what format to use.

Exercise 7 Solution

If the customer ID that you enter is not in the customers table, then creating a new `orders` record will violate that table’s foreign key constraint.

The program creates a new order ID and then uses it to create `order_items` records, so you cannot accidentally use an order ID that violates that table’s foreign key constraints.

However, you can generate some “out of range” errors even if you type valid values. For example, if you enter a date that has a really big year, such as 1/2/100000, then the database can hold the value but you get a `ValueError` when you try to fetch the data. The database can handle this value, but Python cannot.

If you enter a *really* big year, then even the database cannot hold it and will give you a `DatetimeFieldOverflow` error when you try to create the `orders` record.

You can also run afoul of `NumericValueOutOfRange` errors if you try to enter extremely large quantities or prices.

None of those are formatting errors. For example, 1/2/1000000 is a valid date and 1e1000000 is a valid price, at least theoretically. The moral is that every program needs error handling because users may enter invalid values even if they have the correct format.

CHAPTER 19: POSTGRESQL IN C#

Exercise 1 Solution

The `CreateCustomerRecords` and `CreateOrderRecords` methods call `ExecuteScalar` because their SQL statements include a `RETURNING` clause, which returns the newly created records’ customer or order IDs. In contrast, the `CreateOrderItemRecords` method’s SQL statement does not create an ID and does not use a `RETURNING` clause, so there’s no scalar value for `ExecuteScalar` to return.

Exercise 2 Solution

The following code shows my solution:

```
// Exercise 19.2
// Let the user create an order.
static private void LetUserCreateOrder(NpgsqlConnection conn)
{
    // Get the customer ID.
    Console.Write("Customer ID: ");
    string customerId = Console.ReadLine();

    // Get the order date.
    Console.Write("Order Date (m/d/yyyy): ");
```

```

string orderDate = Console.ReadLine();

// Get order items.
List<string[]> items = new List<string[]>();
for (; ;)
{
    Console.Write("    Item Name: ");
    string itemName = Console.ReadLine();
    if (itemName.Length == 0) break;

    Console.Write("    Quantity: ");
    string quantity = Console.ReadLine();

    Console.Write("    Price Each: ");
    string priceEach = Console.ReadLine();

    // Add the new item to the list.
    items.Add(new string[] { itemName, quantity, priceEach });
}

using (NpgsqlTransaction transaction = conn.BeginTransaction())
{
    // Create the order.
    string createOrderStatement =
        "INSERT INTO orders (customer_id, date) " +
        $"VALUES ({customerId}, '{orderDate}')" +
        "RETURNING order_id";
    Console.WriteLine(createOrderStatement);
    int orderId;
    using (NpgsqlCommand cmd =
        new NpgsqlCommand(createOrderStatement, conn))
    {
        cmd.Transaction = transaction;

        orderId = (int)cmd.ExecuteScalar();
        Console.WriteLine($"Order ID: {orderId}\n");
    }

    // Create the order items.
    string createItemStatement =
        @"INSERT INTO order_items
        (order_id, item_number, item_name, quantity, price_each)
VALUES
        (@order_id, @item_number, @item_name, @quantity, @price_each)";
    Console.WriteLine(createItemStatement);
    Console.WriteLine();

    using (NpgsqlCommand cmd =
        new NpgsqlCommand(createItemStatement, conn))
    {
        cmd.Transaction = transaction;

        // Add the parameters.
        NpgsqlParameter orderIdParameter =
            cmd.Parameters.Add("order_id",
                NpgsqlTypes.NpgsqlDbType.Integer);
        NpgsqlParameter itemNumberParameter =

```

```
        cmd.Parameters.Add("item_number",
            NpgsqlTypes.NpgsqlDbType.Integer);
        NpgsqlParameter itemNameParameter =
            cmd.Parameters.Add("item_name",
                NpgsqlTypes.NpgsqlDbType.Text);
        NpgsqlParameter quantityParameter =
            cmd.Parameters.Add("quantity",
                NpgsqlTypes.NpgsqlDbType.Integer);
        NpgsqlParameter priceEachParameter =
            cmd.Parameters.Add("price_each",
                NpgsqlTypes.NpgsqlDbType.Numeric);

        // Prepare the command.
        cmd.Prepare();

        // Create the items.
        int itemNumber = 0;
        foreach (string[] item in items)
        {
            itemNumber++;

            orderIdParameter.Value = orderId;
            itemNumberParameter.Value = itemNumber;
            itemNameParameter.Value = item[0];
            quantityParameter.Value = int.Parse(item[1]);
            priceEachParameter.Value = float.Parse(item[2]);

            cmd.ExecuteNonQuery();
        } // End foreach (string[] item in items)
    } // End using cmd
    transaction.Commit();
} // End using transaction
}
```

Exercise 3 Solution

Figure A.40 shows pgAdmin showing the data in the order_items table. Notice that there are no items named Aardvark.

My results for the orders table did not show any orders with date 1/2/2300.

Exercise 4 Solution

This is no problem. C# and PostgreSQL can handle Unicode characters so the program can save “crème brûlée” in the database.

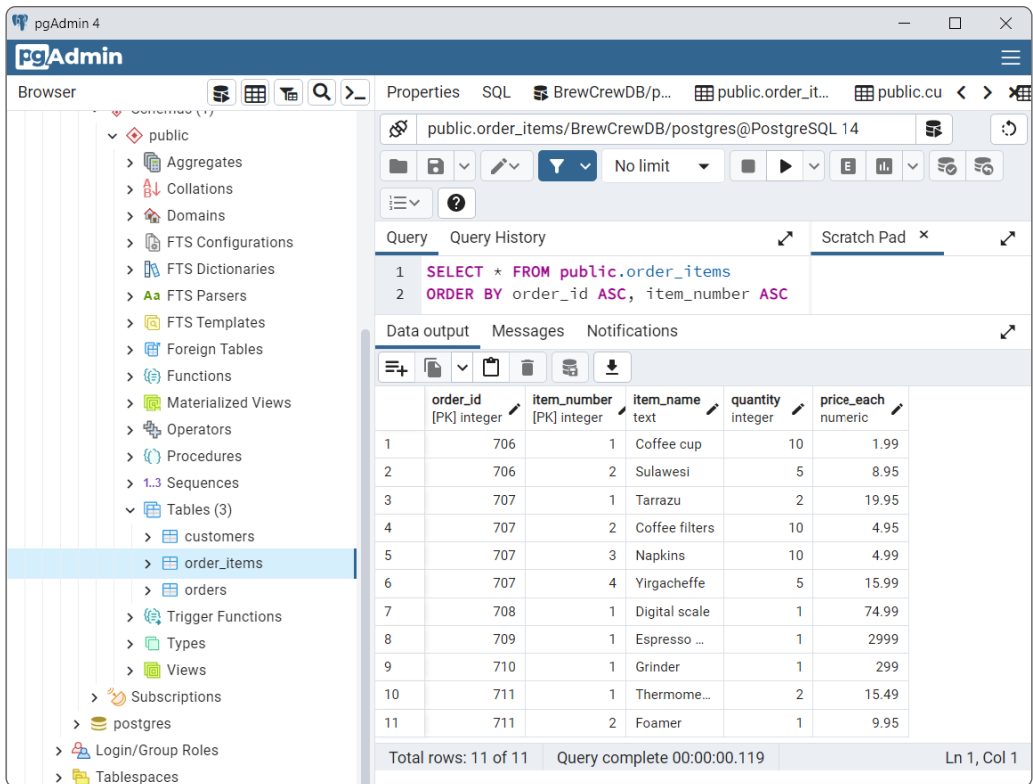


FIGURE A.40

Exercise 5 Solution

If you compose the SQL `INSERT` statements so that they include the record's values, then this is a problem. The following shows the SQL statement that creates the `order_items` record:

```
INSERT INTO order_items
  (order_id, item_number, item_name, quantity, price_each)
VALUES
  (505, 1, 'Amy's cookies', 12, 1)
```

The apostrophe in “Amy’s cookies” matches with the initial apostrophe that starts the item name string so the database thinks that string has ended. Then the remaining text, `s cookies'`, confuses it and you get the following error:

```
Npgsql.PostgresException: '42601: syntax error at or near "s"
```

However, if you use SQL `INSERT` statements that use placeholders and parameters, then this is no problem.

Exercise 6 Solution

By default, the database assumes that dates use the m/d/y format in SQL statements. If you enter a date in d/m/y format, it switches the month and date.

If you enter 20/10/2027, the program crashes with the following error because it assumes you mean the 10th day of the 20th month:

```
Npgsql.PostgresException: '22008: date/time field value out of range:
"20/10/2300"
```

There are various ways that you can make PostgreSQL use a different date format such as setting `datestyle` in the `postgresql.conf` file or executing the following command in the PostgreSQL console:

```
set datestyle to European
```

In any case, it may be best to include the format in the date prompt as in “Date (m/d/yyyy)” so that the user knows what format to use.

Exercise 7 Solution

If the customer ID that you enter is not in the customers table, then creating a new `orders` record will violate that table’s foreign key constraint.

The program creates a new order ID and then uses it to create `order_items` records, so you cannot accidentally use an order ID that violates that table’s foreign key constraints.

However, you can generate some “out of range” errors even if you type valid values. For example, if you enter a date that has a really big year, such as 1/2/100000, then the database can hold the value but you get a `ValueError` when you try to fetch the data. The database can handle this value, but C# cannot.

If you enter a *really* big year, then even the database cannot hold it and will throw an exception similar to the following when you try to create the `orders` record:

```
Npgsql.PostgresException: '22008: date out of range: "1/2/1234567890"
```

You can also run afoul of `NumericValueOutOfRangeException` errors if you try to enter extremely large quantities or prices.

None of those are formatting errors. For example, 1/2/1000000 is a valid date and 1e1000000 is a valid price, at least theoretically, although you might also have trouble getting C# to work with such a large value. The moral is that every program needs error handling because users might enter invalid values even if they have the correct format.

CHAPTER 20: NEO4J AURADB IN PYTHON

Exercise 1 Solution

AuraDB relationships are directional, so if you wanted two players A and B to be related by the same relationship, then you would need to create both of the relationships `A IS_TEAMMATES_WITH B` and `B IS_TEAMMATES_WITH A`.

Exercise 2 Solution

You can use Figure 20.4 to check your results.

- a. That statement would return the people to whom person H reports either directly or indirectly. In this example, that would be people G, C, and A.
- b. That statement would return the people who report directly to person C. In this example, that would be people F and G.

Exercise 3 Solution

Yes. The path between any two nodes in a tree is unique. To get from node A to node B, you move up the tree above node A until you reach the lowest common ancestor of the two nodes. Then you follow the path down from there to node B.

Exercise 4 Solution

This statement returns nodes that are two levels above person I in the org chart—in other words, I’s boss’s boss (or grandboss, if you will).

Exercise 5 Solution

The first statement starts at an arbitrary node and looks for a path *to* node C.

The second statement starts at node C and looks for a path *from* some other node.

The results are the same, so you can use whichever one seems more intuitive to you.

CHAPTER 21: NEO4J AURADB IN C#

Exercise 1 Solution

That statement moves up one level in the tree and then back down one level, so it returns the siblings of node H.

Exercise 2 Solution

For the org chart shown in Figure 21.2, the statement returns node D: Dir Puns and Knock-Knock Jokes.

Exercise 3 Solution

This query returns all pairs of nodes that are siblings.

Exercise 4 Solution

For the org chart shown in Figure 21.2, the statement returns the following pairs of nodes:

- B, C
- C, B
- D, E
- E, D
- F, G
- G, F
- H, I
- I, H

Exercise 5 Solution

That `MATCH` statement returns the names of `REPORTS_TO` relationships that are in a chain three relationships long.

Exercise 6 Solution

For the org chart shown in Figure 21.2, the statement returns the following chains of relationships:

- I->G, G->C, C->A
- H->G, G->C, C->A

Exercise 7 Solution

The following `MATCH` statement returns node A's grandchildren:

```
MATCH
  (:OrgNode { ID: 'A' }) <- [:REPORTS_TO] - (:OrgNode) <- [:REPORTS_TO] - (n:OrgNode)
return n
```


CHAPTER 22: MONGODB ATLAS IN PYTHON

Exercise 1 Solution

The following `find` statement selects documents for people who are assigned to the `Scrat` and who are not `Cook's Mates`:

```
cursor = db.postings.find(
    {"$and": [
        {"Ship": "Scrat"},
        {"Position": {"$ne": "Cook's Mate"}}},
    ]})
```

Exercise 2 Solution

- The old documents do not have a `MiddleInitial` value, so the program cannot try to access `doc["MiddleInitial"]` for the old documents or it will crash.
- The program can use `if "MiddleInitial" in doc` to see if the document has a `MiddleInitial` value.
- The following code saves the `MiddleInitial` value or a space in variable `initial`:

```
# MiddleInitial may be missing.
if 'MiddleInitial' in doc:
    initial = doc["MiddleInitial"]
else:
    initial = ' '
```

Exercise 3 Solution

The following code selects documents for `Frieda's Glory` and returns only the `FirstName` and `Position` values:

```
cursor = db.postings.find(
    {"Ship": "Frieda's Glory"},
    {"_id": False, "FirstName": True, "Position": True})
for doc in cursor:
    print(doc)
```

The second dictionary excludes the `_id` field. It specifically includes the `FirstName` and `Position` fields, so all of the other fields are excluded.

Here's the result:

```
{'FirstName': 'Joshua', 'Position': 'Fuse Tender'}
{'FirstName': 'Al', 'Position': ['Diplomat', 'Interpreter']}
```

The program displays the result in the way Python prints dictionaries. Each result is surrounded by curly braces and contains a sequence of key/value pairs giving the selected field names and values. Because these are dictionaries, you could access a specific value as in `doc["Ship"]`.

Exercise 4 Solution

The following code finds and displays documents for pilots and captains:

```
cursor = db.postings.find(
    {"Position": {"$in": ["Pilot", "Captain"]}})
for doc in cursor:
    print(person_string(doc))
```

CHAPTER 23: MONGODB ATLAS IN C#

Exercise 1 Solution

The following LINQ query selects documents for people who are assigned to the Scrat and who are not Cook's Mates:

```
var scratNonCooks =
    from e in collection.AsQueryable<BsonDocument>()
    where e["Ship"] == "Scrat"
        && e["Position"] != "Cook's Mate"
    select e;
```

In the test data, the only document that matches is for Sally Barker.

Exercise 2 Solution

- The old documents do not have a `MiddleInitial` value, so the program cannot try to access `doc["MiddleInitial"]` for the old documents or it will crash.
- The program can use `doc.Contains("MiddleInitial")` to see if the document has a `MiddleInitial` value.
- The following code saves the `MiddleInitial` value or a space in variable `initial`:

```
string initial = "";
if (doc.Contains("MiddleInitial"))
    initial = doc["MiddleInitial"].AsString;
else
    initial = " ";
```

Exercise 3 Solution

The following code selects and displays documents representing people posted to the Scrat:

```
var scrats = collection.Find(doc => doc["Ship"] == "Scrat").ToList();
foreach (BsonDocument doc in scrats)
{
    Console.WriteLine(PersonString(doc));
}
```

Exercise 4 Solution

The following code deletes the documents that have no Rank value:

```
collection.DeleteMany(doc => doc["Rank"] == BsonNull.Value);
```

CHAPTER 24: APACHE IGNITE IN PYTHON

Exercise 1 Solution

This is almost too easy. Your program would execute the following code to set the message of the day:

```
misc_data_cache.put('motd',
    'Seek success but prepare for vegetables')
```

The users' program would execute the following code to get and display the current message:

```
message = misc_data_cache.get('motd')
print(f'Message of the Day: {message}')
```

The following shows the output:

```
Message of the Day: Seek success but prepare for vegetables.
```

Exercise 2 Solution

Again, this is fairly easy. The following code stores the lists of values:

```
misc_data_cache.put('string_array', ['apple', 'banana', 'cherry'])
misc_data_cache.put('int_array', [1, 1, 2, 3, 5, 8, 13])
misc_data_cache.put('float_array', [3.14159, 2.71828, 1.61803, 6.0221515e23])
```

The following code retrieves and prints the lists:

```
string_array = misc_data_cache.get('string_array')
print(string_array)

int_array = misc_data_cache.get('int_array')
print(int_array)

float_array = misc_data_cache.get('float_array')
print(float_array)
```

The following shows the output:

```
string_array: ['apple', 'banana', 'cherry']
int_array:    [1, 1, 2, 3, 5, 8, 13]
float_array:  [3.14159, 2.71828, 1.61803, 6.0221515e+23]
```

Exercise 3 Solution

Yes, you can use an array as a key.

The following code saves two values that use arrays as keys:

```
misc_data_cache.put([1, 2, 3, 4, 5], 'Counting numbers')
misc_data_cache.put(
    ['Pygmy hog', 'Tasmanian devil'],
    ['Himalayas', 'Tasmania'])
```

The first statement uses the list [1, 2, 3, 4, 5] as the key for the string “Counting numbers.” The second uses a list containing two animal names as the key for another list that indicates where they live.

The following code reads and displays those values:

```
numbers = misc_data_cache.get([1, 2, 3, 4, 5])
print(numbers)

habitats = misc_data_cache.get(['Pygmy hog', 'Tasmanian devil'])
print(habitats)
```

The following shows the output:

```
Counting numbers
['Himalayas', 'Tasmania']
```

It’s hard to think of a good example where you might want to do that, however.

Exercise 4 Solution

- a. The following code adds the phone numbers to the database individually:

```
misc_data_cache.put('Glenn', 'Pennsylvania 6-5000')
misc_data_cache.put('Jenny', '867-5309')
misc_data_cache.put('John', '36 24 36')
misc_data_cache.put('Marvin', 'Beechwood 4-5789')
misc_data_cache.put('Tina', '6060-842')
```

- b. Either of the following two statements retrieves the phone number dictionary from the database. The third statement prints the dictionary.

```
phones = misc_data_cache.get_all(['Glenn', 'Jenny', 'John', 'Marvin', 'Tina'])
phones = misc_data_cache.get_all(('Glenn', 'Jenny', 'John', 'Marvin', 'Tina'))
print(phones)
```

The following shows the output:

```
{'John': '36 24 36', 'Glenn': 'Pennsylvania 6-5000', 'Jenny': '867-5309',
'Marvin': 'Beechwood 4-5789', 'Tina': '6060-842'}
```

The following code shows one way to nicely format the dictionary:

```
import json
print(json.dumps(phones, indent=4))
```

The following shows the formatted output:

```
{
  "John": "36 24 36",
  "Glenn": "Pennsylvania 6-5000",
  "Jenny": "867-5309",
  "Marvin": "Beechwood 4-5789",
  "Tina": "6060-842"
}
```

c. The following code prints Glenn’s and Jenny’s phone numbers:

```
print(f'Glenn: {phones["Glenn"]}')
print(f'Jenny: {phones["Jenny"]}')
```

CHAPTER 25: APACHE IGNITE IN C#

Exercise 1 Solution

The following extension method returns a value or the string “missing.”

```
public static class Extensions
{
    // Get the value for this key.
    // If the key is not present, return "missing."
    public static string GetString<T1, T2>(
        this ICacheClient<T1, T2> cache, T1 key)
    {
        if (cache.ContainsKey(key))
            return cache.Get(key).ToString();
        return "missing";
    }
}
```

The only real trick here is that the `ICacheClient` interface takes two generic type parameters, so the method should also. The code calls `ContainsKey` and, if the key is present, it fetches the corresponding value, calls its `ToString` method, and returns the result. If the key is missing, the method returns “missing.”

Now the code can call the method, as shown in the following code:

```
// Fetch some data.
Console.WriteLine(cache.GetString(100));
Console.WriteLine(cache.GetString("fish"));
Console.WriteLine(cache.GetString(3.14));
```

In a real program, it might be better to return the value's type (in this case T2) and make it nullable so that you don't need to work only with strings.

Exercise 2 Solution

This is almost too easy. Your program would execute the following code to set the message of the day:

```
cache.put('motd', 'Seek success but prepare for vegetables')
```

The users' program would execute the following code to get and display the current message:

```
Console.WriteLine($"Message of the Day: {cache.GetString("motd")}");
```

The following shows the output:

```
Message of The Day: Seek success but prepare for vegetables.
```

Exercise 3 Solution

Again, this is fairly easy. The following code stores the arrays of values:

```
cache.Put("string_array",
    new string[]
    {
        "apple",
        "banana",
        "cherry"
    });
cache.Put("int_array", new int[] { 1, 2, 3, 4, 5 });
cache.Put("double_array",
    new double[] { 3.14159, 2.71828, 1.61803, 6.0221515e23 });
```

The following code retrieves and prints the arrays:

```
string[] strings = (string[])cache.Get("string_array");
Console.WriteLine("{ " + string.Join(", ", strings) + " }");

int[] ints = (int[])cache.Get("int_array");
Console.WriteLine("{ " + string.Join(", ", ints) + " }");

double[] doubles = (double[])cache.Get("double_array");
Console.WriteLine("{ " + string.Join(", ", doubles) + " }");
```

Exercise 4 Solution

Yes, you can use an array as a key.

