Dan Sullivan

NoSQL

FOR MERE MORTALS®

Software-Independent Approach!

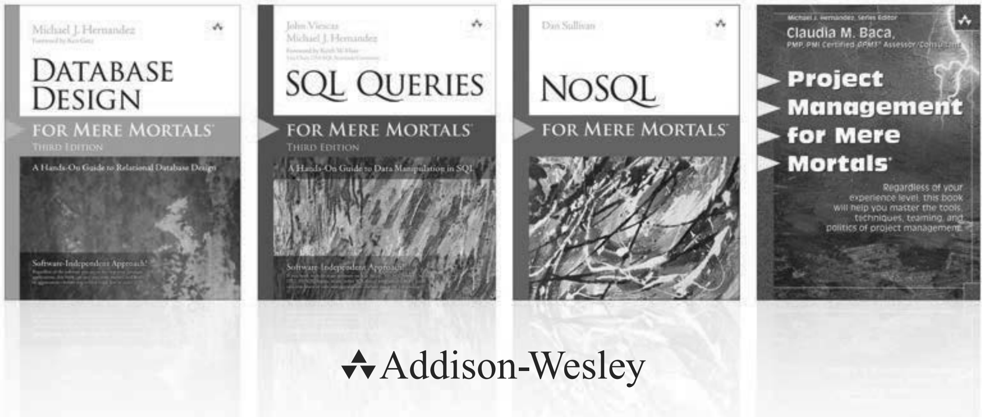
If you find yourself working around the constraints of relational databases, then a NoSQL database might be a better option. This book will help you identify and implement the best NoSQL database for your application.

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

NoSQL for Mere Mortals®

For Mere Mortals Series



Visit **informit.com/formeremortalsseries** for a complete list of available products.

The **For Mere Mortals® Series** presents you with information on important technology topics in an easily accessible, common sense manner. If you have little or no background or formal training on the subjects covered in the series, these guides are for you. This series avoids dwelling on the theoretical and instead takes you right to the heart of the topic with a matter-of-fact, hands-on approach.

"These books will give you the knowledge you need to do your work with confidence."

Mike Hernandez - Series Editor

Are you an instructor? Most *For Mere Mortals* guides have extensive teaching resources and supplements available.

Make sure to connect with us!
informit.com/socialconnect

informIT.com the trusted technology learning source | Addison-Wesley | Safari Books Online

ALWAYS LEARNING PEARSON

NoSQL for Mere Mortals®

Dan Sullivan

Addison-Wesley

Hoboken, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals. The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein. The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For questions about sales outside the U.S., please contact:

International Sales
international@pearsoned.com

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2015935038

Copyright © 2015 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-13-402321-2
ISBN-10: 0-13-402321-8

Text printed in the United States on recycled paper at Edwards Brothers Malloy, Ann Arbor, Michigan.

First printing, April 2015

Editor-in-Chief: Greg Wiegand
Acquisitions Editor: Joan Murray
Development Editor: Mark Renfrow
Managing Editor: Sandra Schroeder
Senior Project Editor: Tonya Simpson
Copy Editor: Karen Annett
Indexer: WordWise Publishing Services
Proofreader: Chuck Hutchinson
Technical Reviewer: Theodor Richardson
Editorial Assistant: Cindy Teeters
Cover Designer: Alan Clements
Compositor: Mary Sudul

For Katherine

This page intentionally left blank

About the Author

Dan Sullivan is a data architect and data scientist with more than 20 years of experience in business intelligence, machine learning, data mining, text mining, Big Data, data modeling, and application design. Dan's project work has ranged from analyzing complex genomics and proteomics data to designing and implementing numerous database applications. His most recent work has focused on NoSQL database modeling, data analysis, cloud computing, text mining, and data integration in life sciences. Dan has extensive experience in relational database design and works regularly with NoSQL databases. Dan has presented and written extensively on NoSQL, cloud computing, analytics, data warehousing, and business intelligence. He has worked in many industries, including life sciences, financial services, oil and gas, manufacturing, health care, insurance, retail, power systems, telecommunications, pharmaceuticals, and publishing.

This page intentionally left blank

Contents

Preface xxi
Introduction xxv

PART I: INTRODUCTION 1

Chapter 1 Different Databases for Different Requirements 3

Relational Database Design 4
E-commerce Application 5
Early Database Management Systems 6
Flat File Data Management Systems 7
Organization of Flat File Data Management Systems 7
Random Access of Data 9
Limitations of Flat File Data Management Systems 9
Hierarchical Data Model Systems 12
Organization of Hierarchical Data Management Systems 12
Limitations of Hierarchical Data Management Systems 14
Network Data Management Systems 14
Organization of Network Data Management Systems 15
Limitations of Network Data Management Systems 17
Summary of Early Database Management Systems 17
The Relational Database Revolution 19
Relational Database Management Systems 19
Organization of Relational Database Management Systems 20
Organization of Applications Using Relational Database Management Systems 26
Limitations of Relational Databases 27
Motivations for Not Just/No SQL (NoSQL) Databases 29
Scalability 29
Cost 31
Flexibility 31
Availability 32
Summary 34

Case Study 35
Review Questions 36
References 37
Bibliography 37
Chapter 2 Variety of NoSQL Databases 39
Data Management with Distributed Databases 41
Store Data Persistently 41
Maintain Data Consistency 42
Ensure Data Availability 44
Consistency of Database Transactions 47
Availability and Consistency in Distributed Databases 48
Balancing Response Times, Consistency, and Durability 49
Consistency, Availability, and Partitioning: The CAP Theorem 51
ACID and BASE 54
ACID: Atomicity, Consistency, Isolation, and Durability 54
BASE: Basically Available, Soft State, Eventually Consistent 56
Types of Eventual Consistency 57
Casual Consistency 57
Read-Your-Writes Consistency 57
Session Consistency 58
Monotonic Read Consistency 58
Monotonic Write Consistency 58
Four Types of NoSQL Databases 59
Key-Value Pair Databases 60
Keys 60
Values 64
Differences Between Key-Value and Relational Databases 65
Document Databases 66
Documents 66
Querying Documents 67
Differences Between Document and Relational Databases 68

Column Family Databases 69
Columns and Column Families 69
Differences Between Column Family and Relational Databases 70
Graph Databases 71
Nodes and Relationships 72
Differences Between Graph and Relational Databases 73
Summary 75
Review Questions 76
References 77
Bibliography 77