The following code saves two values that use arrays as keys:

```
cache.Put(new int[] { 1, 2, 3, 4, 5 }, "Counting numbers");
cache.Put(
    new string[] { "Pygmy hog", "Tasmanian devil" },
    new string[] { "Himalayas", "Tasmania" });
```

The first statement uses { 1, 2, 3, 4, 5 } as the key for the string “Counting numbers.” The second uses a list containing two animal names as the key for another list that indicates where they live.

The following code reads and displays those values:

```
Console.WriteLine(cache.Get(
    new int[] { 1, 2, 3, 4, 5 }));

string[] habitats = (string[])cache.Get(
    new string[] { "Pygmy hog", "Tasmanian devil" });
Console.WriteLine("{ " + string.Join(", ", habitats) + " }");
Console.WriteLine();
```

The following shows the output:

```
Counting numbers
{ Himalayas, Tasmania }
```

It’s hard to think of a good example where you might want to do that, however.

CHAPTER 26: INTRODUCTION TO SQL

Exercise 1 Solution

The following code creates the Venues, Shows, and ShowTimes tables. Note that you must create the tables in this order because you cannot create a foreign key constraint that refers to a table that doesn’t yet exist.

```
CREATE TABLE Venues (
    VenueId    INT           NOT NULL,
    VenueName  VARCHAR(45)   NOT NULL,
    Seating    INT           NOT NULL,
    PRIMARY KEY (VenueId)
);

CREATE TABLE Shows (
    ShowId     INT           NOT NULL,
    ShowName   VARCHAR(45)   NOT NULL,
    VenueId    INT           NOT NULL,
    PRIMARY KEY (ShowId),
    INDEX fk_Shows_Venues (VenueId),
    CONSTRAINT fk_Shows_Venues
        FOREIGN KEY (VenueId)
        REFERENCES Venues (VenueId)
        ON DELETE NO ACTION
        ON UPDATE NO ACTION
```

```
);  
  
CREATE TABLE ShowTimes(  
  ShowId      INT           NOT NULL,  
  Time        TIME         NOT NULL,  
  PRIMARY KEY (ShowId, Time),  
  INDEX fk_ShowTimes_Shows (ShowId),  
  CONSTRAINT fk_ShowTimes_Shows  
    FOREIGN KEY (ShowId)  
    REFERENCES Shows (ShowId)  
    ON DELETE NO ACTION  
    ON UPDATE NO ACTION  
);
```

Exercise 2 Solution

The following code inserts the values for the aquarium show schedule. Note that the statements must insert data in tables used as foreign key constraints before inserting the values that refer to them. For example, the statement that creates the Sherman's Lagoon record in the Venues table must come before the Sherm's Shark Show record that refers to it.

Also note that some of the text contains an apostrophe so that text is delimited by double quotes instead of single quotes. For example, the text "Sherman's Lagoon" contains an apostrophe. Alternatively you could double-up the apostrophes to indicate characters that are part of the text value as in 'Sherman' 's Lagoon' where ' ' are two apostrophes, not a double quote. Double apostrophes can also look like a single double quote and could lead to errors and is just plain confusing. Let's just agree to not go there if at all possible. Now if your text must contain both single and double quotes, some degree of ugliness is likely to ensue.

```
INSERT INTO Venues VALUES (101, "Sherman's Lagoon", 375);  
INSERT INTO Venues VALUES (102, "Peet Amphitheater", 300);  
INSERT INTO Venues VALUES (103, "Ngorongoro Wash", 413);  
  
INSERT INTO Shows VALUES (1, "Sherm's Shark Show", 101);  
INSERT INTO Shows VALUES (2, "Meet the Rays", 101);  
INSERT INTO Shows VALUES (3, "Deb's Daring Dolphins", 102);  
INSERT INTO Shows VALUES (4, "The Walter Walrus Comedy Hour", 102);  
INSERT INTO Shows VALUES (5, "Flamingo Follies", 103);  
INSERT INTO Shows VALUES (6, "Wonderful Waterfowl", 103);  
  
INSERT INTO ShowTimes VALUES (1, "11:15");  
INSERT INTO ShowTimes VALUES (1, "15:00");  
INSERT INTO ShowTimes VALUES (2, "13:15");  
INSERT INTO ShowTimes VALUES (2, "18:00");  
INSERT INTO ShowTimes VALUES (3, "11:00");  
INSERT INTO ShowTimes VALUES (3, "12:00");  
INSERT INTO ShowTimes VALUES (3, "18:30");  
INSERT INTO ShowTimes VALUES (4, "14:00");  
INSERT INTO ShowTimes VALUES (4, "17:27");  
INSERT INTO ShowTimes VALUES (5, "14:00");  
INSERT INTO ShowTimes VALUES (6, "15:00");
```


Exercise 3 Solution

The following statement updates the Flamingo Follies time:

```
UPDATE Shows, ShowTimes SET Time = "14:15"
WHERE Shows.ShowId = ShowTimes.ShowId
AND Time= "14:00"
AND ShowName = "Flamingo Follies";
```

The following statement updates the Sherm's Shark Show time:

```
UPDATE Shows, ShowTimes SET Time = "15:15"
WHERE Shows.ShowId = ShowTimes.ShowId
AND Time= "15:00"
AND ShowName = "Sherm's Shark Show";
```

Exercise 4 Solution

The following code produces the desired result in MySQL:

```
SELECT
    ShowName AS "Show",
    LPAD(DATE_FORMAT(Time, "%l:%i %p"), 8, " ") AS Time,
    VenueName AS Location
FROM Shows, ShowTimes, Venues
WHERE Shows.ShowId = ShowTimes.ShowId
AND Shows.VenueId = Venues.VenueId
ORDER BY "Show", TIME(Time);
```

Exercise 5 Solution

The following code produces the desired result in MySQL:

```
SELECT
    LPAD(DATE_FORMAT(Time, "%l:%i %p"), 8, " ") AS Time,
    ShowName AS "Show",
    VenueName AS Location
FROM Shows, ShowTimes, Venues
WHERE Shows.ShowId = ShowTimes.ShowId
AND Shows.VenueId = Venues.VenueId
ORDER BY TIME(Time), "Show";
```

CHAPTER 27: BUILDING DATABASES WITH SQL SCRIPTS

Exercise 1 Solution

One order in which you could build these tables is: Mpaaratings, Genres, Movies, Persons, MovieProducers, MovieDirectors, RoleTypes, MovieActors.

Exercise 2 Solution

The following code shows one possible SQL script for creating the movie database:

```
CREATE DATABASE MovieDb;
USE MovieDb;

CREATE TABLE MpaaRatings (
    MpaaRaiting    VARCHAR(5)    NOT NULL,
    Description    VARCHAR(40)   NOT NULL,
    PRIMARY KEY (MpaaRaiting)
);

CREATE TABLE Genres (
    Genre          VARCHAR(10)   NOT NULL,
    Description    VARCHAR(40)   NOT NULL,
    PRIMARY KEY (Genre)
);

CREATE TABLE Movies (
    MovieId        INT           NOT NULL    AUTO_INCREMENT,
    Title          VARCHAR(40)   NOT NULL,
    Year           INT           NOT NULL,
    MpaaRating     VARCHAR(5)   NOT NULL,
    Review         TEXT          NULL,
    NumStars       INT           NULL,
    Minutes        INT           NOT NULL,
    Description    TEXT          NULL,
    Genre          VARCHAR(10)   NULL,
    TrailerUrl     VARCHAR(255)  NULL,
    PRIMARY KEY (MovieId),
    INDEX FK_Movies_Ratings (MpaaRating ASC),
    INDEX FK_Movies_Genres (Genre ASC),
    CONSTRAINT FK_Movies_Ratings
        FOREIGN KEY (MpaaRating)
        REFERENCES MovieDb.MpaaRatings (MpaaRaiting)
        ON DELETE NO ACTION
        ON UPDATE NO ACTION,
    CONSTRAINT FK_Movies_Genres
        FOREIGN KEY (Genre)
        REFERENCES MovieDb.Genres (Genre)
        ON DELETE NO ACTION
        ON UPDATE NO ACTION
);

CREATE TABLE Persons (
    PersonId       INT           NOT NULL    AUTO_INCREMENT,
    FirstName      VARCHAR(40)   NOT NULL,
    LastName       VARCHAR(40)   NOT NULL,
    PRIMARY KEY (PersonId)
);

CREATE TABLE MovieProducers (
    MovieId        INT           NOT NULL,
    PersonId       INT           NOT NULL,
```

```

PRIMARY KEY (MovieId, PersonId),
INDEX FK_Producers_Persons (PersonId ASC),
INDEX FK_Producers_Movies (MovieId ASC),
CONSTRAINT FK_Producers_Persons
    FOREIGN KEY (PersonId)
        REFERENCES MovieDb.Persons (PersonId)
    ON DELETE NO ACTION,
    ON UPDATE NO ACTION,
CONSTRAINT FK_Producers_Movies
    FOREIGN KEY (MovieId)
        REFERENCES MovieDb.Movies (MovieId)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION
);

CREATE TABLE MovieDirectors (
    MovieId          INT          NOT NULL,
    PersonId         INT          NOT NULL,
    PRIMARY KEY (MovieId, PersonId),
    INDEX FK_Directors_Persons (PersonId ASC),
    INDEX FK_Directors_Movies (MovieId ASC),
    CONSTRAINT FK_Directors_Persons
        FOREIGN KEY (PersonId)
            REFERENCES MovieDb.Persons (PersonId)
        ON DELETE NO ACTION,
        ON UPDATE NO ACTION,
    CONSTRAINT FK_Directors_Movies
        FOREIGN KEY (MovieId)
            REFERENCES MovieDb.Movies (MovieId)
        ON DELETE NO ACTION
        ON UPDATE NO ACTION
);

CREATE TABLE RoleTypes (
    RoleType        VARCHAR(40)  NOT NULL,
    PRIMARY KEY (RoleType)
);

CREATE TABLE MovieActors (
    MovieId          INT          NOT NULL,
    PersonId         INT          NOT NULL,
    CharacterName    VARCHAR(40)  NOT NULL,
    RoleType         VARCHAR(40)  NULL,
    Review           TEXT         NULL,
    NumStars         INT          NULL,
    PRIMARY KEY (MovieId, PersonId, CharacterName),
    INDEX FK_Actors_Persons (PersonId ASC),
    INDEX FK_Actors_RoleTypes (RoleType ASC),
    INDEX FK_Actors_Movies (MovieId ASC),
    CONSTRAINT FK_Actors_Persons
        FOREIGN KEY (PersonId)
            REFERENCES MovieDb.Persons (PersonId)
        ON DELETE NO ACTION
        ON UPDATE NO ACTION,
    CONSTRAINT FK_Actors_RoleTypes

```

```

    FOREIGN KEY (RoleType)
    REFERENCES MovieDb.RoleTypes (RoleType)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
    CONSTRAINT FK_Actors_Movies
    FOREIGN KEY (MovieId)
    REFERENCES MovieDb.Movies (MovieId)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION
);

DROP DATABASE MovieDb;

```

CHAPTER 28: DATABASE MAINTENANCE

Exercise 1 Solution

The following table shows a backup schedule. In this case, you have time for a full backup every night so you may as well use it.

NIGHT	OFF-PEAK START	OFF-PEAK END	OFF-PEAK HOURS	BACKUP TYPE
Monday	11:00 p.m.	4:00 a.m.	5	Full
Tuesday	11:00 p.m.	4:00 a.m.	5	Full
Wednesday	11:00 p.m.	4:00 a.m.	5	Full
Thursday	11:00 p.m.	4:00 a.m.	5	Full
Friday	11:00 p.m.	6:00 a.m.	7	Full
Saturday	9:00 p.m.	6:00 a.m.	9	Full
Sunday	9:00 p.m.	4:00 a.m.	7	Full

Exercise 2 Solution

Now you can make a full backup only on Saturday night and on other nights you only have time for an incremental backup of changes since the previous night's backup. The following table shows a new backup schedule.

NIGHT	OFF-PEAK START	OFF-PEAK END	OFF-PEAK HOURS	BACKUP TYPE
Monday	12:00 a.m.	3:00 a.m.	3	Incremental from Sunday
Tuesday	12:00 a.m.	3:00 a.m.	3	Incremental from Monday
Wednesday	12:00 a.m.	3:00 a.m.	3	Incremental from Tuesday
Thursday	12:00 a.m.	3:00 a.m.	3	Incremental from Wednesday
Friday	12:00 a.m.	5:00 a.m.	5	Incremental from Thursday
Saturday	10:00 p.m.	5:00 a.m.	7	Full
Sunday	10:00 p.m.	3:00 a.m.	5	Incremental from Saturday

This backup schedule is pretty full, so you should probably start thinking about other strategies to use if your database continues to grow. For example, you might need to perform some backups during peak hours (naturally during the “off-peak” peak hours), or you could partition the database so areas handling different time zones are stored separately so you can back them up separately.

Exercise 3 Solution

The fact that the database now covers multiple time zones causes problems because it means you have fewer non-peak hours for backups. If you partition the data by time zone, you can use the time zones in your favor. Not only will each separate time zone have more non-peak hours separately, but each partition will also hold less data so it will take less time to back up.

The downside is that you'll need to host the partitions on separate computers. Cloud services can often make that easier and more flexible in case you need to add or modify partitions later.

CHAPTER 29: DATABASE SECURITY

Exercise 1 Solution

An order entry clerk doesn't need to read or update any existing order records, so you don't need to set privileges for individual fields in the Orders or OrderItems tables (although you may want the user interface program to read previous orders, so it can copy their values).

The clerk will need to read existing Customers records for existing customers and create Customers records for new customers. Many applications also allow the clerk to update customer data when creating a new order, so the clerk needs Update access to the Customers table. The clerk should not change the CustomerId field, however, because that would disconnect the customer from previous orders. (In general, you should not update primary key values because that causes this kind of problem. Some databases don't even allow you to change primary key values.)

The clerk needs Read access to the InventoryItems table to select the items that the customer wants to buy. (If there isn't enough inventory, assume the clerk creates the order anyway and sets the order's status to Back Order.)

The clerk also needs Read access to the OrderStatuses table to pick an initial status.

The following table lists the privileges that an order entry clerk needs for each table.

TABLE OR FIELD	PRIVILEGES
Customers	C
CustomerId	R
FirstName	RU
LastName	RU
Street	RU
City	RU
State	RU
Zip	RU
Phone	RU
CreditCardType	RU
CreditCardNumber	RU

TABLE OR FIELD	PRIVILEGES
Orders	C
OrderId	–
CustomerId	–
OrderDate	–
ShippedDate	–
OrderStatus	–
OrderItems	C
OrderId	–
SequenceNumber	–
ItemId	–
Quantity	–
InventoryItems	–
ItemId	R
Description	R
Price	R
QuantityInStock	R
OrderStatuses	–
OrderStatus	R

The following SQL statements create an order entry clerk with appropriate privileges:

```
CREATE USER EntryClerk IDENTIFIED BY 'secret';

-- Revoke all privileges for the user.
REVOKE ALL PRIVILEGES, GRANT OPTION FROM EntryClerk;

-- Grant needed privileges.
GRANT INSERT, SELECT ON ShippingDb.Customers TO EntryClerk;
```

```

GRANT UPDATE (FirstName, LastName, Street, City, State, Zip, Phone,
    CreditCardType, CreditCardNumber)
    ON ShippingDb.Customers TO EntryClerk;
GRANT INSERT ON ShippingDb.Orders TO EntryClerk;
GRANT INSERT ON ShippingDb.OrderItems TO EntryClerk;
GRANT SELECT ON ShippingDb.InventoryItems TO EntryClerk;
GRANT SELECT ON ShippingDb.OrderStatuses TO EntryClerk;

```

Exercise 2 Solution

A customer service clerk must be able to read everything to give information about an existing order. This clerk doesn't need to create records but needs to be able to update and delete Orders and OrderItems records for orders that have not yet shipped.

Though the clerk can update Customers data, the CustomerId should never change because that would disconnect it from previous orders.

Note that it doesn't make sense for the clerk to update Orders data. Changing OrderId would disconnect the items from the order, changing CustomerId would disconnect the order from the customer, changing OrderDate would be revising history (popular with politicians but not a good business practice), and changing ShippedDate and OrderStatus is the shipping clerk's job.

This clerk should also not be able to change an OrderItems record's OrderId value because it would disconnect the item from the order.

Whether the clerk can delete Customers records is a business rule. In this case, assume the clerk cannot delete customers, so you don't need to worry about old orders without corresponding Customers records.

Finally, whether the clerk can update OrderItems records or should just delete old records and create new ones is another business rule. In this case, it will probably be easier for the user interface application to delete the old records and create new ones, so the clerk needs Create, Read, and Delete privileges for the OrderItems table.

The following table lists the privileges that a customer service clerk needs for each table.

TABLE OR FIELD	PRIVILEGES
Customers	–
CustomerId	R
FirstName	RU
LastName	RU
Street	RU
City	RU

TABLE OR FIELD	PRIVILEGES
State	RU
Zip	RU
Phone	RU
CreditCardType	RU
CreditCardNumber	RU
Orders	D
OrderId	R
CustomerId	R
OrderDate	R
ShippedDate	R
OrderStatus	R
OrderItems	CD
OrderId	R
SequenceNumber	R
ItemId	R
Quantity	R
InventoryItems	–
ItemId	R
Description	R
Price	R
QuantityInStock	R
OrderStatuses	–
OrderStatus	R

The following SQL statements create a customer service clerk with appropriate privileges:

```
CREATE USER ServiceClerk IDENTIFIED BY 'secret';

-- Revoke all privileges for the user.
REVOKE ALL PRIVILEGES, GRANT OPTION FROM ServiceClerk;

-- Grant needed privileges.
GRANT SELECT ON ShippingDb.Customers TO ServiceClerk;
GRANT UPDATE (FirstName, LastName, Street, City, State, Zip, Phone,
    CreditCardType, CreditCardNumber)
    ON ShippingDb.Customers TO ServiceClerk;
GRANT SELECT, DELETE ON ShippingDb.Orders TO ServiceClerk;
GRANT INSERT, SELECT, DELETE ON ShippingDb.OrderItems TO ServiceClerk;
GRANT SELECT ON ShippingDb.InventoryItems TO ServiceClerk;
GRANT SELECT ON ShippingDb.OrderStatuses TO ServiceClerk;
```

Exercise 3 Solution

The inventory manager's main task is to order new inventory and maintain the InventoryItems table. That requires Create, Read, Update, and Delete privileges on that table.

To change an order's status from Back Ordered to Ordered, the inventory manager must look in the Orders table to find orders in the Back Ordered status, look up the items for that order, and see if there is now enough inventory to fulfill the order. That means the manager must be able to look at the Orders table's OrderId and OrderStatus fields, and update the OrderStatus field. The manager must also be able to look at the OrderItems table's OrderId, ItemId, and Quantity fields.

The following table lists the privileges that an inventory manager needs for each table.

TABLE OR FIELD	PRIVILEGES
Customers	–
CustomerId	–
FirstName	–
LastName	–
Street	–
City	–
State	–
Zip	–

TABLE OR FIELD	PRIVILEGES
Phone	–
CreditCardType	–
CreditCardNumber	–
Orders	–
OrderId	R
CustomerId	–
OrderDate	–
ShippedDate	–
OrderStatus	RU
OrderItems	–
OrderId	R
SequenceNumber	–
ItemId	R
Quantity	R
InventoryItems	CD
ItemId	R
Description	RU
Price	RU
QuantityInStock	RU
OrderStatuses	–
OrderStatus	R

The following SQL statements create an inventory manager with appropriate privileges:

```
CREATE USER InventoryManager IDENTIFIED BY 'secret';

-- Revoke all privileges for the user.
REVOKE ALL PRIVILEGES, GRANT OPTION FROM InventoryManager;

-- Grant needed privileges.
GRANT SELECT (OrderId, OrderStatus) ON ShippingDb.Orders TO InventoryManager;
GRANT UPDATE (OrderStatus) ON ShippingDb.Orders TO InventoryManager;
GRANT SELECT (OrderId, ItemId, Quantity)
    ON ShippingDb.OrderItems TO InventoryManager;
GRANT INSERT, SELECT, UPDATE, DELETE
    ON ShippingDb.InventoryItems TO InventoryManager;
GRANT SELECT ON ShippingDb.OrderStatuses TO InventoryManager;
```

B

Sample Relational Designs

When you break a data model down into small pieces, there are really only three types of data relationships: one-to-one, one-to-many, and many-to-many (modeled with one-to-many relationships). If you think in those terms, then you really don't need examples. Just break the problem into small enough pieces and start assembling these three kinds of relationships. (Inheritance and subtyping forms another kind of logical relationship that you can also model.)

However, it may be useful to see more complete examples that include several different entities associated in typical ways. This appendix is intended to show you those kinds of examples.