PART II: KEY-VALUE DATABASES 79

Chapter 3 Introduction to Key-Value Databases 81
From Arrays to Key-Value Databases 82
Arrays: Key Value Stores with Training Wheels 82
Associative Arrays: Taking Off the Training Wheels 84
Caches: Adding Gears to the Bike 85
In-Memory and On-Disk Key-Value Database: From Bikes to Motorized Vehicles 89
Essential Features of Key-Value Databases 91
Simplicity: Who Needs Complicated Data Models Anyway? 91
Speed: There Is No Such Thing as Too Fast 93
Scalability: Keeping Up with the Rush 95
Scaling with Master-Slave Replication 95
Scaling with Masterless Replication 98
Keys: More Than Meaningless Identifiers 103
How to Construct a Key 103
Using Keys to Locate Values 105
Hash Functions: From Keys to Locations 106
Keys Help Avoid Write Problems 107

Values: Storing Just About Any Data You Want 110
Values Do Not Require Strong Typing 110
Limitations on Searching for Values 112
Summary 114
Review Questions 115
References 116
Bibliography 116
Chapter 4 Key-Value Database Terminology 117
Key-Value Database Data Modeling Terms 118
Key 121
Value 123
Namespace 124
Partition 126
Partition Key 129
Schemaless 129
Key-Value Architecture Terms 131
Cluster 131
Ring 133
Replication 135
Key-Value Implementation Terms 137
Hash Function 137
Collision 138
Compression 139
Summary 141
Review Questions 141
References 142
Chapter 5 Designing for Key-Value Databases 143
Key Design and Partitioning 144
Keys Should Follow a Naming Convention 145
Well-Designed Keys Save Code 145
Dealing with Ranges of Values 147
Keys Must Take into Account Implementation Limitations 149
How Keys Are Used in Partitioning 150

Designing Structured Values 151
Structured Data Types Help Reduce Latency 152
Large Values Can Lead to Inefficient Read and Write Operations 155
Limitations of Key-Value Databases 159
Look Up Values by Key Only 160
Key-Value Databases Do Not Support Range Queries 161
No Standard Query Language Comparable to SQL for Relational Databases 161
Design Patterns for Key-Value Databases 162
Time to Live (TTL) Keys 163
Emulating Tables 165
Aggregates 166
Atomic Aggregates 169
Enumerable Keys 170
Indexes 171
Summary 173
Case Study: Key-Value Databases for Mobile Application Configuration 174
Review Questions 177
References 178

PART III: DOCUMENT DATABASES 179

Chapter 6 Introduction to Document Databases 181
What Is a Document? 182
Documents Are Not So Simple After All 182
Documents and Key-Value Pairs 187
Managing Multiple Documents in Collections 188
Getting Started with Collections 188
Tips on Designing Collections 191
Avoid Explicit Schema Definitions 199
Basic Operations on Document Databases 201
Inserting Documents into a Collection 202

Deleting Documents from a Collection 204
Updating Documents in a Collection 206
Retrieving Documents from a Collection 208
Summary 210
Review Questions 210
References 211
Chapter 7 Document Database Terminology 213
Document and Collection Terms 214
Document 215
Documents: Ordered Sets of Key-Value Pairs 215
Key and Value Data Types 216
Collection 217
Embedded Document 218
Schemaless 220
Schemaless Means More Flexibility 221
Schemaless Means More Responsibility 222
Polymorphic Schema 223
Types of Partitions 224
Vertical Partitioning 225
Horizontal Partitioning or Sharding 227
Separating Data with Shard Keys 229
Distributing Data with a Partitioning Algorithm 230
Data Modeling and Query Processing 232
Normalization 233
Denormalization 235
Query Processor 235
Summary 237
Review Questions 237
References 238
Chapter 8 Designing for Document Databases 239
Normalization, Denormalization, and the Search for Proper Balance 241
One-to-Many Relations 242
Many-to-Many Relations 243

The Need for Joins 243
Executing Joins: The Heavy Lifting of Relational Databases 245
Executing Joins Example 247
What Would a Document Database Modeler Do? 248
The Joy of Denormalization 249
Avoid Overusing Denormalization 251
Just Say No to Joins, Sometimes 253
Planning for Mutable Documents 255
Avoid Moving Oversized Documents 258
The Goldilocks Zone of Indexes 258
Read-Heavy Applications 259
Write-Heavy Applications 260
Modeling Common Relations 261
One-to-Many Relations in Document Databases 262
Many-to-Many Relations in Document Databases 263
Modeling Hierarchies in Document Databases 265
Parent or Child References 265
Listing All Ancestors 266
Summary 267
Case Study: Customer Manifests 269
Embed or Not Embed? 271
Choosing Indexes 271
Separate Collections by Type? 272
Review Questions 273
References 273

PART IV: COLUMN FAMILY DATABASES 275

Chapter 9 *Introduction to Column Family Databases* 277

In the Beginning, There Was Google BigTable 279
Utilizing Dynamic Control over Columns 280
Indexing by Row, Column Name, and Time Stamp 281
Controlling Location of Data 282

Reading and Writing Atomic Rows 283
Maintaining Rows in Sorted Order 284
Differences and Similarities to Key-Value and Document Databases 286
Column Family Database Features 286
Column Family Database Similarities to and Differences from Document Databases 287
Column Family Database Versus Relational Databases 289
Avoiding Multirow Transactions 290
Avoiding Subqueries 291
Architectures Used in Column Family Databases 293
HBase Architecture: Variety of Nodes 293
Cassandra Architecture: Peer-to-Peer 295
Getting the Word Around: Gossip Protocol 296
Thermodynamics and Distributed Database: Why We Need Anti-Entropy 299
Hold This for Me: Hinted Handoff 300
When to Use Column Family Databases 303
Summary 304
Review Questions 304
References 305
Chapter 10 Column Family Database Terminology 307
Basic Components of Column Family Databases 308
Keyspace 309
Row Key 309
Column 310
Column Families 312
Structures and Processes: Implementing Column Family Databases 313
Internal Structures and Configuration Parameters of Column Family Databases 313
Old Friends: Clusters and Partitions 314
Cluster 314
Partition 316