Note that many different problems can be modeled in very similar ways. For example, consider a typical library. It has one or more copies of a whole bunch of books and lends them to patrons for a specific amount of time. Patrons can renew a book once and pay late fees if they fail to return the book on time. Some libraries have other policies such as not charging late fees on children's books, not charging late fees at all, or sending out specially trained book-sniffing dogs to hunt you down if you don't return a book. (I'm guessing on that last one.)

Now, consider a business that rents party supplies such as tables, chairs, big tents, bouncy castles, dunk tanks, doves, and so forth. Like a library, this business has multiple copies of many of these items. (It probably has dozens of tables and hundreds of chairs.) Also like a library, this business "loans" (for a fee) its items to customers and charges a late fee if an item isn't returned on time.

Though a library and a party rental store are very different organizations, the structure of their databases is quite similar.

As you look at the examples in this appendix, think about variations that might use a similar structure. Though your application may not fit these examples exactly, you may find an example that uses a similar structure.

Also note that different applications might use very different database designs for the same data. The exact fields and sometimes even the tables included in the design depend on the application's focus.

For example, consider a large company's employee data. If you're building an application to assign employees to tasks for which they are qualified, your database will probably have an `EmployeeSkills` table that matches employee records to their skills. You'll also need a `Tasks` table that describes tasks and lists the skills that they require.

In contrast, suppose you need to build a human resources application that tracks employee payroll deductions for retirement contributions, medical coverage, and so forth. Although this application deals with the same employees, it doesn't need skill or task data, so it doesn't need the `EmployeeSkills` or `Tasks` tables. It also needs new employee data not required by the work assignment project, such as employee Social Security number, bank account number, next of kin, and anniversary date.

These two applications deal with the same physical entities (employees) but have very different data needs. Because the type of data that you might need to store in a particular table depends on your application, these examples don't try to be exhaustive. For employee data, a table might include a few fields such as `FirstName`, `LastName`, and `HireDate` to give you an idea of what the table might include, but you'll have to fill in the details for your application.

The following sections each describe a single example database design and give some ideas for variations that you may find useful. In particular, many of these models can include all sorts of address information such as `Street`, `Suite`, `Building`, `Office`, `MailStop`, `POBox`, `RuralRoute`, `City`, `Urbanization`, `State`, `Province`, `Zip`, `PostalCode`, `Country`, `Phone`, and `Extension`. To keep the models simple, some tables include a single `Address` entry to represent address information. You should use whatever address information is appropriate for your application.

These models are basically relational, but some might be interesting to study in NoSQL databases. For example, you could use a graph database such as Neo4j AuraDB to store movie information. Then you could examine the relationships among actors, directors, and movies. That might let you find patterns such as groups of actors who produce good movies. Add in customer ratings and you can search for preference patterns. "If you like *this* movie, then you may also like *that* movie."

Many simpler designs would also work well in a document database such as MongoDB Atlas. Any time you need to store self-contained records, you can put the information in JSON files and store them in a document database. For example, cooking recipes, Pokémon card data, fantasy football player stats, and Cub Scout popcorn preorders (so you know where to deliver later) are all relatively self-contained and would fit well in JSON documents.

For example, you the database's JSON files could hold recipes. Then you could search the `Keywords` field for "holiday" or search the `Ingredients` field for "goji berries."

BOOKS

The entities in a books database include the books themselves, authors, and publishers. Depending on your application, you may also want to include information about the book's various editors (acquisitions editors, managing editors, production editors, technical editors, and copy editors), artists, photographers, and all of the other many people associated with the book. If the book is a compilation (for example, a collection of articles or short stories), the "Editor" who put it all together is similar to the author of the work as a whole.

Some of these people may be associated with the publisher. For example, many editors work for a particular publisher, although others are hired by a publisher as contractors and editors often move

from one publisher to another over time. So if you want to record these types of associations, you will probably need to allow editors to have a separate publisher affiliation for each book.

Books may have many printings and editions. Different printings contain few or no differences. Different editions may contain completely different content or may come in different media such as paperback, hardcover, audio CD, online video, DVD, PDF file, large print edition, and so forth.

Figure B.1 shows a simple book database design that models books, authors, and publishers. Book categories include Cooking, Home & Garden, Science Fiction, Professional, Bodice Ripper, and so forth. Recall that dashed lines represent lookup tables.

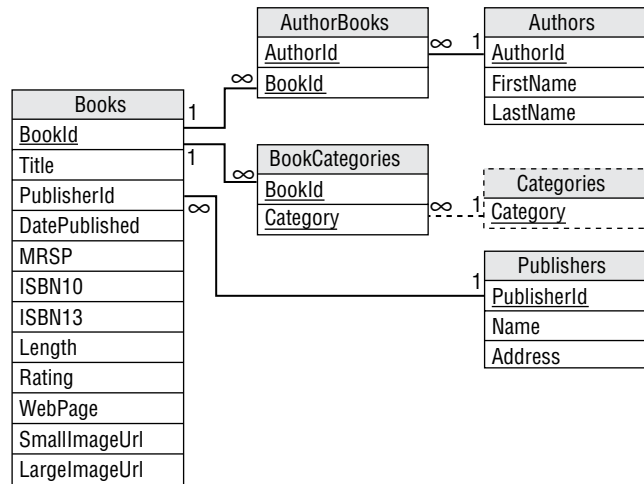


FIGURE B.1

Figure B.2 shows a more complex design that includes media and edition information. The LengthUnit field depends on an item's medium. For example, audio books have length measured in minutes, whereas printed books have length measured in pages.

FUNNY PAGES

I'm not sure how you should measure length for e-books where the number of "pages" depends on the book reader's format. For example, one search online said an e-book version of *War and Peace* was 2,118 pages long, but the actual number of pages depends on the size of your screen. For example, I sometimes read e-books on an e-reader (small), tablet (larger), or laptop (larger still). I knew one person who read very long books on his cellphone (although otherwise he seemed normal enough).

The number of pages will also depend on the font size that you use, which in turn may depend on whether your glasses are handy.

I think I would pick a number of pages at some standard screen size and go with that, keeping in mind that the actual number of "pages" will vary.

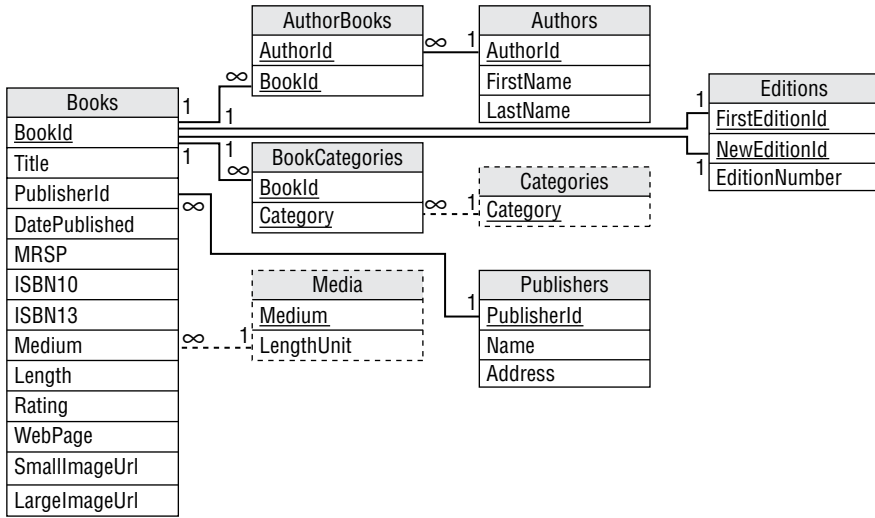


FIGURE B.2

Figure B.3 generalizes the author data to include other types of people such as editors, artists, and photographers.

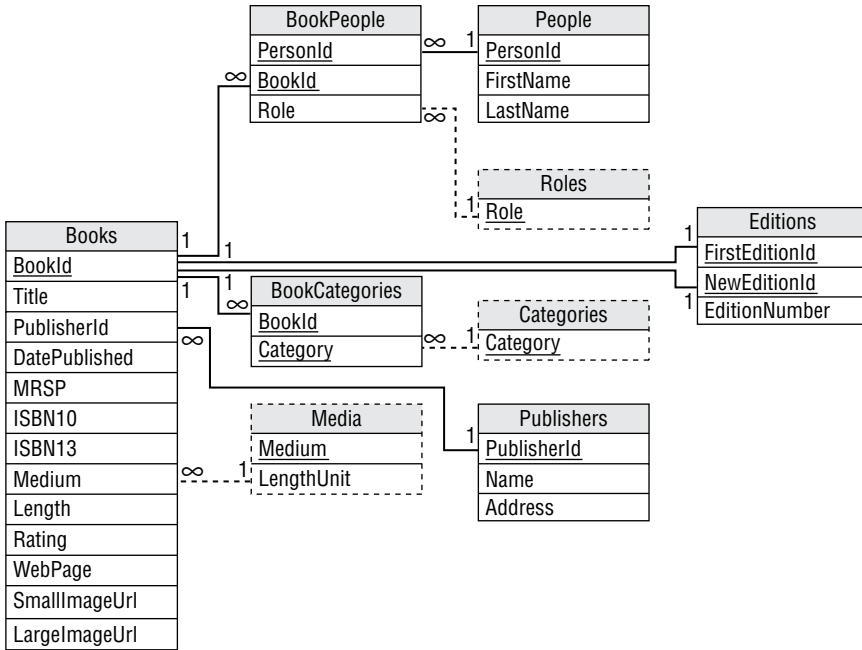


FIGURE B.3

MOVIES

In many ways, movies are similar to books. Both are created by a team of people who may work on other projects as well, both are owned by some sort of entity (publisher or studio), and both have similar basic information such as lengths, descriptions, and URLs. Many of the details are different (movies have actors instead of authors and directors instead of editors), but some of the basic structure of the data is similar.

Figure B.4 shows a version of Figure B.3 that has been modified to hold movie data. To avoid confusion, this model uses the word “job” to represent a person’s responsibility in the project (Actor, Director, Producer, Grip, Best Boy, Python Wrangler, and so forth) and “actor role” to represent a part played by an actor.

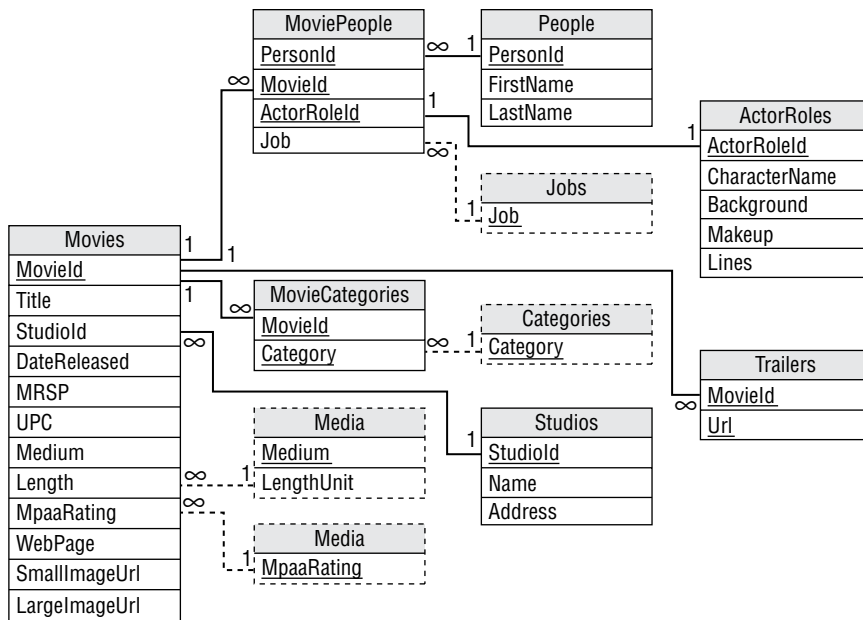


FIGURE B.4

Notice that a single person may appear in more than one *MoviePeople* records for a movie. For example, an actor may play more than one role and may be the director. (I suppose an actor could also be a crew member but it’s hard to imagine Orlando Bloom catering or Julia Roberts stringing cables.)

Notice also the one-to-one relationship between the *MoviePeople* and *ActorRoles* records. Each *ActorRoles* record represents a single role in a particular movie. If someone later shoots a remake of a movie, this model assumes that each of the characters gets a new *ActorRoles* record because the characters will represent new interpretations of the originals. If the *ActorRoles* records are

abbreviated enough that they will be the same for different versions, you could make this a one-to-many relationship. You might also want to add version information similar to the Editions table in Figure B.3.

(I'll know a lot more about the movie industry so I can build better models after Steven Spielberg makes a movie out of this book.)

MUSIC

Though music collections, books, and movies all have similarities (for example, they are all produced by a team of people), some important differences exist. Songs are grouped by album and albums are grouped by band. Some or all of a band may participate in any given song on an album. Over time, the members of a band may change (except for U2), and one artist may be in many bands or even the same band more than once (in case they have a falling out and then later decide to make a reunion tour when the money runs out).

Although you could make similar distinctions for books (you could group books by series and you might define working combinations of authors), it doesn't make as much sense unless your application really needs to focus on that kind of information. Figure B.5 shows a design to hold music album data.

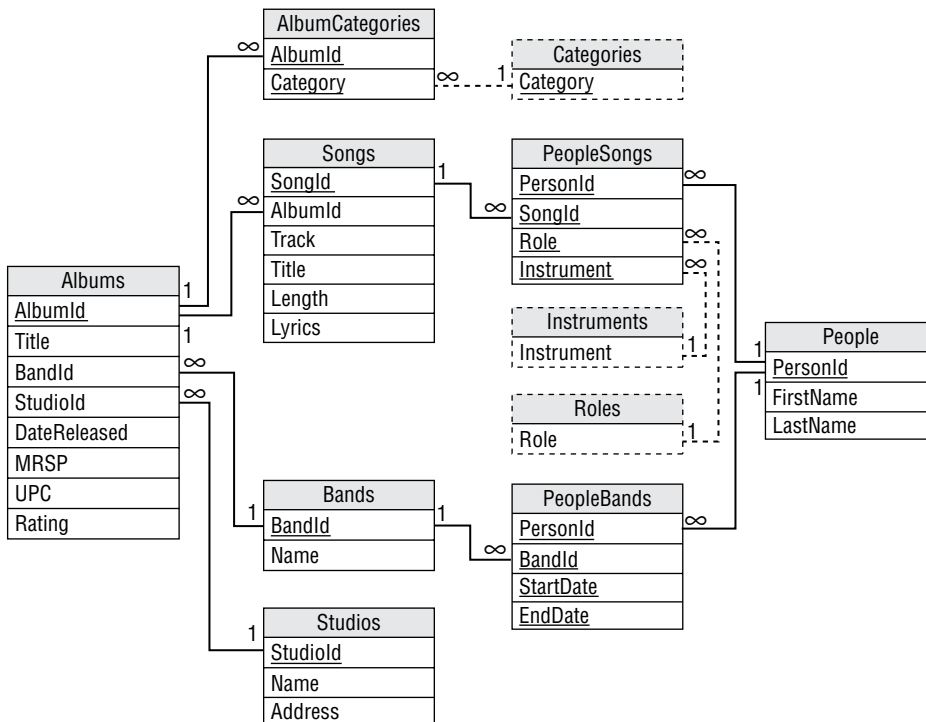


FIGURE B.5

In this model, roles might include such values as Lead Vocal, Song Writer, Choreographer, and Costume Designer. The Instruments table holds values such as Electric Guitar, Drums, Zither, and Electric Stringed Didgeridoo.

Notice that one person may play multiple roles in the same song. Often, this will be singer and songwriter, but some artists sing while they play and really nimble artists may play multiple instruments.

DOCUMENT MANAGEMENT

A document management system provides features such as storing documents, managing concurrent use, controlling permissions, allowing you to create new versions over time, and showing you the changes between different versions. They also let you “fork” documents to create different branches that form a document hierarchy. For example, you might make a cookbook and then fork a copy to modify into a gluten-free cookbook. Those two branches could then evolve separately over time.

This is different from a NoSQL document database, which focuses on storing and querying documents. You could probably use a document database to build a document management system, but that’s not what I’m talking about here.

Before you rush out and build a document management system, you might consider using one that is already available. Systems such as Git (<https://git-scm.com>), Concurrent Versions System (CVS—www.nongnu.org/cvs), and Apache Subversion (<https://subversion.apache.org>) manage multiple document versions quite effectively. They may not provide all of the features that you might add (such as advanced keyword queries), but they provide enough features to be quite useful without all of the work of building your own system.

However, Figure B.6 shows a data model that you could use to manage multiple document versions. This model assumes that a single author makes each version of a document and that multiple versions have major and minor version numbers as in 1.0, 1.1, 2.0, and so forth. The model allows you to store keywords and comments for the document as a whole and for each version.

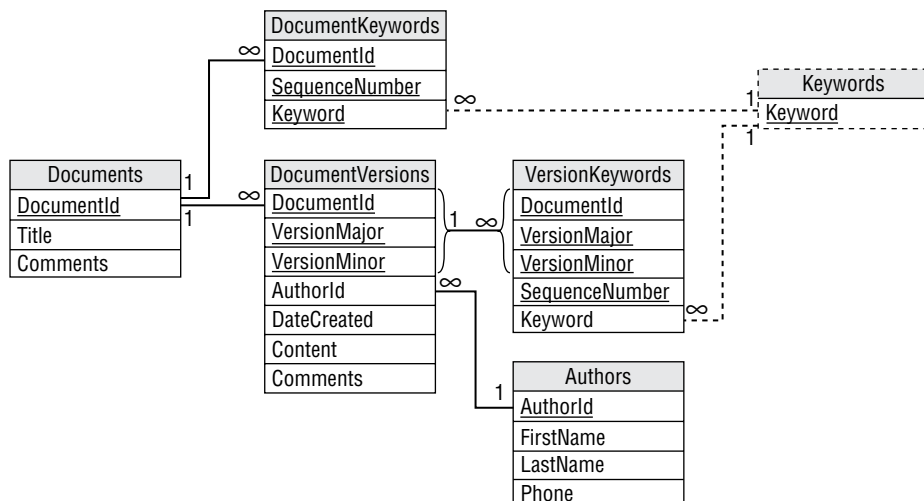


FIGURE B.6

The DocumentVersions table’s Content field can hold either the complete document version or a list of differences between this version and the previous one.

CUSTOMER ORDERS

Several of the examples described in this book include data to record customer orders. Figure B.7 shows one of the simpler variations. It assumes a customer has a single address and all orders for that customer are shipped to that address.

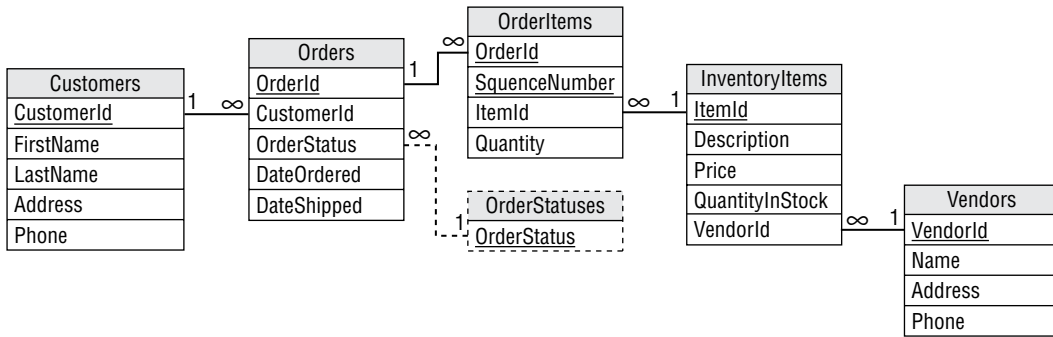


FIGURE B.7

EMPLOYEE SHIFTS AND TIMESHEETS

Employee shifts and timesheet records are very similar. Both record a date and hours scheduled or hours worked for an employee. Figure B.8 shows a simple model for storing shift and timesheet data.

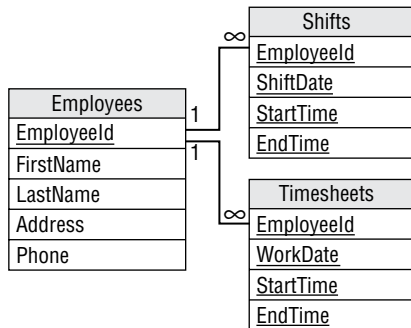


FIGURE B.8

EMPLOYEES, PROJECTS, AND DEPARTMENTS

Figure B.9 shows a model for storing employee, project, and department data. This model assumes that an employee can be in any number of projects but only one department. The DepartmentRoles table contains values such as Manager, Secretary, Member of Technical Staff, and Sycophant. The ProjectRoles table contains values such as Project Manager, Lead Developer, Toolsmith, and Tester. The primary key for EmployeeProjects includes the ProjectRole field, so a single employee can play multiple roles in a single project (for example, Project Manager and Doomsayer).

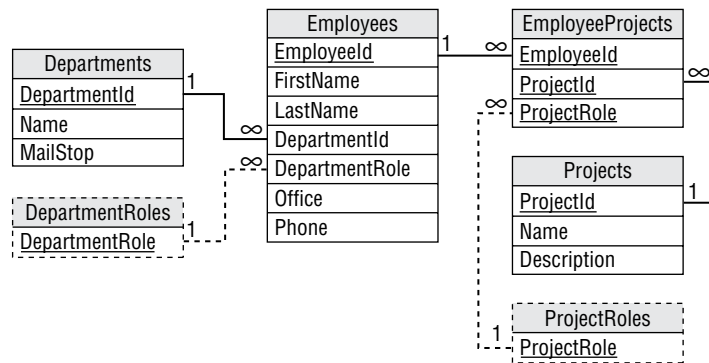


FIGURE B.9

You can use this design for matrix management with only a few changes. In matrix management, an employee has a *functional manager* who coordinates employees who have similar functions (mechanical engineering, optical design, software development, and so forth). The functional manager guides the employee's career development and handles project reviews.

Project managers determine what the employee does on a particular project.

For example, an electronics technician might be in the Electronics department, report to a functional manager in that department, work on several projects in various other departments, and report to project managers in those departments.

If each functional department has a single manager, you can use Figure B.9 by simply adding the functional managers as members of their departments with DepartmentRole set to Functional Manager. If there is not a simple one-to-one relationship between departments and functional managers, you can add a FunctionalId field to the Employees table as shown in Figure B.10.

EMPLOYEE SKILLS AND QUALIFICATIONS

Employee skills and qualifications are important when certain jobs require them. For example, machining a particular glass part might require a technician who is certified to use a computerized ultrasonic cutter.

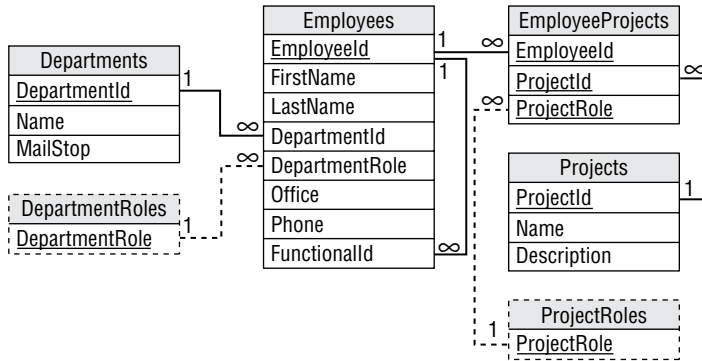


FIGURE B.10

Depending on your application, some qualifications expire, so you need to take a class and/or test to remain current. For example, a Red Cross CPR certification lasts 2 years and the USA Gymnastics Safety Certification lasts 4 years (assuming you pay the annual fee). The design shown in Figure B.11 assumes that all skills expire. If a skill does not expire, you can set its ValidDuration to a really large value such as 100 years (or perhaps 1 million years for a longevity coach certification). If none of the skills you track expire, you can remove the ValidDuration field.

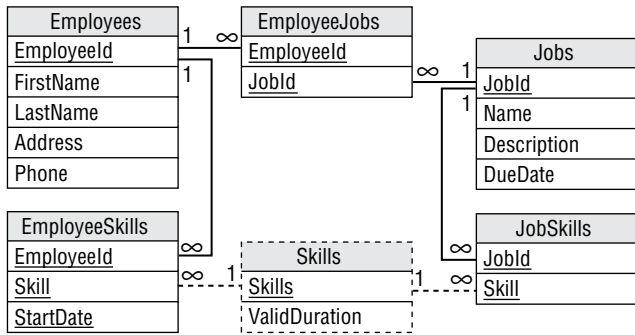


FIGURE B.11

In a more formal setting, you might need to add more fields to the EmployeeSkills table. For example, you might need to track a certification number, issuing agency, and so forth to prove an employee has a certain skill.

Note that this design tracks employee skills but doesn't do anything with them. It would be up to the user interface application to ensure that only employees with the proper skills are assigned to a given job.

IDENTICAL OBJECT RENTAL

Just about any situation where something is given to a customer for a limited period of time can be modeled as a rental. For example, construction equipment rentals, kayak rentals, hourly contractors, and hotel rooms can all be treated as different kinds of rentals.

Figure B.12 shows a design that holds data for simple rental of identical objects. For example, a surf shop might have a dozen each of six different surfboard models and you don't really care *which* 8' Wavestorm you get (as long as it's not covered in tooth marks).

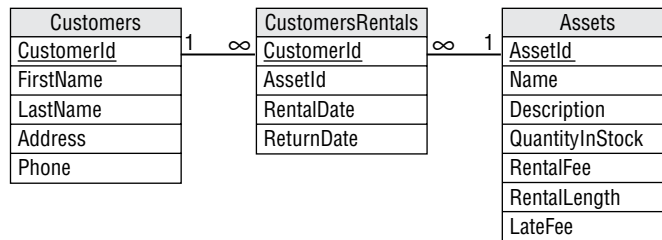


FIGURE B.12

Some businesses have items that are distinct, but the differences may not matter to you. For example, a kimono rental store might have hundreds of kimonos with different colors and patterns, but you may want to treat them as interchangeable in the database.

If you want to track specific instances of rented assets (for example, to keep track of the number of times each surfboard or kimono is rented), you can add an AssetInstances table, as shown in Figure B.13.

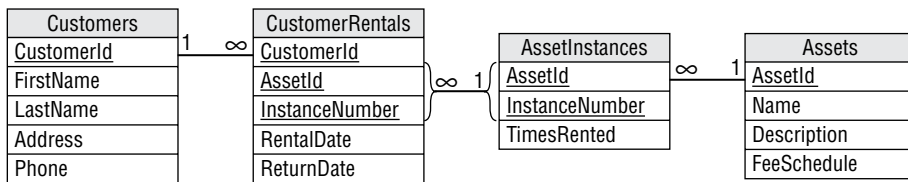


FIGURE B.13

There are many variations on rental and late fees. A bicycle rental store might charge a daily fee and no late fees. A store that rents heavy equipment such as backhoes and pile drivers might charge an hourly fee and large late fees. A public library might charge no rental fee and a small daily late fee. (My local library charges no fees for children's books.)

DISTINCT OBJECT RENTAL

If the objects that are rented are distinct, then you need to modify the design slightly. Figure B.14 shows a rental variation that is more appropriate for a company that hires its employees as contractors.

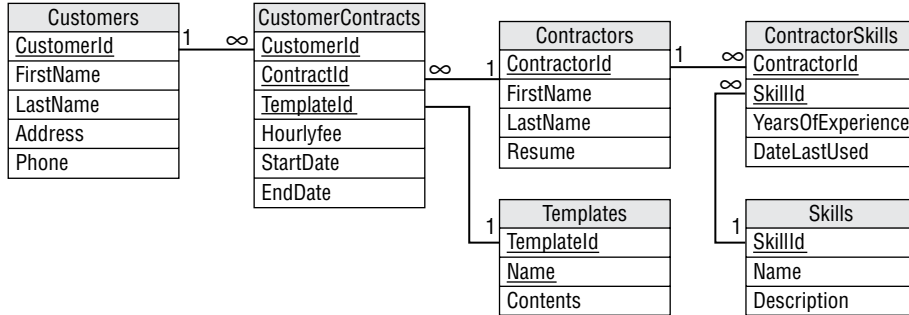


FIGURE B.14

Unlike kayak rental and libraries, the contractor “rental” design models distinct entities because different contractors are not all the same (unless you’re the Army or a mega-corporation that treats people as interchangeable “assets”).

Other businesses can model distinct entities in a similar manner, although the exact details will usually differ. For example, Figure B.15 shows a model designed for hotel reservations.

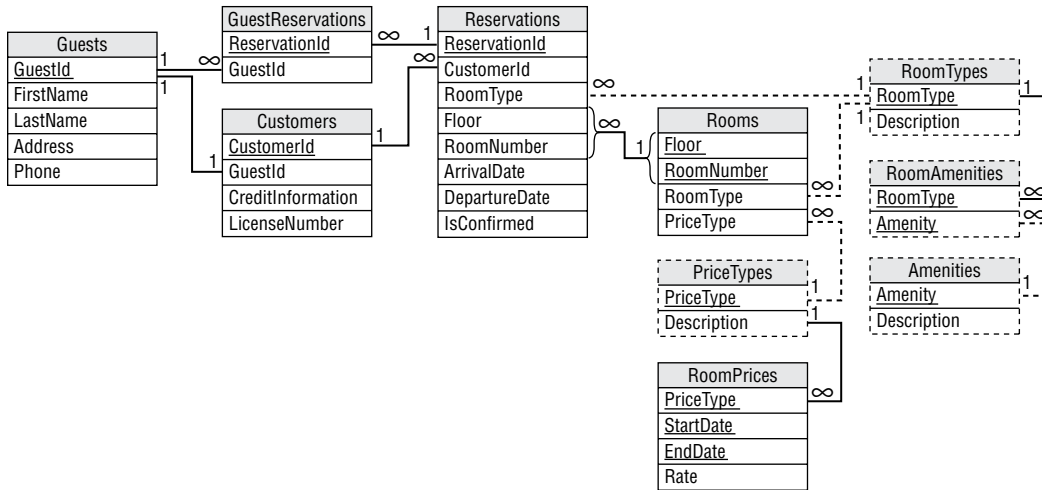


FIGURE B.15

A customer calls and makes a reservation. Initially the reservation is for a type of room, not a particular room. When the guests check in, the clerk fills in the floor and room number.

A Guest is a person staying in the room. (Several people may be staying in the same room.) A Customer is a person who pays for a reservation (only one person pays). If the IsConfirmed field has the value True, the reservation is confirmed for late arrival, so the hotel will hold the room and charge the Customer's credit card if they show up at midnight. (However, I've had my room sold to another customer despite being confirmed for late arrival, so the model may need tweaking in the real world.)

A Room Type defines the amenities in the room. Amenities include such things as hot tubs, balconies, bathrooms, nonsmoking, pets allowed, and king-sized, heart-shaped rotating beds that vibrate.

A Price Type defines the prices for a room. Price Types include values such as Business, Preferred, Frequent Visitor, Walk In, and Chump Whose Flight Was Canceled At The Last Minute And Is Desperate. (Hotels typically code price types as A, B, C, and so forth so the Chump doesn't notice he's paying four times as much as the family from Des Moines who booked three months in advance.)

STUDENTS, COURSES, AND GRADES

Figure B.16 shows a model for storing student, course, and grade data.

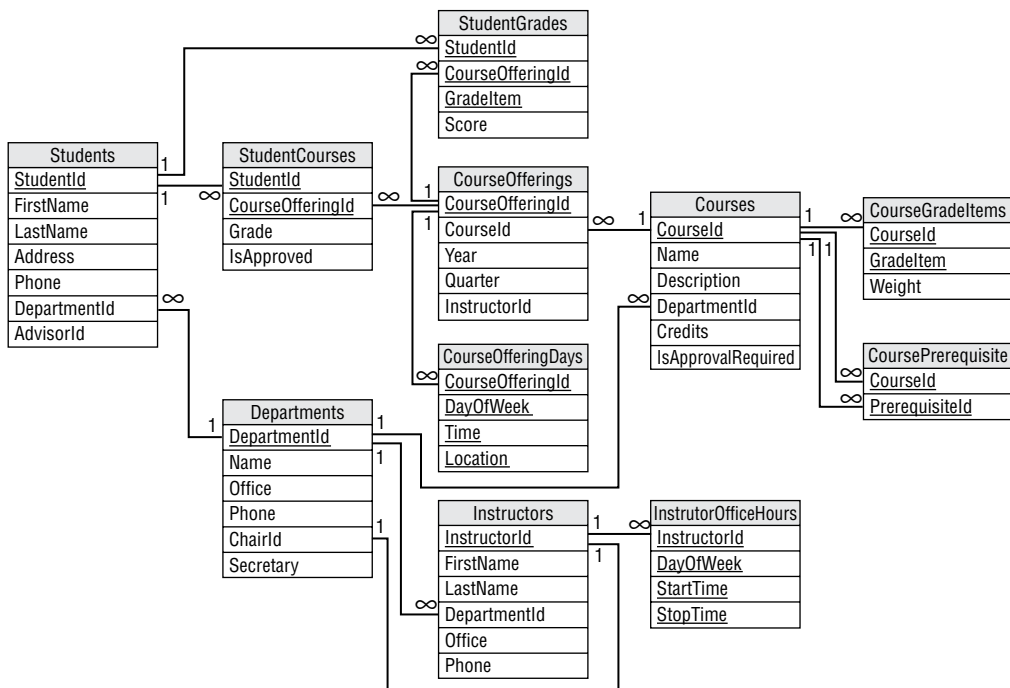


FIGURE B.16

A Course represents a type of class (Introduction to Database Design). A Course Offering is a particular instance of a Course (this year's winter term Introduction to Database Design class on Tuesdays and Thursdays from 9:00 to 10:30 in Building 12, room B-16).

A Grade Item is something in the course that receives a grade, such as Quiz 1, Midterm, Term Paper 2, and Attendance. The CourseGradeItem table's Weight field lets you assign different weights to different Grade Items. For example, you might have 10 quizzes with a weight of 10 each and a final with a weight of 100, so it's worth half of the total grade.

Grade Items are somewhat tricky because a StudentGrades record should have an appropriate GradeItem value. You might like to make the record's combined CourseOfferingId/GradeItem be a foreign key into the CourseGradeItems table, but that table uses CourseId as a key, not CourseOfferingId. Ensuring that the CourseGradeItems record has a valid GradeItem must be handled as a business rule.

This database will probably be stored in two pieces: an online piece holding the current school year's data and a data warehouse piece holding older data. The data would undergo final consistency checks before it is moved from the current database to the warehouse. For example, you would verify that students have grades for every CourseGradeItem defined for their classes.

Other Students fields in the online database would probably record summary information. For example, a GPA field could record the student's grade point average for courses in the data warehouse. That field would be redundant because you could recalculate it from the data in the data warehouse, but placing it in the online database would let you avoid opening the data warehouse for day-to-day queries.

TEAMS

Figure B.17 shows a relatively simple data model for team sports. This design is based on a typical volleyball league. Players belong to a team and teams play in tournaments.

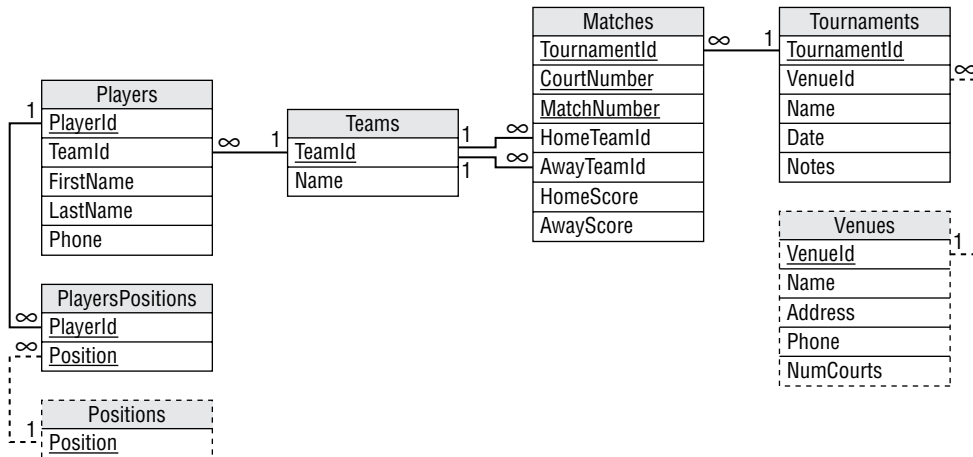


FIGURE B.17

This model allows each player to be associated with several positions. You can include special

non-playing “positions” such as captain, coach, and water boy, or you can add them as new fields in the Players or Teams table depending on your needs.

Tournaments occur at venues that have a given number of courts. A match is a game between two teams. (In practice, a match will include several games, so the scores may be games won rather than points won. In a really serious competition, you would need to expand the model to save scores for individual games in a match, so you can compare points head-to-head in case there’s a tie based on games alone. In fact, official volleyball record sheets include so much detail that you can figure out exactly when each point was made by each team and every player’s location at the time—at least you can if you’re good at solving puzzles.)

In a normal tournament, teams play against each other in *pools*. For example, in a tournament of 12 teams, the teams might be divided into two pools of six with each pool on a separate court. Each pool would play a round-robin where every team plays against every other team in its pool. Then the top two teams from each pool would enter single-elimination playoffs.

You can modify the simple design shown in Figure B.17 to handle non-tournament situations. For example, in many soccer leagues, teams play one game a week so there isn’t really a notion of a tournament. In that case, you can pull the relevant tournament fields (VenueId, Date, Notes) into the Matches table. You might also want to make some cosmetic changes such as changing “court” to “field” or “pitch.”

INDIVIDUAL SPORTS

An individual sport such as running doesn’t need all of the team information recorded in the previous model. Instead, its database can focus on individual statistics and accomplishments.

Figure B.18 shows a model to hold running information. If you only store basic race information, you can treat races like any other run. If you’re more competitive and want to record race data such as finishing position, position in age group, mile times, number of bathroom breaks, and so forth, you can add new Races and RunnerRaces tables similar to the Runs and RunnerRuns.

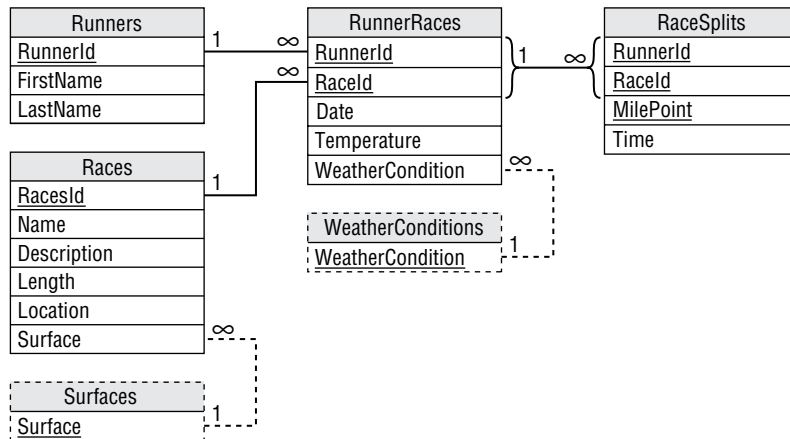


FIGURE B.18

VEHICLE FLEETS

Fleet tracking can be quite complex. Different parts of a business might want to track the vehicles' cargo and weights, current location, special equipment and tools, leases, repairs and maintenance, mileage, equipment, drivers, taxes, fuel use and taxes, and so forth.

For this example, you should know a little about the International Fuel Tax Agreement (IFTA). IFTA is an agreement among most U.S. states and Canadian provinces to distribute fuel taxes fairly.

Each state and province charges a different tax rate on various kinds of fuel such as gasoline, diesel, propane, E-85, A55, and several others. (Perhaps the list will soon include hydrogen, capacitance gel, and antimatter.) The taxes are included in the price at the pump and you've been paying them for years, probably without thinking about it.

The system is simple and makes sense until you consider a big fleet of vehicles that buys fuel in one state and then drives mostly in another state. For example, suppose your business is in Kansas City, Missouri but you do most of your driving across the river in Kansas City, Kansas. Fuel is cheaper in Missouri so you buy yours there. Kansas screams, "No fair! You're paying fuel taxes to Missouri but using our roads!" Enter ITFA.

Each quarter, you need to file IFTA tax forms listing every mile you drove and every drop of fuel you purchased in every state or province. You then need to pay any extra taxes that you owe based on how much tax you paid in each state and where you drove. In this example, you probably owe Kansas some money. The net result is there's much less incentive for you to cross borders to buy fuel. The IFTA agency gathers all of these records from fleets all over North America, performs lengthy calculations, and then makes the states pay each other the differences between their taxes collected and what they should have collected based on miles driven. The numbers tend to cancel out so the grand totals aren't necessarily big.

Figure B.19 shows a model designed to hold license, permit, fuel, and mileage data. Each field marked with an asterisk should be validated against a States lookup table, but to keep the model simple (relatively simple, anyway), the States table and its links aren't shown.

The model's FuelPurchases table records the states in which fuel is purchased. The TripStateEntries records the mileages at which a vehicle entered a new state. By subtracting Mileage values in subsequent TripStateEntries, you can calculate the number of miles driven in each state.

Another interesting case is modeling jobs, employees, and vehicles with special requirements, tools, and skills. Employees have tools (such as wrenches, ohm meters, and chainsaws) and skills (such as the ability to fix dishwashers, install phones, and juggle). Vehicles have equipment such as pipe benders and threaders, cherry pickers, and pole setters.

Finally, jobs require certain skills and tools. For example, if you need to haul a lot of logs, you need a vehicle with a tree grapple and an employee who has tree grappling as a skill. ("Tree grappling" sounds like a wrestling move but it's not.)