Taking a Look Under the Hood: More Column Family Database Components 317
Commit Log 317
Bloom Filter 319
Consistency Level 321
Processes and Protocols 322
Replication 322
Anti-Entropy 323
Gossip Protocol 324
Hinted Handoff 325
Summary 326
Review Questions 327
References 327
Chapter 11 Designing for Column Family Databases 329
Guidelines for Designing Tables 332
Denormalize Instead of Join 333
Make Use of Valueless Columns 334
Use Both Column Names and Column Values to Store Data 334
Model an Entity with a Single Row 335
Avoid Hotspotting in Row Keys 337
Keep an Appropriate Number of Column Value Versions 338
Avoid Complex Data Structures in Column Values 339
Guidelines for Indexing 340
When to Use Secondary Indexes Managed by the Column Family Database System 341
When to Create and Manage Secondary Indexes Using Tables 345
Tools for Working with Big Data 348
Extracting, Transforming, and Loading Big Data 350
Analyzing Big Data 351
Describing and Predicting with Statistics 351
Finding Patterns with Machine Learning 353
Tools for Analyzing Big Data 354

Tools for Monitoring Big Data 355
Summary 356
Case Study: Customer Data Analysis 357
Understanding User Needs 357
Review Questions 359
References 360

PART V: GRAPH DATABASES 361
Chapter 12 Introduction to Graph Databases 363
What Is a Graph? 363
Graphs and Network Modeling 365
Modeling Geographic Locations 365
Modeling Infectious Diseases 366
Modeling Abstract and Concrete Entities 369
Modeling Social Media 370
Advantages of Graph Databases 372
Query Faster by Avoiding Joins 372
Simplified Modeling 375
Multiple Relations Between Entities 375
Summary 376
Review Questions 376
References 377
Chapter 13 Graph Database Terminology 379
Elements of Graphs 380
Vertex 380
Edge 381
Path 383
Loop 384
Operations on Graphs 385
Union of Graphs 385
Intersection of Graphs 386
Graph Traversal 387

Properties of Graphs and Nodes 388
Isomorphism 388
Order and Size 389
Degree 390
Closeness 390
Betweenness 391
Types of Graphs 392
Undirected and Directed Graphs 392
Flow Network 393
Bipartite Graph 394
Multigraph 395
Weighted Graph 395
Summary 396
Review Questions 397
References 397
Chapter 14 Designing for Graph Databases 399
Getting Started with Graph Design 400
Designing a Social Network Graph Database 401
Queries Drive Design (Again) 405
Querying a Graph 408
Cypher: Declarative Querying 408
Gremlin: Query by Graph Traversal 410
Basic Graph Traversal 410
Traversing a Graph with Depth-First and Breadth-First Searches 412
Tips and Traps of Graph Database Design 415
Use Indexes to Improve Retrieval Time 415
Use Appropriate Types of Edges 416
Watch for Cycles When Traversing Graphs 417
Consider the Scalability of Your Graph Database 418
Summary 420
Case Study: Optimizing Transportation Routes 420
Understanding User Needs 420
Designing a Graph Analysis Solution 421

Review Questions 423
References 423

PART VI: CHOOSING A DATABASE FOR YOUR APPLICATION 425
Chapter 15 Guidelines for Selecting a Database 427
Choosing a NoSQL Database 428
Criteria for Selecting Key-Value Databases 429
Use Cases and Criteria for Selecting Document Databases 430
Use Cases and Criteria for Selecting Column Family Databases 431
Use Cases and Criteria for Selecting Graph Databases 433
Using NoSQL and Relational Databases Together 434
Summary 436
Review Questions 436
References 437

PART VII: APPENDICES 441
Appendix A: Answers to Chapter Review Questions 443
Appendix B: List of NoSQL Databases 477
Glossary 481
Index 491

Preface

"Whatever there be of progress in life comes not through adaptation but through daring."
—Henry Miller

It is difficult to avoid discussions about data. Individuals are concerned about keeping their personal data private. Companies struggle to keep data out of the hands of cybercriminals. Governments and businesses have an insatiable appetite for data. IT analysts trip over themselves coming up with new terms to describe data: Big Data, streaming data, high-velocity data, and unstructured data. There is no shortage of terms for ways to store data: databases, data stores, data warehouses, and data lakes. Someone has gone so far as to coin the phrase data swamp.

While others engage in sometimes heated discussions about data, there are those who need to collect, process, analyze, and manage data. This book is for them.

NoSQL databases emerged from unmet needs. Data management tools that worked well for decades could not keep up with demands of Internet applications. Hundreds and thousands of business professionals using corporate databases were no longer the most challenging use case. Companies such as Google, Amazon, Facebook, and Yahoo! had to meet the needs of users that measured in the millions.

The theoretically well-grounded relational data model that had served us so well needed help. Specialized applications, like Web crawling and online shopping cart management, motivated the enhancement and creation of nonrelational databases, including key-value, document, column family, and graph databases. Relational databases are still needed and face no risk of being replaced by NoSQL databases.

Instead, NoSQL databases offer additional options with different performance and functional characteristics.

This book is intended as a guide to introduce NoSQL databases, to discuss when they work well and when they do not, and, perhaps most important, to describe how to use them effectively to meet your data management needs.

You can find PowerPoints, chapter quizzes, and an accompanying instructor's guide in Pearson's Instructor Resource Center (IRC) via the website pearsonhighered.com.

Acknowledgments

This book is the product of a collaboration, not a single author as the cover may suggest. I would like to thank my editor, Joan Murray, for conceiving of this book and inviting me into the ranks of the well-respected authors and publishing professionals who have created the For Mere Mortals series.

Tonya Simpson patiently and professionally took a rough draft of *NoSQL for Mere Mortals* and turned it into a polished, finished product. Thanks to Sondra Scott, Cindy Teeters, and Mark Renfrow of Pearson for their help in seeing this book to completion. Thank you to Karen Annett for copyediting this book; I know I gave you plenty to do.

Thanks to Theodor Richardson for his thoughtful and detail-oriented technical edit.

My family was a steadfast support through the entire book writing process.

My father-in-law, Bill Aiken, is my number-one fan and my constant source of encouragement.

I am thankful for the encouragement offered by my children Nicole, Charles, and Kevin and their partners Katie and Sara.

I would like to especially thank my sons, Nicholas and James. Nicholas read chapters and completed review questions as if this were a textbook in a course. He identified weak spots and was a resource for improving the explanations throughout the text. James, a professional technology writer himself, helped write the section on graph databases. He did not hesitate to make time in his schedule for yet another unexpected request for help from his father, and as a result, the quality of those chapters improved.

Neither this book nor the other professional and personal accomplishments I have had over the past three decades could have occurred without the ever-present love and support of my partner, Katherine. Others cannot know, and probably do not even suspect, that much of what I appear to have done myself is really what we have accomplished together. This book is just one of the many products of our journey.