Figure B.20 shows a data model to store this information. The model is simplified and leaves out a lot of information. For example, you may need to add job addresses, appointments, site contact, and so forth.

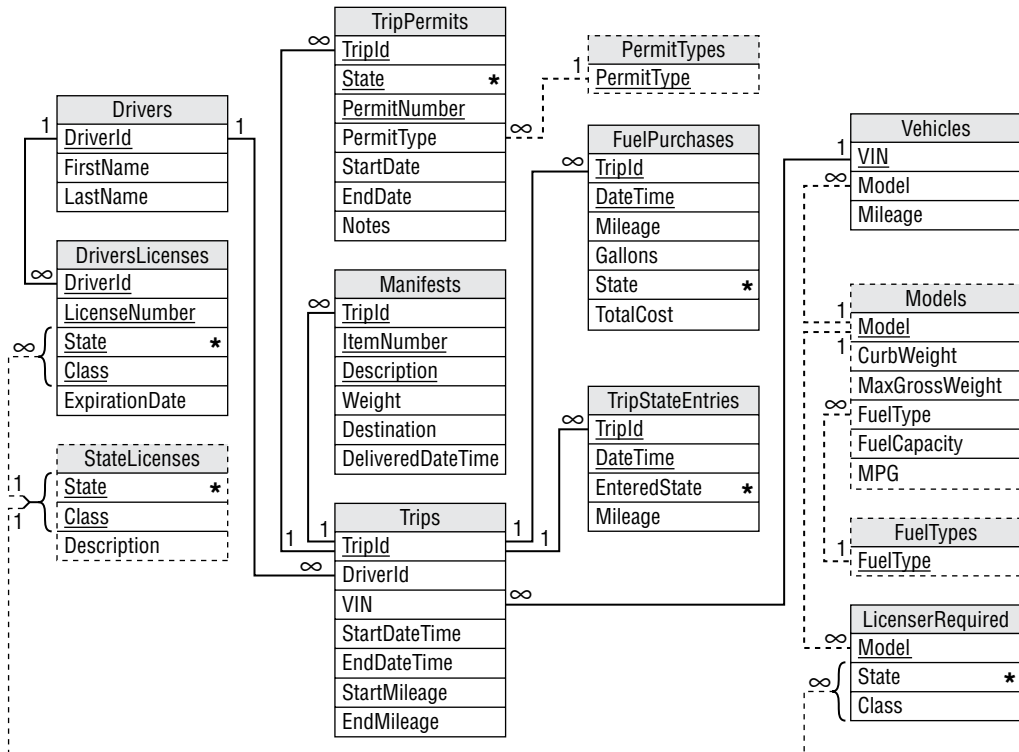


FIGURE B.19

Note that assigning a tool to an employee implies that the employee can use the tool. You wouldn't give a defibrillator to an employee who didn't know how to use it. However, if you also want to model employees signing equipment in and out, you might need to make this assumption explicit by giving the employee a Defibrillator skill.

This model assumes that multiple employees may be assigned to a single job. You could allow an Assignments record to have a null VehicleId value to allow two employees to ride in the same vehicle.

Of course, once you have this data stored, someone will need to figure out a way to match employees, vehicles, and jobs to get the most work done as efficiently as possible, and that's a whole different book.

CONTACTS

The most obvious application that needs to store contact information is an address book, but many other applications store contact information too. Most complex applications that involve interaction among customers, employees, vendors, and other people can benefit from a contact database. For example, an order placement and processing application can use contact data to keep track of customer calls that place orders, change orders, request returns, and register complaints.

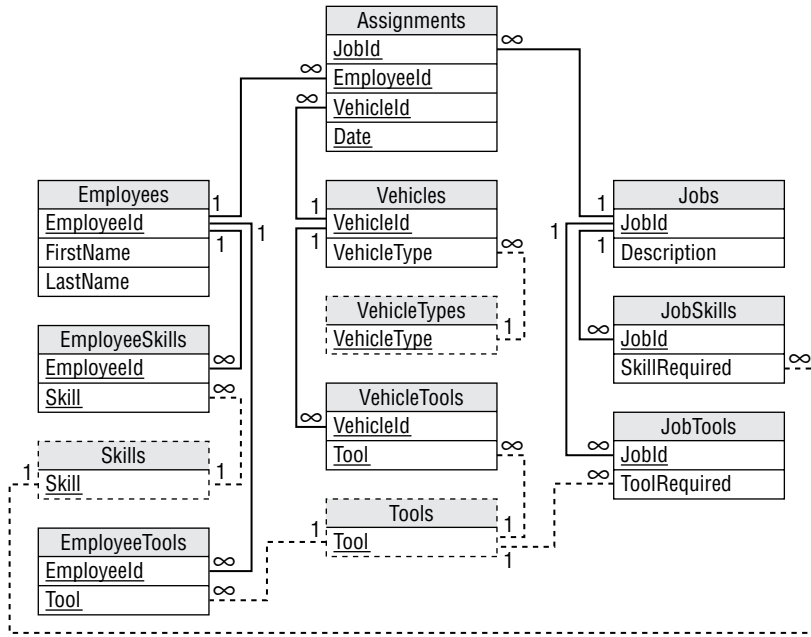


FIGURE B.20

Figure B.21 shows a general contact data model. An application can use these tables to remember contacts at different times covering different topics.

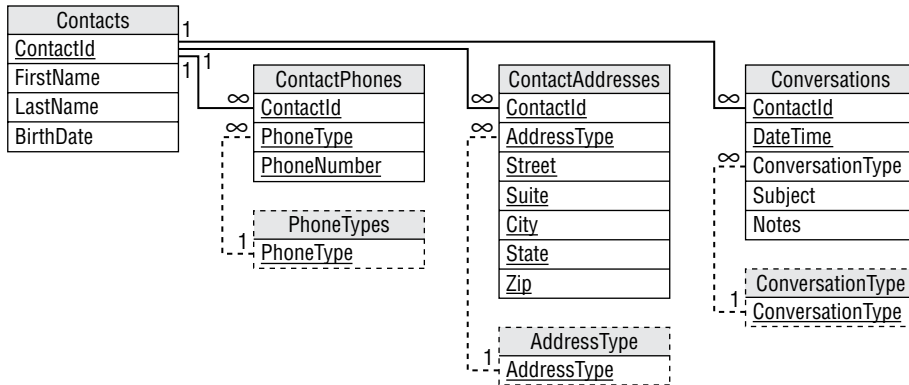


FIGURE B.21

If you want to integrate contact data in an application involving multiple employees, you may want to add an EmployeeId field to the Conversations table so that you know who talked to the customer. You might also want to add fields to refer to a customer order to help further define the conversation. That would allow you to search for all of the conversations related to a particular order.

PASSENGERS

There are several ways you might like to model vehicles with passengers. For example, typically city buses don't take reservations and don't care where passengers sit as long as the number of passengers doesn't exceed the vehicle's capacity (which in larger cities seems to be approximately two passengers per cubic foot of space).

Figure B.22 shows a simple design to track the number of passengers on a bus.

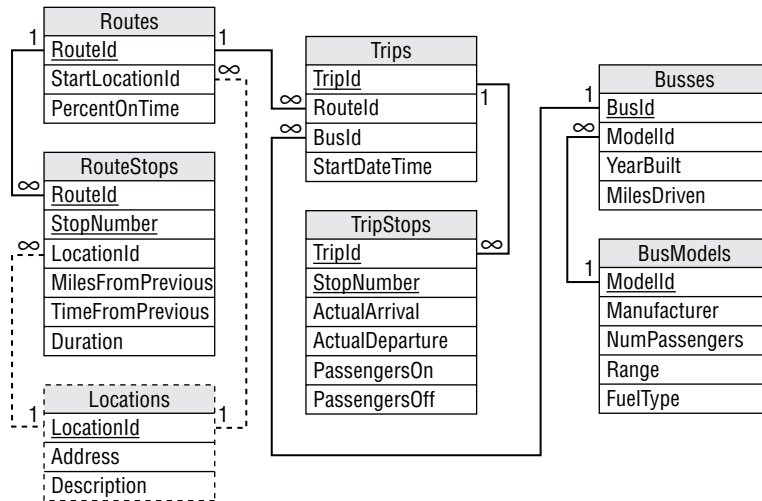


FIGURE B.22

A route defines the stops that a bus will take. The information for each stop includes the time it should take to get to that stop and the duration of time that the bus should ideally wait at that stop.

A trip represents a bus traveling over a route. The TripStops records correspond to RouteStops records and record actual times and passenger counts. (In practice I don't know how often drivers record passenger numbers. As long as one more passenger can pile on top, most drivers don't seem too bothered.)

Figure B.23 shows a slightly more complex model that allows passengers to reserve room on a bus but not to reserve individual seats. This model is intended for long-distance common carriers such as long-distance buses (Greyhound, Trailways) and railroads. In this model, the customer makes reservations to ensure that a seat is available on each leg of the trip but specific seats are not assigned.

This model is very similar to the previous one except it includes Reservations and ReservationSeats tables. Each Reservations record records information about a customer's trip. A ReservationSeats record holds information about a set of seats on a particular bus trip. The collection of ReservationSeats records corresponding to a particular Reservations record contains all of the information about the buses that a passenger's trip will use.

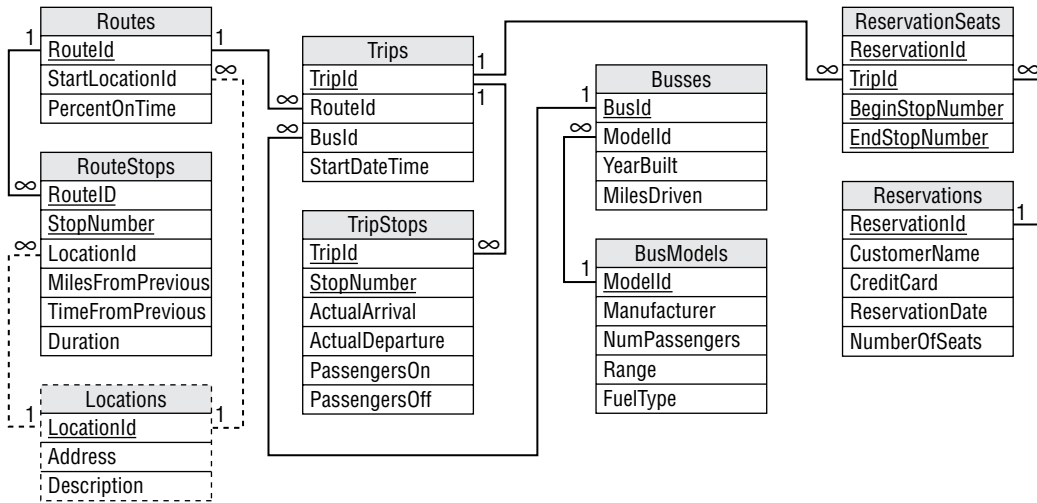


FIGURE B.23

You can model airline and other travel where passengers have previously assigned seats using a very similar model. The only change you need (at least to model this part of the system) is to add assigned seat information to the ReservationSeats data. You could also add meal selection and other special information to each seat.

Note that these databases are typically enormous. For example, a typical large airline runs several thousand flights per day holding up to a few hundred passengers each. That means the Trips and TripStops tables grow by a few thousand records per day and the ReservationSeats table might grow by a few hundred thousand records per day. If you allow passengers to reserve seats up to a year in advance, the database must be able to hold several hundred million records.

Keeping such a large and quickly changing database running efficiently 24 hours a day is a Herculean effort. It may require huge disk farms, segmented data, special route-finding algorithms, and massive backup and warehousing processes. In other words, don't try this at home.

RECIPES

This may seem like a silly example, but it demonstrates how to store a set of instructions that require special equipment such as pots, pans, and ingredients.

A recipe database needs to store basic information about recipes such as their names, difficulty, and tastiness rating. It also needs an ingredient list and instructions. Figure B.24 shows a simple recipe database design. This model assumes the Difficulty and Rating fields are simple numeric values (for example, on a 1 to 10 scale). If you wanted to, you could change them to values such as Easy, Medium, and Hard, and make them foreign keys to lookup tables.

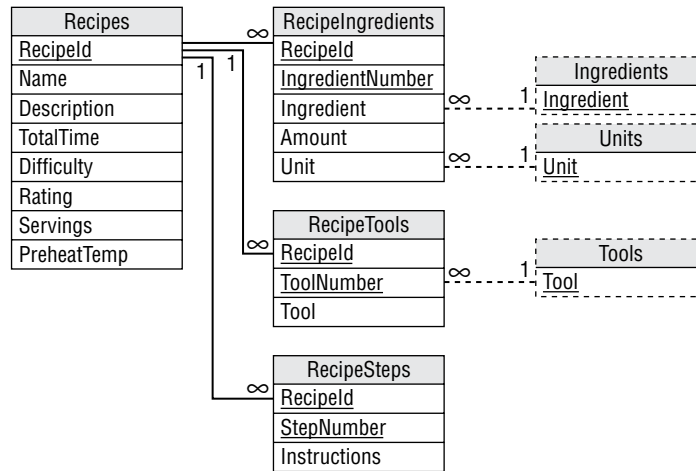


FIGURE B.24

You can use this design to store information about other assembly tasks (such as putting together skateboards, tuning a car, building a tokamak reactor, and so forth) or more generally for giving instructions for complex tasks (troubleshooting a wireless network or deciphering a long-distance calling plan).

Unfortunately, generalizing this model to pull information out of the steps is trickier than it might initially seem. For example, you might like to make an instruction record refer to an ingredient and then tell you what to do with it as in “Oatmeal, 2 cups, mix.” That instruction would work but others are more complex.

For example, a recipe might ask you to mix different ingredients in separate bowls and then combine them. To break that information out, you would probably need to record the bowls as equipment and then somehow associate ingredients with each bowl. Some recipes call for even more complex steps such as separating eggs, scalding milk, caramelizing sugar, changing temperatures during cooking, and even lighting food on fire.

With enough time and effort, you might be able to write a cooking language to let you represent all of these operations (you could call it CML—Cooking Markup Language) but what would you have gained? Breaking instructions down to that level would let you do amazing things like finding all recipes that require you to perform certain tasks such as “powderizing” oatmeal in a food processor, but how often will you need to perform those kinds of searches?

The simpler model already lets you search for specific tools, ingredients, and temperatures, so it’s probably best to stick with that model unless you have a very specialized need with well-defined steps. If necessary, you can add keywords to the recipes to let you search for particular unusual tools and techniques such as flambé and fossil-shaped gelatin molds.

A simple recipe database would also fit naturally in document database. Each document would be a JSON file describing a recipe and having fields that let you search for keywords or specific ingredients.

GLOSSARY

This appendix contains a glossary of useful database terms. You may find them handy when you read other books and articles about databases. You may also want to look for database and related glossaries online and in print. For example, <https://raima.com/database-terminology> and www.prisma.io/dataguide/intro/database-glossary provide good online glossaries.

1NF See *first normal form*.

2NF See *second normal form*.

3NF See *third normal form*.

4NF See *fourth normal form*.

5NF See *fifth normal form*.

6NF See *domain key normal form*.

A

ACID An acronym for a set of database properties needed for reliable transactions. ACID stands for atomicity, consistency, isolation, durability (see the entries for those terms).

alternate key A candidate key that is not used as the table's primary key.

Apache Ignite A NoSQL key-value database that can run locally or in the cloud. See <https://ignite.apache.org>.

association object In a semantic object model, an object used to represent a relationship between two other objects.

atom A piece of data that cannot be meaningfully divided. For example, a Social Security number of the form 123-45-6789 contains three pieces. There is structure in those pieces (see www.ssa.gov/history/ssn/geocard.html), but it's a pretty unusual application that needs to look them up separately so the number as a whole is usually treated as an atom.

atomic transaction A possibly complex series of actions that is considered as a single operation by those not involved directly in performing the transaction.

atomicity The requirement that tasks within a transaction occur as a group as if they were a single complex task. The tasks are either all performed or none of them are performed. It's all or nothing.

attribute The formal database term for column (see *column*).

B

b+tree (Pronounced “bee plus tree.”) A self-balancing tree data structure that allows efficient searching of indexes. A b+tree stores data records only in leaf nodes.

BCNF See *Boyce-Codd normal form*.

BLOB Binary large object. A data type that can hold large objects of arbitrary content such as video files, audio files, images, and so forth. Because the data can be any arbitrary chunk of binary data, the database does not understand its contents so you cannot search in these fields.

Boyce-Codd normal form (BCNF) A table is in BCNF if every determinant is also a candidate key (minimal superkey). See *determinant* and *candidate key*.

b-tree (Pronounced “bee tree.”) A self-balancing tree data structure that allows efficient searching of indexes. A b-tree stores data records in internal and leaf nodes.

business rule Business-specific rule that constrains the data. For example, “all orders require a valid existing Customers record” and “orders for \$50 or more get free shipping” are business rules.

C

candidate key A minimal superkey (see *superkey*). In other words, the fields in a candidate key uniquely define the records in a table and no subset of those fields also uniquely defines the records.

cardinality A representation of the minimum and maximum allowed number of values for an attribute. In semantic object models, written as L.U, where L and U are the lower and upper bounds. For example, 1.10 means an attribute must occur between 1 and 10 times.

catalog A directory storing metadata.

check constraint A record-level validation that is performed when a record is created or updated.

cloud Servers, software, databases, and services that are accessed over the Internet.

column A piece of data that may be recorded for each row in a table. The corresponding formal database term is attribute.

column-oriented database (aka wide-column database or column family database) These allow you to treat data in a table-like format, although they may allow different rows in a table to contain different fields. They are optimized to efficiently fetch or search data in columns, so they make aggregation analysis faster.

commit To make changes performed within a transaction permanent. The SQL `COMMIT` command performs this action.

composite index An index that includes two or more fields. Also called a *compound index* or *concatenated index*.

composite key A key that includes two or more fields. Also called a *compound key* or *concatenated key*.

composite object In a semantic object model, an object that contains at least one multivalued, non-object attribute.

compound index An index that includes two or more fields. Also called a *composite index* or *concatenated index*.

compound key A key that includes two or more fields. Also called a *composite key* or *concatenated key*.

compound object In a semantic object model, an object that contains at least one object attribute.

concatenated index An index that includes two or more fields. Also called a *compound index* or *composite index*.

concatenated key A key that includes two or more fields. Also called a *compound key* or *composite key*.

consistency The requirement that a transaction should leave the database in a consistent state. If a transaction would put the database in an inconsistent state, the transaction is canceled.

CRUD An acronym for the four main database operations: create, read, update, delete. These operations correspond to the SQL statements `INSERT`, `SELECT`, `UPDATE`, and `DELETE`.

CSV file Comma-separated value file. A text file where each row contains the data for one record and field values are separated by commas.

cursor An object that allows a program to work through the records returned by a query one at a time. Some databases allow cursors to move forward and backward through the set of returned records, whereas others allow only forward movement.

cyclic dependency Occurs when objects depend on each other directly or indirectly. For example, a cyclic dependency occurs in a table if field A depends on field B, field B depends on field C, and field C depends on field A.

D

Data Control Language (DCL) The SQL commands that control access to data such as `GRANT` and `REVOKE`.

Data Definition Language (DDL) The SQL commands that deal with creating the database's structure such as `CREATE TABLE`, `CREATE INDEX`, and `DROP TABLE`.

data dictionary A list of descriptions of data items to help developers stay on the same track.

Data Manipulation Language (DML) The SQL commands that manipulate data in a database. These include `INSERT`, `SELECT`, `UPDATE`, and `DELETE`.

data mart A smaller data warehouse that holds data of interest to a particular group. Also see *data warehouse*.

data mining Digging through data (usually in a data warehouse or data mart) to identify interesting patterns.

data scrubbing Processing data to remove or repair inconsistencies.

data type The type of data that a column can hold. Types include numbers, fixed-length strings, variable-length strings, BLOBs, and so forth.

data warehouse A repository of data for offline use in building reports and analyzing historical data. Also see *data mart*.

database An entity that holds data in some useful way and provides CRUD methods (see *CRUD*). Modern databases also provide sophisticated methods for joining, sorting, grouping, and otherwise manipulating the data.

database administrator (DBA) Someone who manages the database, optimizes performance, performs backups, and so forth.

DBA See *database administrator*.

DBMS Database management system. A product or tool that manages any kind of database, not just relational databases.

DDBMS Distributed database management system. See *DBMS*.

DDL See *Data Definition Language*.

delete To remove one or more rows from a table. The SQL `DELETE` command performs this action. Represents the D (for Delete) in CRUD.

deletion anomaly Occurs when deleting a record can destroy information.

determinant A field that at least partly determines the value in another field.

dimensional database A database that treats the data as if it is stored in cells within a multidimensional array (see *multidimensional array*).

distributed database A database with pieces stored on multiple computers on a network.

DKNF See *domain key normal form*.