Dan Sullivan
Portland, Oregon
2015

Introduction

"Just when I think I have learned the way to live, life changes."
—Hugh Prather

Databases are like television. There was a time in the history of both when you had few options to choose from and all the choices were disappointingly similar. Times have changed. The database management system is no longer synonymous with relational databases, and television is no longer limited to a handful of networks broadcasting indistinguishable programs.

Names like PostgreSQL, MySQL, Oracle, Microsoft SQL Server, and IBM DB2 are well known in the IT community, even among professionals outside the data management arena. Relational databases have been the choice of data management professionals for decades. They meet the needs of businesses tracking packages and account balances as well as scientists studying bacteria and human diseases. They keep data logically organized and easily retrieved. One of their most important characteristics is their ability to give multiple users a consistent view of data no matter how many changes are under way within the database.

Many of us in the database community thought we understood how to live with databases. Then life changed. Actually, the Internet changed. The Internet emerged from a military-sponsored network called ARPANET to become a platform for academic collaboration and eventually for commercial and personal use. The volume and types of data expanded. In addition to keeping our checking account balances, we want our computers to find the latest news, help with homework, and summarize reviews of new films. Now, many of us depend on the Internet to keep in touch with family, network with colleagues, and pursue professional education and development.

It is no surprise that such radical changes in data management requirements have led to radically new ways to manage data. The latest generation of data management tools is collectively known as NoSQL databases. The name reflects what these systems are not instead of what they are. We can attribute this to the well-earned dominance of relational databases, which use a language called SQL.

NoSQL databases fall into four broad categories: key-value, document, column family, and graph databases. (Search-oriented systems, such as Solr and Elasticsearch are sometimes included in the extended family of NoSQL databases. They are outside the scope of this book.)

Key-value databases employ a simple model that enables you to store and look up a datum (also known as the value) using an identifier (also known as the key). BerkleyDB, released in the mid-1990s, was an early key-value database used in applications for which relational databases were not a good fit.

Document databases expand on the ideas of key-value databases to organize groups of key values into a logical structure known as a document. Document databases are high-performance, flexible data management systems that are increasingly used in a broad range of data management tasks.

Column family databases share superficial similarities to relational databases. The name of the first implementation of a column family database, Google BigTable, hints at the connection to relational databases and their core data structure, the table. Column family databases are used for some of the largest and most demanding, data-intensive applications.

Graph databases are well suited to modeling networks—that is, things connected to other things. The range of use cases spans computers communicating with other computers to people interacting with each other.

This is a dynamic time in database system research and development. We have well-established and widely used relational databases that are good fits for many data management problems. We have long-established alternatives, such as key-value databases, as well as more recent designs, including document, column family, and graph databases.

One of the disadvantages of this state of affairs is that decision making is more challenging. This book is designed to lessen that challenge. After reading this book, you should have an understanding of NoSQL options and when to use them.

Keep in mind that NoSQL databases are changing rapidly. By the time you read this, your favorite NoSQL database might have features not mentioned here. Watch for increasing support for transactions. How database management systems handle transactions is an important distinguishing feature of these systems. (If you are unfamiliar with transactions, don't worry. You will soon know about them if you keep reading.)

Who Should Read This Book?

This book is designed for anyone interested in learning how to use NoSQL databases. Novice database developers, seasoned relational data modelers, and experienced NoSQL developers will find something of value in this book.

Novice developers will learn basic principles and design criteria of data management in the opening chapters of the book. You'll also get a bit of data management history because, as we all know, history has a habit of repeating itself.

There are comparisons to relational databases throughout the book. If you are well versed in relational database design, these comparisons might help you quickly grasp and assess the value of NoSQL database features.

For those who have worked with some NoSQL databases, this book may help you get up to speed with other types of NoSQL databases. Key-value and document databases are widely used, but if you haven't encountered column family or graph databases, then this book can help.

If you are comfortable working with a variety of NoSQL databases but want to know more about the internals of these distributed systems, this book is a starting place. You'll become familiar with implementation features such as quorums, Bloom filters, and anti-entropy. The references will point you to resources to help you delve deeper if you'd like.

This book does not try to duplicate documentation available with NoSQL databases. There is no better place to learn how to insert data into a database than from the documentation. On the other hand, documentation rarely has the level of explanation, discussion of pros and cons, and advice about best practices provided in a book such as *NoSQL for Mere Mortals*. Read this book as a complement to, not a replacement for, database documentation.

The Purpose of This Book

The purpose of this book is to help someone with an interest in data to use NoSQL databases to help solve problems. The book is built on the assumption that the reader is not a seasoned database professional. If you are comfortable working with Excel, then you are ready for the topics covered in this book.

With this book, you'll not only learn about NoSQL databases, but also how to apply design principles and best practices to solve your data management requirements. This is a book that will take you into the internals of NoSQL database management systems to explain how distributed databases work and what to do (and not do) to build scalable, reliable applications.

The hallmark of this book is pragmatism. Everything in this book is designed to help you use NoSQL databases to solve problems. There is

a bit of computer science theory scattered through the pages but only to provide more explanation about certain key topics. If you are well versed in theory, feel free to skip over it.

How to Read This Book

For those who are new to database systems, start with Chapters 1 and 2. These will provide sufficient background to read the other chapters.

If you are familiar with relational databases and their predecessors, you can skip Chapter 1. If you are already experienced with NoSQL, you could skip Chapter 2; however, it does discuss all four major types of NoSQL databases, so you might want to at least skim the sections on types you are less familiar with.

Everyone should read Part II. It is referenced throughout the other parts of the book. Parts III, IV, and V could be read in any order, but there are some references to content in earlier chapters. To achieve the best understanding of each type of NoSQL database, read all three chapters in Parts II, III, IV, and V.

Chapter 15 assumes familiarity with the content in the other chapters, but you might be able to skip parts on NoSQL databases you are sufficiently familiar with. If your goal is to understand how to choose between NoSQL options, be sure to read Chapter 15.

How This Book Is Organized

Here's an overview of what you'll find in each part and each chapter.

Part I: Introduction

NoSQL databases did not appear out of nowhere. This part provides a background on relational databases and earlier data management systems.

Chapter 1, "Different Databases for Different Requirements," introduces relational databases and their precursor data management systems along with a discussion about today's need for the alternative approaches provided by NoSQL databases.

Chapter 2, "Variety of NoSQL Databases," explores key functionality in databases, challenges to implementing distributed databases, and the trade-offs you'll find in different types of databases. The chapter includes an introduction to a series of case studies describing realistic applications of various NoSQL databases.

Part II: Key-Value Databases

In this part, you learn how to use key-value databases and how to avoid potential problems with them.

Chapter 3, "Introduction to Key-Value Databases," provides an overview of the simplest of the NoSQL database types.

Chapter 4, "Key-Value Database Terminology," introduces the vocabulary you need to understand the structure and function of key-value databases.

Chapter 5, "Designing for Key-Value Databases," covers principles of designing key-value databases, the limitations of key-value databases, and design patterns used in key-value databases. The chapter concludes with a case study describing a realistic use case of key-value databases.

Part III: Document Databases

This part delves into the widely used document database and provides guidance on how to effectively implement document database applications.

Chapter 6, "Introduction to Document Databases," describes the basic characteristics of document databases, introduces the concept of schemaless databases, and discusses basic operations on document databases.

Chapter 7, "Document Database Terminology," acquaints you with the vocabulary of document databases.

Chapter 8, "Designing for Document Databases," delves into the benefits of normalization and denormalization, planning for mutable documents, tips on indexing, as well as common design patterns. The chapter concludes with a case study using document databases for a business application.

Part IV: Column Family Databases

This part covers Big Data applications and the need for column family databases.

Chapter 9, "Introduction to Column Family Databases," describes the Google BigTable design, the difference between key-value, document, and column family databases as well as architectures used in column family databases.

Chapter 10, "Column Family Database Terminology," introduces the vocabulary of column family databases. If you've always wondered "what is anti-entropy?" this chapter is for you.

Chapter 11, "Designing for Column Family Databases," offers guidelines for designing tables, indexing, partitioning, and working with Big Data.

Part V: Graph Databases

This part covers graph databases and use cases where they are particularly appropriate.

Chapter 12, “Introduction to Graph Databases,” discusses graph and network modeling as well as the benefits of graph databases.

Chapter 13, “Graph Database Terminology,” introduces the vocabulary of graph theory, the branch of math underlying graph databases.

Chapter 14, “Designing for Graph Databases,” covers tips for graph database design, traps to watch for, and methods for querying a graph database. This chapter concludes with a case study example of graph database applied to a business problem.

Part VI: Choosing a Database for Your Application

This part deals with applying what you have learned in the rest of the book.

Chapter 15, “Guidelines for Selecting a Database,” builds on the previous chapters to outline factors that you should consider when selecting a database for your application.

Part VII: Appendices

Appendix A, “Answers to Chapter Review Questions,” contains the review questions at the end of each chapter along with answers.

Appendix B, “List of NoSQL Databases,” provides a nonexhaustive list of NoSQL databases, many of which are open source or otherwise free to use.

The Glossary contains definitions of NoSQL terminology used throughout the book.

8

Designing for Document Databases

"Making good decisions is a crucial skill at every level."
—Peter Drucker
Author and Management Consultant

Topics Covered In This Chapter

Normalization, Denormalization, and the Search for Proper Balance

Planning for Mutable Documents

The Goldilocks Zone of Indexes

Modeling Common Relations

Case Study: Customer Manifests

Designers have many options when it comes to designing document databases. The flexible structure of JSON and XML documents is a key factor in this—flexibility. If a designer wants to embed lists within lists within a document, she can. If another designer wants to create separate collections to separate types of data, then he can. This freedom should not be construed to mean all data models are equally good—they are not.

The goal of this chapter is to help you understand ways of assessing document database models and choosing the best techniques for your needs.

Relational database designers can reference rules of normalization to help them assess data models. A typical relational data model is

designed to avoid data anomalies when inserts, updates, or deletes are performed. For example, if a database maintained multiple copies of a customer's current address, it is possible that one or more of those addresses are updated but others are not. In that case, which of the current databases is actually the current one?

In another case, if you do not store customer information separately from the customer's orders, then all records of the customer could be deleted if all her orders are deleted. The rules for avoiding these anomalies are logical and easy to learn from example.

❖ **Note** Document database modelers depend more on heuristics, or rules of thumb, when designing databases. The rules are not formal, logical rules like normalization rules. You cannot, for example, tell by looking at a description of a document database model whether or not it will perform efficiently. You must consider how users will query the database, how much inserting will be done, and how often and in what ways documents will be updated.

In this chapter, you learn about normalization and denormalization and how it applies to document database modeling. You also learn about the impact of updating documents, especially when the size of documents changes. Indexes can significantly improve query response times, but this must be balanced against the extra time that is needed to update indexes when documents are inserted or updated. Several design patterns have emerged in the practice of document database design. These are introduced and discussed toward the end of the chapter.

This chapter concludes with a case study covering the use of a document database for tracking the contents of shipments made by the fictitious transportation company introduced in earlier chapters.

Normalization, Denormalization, and the Search for Proper Balance

Unless you have worked with relational databases, you probably would not guess that normalization has to do with eliminating redundancy. Redundant data is considered a bad, or at least undesirable, thing in the theory of relational database design. Redundant data is the root of anomalies, such as two current addresses when only one is allowed.

In theory, a data modeler will want to eliminate redundancy to minimize the chance of introducing anomalies. As Albert Einstein observed, "In theory, theory and practice are the same. In practice, they are not." There are times where performance in relational databases is poor because of the normalized model. Consider the data model shown in Figure 8.1.

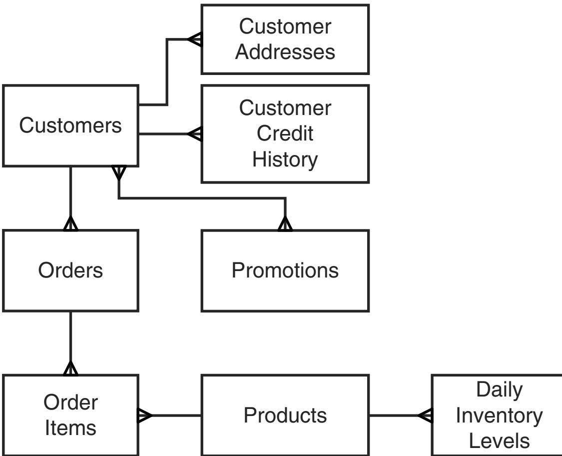


Figure 8.1 *Normalized databases have separate tables for entities. Data about entities is isolated and redundant data is avoided.*

Figure 8.1 depicts a simple normalized model of customers, orders, and products. Even this simple model requires eight tables to capture a basic set of data about the entities. These include the following:

- Customers table with fields such as name, customer ID, and so on
- Loyalty Program Members, with fields such as date joined, amount spent since joining, and customer ID
- Customer Addresses, with fields such as street, city, state, start date, end date, and customer ID
- Customer Credit Histories report with fields such as credit category, start date, end date, and customer ID
- Orders, with fields such as order ID, customer ID, ship date, and so on
- Order Items, with fields such as order ID, order item ID, product ID, quantity, cost, and so on
- Products, with fields such as product ID, product name, product description, and so on
- Daily Inventory Levels, with fields such as product ID, date, quantity available, and so on
- Promotions, with fields such as promotion ID, promotion description, start date, and so on
- Promotion to Customers, with fields such as promotion ID and customer ID

Each box in Figure 8.1 represents an entity in the data model. The lines between entities indicate the kind of relationship between the entities.