DML See *Data Manipulation Language*.

document database (aka document store or document-oriented database) Stores documents, usually in XML, JSON, BSON, or a similar format. These allow you to query on fields defined within the documents.

domain The values that are allowed for a particular column. For example, the domain of the `AverageSpeed` field in a database of downhill speed skiers might allow values between 0 and 200 miles per hour (although if your average speed is 0, you might consider another sport).

domain key normal form (DKNF) A table is in DKNF if it contains no constraints except domain constraints and key constraints.

durability The requirement that a completed transaction is safely recorded in the database and will not be lost even if the database crashes.

E

Easter egg A feature or message hidden in software or another medium.

Edgar Codd IBM researcher who laid the groundwork for modern relational databases and SQL starting in 1970.

edge In a graph database, a connection between two nodes. An edge represents a relationship between its nodes, so it is sometimes called a relationship.

entity In entity-relationship modeling, an object or item of interest such as a customer, invoice, vehicle, or product.

entity integrity Requires that all tables have a primary key. The values in the primary key fields must be non-null and no two records can have the same primary key values.

entity-relationship diagram (ER diagram) A diagram that shows entities (rectangles), their attributes (ellipses), and the relationships among them (diamonds).

ER diagram See *entity-relationship diagram*.

Extensible Markup Language (XML) A language that uses nested tokens to represent hierarchical data. Also see *JSON*.

F

field Another informal term for column (see *column*).

fifth normal form (5NF) A table is in 5NF if it is in 4NF and contains no related multivalued dependencies.

first normal form (1NF) A table is in 1NF if it satisfies basic conditions to be a relational table.

flat file A plain-old text file used to store data. A flat file isn't very fancy and provides few tools for querying, sorting, grouping, and performing other database operations, but flat files are very easy to use.

foreign key One or more columns that are related to values in corresponding columns in another table.

fourth normal form (4NF) A table is in 4NF if it is in BCNF and contains no unrelated multivalued dependencies.

G

graph database A database that stores nodes that represent pieces of data connected by edges (aka links or relationships) that represent relationships between the nodes. These are good when you want to study relationships among objects.

Graph Query Language (GQL) Upcoming standard language for querying graph databases. Used by Neo4j AuraDB. See www.gqlstandards.org.

H

HOLAP Hybrid online analytical processing. A combination of MOLAP and ROLAP. Typically, this combines relational storage for some data and specialized storage for other data. The exact definition of HOLAP isn't clear, so you can use it as a conversation starter at cocktail parties. Also see *OLAP*.

hybrid object In a semantic object model, an object that contains a combination of multivalued and object attributes.

hypercube A multidimensional array (see *multidimensional array*). To be a true hypercube, each dimension should have the same length or number of entries.

I

identifier In a semantic object model, one or more attributes that are used to identify individual objects. Indicated by writing ID to the left of the attribute(s), underlined if the identifier is unique.

index A data structure that uses one or more columns to make looking up values on those columns faster.

insert To create a new record in a table. The SQL `INSERT` command performs this action. The C (for Create) in CRUD.

insertion anomaly Occurs when you cannot store certain kinds of information because it would violate the table's primary key constraints.

instance A particular occurrence of an entity. For example, if VicePresident is an entity (class), then William Rufus King was an instance of that class.

isolation The requirement that no one should be able to peek into the database and see changes while a transaction is underway. Anyone looking at the data will either see it as it is before the transaction or after the transaction but cannot see the transaction partly completed.

J

JavaScript Object Notation (JSON) A language that uses nested tokens to represent hierarchical data. Also see *XML*.

join A query that selects data from more than one table, usually using a `JOIN` or `WHERE` clause to indicate which records in the two tables go together.

JOLAP Java Online Analytical Processing. A Java API for online analytical processing. Also see *OLAP*.

JSON See *JavaScript Object Notation*.

K

key One or more fields used to locate or arrange the records in a table. Also see *index*.

key constraint In an ER diagram, a key constraint means an entity can participate in at most one instance of a relationship. For example, during flight a pilot can fly at most one hang glider.

key-value database (aka key-value store) A database that holds values that you can index by using a key. This is similar to the hash tables, dictionaries, and associative arrays that are provided by different programming languages.

L

link See *edge*.

lock Used to control access to part of the database. For example, while one user updates a row, the database places a lock on the row so that other users cannot interfere with the update. Different databases may lock data by rows, table, or disk page.

M

many-to-many relationship A relationship where one object of one type may correspond to many objects of another type, and vice versa. For example, one COURSE may include many STUDENTS and one STUDENT may be enrolled in many COURSEs. Normally you implement this kind of relationship by using an intermediate table that has one-to-many relationships with the original tables.

MariaDB An open source relational database management system. See <https://mariadb.org>. Also see *Michael Widenius*.

MaxDB Relational database management system from SAP AG. Also see *Michael Widenius*.

memo A text data type that can hold very large chunks of text.

metabase A database that stores metadata.

metadata Data about the database such as table names, column names, column data types, column lengths, keys, and indexes. Some relational databases allow you to query tables that contain the database's metadata.

Michael Widenius Main author of the MySQL, MariaDB, and MaxDB databases, which are named for his children My, Maria, and Max.

MOLAP Multidimensional analytical processing. The “classic” version of OLAP and is sometimes referred to as simply OLAP. See *OLAP*.

MongoDB Atlas A NoSQL cloud document database. See www.mongodb.com.

multidimensional array A multidimensional rectangular block of cells containing values. Picture a row of bricks where each brick is a cell. A wall made of bricks arranged in rows and columns (which would not be very architecturally sound) would be a two-dimensional array. A series of walls closely packed together (which would be architecturally bizarre) would be a three-dimensional array. Use your imagination for higher dimensions.

multivalued dependency When one field implies the values in two other fields that are unrelated. For example, a table has a multivalued dependency if field A implies values in field B, and field A implies values in field C, but the values in fields B and C are not related.

MySQL A relational database management system that was formerly open source but that is now mostly commercialized. See www.mysql.com. Also see *Michael Widenius*.

N

Neo4j AuraDB Fully managed graph database service in the cloud. See www.neo4j.com. Uses the Graph Query Language (see *Graph Query Language*).

node In a graph database, a node represents some kind of object. Edges define relationships between nodes.

normalization The process of transforming the database's structure to minimize the chances of certain kinds of data anomalies.

NoSQL Either “not SQL” or “not only SQL.” When speaking about database types, this generally refers to nonrelational databases, which include the four main categories document database, key-value database, column-oriented database, and graph database. Many of these can run locally or in the cloud.

null A special column value that means “this column has no value.”

O

object An instance of an item of interest to the data model. See *instance*.

object database See *object-oriented database*.

object database management system (ODBMS) A product or tool for managing object-oriented databases. See *object-oriented database*.

object store See *object-oriented database*.

object-oriented database A database that provides tools to allow a program to create, read, update, and delete objects. The database automatically handles object persistence (changes to the object are automatically saved) and concurrency (two users accessing the same object will not interfere with each other).

object-relational database (ORD) A database that provides relational operations plus additional features for creating, reading, updating, and deleting objects.

object-relational database management system (ORDBMS) See *object-relational database*.

object-relational mapping A translation layer that converts objects to and from entries in a relational database.

ODBMS See *object database management system*.

OLAP See *online analytical processing*.

one-to-many relationship A relationship where one object of one type may correspond to many objects of another type. For example, one INSTRUCTOR may teach many COURSEs, but each COURSE has only one INSTRUCTOR.

one-to-one relationship Occurs when one record in a table corresponds to exactly one record in another table. For example, each US_STATE has one GOVERNOR and each GOVERNOR belongs to one US_STATE.

online analytical processing (OLAP) A data mining approach for analyzing information from multiple data sources.

OODBMS Object-oriented database management system. See *object database management system*.

Oracle An enterprise-level relational database management system. See www.oracle.com/database.

ORD See *object-relational database*.

ORDBMS Object-relational database management system. See *object-relational database*.

P

participation constraint In an ER diagram, a participation constraint means every entity in an entity set must participate in a relationship set. The constraint is drawn with a thick or double line. For example, during flight a hang glider must participate in the “Pilot Flies HangGlider” relationship.

PL/SQL Procedural Language/Structured Query Language. Oracle’s extension to SQL used to write stored procedures in Oracle.

PostgreSQL (aka Postgres) Open source object-relational database system. See www.postgresql.org.

primary key A candidate key that is singled out as the table’s “main” method for uniquely identifying records. Most databases automatically build an index for a table’s primary key and enforce uniqueness.

primary key constraint Requires that each table’s primary key behavior be valid. In particular, this requires that no two records in a table have exactly the same primary key values and that all records’ primary key values be non-null.

Q

query A command that extracts data from a database. In SQL, a SELECT statement.

R

RDBMS See *relational database management system*.

record Another informal term for row (see *row*).

referential integrity Requires that relationships among tables be consistent. For example, foreign key constraints must be satisfied. You cannot accept a transaction until referential integrity is satisfied.

regret What you feel when you accidentally omit the `WHERE` clause in an `UPDATE` or `DELETE` statement so that you update or delete every record in the table.

relation The database theoretical term for a table. For example, the Customer table is a relation holding attributes such as `FirstName`, `LastName`, `Street`, `City`, `State`, `ZipCode`, `PacManTopScore`, and `WarcraftName`.

relational database A database that stores data in tables containing rows and columns and that allows queries representing relationships among records in different tables.

relational database management system (RDBMS) A product or tool that manages a relational database such as MySQL, PostgreSQL, MariaDB, or SQL Server.

relationship An association between two tables. For example, if an order contains several order items, there is a one-to-many relationship between the `Orders` and `OrderItems` tables. Don't confuse this term with "relation."

replication The process of storing data in multiple databases while ensuring that it remains consistent. For example, one database might contain a master copy of the data and other satellite databases might hold read-only copies to let clerks or customers view data quickly without impacting the main database.

report The results of a query displayed in a nice format. Sometimes, this term is used to mean the format that will produce the report when data is added to it.

ROLAP Relational online analytical processing. OLAP performed with a relational database. See *OLAP*.

roll back To undo changes performed within a transaction before the transaction is committed. The SQL `ROLLBACK` command performs this action.

row A group of related column values in a table. The corresponding formal database term is tuple. Also see *record*.

S

savepoint A position within a transaction that you may want to roll back to later. The program can roll back the entire transaction or to a savepoint. The SQL `SAVEPOINT` command creates a savepoint.

second normal form (2NF) A table is in 2NF if it is in 1NF and every field that is not part of the primary key depends on every part of the primary key.

select To extract data from a database, possibly from multiple tables. The SQL `SELECT` command performs this action. This is the R (for Read) in CRUD.

semantic attribute A characteristic of a semantic object.

semantic class A named collection of attributes sufficient to specify an entity of interest.

semantic object (SO) An instance of a semantic class with specific attribute values.

semantic object model (SOM) A model that uses classes, objects, and relationships to provide understanding of a system. Classes have attributes that describe instances. Object attributes provide the relationships among objects.

simple object In a semantic object model, an object that has only single-valued simple attributes.

SOM See *semantic object model*.

splay tree A self-balancing tree data structure that allows efficient searching of indexes.

SQL See *Structured Query Language*.

SQL Server Microsoft's enterprise-level relational database management system. See www.microsoft.com/sql.

stored procedure A piece of code stored in the database that can be executed by other pieces of code such as check constraints or application code. Stored procedures are a good place to store business logic that should not be built into the database's structure.

Structured Query Language (SQL) An industry standard English-like language for building and manipulating relational databases.

subclass object An object that represents a subset of the objects in a larger class. For example, the Vehicle class could have a Truck subclass, which could have its own PickupTruck subclass, which could in turn have its own BrokenPickupTruck class.

superkey A set of fields that uniquely define the records in a table. (Not a key that wears a cape and fights crime.)

T

table A collection of rows holding similar columns of data. The corresponding formal database term is *relation*.

temporal database A database that associates times with data. See *valid time*.

third normal form (3NF) A table is in 3NF if it is in 2NF and it contains no transitive dependencies.

transaction A series of database operations that should be treated as a single atomic operation, so either they all occur or none of them occur. In SQL a transaction normally begins with a `START TRANSACTION` or `BEGIN TRANSACTION` command and ends with a `COMMIT` or `ROLLBACK` command. Also see *commit*, *roll back*, and *savepoint*.

Transaction Control Language (TCL) The SQL commands that control transactions such as `COMMIT`, `ROLLBACK`, and `SAVEPOINT`.

Transact-SQL (TSQL) Microsoft's version of SQL used in SQL Server. See *SQL*.

transitive dependency When one non-key field's value depends on another non-key field's value. Typically, this shows up as duplicated data. For example, suppose a table holds people's favorite book

information and includes fields Person, Title, and Author. The primary key is Person but Title determines Author, so there is a transitive dependency between Title and Author.

trigger A stored procedure that executes when certain conditions occur such as when a record is created, modified, or deleted. Triggers can perform special actions like creating other records or validating changes.

TSQL See *Transact-SQL*.

tuple The formal database term for a record or row (see *row*).

U

unique constraint (aka uniqueness constraint) Requires that the values in one or more columns be unique within a table.

update To modify the values in a record's fields. The SQL `UPDATE` command performs this action. This is the U (for Update) in CRUD.

update anomaly Occurs when a change to a row leads to inconsistent data.

V

valid time The time during which a piece of data is valid in a temporal database.

view The result of a query that can be treated as if it were a virtual table. For example, you could define views that return only selected records from a table or that return only certain columns.

W

weak entity In an ER diagram, an entity that cannot be identified by its own attributes. Instead you need to use the attributes of some other associated entity to find the weak entity. Drawn with a bold or double rectangle and attached to its identifying relationship by a bold or double arrow.

X

XML See *Extensible Markup Language (XML)*.

INDEX

A

- action methods, in Neo4j AuraDB, 405–409, 419–423
- actors, in use cases, 102
- address constraints, 305–306
- algorithms, 56
- ALTER command, 496
- alternate key, 36, 671
- alternative flow, in use cases, 102
- Amazon Web Services (AWS), 435
- American National Standards Institute (ANSI), 492
- Anaconda, 329
- Apache Ignite
 - about, 328
 - in C#
 - about, 477, 483–484
 - building class, 480
 - creating programs, 477–483
 - demonstrating persistent data, 483
 - demonstrating volatile data, 483
 - exercise solutions, 633–635
 - exercises, 484–485
 - installing Ignite database adapter, 478–479
 - main program, 479–480
 - ReadData method, 482–483
 - WriteData method, 480–481
 - defined, 671
 - in Python
 - about, 467–468, 474
 - creating programs, 470–474
 - defining Building class, 471
 - demonstrating persistent data, 474
 - demonstrating volatile data, 473–474
 - exercise solutions, 631–633
 - exercises, 474–475
 - installing Apache Ignite, 468
 - installing pyignite database adapter, 471
 - with persistence, 470
 - reading data, 473
 - saving data, 471–472
 - starting nodes, 468–470
 - without persistence, 469
- Apache Subversion, 521
- application programming interface (API), 43
- applications
 - multi-tier, 154–158
 - social media, 55
- architecture, multi-tier, 205
- as a service (aaS), 23
- association objects, 126–128
- associations
 - about, 216
 - many-to-many, 216
 - multiple many-to-many, 216–218
 - multiple-object, 218–221
 - reflexive, 222–231
 - repeated attribute, 221–222
- atom, 671
- atomic objects, 124
- atomic transactions
 - defined, 8, 671
 - as a desirable feature of databases, 12–13, 25
- atomicity
 - defined, 671
 - lack of, in NoSQL databases, 198
- Atomicity, Consistency, Isolation, and Durability (ACID)
 - about, 561

defined, 671
 as a desirable feature of databases, 13–16

attributes
 defined, 671
 entity-relationship models (ER diagram/ERD)
 and, 131–132
 in relational databases, 34

audit trails, 236

availability, in CAP theorem, 22

AVERAGE function, 511

B

background, of Structured Query Language (SQL), 491

backups
 for database maintenance, 533–537
 as a desirable feature of databases, 17–18, 25

basic initialization scripts, 520

basic syntax, for Structured Query Language (SQL), 495

Basically Available, Soft state, and Eventually consistent (BASE)
 about, 561–562
 as a desirable feature of databases, 16

BEGIN command, 498

Beginning Software Engineering, Second Edition (Stephens), 99, 204

Binary JavaScript Object Notation (BSON), in MongoDB Atlas, 432–434

Binary Large Object (BLOB) data type, 52, 499, 672

Boeing, 554

books, database design for, 650–652

Boolean data type, 499

Boyce-Codd Normal Form (BCNF), 181–185, 672

brainstorming
 about, 94–95
 with customers, 265–275

Brewer, Eric (computer scientist), 22

Brewer's theorem, 22

b-tree, 672

b+tree, 672

bugs, 249

building
 combined ER diagrams, 291–292

data models, 283–301

databases with SQL scripts, 519–532

ER diagrams, 289–290

initial semantic object models, 283–286

multiple-object associations, 220–221

tables, 500–503

Building class, in Apache Ignite, 471, 480

Building the Data Warehouse 4th Edition (Inmon), 538

build_org_chart method, 410

BuildOrgChart method, 424

business rules
 about, 145–147, 158–159, 303, 310–311
 defined, 672
 drawing relational models, 310
 exercise solutions, 581–587, 608–610
 exercises, 159–161, 311
 extracting, 152–154, 303–311
 identifying, 147–152, 303–309
 multi-tier applications, 154–158

business tier/layer, 154

C

C#
 about, 327

Apache Ignite in
 about, 477, 483–484
 building class, 480
 creating programs, 477–483
 demonstrating persistent data, 483
 demonstrating volatile data, 483
 exercises, 484–485
 installing Ignite database adapter, 478–479
 main program, 479–480
 ReadData method, 482–483
 WriteData method, 480–481

MariaDB in
 about, 355, 366
 creating data, 360–363
 creating databases, 356–358
 creating programs, 355–365
 defining tables, 358–360
 exercises, 366–367
 fetching data, 364–365
 installing MySQLConnector, 356

- MongoDB Atlas in
 - about, 453–454, 465
 - creating programs, 454–465
 - exercises, 466
 - helper methods, 454–462
 - installing MongoDB database adapter, 454
 - main program, 462–465
- Neo4j AuraDB in
 - about, 417–418, 428
 - action methods, 419–423
 - creating programs, 418–427
 - exercises, 428–429
 - installing Neo4j driver, 418–419
 - main program, 426–427
 - org chart methods, 423–425
- PostgreSQL in
 - about, 389, 399
 - connecting to database, 390
 - creating customer data, 392–393
 - creating order data, 393–395
 - creating order item data, 395–396
 - creating programs, 389–399
 - deleting old data, 391–392
 - displaying orders, 396–399
 - exercises, 399–400
 - installing Npgsql, 389–390
- C++, 327
- calculation parameters, business rules for, 148
- camel case, 131
- candidate key, 35, 36, 181, 672
- CAP theorem
 - about, 562
 - as a desirable feature of databases, 22
- cardinality
 - defined, 672
 - entity-relationship models (ER diagram/ERD)
 - and, 133–134
 - semantic object models (SOMs) and, 120
- Cartesian product, as a database operation, 40
- case sensitivity
 - in entity-relationship models (ER diagram/ERD), 131
 - in PostgreSQL, 376
 - in SQL, 495
- catalog, 672
- categories, of SQL scripts, 520–522
- cd command, 469, 470
- changeable data models, 198
- CHAR data type, 499
- check constraints, 37–38, 672
- CIH virus, 535
- classes, semantic object models (SOMs) and, 119–120, 124–129
- cleanup scripts, 521
- CLOSE command, 497
- closing connections, in PostgreSQL, 384
- cloud
 - considerations, as a desirable feature of databases, 22–23
 - defined, 672
 - NoSQL databases and, 47–49
- Codd, Edgar (researcher), 675
- column family databases. *See* column-oriented databases
- column-oriented databases
 - about, 53
 - defined, 672
 - uses for, 75
- columns, in relational databases, 32–34
- commands, in Structured Query Language (SQL), 495–498
- commas, in tuples, 346
- comma-separated value (CSV) files, 673
- comments
 - entity-relationship models (ER diagram/ERD)
 - and, 137
 - semantic object models (SOMs) and, 129–130
- commit, 672
- COMMIT command, 498
- Commit method, 363
- communication networks, 55
- compacting databases, 538
- complex calculations, ability to perform, as a desirable feature of databases, 21–22, 26
- complex object, 124
- complicated checks, business rules for, 148–149
- composite index. *See* concatenated index
- composite key, 35, 36, 672, 673
- composite objects, 124–125, 673
- compound index. *See* concatenated index
- compound key. *See* composite key
- compound objects, 125, 673