One-to-Many Relations

When a single line ends at an entity, then one of those rows participates in a single relation. When there are three branching lines ending at an entity, then there are one or more rows in that relationship. For example, the relation between Customer and Orders indicates that a

customer can have one or more orders, but there is only one customer associated with each order.

This kind of relation is called a one-to-many relationship.

Many-to-Many Relations

Now consider the relation between Customers and Promotions. There are branching lines at both ends of the relationship. This indicates that customers can have many promotions associated with them. It also means that promotions can have many customers related to them. For example, a customer might receive promotions that are targeted to all customers in their geographic area as well as promotions targeted to the types of products the customer buys most frequently.

Similarly, a promotion will likely target many customers. The sales and marketing team might create promotions designed to improve the sale of headphones by targeting all customers who bought new phones or tablets in the past three months. The team might have a special offer on Bluetooth speakers for anyone who bought a laptop or desktop computer in the last year. Again, there will be many customers in this category (at least the sales team hopes so), so there will be many customers associated with this promotion.

These types of relations are known as many-to-many relationships.

The Need for Joins

Developers of applications using relational databases often have to work with data from multiple tables. Consider the Order Items and Products entities shown in Figure 8.2.

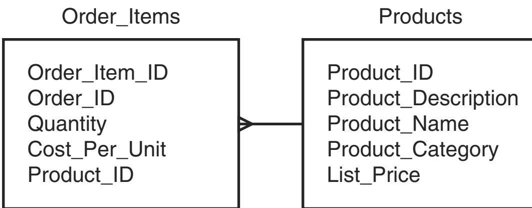


Figure 8.2 *Products and Order Items are in a one-to-many relationship. To retrieve Product data about an Order item, they need to share an attribute that serves as a common reference. In this case, Product_ID is the shared attribute.*

If you were designing a report that lists an order with all the items on the order, you would probably need to include attributes such as the name of the product, the cost per unit, and the quantity. The name of the product is in the Product table, and the other two attributes are in the Order Items table (see Figure 8.3).

❖ **Note** If you are familiar with the difference in logical and physical data models, you will notice a mix of terminology. Figures 8.1 and 8.2 depict logical models, and parts of these models are referred to as *entities* and *attributes*. If you were to write a report using the database, you would work with an implementation of the physical model.

For physical models, the terms *tables* and *columns* are used to refer to the same structures that are called *entities* and *attributes* in the logical data model. There are differences between entities and tables; for example, tables have locations on disks or in other data structures called table spaces. Entities do not have such properties.

For the purpose of this chapter, *entities* should be considered synonymous with *tables* and *attributes* should be considered synonymous with *columns*.

Order Items				
Order_Item_ID	**Order_ID**	**Quantity**	**Cost_Per_Unit**	**Product_ID**
1298	789	1	$25.99	345
1299	789	2	$20.00	372
1300	790	1	$12.50	591
1301	790	1	$20.00	372
1302	790	3	$12.99	413

Products				
Product_ID	**Product_Description**	**Product_Name**	**Product_Category**	**List_Price**
345	Easy clean tablet cover that fits most 10" Android tablets.	Easy Clean Cover	Electronic Accessories	25.99
372	Lightweight blue ear buds with comfort fit.	Acme Ear Buds	Electronic Accessories	20
413	Set of 10 dry erase markers.	10-Pack Markers	Office Supplies	15
420	60"×48" whiteboard with marker and eraser holder.	Large Whiteboard	Office Supplies	56.99
591	Pack of 100 individually wrapped screen wipes.	Screen Clean Wipes	Office Supplies	12.99

Figure 8.3 *To be joined, tables must share a common value known as a foreign key.*

In relational databases, modelers often start with designs like the one you saw earlier in Figure 8.1. Normalized models such as this minimize redundant data and avoid the potential for data anomalies. Document database designers, however, often try to store related data together in the same document. This would be equivalent to storing related data in one table of a relational database. You might wonder why data modelers choose different approaches to their design. It has to do with the trade-offs between performance and potential data anomalies.

To understand why normalizing data models can adversely affect performance, let's look at an example with multiple joins.

Executing Joins: The Heavy Lifting of Relational Databases

Imagine you are an analyst and you have decided to develop a promotion for customers who have bought electronic accessories in the past 12 months. The first thing you want to do is understand who those customers are, where they live, and how often they buy from your business. You can do this by querying the Customer table.

You do not want all customers, though—just those who have bought electronic accessories. That information is not stored in the Customer table, so you look to the Orders table. The Orders table has some information you need, such as the date of purchase. This enables you to filter for only orders made in the past 12 months.

The Orders table, however, does not have information on electronic accessories, so you look to the Order Items table. This does not have the information you are looking for, so you turn to the Products table. Here, you find the information you need. The Products table has a column called Product_Category, which indicates if a product is an electronic accessory or some other product category. You can use this column to filter for electronic accessory items.

At this point, you have all the data you need. The Customer table has information about customers, such as their names and customer IDs. The Orders table has order date information, so you can select only orders from the past 12 months. It also allows you to join to the Order_Items table, which can tell you which orders contained products in the electronic accessories category. The category information is not directly available in the Order_Items table, but you can join the Order_Items table to the Products table to get the product category (see Figure 8.4).