- concatenated index, 672, 673
- concatenated key. *See* composite key
- Concurrent Versions System (CVS), 521, 655
- configuration, initial, privileges and, 553
- connections
 - closing in PostgreSQL, 384
 - to databases in PostgreSQL, 379–380, 390
 - finding in MongoDB Atlas, 436–439
- connect_to_db method, 440, 441–442
- consistency
 - in CAP theorem, 22
 - defined, 673
 - as a desirable feature of databases, 10, 25
 - importance of, 48–49
 - in NoSQL databases, 198
- constraints
 - about, 37
 - address, 305–306
 - check, 37–38
 - domain, 37
 - foreign key, 38–39
 - not enforcing, as a design pitfall, 253
 - primary key, 38
 - unique, 38
- contacts, database design for, 665–666
- continuous shadowing, 16
- conventions, entity-relationship models (ER diagram/ERD) and, 136–137
- Convert, 90
- converting
 - domains into tables, 205–206
 - entity-relationship models, 140–141
 - semantic object models, 138–140
- Coordinated Universal Time (UCT), 68
- cost, as a desirable feature of databases, 18–19, 26
- Create, Read, Update, and Delete (CRUD)
 - about, 497
 - defined, 673
 - as a desirable feature of databases, 9, 25
- CREATE command, 407, 408, 420, 421, 496
- CREATE DATABASE statement, 519
- CREATE INDEX statement, in Structured Query Language (SQL), 503–504
- CREATE TABLE statement
 - about, 348, 360, 500–503
 - in Structured Query Language (SQL), 498–503
- CreateCustomerRecords method, 392, 396
- create_data method, 440, 442–444, 448
- CreateData method, 360–362, 455, 457–458, 464
- CreateDatabase method, 356–357
- CreateOrderItemRecords method, 395–396
- CreateOrderRecords method, 393–394, 396
- CreateTables method, 358–359
- creating
 - backup plans, 536–537
 - customer data in PostgreSQL, 380–381, 392–393
 - data in MariaDB, 348–350, 360–363
 - databases
 - in MariaDB, 356–358
 - in PostgreSQL, 373–374
 - MariaDB databases, 344–346
 - order data in PostgreSQL, 382–383, 393–395
 - order item data in PostgreSQL, 383–384, 395–396
 - programs
 - in Apache Ignite, 470–474, 477–483
 - in MariaDB, 343–352, 355–365
 - in MongoDB Atlas, 439–450, 454–465
 - in Neo4j AuraDB, 405–413, 418–427
 - in PostgreSQL, 378–386, 389–399
 - requirements document, 101
 - use cases, 102–105
 - users in PostgreSQL, 371–372
- Crockford, Douglas, 433
- cross-record checks, business rules for, 148
- cross-table checks, business rules for, 148
- crow’s foot symbol, 133
- current operations, studying, 94
- cursor, 673
- Customer Champion, 89
- customer data, creating in PostgreSQL, 380–381, 392–393
- customer orders, database design for, 656
- Customer Representative, 89
- customers
 - brainstorming with, 265–275
 - meeting, 88–89, 263–265
 - picking the brains of, 93
 - understanding reasoning of, 96–97
- cyclic dependency, 673
- Cypher, in Neo4j AuraDB, 404–405

D

- data. *See also* temporal data
 - arranging
 - in BCNF, 184–185
 - in domain/key normal form (DKNF), 194–195
 - in fifth normal form, 192–193
 - in first normal form (1NF), 167–172
 - in fourth normal form, 188–189
 - in second normal form (2NF), 175–177
 - in third normal form (3NF), 179–181
 - creating in MariaDB, 348–350, 360–363
 - deleting in PostgreSQL, 380, 391–392
 - determining
 - how pieces of data are related, 269–271
 - needs for user interface, 268–269
 - origin of, 269
 - fetching in MariaDB, 350–352, 364–365
 - reading in Apache Ignite, 473
 - redundant, 210–211
 - saving in Apache Ignite, 471–472
- data anomalies, in NoSQL databases, 198
- Data Control Language (DCL), 495, 497–498, 673
- Data Definition Language (DDL), 495–496, 673
- data dictionary, 673
- data disasters, 20
- data elements. *See* attributes
- data initialization scripts, 520–521
- data integrity
 - determining needs for, 273–275
 - user needs and, 86–87
- Data Manipulation Language (DML), 495, 497, 673
- data mart, 673
- data mining, 673
- data models/modeling
 - about, 111–114, 142, 283, 298–300
 - building, 283–301
 - entity-relationship modeling, 130–137, 289–294
 - exercise solutions, 573–580, 605–607
 - exercises, 142–143, 300–301
 - relational modeling, 137–141, 294–298
 - semantic object modeling, 118–130, 283–288
 - translating user needs into, 111–143
 - user interface models, 114–118
- data needs, user needs and, 86
- data normalization
 - about, 163–164, 199, 311, 323–324
 - best level of, 197
 - Boyce-Codd Normal Form (BCNF), 181–185
 - defined, 678
 - domain/key normal form (DKNF), 193–195
 - essential redundancy, 195–197
 - exercise solutions, 587–592, 610–613
 - exercises, 199–201, 324
 - fifth normal form (5NF), 190–193
 - first normal form (1NF), 164–172
 - fourth normal form (4NF), 185–189
 - improving flexibility, 311–315
 - insufficient, as a design pitfall, 248–249
 - normal *vs.* abnormal, 432
 - NoSQL, 197–198
 - second normal form (2NF), 173–177
 - stopping at third normal form, 181
 - third normal form (3NF), 177–181
 - too much, as a design pitfall, 248
 - verifying
 - first normal form, 315–318
 - second normal form, 318–321
 - third normal form, 321–323
- data residency, 19
- data scrubbing, 674
- data sovereignty, 19
- data tier, 154
- data types
 - about, 499
 - business rules for, 149
 - of columns, 32–33
 - defined, 674
- data warehouse
 - for database maintenance, 537–538
 - defined, 674
- database adapters, 332–333
- database administrator (DBA), 30, 674
- database connector. *See* database adapters
- database creation scripts, 520
- database design
 - about, 26–27, 212–213, 241, 257
 - allowing redundant data, 210–211
 - avoiding pitfalls of, 241–259

- consequences of good and bad, 24–26
- converting domains into tables, 205–206
- of databases for software support, 203–214
- desirable features, 9–26
- documenting everything, 204–205
- exercise solutions, 557–562, 592–593, 598–600
- exercises, 27–28, 213–214, 257–259
- focused tables, 206–207
- goals for, 3–28
- importance of, 4–5
- information containers, 6–8
- insufficient normalization for, 248–249
- insufficient testing for, 249
- lack of preparation for, 241–242
- mishmash tables for, 250–252
- multi-tier architecture, 205
- naming conventions, 209–210
- not defining natural keys with, 256–257
- not enforcing constraints for, 253
- not planning for change with, 245–247
- not squeezing in everything, 211–212
- obsession with IDs for, 253–256
- performance anxiety for, 249–250
- planning ahead, 204
- poor documentation for, 242
- poor naming standards for, 242–244
- in PostgreSQL, 371
- samples, 649–669
- strengths and weaknesses of information containers, 8–9
- thinking too small for, 244–245
- too much normalization with, 248
- types of tables, 207–209
- database design patterns
 - about, 215, 238
 - associations, 216–231
 - exercise solutions, 594–597
 - exercises, 238–240
 - locking, 236–238
 - logging, 236–238
 - temporal data, 232–236
- database engine, 332
- database maintenance
 - about, 533, 542
 - backups, 533–537
 - compacting databases, 538
 - data warehousing, 537–538
 - exercise solutions, 640–641
 - exercises, 542–543
 - performance tuning, 538–542
 - repairing databases, 538
- database management systems (DBMSs), 30
- database operations, 40–41
- database security
 - about, 545, 555–556
 - exercise solutions, 642–648
 - exercises, 556
 - initial configuration and privileges, 553
 - levels of, 545
 - passwords, 546–548
 - physical security, 554–555
 - privileges, 548–553
 - too much, 553–554
- database tier/layer, 154
- database transaction units (DTUs), 18–19
- databases
 - building with SQL scripts, 519–532
 - compacting, 538
 - connecting to, in PostgreSQL, 379–380, 390
 - creating
 - in MariaDB, 356–358
 - in PostgreSQL, 373–374
 - deductive, 70
 - defined, 6, 674
 - designing
 - in PostgreSQL, 371
 - for software support, 203–214
 - desirable features of, 9–26
 - dimensional, 70–71
 - exercise solutions, 563–564
 - MariaDB, creating, 344–346
 - object, 70
 - object-oriented, 70
 - repairing, 538
 - semi-structured, 64
 - temporal, 71–72
- DATE data type, 499
- DATETIME data type, 499
- DECIMAL data type, 499
- DECLARE command, 497

- dedicated service, 434
 - deductive database, 70
 - defining
 - Building class in Apache Ignite, 471
 - tables
 - about, 346–348
 - in MariaDB, 358–360
 - in PostgreSQL, 374–378
 - The Definitive Guide to Dimensional Modeling, 3rd Edition* (Kimball), 538
 - DELETE command, 380, 497
 - DELETE FROM statement, 350, 380, 392
 - DELETE statement
 - about, 391, 392
 - in Structured Query Language (SQL), 514–515
 - delete_all_nodes method, 406–407, 412
 - DeleteAllNodes method, 419–420
 - deleted objects, temporal data and, 233–234
 - DeleteMany method, 456
 - delete_old_data method, 440, 442
 - DeleteOldData method, 455, 456
 - DeleteRecords method, 391
 - deleting
 - data in PostgreSQL, 380
 - defined, 674
 - old data in PostgreSQL, 391–392
 - deletion anomaly, 174, 674
 - deliverables, 101
 - demanding feedback, 280–281
 - demonstrating
 - persistent data in Apache Ignite, 474, 483
 - volatile data in Apache Ignite, 473–474, 483
 - determinant, 182, 674
 - development, methodologies for, 99
 - Devil’s Advocate, 90
 - difference, as a database operation, 40
 - differential backup, 534
 - dimensional database, 70–71, 674
 - directed relationships, 404
 - displaying orders, in PostgreSQL, 396–399
 - DisplayOrders method, 396–397
 - Dispose method, 357, 363
 - distinct object rental, database design for, 660–661
 - distributed database, 674
 - divide, as a database operation, 40
 - document databases
 - about, 51–52, 490
 - defined, 674
 - uses for, 75
 - document management, database design for, 655–656
 - document stores. *See* document databases
 - documentation
 - of everything, 204–205
 - poor, as a design pitfall, 242
 - document-oriented databases. *See* document databases
 - domain constraints, 37
 - domain/key normal form (DKNF), 193–195, 674
 - domains
 - of columns, 32
 - converting into tables, 205–206
 - defined, 674
 - double quotes, 505
 - drawing relational models, 310
 - DROP DATABASE statement, 519
 - DROP statement, 496, 504
 - DROP TABLE statement, 347, 357, 359–360, 494, 504
 - durability, 15, 674
- ## E
- ease of use, as a desirable feature of databases, 19, 26
 - Easter egg, 675
 - e-commerce programs, 55
 - edges, 54, 675
 - broken, 293–294
 - defined, 680
 - entity-relationship models (ER diagram/ERD)
 - and, 132
 - in Neo4j AuraDB, 404
 - effective dates, temporal data and, 232–233
 - employee shifts and timesheets, database design for, 656
 - employee skills and qualifications, database design for, 657–658
 - employees, projects, and departments, database design for, 657

- enrollment use cases, 104–105
- entities
 - defined, 675
 - entity-relationship models (ER diagram/ERD)
 - and, 131–132
- entity classes, 131
- entity integrity, 675
- entity sets, 131
- entity-relationship diagram (ER diagram), 675
- entity-relationship models (ER diagram/ERD)
 - about, 289
 - additional conventions, 136–137
 - attributes, 131–132
 - building combined ER diagrams, 291–292
 - building ER diagrams, 289–290
 - cardinality, 133–134
 - case, 131
 - comments, 137
 - converting, 140–141
 - defined, 113
 - entities, 131–132
 - identifiers, 131–132
 - improving entity-relationship diagrams, 293–294
 - inheritance, 134–136
 - notes, 137
 - relationships, 132
 - translating user needs into, 130–137
- enumerated values, business rules for, 148, 149
- environment, user needs and, 88
- Error Between Chair and Keyboard (EBCAK)
 - problems, 535
- error correction, as a desirable feature of databases, 11, 25
- Essential Algorithms: A Practical Approach to Computer Algorithms Using Python and C#* (Stephens), 525
- essential redundancy, 195–197
- eventual consistency, 49
- example programs
 - about, 336–337
 - database adapters, 332–336
 - exercise solutions, 613–615
 - exercises, 337
 - Jupyter Notebook, 329–331
 - program passwords, 336
 - tool choices, 327–328
 - Visual Studio, 331–332
- EXECUTE statement, 382–384
- execute_node_query method, 406, 408, 411
- ExecuteNonQuery method, 419, 422, 425
- ExecuteNonQuery method, 357
- ExecuteScalar method, 393, 394
- Executive Champion, 89
- executive summary, 280
- exercise solutions
 - Apache Ignite
 - in C#, 633–635
 - in Python, 631–633
 - business rules, 581–587, 608–610
 - data models, 573–580, 605–607
 - data normalization, 587–592, 610–613
 - database design, 557–562, 592–593, 594–597, 598–600
 - database maintenance, 640–641
 - database security, 642–648
 - databases, 563–564
 - examples, 613–615
 - MariaDB
 - in C#, 617–619
 - in Python, 615–617
 - MongoDB Atlas
 - in C#, 630–631
 - in Python, 629–630
 - Neo4j AuraDB
 - in C#, 627–628
 - in Python, 627
 - NoSQL, 564–568
 - PostgreSQL
 - in C#, 622–626
 - in Python, 619–622
 - Structured Query Language (SQL), 635–640
 - user needs, 568–573, 600–605
- exercises
 - Apache Ignite
 - in C#, 484–485
 - in Python, 474–475
 - business rules, 159–161, 311
 - data models, 142–143, 300–301
 - data normalization, 199–201, 324
 - database design, 27–28, 213–214, 257–259
 - database maintenance, 542–543
 - database security, 556

- design patterns, 238–240
- example programs, 337
- MariaDB, 366–367
- MongoDB Atlas
 - in C#, 466
 - in Python, 450–451
- Neo4j AuraDB
 - in C#, 428–429
 - in Python, 414–415
- NoSQL databases, 78–79
- PostgreSQL, 399–400
- relational databases (RDBs), 44–46
- Structured Query Language (SQL), 515–517
 - scripts, 532
- user needs, 107–109, 281–282
- extending the partial ordering, 525
- extensibility, as a desirable feature of databases, 18–19, 26
- Extensible Markup Language (XML)
 - defined, 675
 - in document databases, 51
 - hierarchical nature of, 229
 - in MongoDB Atlas, 432–434
- extracting business rules, 152–154, 303–311

F

- feasibility, 106
- feedback, demanding, 280–281
- FETCH command, 497
- fetchall method, 351, 385, 494
- FetchData method, 364–365
- fetching data, in MariaDB, 350–352, 364–365
- field, 675
- field-level check constraint, 38
- fifth normal form (5NF), 190–193, 675
- files
 - flat, 59–60
 - JSON, 67–69
 - schema, 64
- final deliverables, 101
- Find method, 459–460
- find statement, 449
- finding
 - connection code in MongoDB Atlas, 436–439
 - more information for SQL, 491–492

- find_path method, 406, 409, 411
- FindPath method, 419, 422–423
- first normal form (1NF)
 - about, 164–172, 212
 - defined, 675
 - verifying, 315–318
- fixed values, business rules for, 149
- flat files
 - about, 59–60
 - defined, 675
 - uses for, 75
- flexibility, improving, 311–315
- flexible data models, 198
- focusing tables, 206–207
- foreign key, 675
- foreign key constraints, 38–39
- fourth normal form (4NF), 185–189, 675
- FROM clause, in Structured Query Language (SQL), 507–511
- full backup, 534
- functionality, user needs and, 85

G

- garbage-in, garbage-out (GIGO) principle, 28
- General Data Protection Regulation (GDPR), 20
- generalizable constraints, business rules for, 148
- Generic Villain, 90
- GetInt32 method, 398
- get_or_create_cache method, 472, 473
- GetOrCreateCache method, 481
- Git, 521, 655
- GitHub, 521
- global statement, 442
- globally unique identifier (GUID), 254
- goals, in use cases, 102
- Google Cloud, 235
- Google Sheets, 43
- GRANT command, 497
- graph databases
 - about, 53–56, 490
 - defined, 675
 - uses for, 75
- Graph Query Language (GQL), 675
- Gravina Island Bridge, 538

group attribute, 119
GROUP BY clause, in Structured Query Language (SQL), 511

H

Health Insurance Portability and Accountability Act (HIPAA), 23
HeidiSQL, running, 340–343
helper methods, in MongoDB Atlas, 440–449, 454–462
hierarchical data
 about, 225–226
 with NoSQL, 228–229
 working with, 226–228
hierarchical databases, 56–59
hybrid objects, 125–126, 676
hybrid online analytical processing (HOLAP), 676
hypercube, 676

I

IBM Db2, 42–43
identical object rental, database design for, 659
identifiers
 defined, 676
 entity-relationship models (ER diagram/ERD) and, 131–132
 semantic object models (SOMs) and, 120–121
identifying
 business rules, 303–309
 IDs, 297–298
IDs
 identifying, 297–298
 obsession with, as a design pitfall, 253–256
Ignite database adapter, installing, 478–479
implicit relationships, 404
import statement, 406, 472
improving
 entity-relationship diagrams, 293–294
 flexibility, 311–315
 semantic object models, 286–288
incremental backup, 534
indexes, 36–37, 676
individual passwords, 546–547

individual sports, database design for, 663
information containers
 about, 6–8
 strengths and weaknesses of, 8–9
inheritance, entity-relationship models (ER diagram/ERD) and, 134–136
inherited objects, 128–129
Inmon, W. H. (author)
 Building the Data Warehouse 4th Edition, 538
INNER JOIN clause, 508
insert, 676
 INSERT command, 497
 INSERT INTO statement, 350
 INSERT statement, 363, 381, 382, 383, 392, 394, 494, 504–506
insertion anomaly, 174, 676
installing
 Apache Ignite, 468
 Ignite database adapter, 478–479
 MariaDB, 340
 MongoDB Atlas, 434–436
 MongoDB database adapter, 454
 MySQLConnector, 356
 Neo4j AuraDB, 402–403
 Neo4j database adapter, 405
 Neo4j driver, 418–419
 Npgsql in PostgreSQL, 389–390
 PostgreSQL, 370–371
 Psycopg, 379
 pyignite database adapter, 471
 PyMongo database adapter, 439–440
 pymysql, 344
instance, 676
INT data type, 499
International Organization for Standardization (ISO), 492
intersection, as a database operation, 40
isolation, 15, 676
IssonArray function, 456

J

Java, 327
Java Online Analytical Processing (JOLAP), 676
JavaScript Object Notation (JSON)

- defined, 676
- in document databases, 51
- files, 67–69
- hierarchical nature of, 229
- in MongoDB Atlas, 432–434
- uses for files, 76

join

- as a database operation, 40
- defined, 676

Jupyter Notebook

- about, 329–331
- packages in, 333–334

K

Kanban manufacturing, 152

kebab case, 131

key constraint, 136, 677

keys

- defined, 676
- in relational databases, 34–36

key-value databases

- about, 52
- defined, 677
- uses for, 75

Kimball, Ralph (author)

The Definitive Guide to Dimensional Modeling, 3rd Edition, 538

L

LEFT JOIN clause, 509

legal considerations, as a desirable feature of databases, 23–24

line symbol, 133

links. *See* edges

LINQ queries, 459

Linux, 370

locking. *See* logging and locking

logging and locking

- audit trails, 236
- defined, 677
- turnkey records, 237–238

lookup tables, 206

M

main method, 355, 479

main program

- in Apache Ignite, 479–480
- in MongoDB Atlas, 449–450, 462–465
- in Neo4j AuraDB, 412–413, 426–427

maintenance, of databases. *See* database maintenance

make_link method, 406, 407–408, 410

MakeLink method, 419, 421, 424

make_node method, 406, 407, 410

MakeNode method, 419, 420–421, 424

many-to-many associations, 216–218

many-to-many relationship, 677

MariaDB

- about, 42–43, 328
- in C#
 - about, 355, 366
 - creating data, 360–363
 - creating databases, 356–358
 - creating programs, 355–365
 - defining tables, 358–360
 - exercise solutions, 617–619
 - exercises, 366–367
 - fetching data, 364–365
 - installing MySqlConnection, 356
- defined, 677
- in Python
 - about, 339–340
 - creating data, 348–350
 - creating databases, 344–346
 - creating programs, 343–352
 - defining tables, 346–348
 - exercise solutions, 615–617
 - fetching data, 350–352
 - installing, 340
 - installing pymysql, 344
 - running HeidiSQL, 340–343
- website, 491