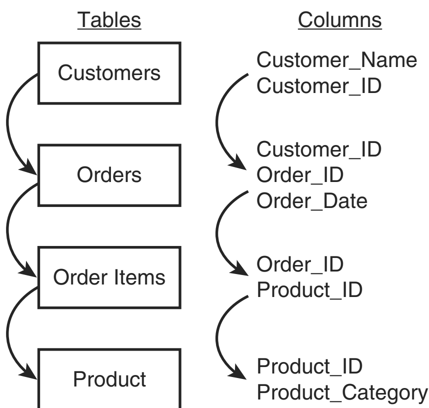


Figure 8.4 *Analyzing customers who bought a particular type of product requires three joins between four tables.*

To get a sense of how much work is involved in joining tables, let's consider pseudocode for printing the name of customers who have purchased electronic accessories in the last 12 months:

```
for cust in get_customers():
   for order in get_customer_orders(cust.customer_id):
      if today() - 365 <= order.order_date:
         for order_item in get_order_items
           (order.order_id):
          if 'electronic accessories' =
              get_product_category(order_item.product_id):
                 customer_set = add_item
                   (customer_set,cust.name);

for customer_name in customer_set:
   print customer_name;
```

In this example, the functions `get _ customers`, `get _ customer _ orders`, and `get _ order _ items` return a list of rows. In the case of `get _ customers()`, all customers are returned.

Each time `get _ customer _ orders` is called, it is given a `customer _ id`. Only orders with that customer ID are returned. Each time `get _ order _ items` is called, it is given an `order _ id`. Only order items with that `order _ id` are returned.

The dot notation indicates a field in the row returned. For example, `order.order _ date` returns the `order _ date` on a particular order. Similarly, `cust.name` returns the name of the customer currently referenced by the `cust` variable.

Executing Joins Example

Now to really see how much work is involved, let's walk through an example. Let's assume there are 10,000 customers in the database. The first `for` loop will execute 10,000 times. Each time it executes, it will look up all orders for the customer. If each of the 10,000 customers

has, on average, 10 orders, then the `for order` loop will execute 100,000 times. Each time it executes, it will check the order date.

Let's say there are 20,000 orders that have been placed in the last year. The `for order_item` loop will execute 20,000 times. It will perform a check and add a customer name to a set of customer names if at least one of the order items was an electronic accessory.

Looping through rows of tables and looking for matches is one—rather inefficient—way of performing joins. The performance of this join could be improved. For example, indexes could be used to more quickly find all orders placed within the last year. Similarly, indexes could be used to find the products that are in the electronic accessory category.

Databases implement query optimizers to come up with the best way of fetching and joining data. In addition to using indexes to narrow down the number of rows they have to work with, they may use other techniques to match rows. They could, for example, calculate hash values of foreign keys to quickly determine which rows have matching values.

The query optimizer may also sort rows first and then merge rows from multiple tables more efficiently than if the rows were not sorted. These techniques can work well in some cases and not in others. Database researchers and vendors have made advances in query optimization techniques, but executing joins on large data sets can still be time consuming and resource intensive.

What Would a Document Database Modeler Do?

Document data modelers have a different approach to data modeling than most relational database modelers. Document database modelers and application developers are probably using a document database for its scalability, its flexibility, or both. For those using document databases, avoiding data anomalies is still important, but they are willing to assume more responsibility to prevent them in return for scalability and flexibility.

For example, if there are redundant copies of customer addresses in the database, an application developer could implement a customer address update function that updates all copies of an address. She would always use that function to update an address to avoid introducing a data anomaly. As you can see, developers will write more code to avoid anomalies in a document database, but will have less need for database tuning and query optimization in the future.

So how do document data modelers and application developers get better performance? They minimize the need for joins. This process is known as denormalization. The basic idea is that data models should store data that is used together in a single data structure, such as a table in a relational database or a document in a document database.

The Joy of Denormalization

To see the benefits of denormalization, let's start with a simple example: order items and products. Recall that the `Order_Items` entity had the following attributes:

- `order_item_ID`
- `order_id`
- `quantity`
- `cost_per_unit`
- `product_id`

The Products entity has the following attributes:

- `product_ID`
- `product_description`
- `product_name`
- `product_category`
- `list_price`

An example of an order items document is

```
{
order_item_ID : 834838,
   order_ID: 8827,
   quantity: 3,
   cost_per_unit: 8.50,
   product_ID: 3648
}
```

An example of a product document is

```
{
   product_ID: 3648,
   product_description: "1 package laser printer paper.
     100% recycled.",
   product_name : "Eco-friendly Printer Paper",
   product_category : "office supplies",
   list_price : 9.00
}
```

If you implemented two collections and maintained these separate documents, then you would have to query the order items collection for the order item you were interested in and then query the products document for information about the product with `product _ ID 3648`. You would perform two lookups to get the information you need about one order item.

By denormalizing the design, you could create a collection of documents that would require only one lookup operation. A denormalized version of the order item collection would have, for example:

```
 {
order_item_ID : 834838,
   order_ID: 8827,
   quantity: 3,
   cost_per_unit: 8.50,
   product :
        {
```

```
              product_description: "1 package laser printer
                paper. 100% recycled.",
              product_name : "Eco-friendly Printer Paper",
              product_category : "office supplies",
              list_price : 9.00
         }
}
```

❖ **Note** Notice that you no longer need to maintain `product _ ID` fields. Those were used as database references (or foreign keys in relational database parlance) in the `Order _ Items` document.

Avoid Overusing Denormalization

Denormalization, like all good things, can be used in excess. The goal is to keep data that is frequently used together in the document. This allows the document database to minimize the number of times it must read from persistent storage, a relatively slow process even when using solid state devices (SSDs). At the same time, you do not want to allow extraneous information to creep into your denormalized collection (see Figure 8.5).

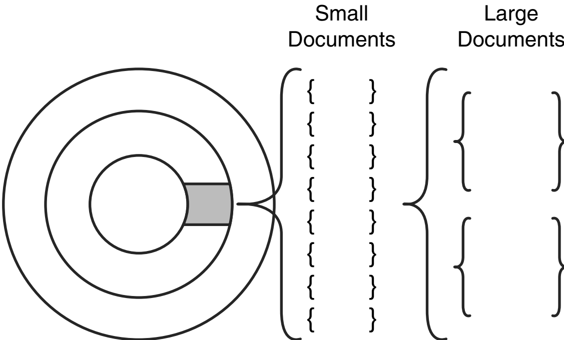


Figure 8.5 *Large documents can lead to fewer documents retrieved when a block of data is read from persistent storage. This can increase the total number of data block reads to retrieve a collection or subset of collections.*

To answer the question "how much denormalization is too much?" you should consider the queries your application will issue to the document database.