MATCH command, 420

MATCH statement, 408, 421, 422

MaxDB, 677

meeting customers, 263–265

memo, 677

- metabase, 677
 - metadata, 677
 - methodologies, for development, 99
 - Microsoft Access, 42–43
 - Microsoft Azure SQL Database, 42–43
 - Microsoft Excel, 43
 - Microsoft SQL Server, 42–43
 - Microsoft Transact-SQL (website), 491
 - milestones. *See* deliverables
 - Minnesota Department of Transportation, 267
 - mishmash tables, as a design pitfall, 250–252
 - mission statement, 279
 - MongoDB Atlas
 - about, 328
 - in C#
 - about, 453–454, 465
 - creating programs, 454–465
 - exercise solutions, 630–631
 - exercises, 466
 - helper methods, 454–462
 - installing MongoDB database adapter, 454
 - main program, 462–465
 - defined, 677
 - in Python
 - about, 431, 450
 - Binary JavaScript Object Notation (BSON), 432–434
 - creating programs, 439–450
 - exercise solutions, 629–630
 - exercises, 450–451
 - Extensible Markup Language (XML), 432–434
 - finding connection code, 436–439
 - helper methods, 440–449
 - installing MongoDB Atlas, 434–436
 - installing PyMongo database adapter, 439–440
 - JavaScript Object Notation (JSON), 432–434
 - main program, 449–450
 - normal *vs.* abnormal, 432
 - MongoDB database adapter, installing, 454
 - movies, database design for, 653–654
 - Mueller, John (tech editor), 370
 - multidimensional analytical processing (MOLAP), 677
 - multidimensional array, 677
 - multidimensional database. *See* dimensional database
 - multiple many-to-many associations, 216–218
 - multiple-object associations, 218–221
 - multistatement commands, in Structured Query Language (SQL), 493–494
 - multi-tier applications, 154–158
 - multi-tier architecture, 205
 - multivalued dependency, 678
 - music, database design for, 654–655
 - Must, Should, Could, Won't (MOSCOW), 98
 - MySQL
 - about, 42–43
 - defined, 678
 - website, 491
 - MySQLConnector, installing, 356
- ## N
- naming conventions
 - about, 209–210
 - as a design pitfall, 242–244
 - natural keys, not defining, as a design pitfall, 256–257
 - nearline, 235
 - Neo4j AuraDB
 - about, 328
 - in C#
 - about, 417–418, 428
 - action methods, 419–423
 - creating programs, 418–427
 - exercise solutions, 627–628
 - exercises, 428–429
 - installing Neo4j driver, 418–419
 - main program, 426–427
 - org chart methods, 423–425
 - defined, 678
 - installing, 402–403
 - in Python
 - about, 401, 414
 - action methods, 405–409
 - creating programs, 405–413
 - Cypher, 404–405
 - exercise solutions, 627
 - exercises, 414–415

- installing Neo4j AuraDB, 402–403
- installing Neo4j database adapter, 405
- main program, 412–413
- nodes, 404
- org chart methods, 410–412
- relationships, 404

Neo4j database adapter, installing, 405

Neo4j driver, installing, 418–419

network data

- about, 229–231
- with NoSQL, 231

networks

- communication, 55
- street, 54–55

NewSQL, as a desirable feature of databases, 17

nodes

- defined, 53, 678
- in Neo4j AuraDB, 404
- starting in Apache Ignite, 468–470

normal flow, in use cases, 102

normalization. *See* data normalization

NoSQL databases

- about, 47, 76–77, 114
- cloud, 47–49
- column-oriented databases, 53
- deductive databases, 70
- defined, 678
- dimensional databases, 70–71
- document databases, 51–52
- exercise solutions, 564–568
- exercises, 78–79
- flat files, 59–60
- graph databases, 53–56
- hierarchical data with, 228–229
- hierarchical databases, 56–59
- JSON files, 67–68
- key-value databases, 52
- network data with, 231
- object databases, 70
- philosophy, 50
- pros and cons of, 72–76
- selecting, 50
- spreadsheets, 69
- SQL and, 489
- temporal databases, 71–72
- XML files, 60–67

NoSQL normalization, 197–198

notes

- entity-relationship models (ER diagram/ERD) and, 137
- semantic object models (SOMs) and, 129–130
- in use cases, 102

Npgsql, installing in PostgreSQL, 389–390

NuGet package, 335

null, 678

NUMBER data type, 499



object attribute, 120

object database management system (ODBMS), 70, 678

object databases, 70, 76

object identifier, 120–121

object-oriented databases, 70, 678

object-relational database (ORD), 30, 678

object-relational database management system (ORDBMS), 30

object-relational mapping, 678

objects

- defined, 678
- semantic object models (SOMs) and, 119–120

one-to-many relationship, 224–225, 679

one-to-one relationship, 223–224, 679

online analytical processing (OLAP), 679

online transaction processing (OLTP) systems, 17

openCypher, 404–405

operating system passwords, 547

operations, database, 40–41

Oracle, 42–43, 679

Oracle SQL (website), 491

ORDER BY clause

- about, 397
- in Structured Query Language (SQL), 512–513

order data, creating in PostgreSQL, 382–383, 393–395

order item data, creating in PostgreSQL, 383–384, 395–396

ordering

- SQL commands, 522–531
- tables, 524–531

orders, displaying in PostgreSQL, 396–399
org chart methods, in Neo4j AuraDB, 410–412,
423–425
outer joins, 510

P

packages
 in Jupyter Notebook, 333–334
 in Visual Studio, 334–336
participation constraint, 136, 679
partition tolerance, in CAP theorem, 22
Pascal case, 131
passengers, database design for, 667–668
passwords
 about, 546
 individual, 546–547
 operating system, 547
 for programs, 336
 quality of, 547–548
 single-password databases, 546
pause command, 469
performance
 determining needs for, 271–272
 performance anxiety as a design pitfall, 249–250
 tuning for database maintenance, 538–542
performing queries, in PostgreSQL, 384–386
persistence
 Apache Ignite and, 469, 470
 as a desirable feature of databases, 17–18, 25
persistent data, demonstrating in Apache Ignite,
 474, 483
personally identifiable information (PII), 23–24
person_string method, 440–441
PersonString method, 455–456, 465
pgAdmin, running, 371
philosophy, NoSQL, 50
physical security, 554–555
Pip, 333
pip commands, 333
planning
 for change (not), as a design pitfall, 245–247
 for user needs, 84–85
plus sign (+), 377
points of view, relational, 31–32
portability, as a desirable feature of databases,
 19–20, 26
post-conditions, in use cases, 102
PostgreSQL
 about, 42–43, 328
 in C#
 about, 389, 399
 connecting to database, 390
 creating customer data, 392–393
 creating order data, 393–395
 creating order item data, 395–396
 creating programs, 389–399
 deleting old data, 391–392
 displaying orders, 396–399
 exercise solutions, 622–626
 exercises, 399–400
 installing Npgsql, 389–390
 case sensitivity in, 376
 defined, 679
 installing, 370–371
 in Python
 about, 369–370
 closing connections, 384
 connecting to databases, 379–380
 creating customer data, 380–381
 creating databases, 373–374
 creating order data, 382–383
 creating order item data, 383–384
 creating programs, 378–386
 creating users, 371–372
 defining tables, 374–378
 deleting old data, 380
 designing databases, 371
 exercise solutions, 619–622
 installing PostgreSQL, 370–371
 installing Psycopg, 379
 performing queries, 384–386
 running pgAdmin, 371
 website, 491
 practices, XML, 64–66
 pre-conditions, in use cases, 102
 predecessor list, 525
 preparation, lack of, as a design pitfall, 241–242
 prepare method, 394, 396
 PREPARE statement, 382, 383

- primary key
 - about, 36
 - defined, 679
 - MariaDB and, 348, 360
- primary key constraints, 38, 679
- `PrintReader` method, 364–365
- prioritizing, 98–99
- privileges
 - about, 548–553
 - initial configuration and, 553
- Procedural Language/Structured Query Language (PL/SQL), 679
- product requirements document (PRD), 101
- programs
 - creating
 - in Apache Ignite, 470–474, 477–483
 - in MariaDB, 343–352, 355–365
 - in MongoDB Atlas, 439–450, 454–465
 - in Neo4j AuraDB, 405–413, 418–427
 - in PostgreSQL, 378–386, 389–399
 - passwords for, 336
- projection, as a database operation, 40
- project-join normal form. *See* fifth normal form (5NF)
- projects, determining how they should look, 267–268
- property, 54
- PyScopg, installing, 379
- pyignite database adapter, installing, 471
- pyinstall, 333
- PyMongo database adapter, installing, 439–440
- `pymysql`, installing, 344
- Python
 - about, 327
 - Apache Ignite in
 - about, 467–468, 474
 - creating programs, 470–474
 - defining `Building` class, 471
 - demonstrating persistent data, 474
 - demonstrating volatile data, 473–474
 - exercises, 474–475
 - installing Apache Ignite, 468
 - installing pyignite database adapter, 471
 - with persistence, 470
 - reading data, 473
 - saving data, 471–472
 - starting nodes, 468–470
 - without persistence, 469
 - MariaDB in
 - about, 339–340
 - creating data, 348–350
 - creating databases, 344–346
 - creating programs, 343–352
 - defining tables, 346–348
 - fetching data, 350–352
 - installing, 340
 - installing `pymysql`, 344
 - running HeidiSQL, 340–343
 - MongoDB Atlas in
 - about, 431, 450
 - Binary JavaScript Object Notation (BSON), 432–434
 - creating programs, 439–450
 - exercises, 450–451
 - Extensible Markup Language (XML), 432–434
 - finding connection code, 436–439
 - helper methods, 440–449
 - installing MongoDB Atlas, 434–436
 - installing PyMongo database adapter, 439–440
 - JavaScript Object Notation (JSON), 432–434
 - main program, 449–450
 - normal *vs.* abnormal, 432
 - Neo4j AuraDB in
 - about, 401, 414
 - action methods, 405–409
 - creating programs, 405–413
 - Cypher, 404–405
 - exercises, 414–415
 - installing Neo4j AuraDB, 402–403
 - installing Neo4j database adapter, 405
 - main program, 412–413
 - nodes, 404
 - org chart methods, 410–412
 - relationships, 404
 - PostgreSQL in
 - about, 369–370
 - closing connections, 384

- connecting to databases, 379–380
- creating customer data, 380–381
- creating databases, 373–374
- creating order data, 382–383
- creating order item data, 383–384
- creating programs, 378–386
- creating users, 371–372
- defining tables, 374–378
- deleting old data, 380
- designing databases, 371
- installing PostgreSQL, 370–371
- installing Psycopg, 379
- performing queries, 384–386
- running pgAdmin, 371

Q

- quality, of passwords, 547–548
- queries
 - defined, 679
 - performing in PostgreSQL, 384–386
- query results, 41
- query_data method, 440, 444–449
- QueryData method, 455, 458–462
- query_org_chart method, 411–412
- QueryOrgChart method, 424–425
- quotes, 505

R

- Read method, 398
- ReadData method, in Apache Ignite, 482–483
- reading data, in Apache Ignite, 473
- read_transaction method, 405, 412
- recipes, database design for, 668–669
- records, 33, 679
- recursive associations. *See* reflexive associations
- redundant data, 210–211
- referential integrity, 680
- referential integrity constraint, 38
- refining. *See* data normalization
- reflexive associations
 - about, 222–223
 - hierarchical, 225–229
 - hierarchical data with NoSQL, 228–229
 - network data, 229–231

- network data with NoSQL, 231
- one-to-many, 224–225
- one-to-one, 223–224
- regret, 680
- relational database management system (RDBMS),
 - 30, 680
- relational databases (RDBs)
 - about, 29–30, 44
 - attributes, 34
 - columns, 32–34
 - constraints, 37–39
 - database operations, 40–41
 - defined, 680
 - exercises, 44–46
 - indexes, 36–37
 - keys, 34–36
 - points of view, 31–32
 - popular, 41–43
 - pros and cons of, 71–72
 - relations, 34
 - rows, 32–34
 - selecting, 30–31
 - spreadsheets, 43
 - tables, 32–34
 - tuples, 34
 - uses for, 74
- relational models/modeling
 - about, 294–298
 - defined, 112
 - drawing, 310
 - identifying IDs, 297–298
 - translating user needs into, 137–141
- relational online analytical processing (ROLAP),
 - 680
- relations
 - defined, 680
 - in relational databases, 34
- relationships. *See* edges
- repairing databases, 538
- repeated attribute associations, 221–222
- replication, 680
- report, 680
- required values, business rules for, 149
- requirements documents
 - creating, 101
 - writing, 279–280

retrieval, as a desirable feature of databases, 9–10, 25

return merchandise authorization (RMA), 100

REVOKE command, 497

RIGHT JOIN clause, 509

ring symbol, 133

roles and responsibilities, user needs and, 89–92

roll back, 680

ROLLBACK command, 498

rows

- defined, 680
- in relational databases, 32–34

running

- HeidiSQL, 340–343
- pgAdmin, 371

S

samples, of database design, 649–669

sanity check constraints, 303

sanity checks, business rules for, 149

SAVE command, 498

savepoint, 680

saving

- data in Apache Ignite, 471–472
- scripts, 521–522

scalable data models, 198

scalar, 393

schema files, 64

schemas, 73

scripts, SQL. *See* Structured Query Language (SQL), scripts

second normal form (2NF)

- about, 173–177
- defined, 680
- verifying, 318–321

security

- considerations, as a desirable feature of databases, 23–24
- database (*See* database security)
- as a desirable feature of databases, 20–21, 26
- determining needs for, 272–273
- user needs and, 87–88

SELECT clause, 41, 506–507, 511, 512–513

SELECT command, 497, 506

SELECT statement, 491, 494, 512–513

selecting

- as a database operation, 40
- defined, 680
- NoSQL databases, 50
- relational databases, 30–31

semantic attribute, 680

semantic class, 680

semantic object (SO), 681

semantic object models/modeling (SOMs)

- about, 121–122, 283
- building initial semantic object models, 283–286
- cardinality, 120
- class types, 124–129
- classes, 119–120
- comments, 129–130
- converting, 138–140
- defined, 113, 681
- identifiers, 120–121
- improving semantic object models, 286–288
- notes, 129–130
- objects, 119–120
- semantic views, 122–124
- translating user needs into, 118–130

semantic views, semantic object models (SOMs)

- and, 122–124

semi-structured database, 64

serverless service, 434

shadowing, 16

shared service, 434

sharing, as a desirable feature of databases, 21, 26

shortestPath method, 409, 423

Short-Timer, 90

SHOW DATABASES command, 358

Sidekick/Gopher, 90

simple attribute, 119

simple objects, 124, 681

single method, 409

single quotes, 505

single-password databases, 546

snake case, 131

Snowflake, 42–43

social media apps, 55

software as a service (SaaS), 23

software support, designing databases for, 203–214

- speed, as a desirable feature of databases, 12, 25
- spilled soda, 535
- splay tree, 681
- spreadsheets
 - about, 43, 69
 - uses for, 76
- SQL server, 681
- SQLite, 42–43
- Stakeholder, 89–90
- standardization, in NoSQL databases, 198
- standards, in Structured Query Language (SQL), 492–493
- starting nodes, in Apache Ignite, 468–470
- Stephens, Rod (author)
 - Beginning Software Engineering, Second Edition*, 99, 204
 - Essential Algorithms: A Practical Approach to Computer Algorithms Using Python and C#*, 525
- stopping, at third normal form (3NF), 181
- stored efficiencies, 153
- stored procedure, 681
- street networks, 54–55
- strong consistency, 49
- Structured Query Language (SQL)
 - about, 41, 489–490, 515
 - background, 491
 - basic syntax, 495
 - case sensitivity of, 495
 - FROM clause, 507–511
 - commands, 495–498
 - CREATE INDEX statement, 503–504
 - CREATE TABLE statement, 498–503
 - defined, 681
 - DELETE statement, 514–515
 - DROP statement, 504
 - exercise solutions, 635–637
 - exercises, 515–517
 - finding more information, 491–492
 - GROUP BY clause, 511
 - INSERT statement, 504–506
 - multistatement commands, 493–494
 - ORDER BY clause, 512–513
 - scripts
 - about, 519, 531–532
 - building databases with, 519–532
 - categories of scripts, 520–522
 - exercise solutions, 637–640
 - exercises, 532
 - importance of, 519–520
 - ordering SQL commands, 522–531
 - SELECT clause, 506–507
 - SELECT command, 506
 - standards, 492–493
 - tutorials, 42, 491
 - UPDATE statement, 513–514
 - versions, 491–492
 - WHERE clause, 511
- students, courses, and grades, database design for, 661–662
- studying current operations, 94
- subclass object, 681
- SUM function, 511
- summary, in use cases, 102
- superkey, 35, 36, 181, 681
- systems, determining requirements for, 265–267

T

- table-level check constraint, 38
- tables
 - building, 500–503
 - converting domains into, 205–206
 - defined, 681
 - defining
 - about, 346–348
 - in MariaDB, 358–360
 - in PostgreSQL, 374–378
 - focusing, 206–207
 - lookup, 206
 - ordering, 524–531
 - in relational databases, 32–34
 - types of, 207–209
- teams, database design for, 662–663
- Temp directory, 56
- temporal data
 - about, 232
 - deleted objects, 233–234
 - determining what to temporalize, 234–236
 - effective dates, 232–233

temporal database, 71–72, 681
 testing, insufficient, as a design pitfall, 249
 thinking small, as a design pitfall, 244–245, 250
 third normal form (3NF)
 about, 177–181
 defined, 681
 stopping at, 181
 verifying, 321–323
 3-2-1 backup strategy, 535
 TIME data type, 499
 TIMESTAMP data type, 499
 tools, choices for, 327–328
 transaction, 681
 Transaction Control Language (TCL), 495, 498, 681
 Transact-SQL (TSQL), 681
 transitive dependency, 177, 681–682
 transitory transactions, 393
 trigger, 682
 TRUNCATE command, 497
 tuples
 commas in, 346
 defined, 682
 in relational databases, 34
 turnkey records, 237–238

U

understanding, verifying, 99–100
 Unified Modeling Language (UML), 103
 uniform resource identifier (URI), 412, 426
 union, as a database operation, 40
 unique constraints, 38, 682
 unique key, 35, 36
 universally unique identifier (UUID), 254
 unrelated multivalued dependency, 186
 update, 682
 update anomaly, 173–174, 682
 UPDATE statement, 497, 513–514
 UPDATE STATISTICS statement, 539
 use cases
 creating, 102–105
 enrollment, 104–105
 writing, 275–278
 user interface models

 defined, 113
 determining what data is needed for, 268–269
 translating user needs into, 114–118
 user interface tier/layer, 154
 user needs
 about, 83–84, 97–98, 106–107, 263, 281
 brainstorming
 about, 94–95
 with customers, 265–275
 creating requirements document, 101
 customer reasoning, 96–97
 customers, 88–89
 data integrity questions, 86–87
 data needs questions, 86
 deciding on feasibility, 106
 defining, 263–282
 demanding feedback, 280–281
 environment questions, 88
 exercise solutions, 568–573, 600–605
 exercises, 107–109, 281–282
 following customers, 93–94
 functionality questions, 85
 future predictions, 95–96
 making use cases, 102–105
 meeting customers, 263–265
 planning for, 84–85
 prioritizing, 98–99
 roles and responsibilities, 89–92
 security questions, 87–88
 studying current operations, 94
 talking to customers, 93
 translating into data models, 111–143
 verifying understanding, 99–100
 writing
 requirements document, 279–280
 use cases, 275–278
 user requirements, defining, 263–282
 users, creating in PostgreSQL, 371–372

V

valid time, 71, 682
 validity
 business rules for, 148
 as a desirable feature of databases, 10–11, 25

VARCHAR data type, 499
 vehicle fleets, database design for, 664–665
 verifying
 first normal form (1nf), 315–318
 second normal form (2NF), 318–321
 third normal form (3NF), 321–323
 understanding, 99–100
 view, 682
 view maps, 123–124
 virtual machines (VMs), 22–23
 virtual tables, 41
 Visual Basic, 266
 Visual Basic for Applications (VBA), 43
 Visual Studio
 about, 331–332
 packages in, 334–336
 volatile data, demonstrating in Apache Ignite,
 473–474, 483

W

W3School (website), 67
 weak entity, 137, 682
 websites
 Apache Ignite, 468
 Apache Subversion, 521
 AuraDB, 402
 CIH virus, 535
 Concurrent Versions System (CVS), 521, 655
 Cypher, 404
 database naming conventions, 243
 Git, 521, 655
 globally unique identifier (GUID), 254

Google Cloud, 235
 Gravina Island Bridge, 538
 Jupyter Notebook, 329
 MariaDB, 491
 Microsoft Transact-SQL, 491
 MongoDB Atlas, 434
 MySQL, 491
 openCypher, 404
 Oracle SQL, 491
 PostgreSQL, 370, 491
 SQL tutorials, 491
 SQL versions, 491–492
 Visual Basic, 264
 Visual Studio, 331
 W3School, 67
 WHERE clause, 350, 363, 397, 511, 513
 wide-column databases. *See* column-oriented
 databases
 Widenius, Michael (author), 677
 windows directory, 57
 WriteData method, 480–481, 482–483
 write_transaction method, 405, 412
 writing
 requirements documents, 279–280
 use cases, 275–278

X

XML files
 about, 60–61, 66
 basics of, 61–64
 practices, 64–66
 uses for, 75–76

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley's ebook EULA.