Let's assume you will use two types of queries: one to generate invoices and packing slips for customers and one to generate management reports. Also, assume that 95% of the queries will be in the invoice and packing slip category and 5% of the queries will be for management reports.

Invoices and packing slips should include, among other fields, the following:

- order_ID
- quantity
- cost_per_unit
- product_name

Management reports tend to aggregate information across groups or categories. For these reports, queries would include product category information along with aggregate measures, such as total number sold. A management report showing the top 25 selling products would likely include a product description.

Based on these query requirements, you might decide it is better to not store product description, list price, and product category in the `Order_Items` collection. The next version of the `Order_Items` document would then look like this:

```
{
   order_item_ID : 834838,
   order_ID: 8827,
   quantity: 3,
   cost_per_unit: 8.50,
   product_name : "Eco-friendly Printer Paper"
}
```

and we would maintain a `Products` collection with all the relevant product details; for example:

```
{
     product_description: "1 package laser printer paper.
       100% recycled.",
     product_name : "Eco-friendly Printer Paper",
     product_category : 'office supplies',
     list_price : 9.00
   }
```

`Product _ name` is stored redundantly in both the `Order _ Items` collection and in the `Products` collection. This model uses slightly more storage but allows application developers to retrieve information for the bulk of their queries in a single lookup operation.

Just Say No to Joins, Sometimes

Never say never when designing NoSQL models. There are best practices, guidelines, and design patterns that will help you build scalable and maintainable applications. None of them should be followed dogmatically, especially in the presence of evidence that breaking those best practices, guidelines, or design patterns will give your application better performance, more functionality, or greater maintainability.

If your application requirements are such that storing related information in two or more collections is an optimal design choice, then make that choice. You can implement joins in your application code. A worst-case scenario is joining two large collections with two `for` loops, such as

```
for doc1 in collection1:
   for doc2 in collection2:
         <do something with both documents>
```

If there are *N* documents in collection1 and *M* documents in collection2, this statement would execute $N \times M$ times. The execution time for such loops can grow quickly. If the first collection has 100,000 documents and the second has 500,000, then the statement would execute 50,000,000,000 (5×10^5) times. If you are dealing with collections

this large, you will want to use indexes, filtering, and, in some cases, sorting to optimize your join by reducing the number of overall operations performed (see Figure 8.6).

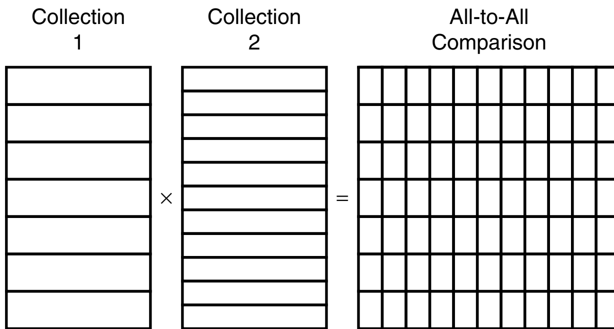


Figure 8.6 *Simple join operations that compare all documents in one collection to all documents in another collection can lead to poor performance on large collections. Joins such as this can be improved by using indexes, filtering, and, in some cases, sorting.*

Normalization is a useful technique for reducing the chances of introducing data anomalies. Denormalization is also useful, but for (obviously) different reasons. Specifically, denormalization is employed to improve query performance. When using document databases, data modelers and developers often employ denormalization as readily as relational data modelers employ normalization.

❖ **Tip** Remember to use your queries as a guide to help strike the right balance of normalization and denormalization. Too much of either can adversely affect performance. Too much normalization leads to queries requiring joins. Too much denormalization leads to large documents that will likely lead to unnecessary data reads from persistent storage and other adverse effects.

There is another less-obvious consideration to keep in mind when designing documents and collections: the potential for documents to change size. Documents that are likely to change size are known as mutable documents.

Planning for Mutable Documents

Things change. Things have been changing since the Big Bang. Things will most likely continue to change. It helps to keep these facts in mind when designing databases.

Some documents will change frequently, and others will change infrequently. A document that keeps a counter of the number of times a web page is viewed could change hundreds of times per minute. A table that stores server event log data may only change when there is an error in the load process that copies event data from a server to the document database. When designing a document database, consider not just how frequently a document will change, but also how the size of the document may change.

Incrementing a counter or correcting an error in a field will not significantly change the size of a document. However, consider the following scenarios:

- Trucks in a company fleet transmit location, fuel consumption, and other operating metrics every three minutes to a fleet management database.
- The price of every stock traded on every exchange in the world is checked every minute. If there is a change since the last check, the new price information is written to the database.
- A stream of social networking posts is streamed to an application, which summarizes the number of posts; overall sentiment of the post; and the names of any companies, celebrities, public officials, or organizations. The database is continuously updated with this information.

Over time, the number of data sets written to the database increases. How should an application designer structure the documents to handle such input streams? One option is to create a new document for each

new set of data. In the case of the trucks transmitting operational data, this would include a truck ID, time, location data, and so on:

```
{
   truck_id: 'T87V12',
   time: '08:10:00',
   date :  '27-May-2015',
   driver_name: 'Jane Washington',
   fuel_consumption_rate: '14.8 mpg',
   …
}
```

Each truck would transmit 20 data sets per hour, or assuming a 10-hour operations day, 200 data sets per day. The `truck _ id`, `date`, and `driver _ name` would be the same for all 200 documents. This looks like an obvious candidate for embedding a document with the operational data in a document about the truck used on a particular day. This could be done with an array holding the operational data documents:

```
{
   truck_id: 'T87V12',
   date :  '27-May-2015',
   driver_name: 'Jane Washington',
   operational_data:
               [
                  {time : '00:01',
                   fuel_consumption_rate: '14.8 mpg',
                   …},
                    {time : '00:04',
                   fuel_consumption_rate: '12.2 mpg',
                   …},
                    {time : '00:07',
                   fuel_consumption_rate: '15.1 mpg',
                   …},
                ...]
}
```

The document would start with a single operational record in the array, and at the end of the 10-hour shift, it would have 200 entries in the array.

From a logical modeling perspective, this is a perfectly fine way to structure the document, assuming this approach fits your query requirements. From a physical model perspective, however, there is a potential performance problem.

When a document is created, the database management system allocates a certain amount of space for the document. This is usually enough to fit the document as it exists plus some room for growth. If the document grows larger than the size allocated for it, the document may be moved to another location. This will require the database management system to read the existing document and copy it to another location, and free the previously used storage space (see Figure 8.7).

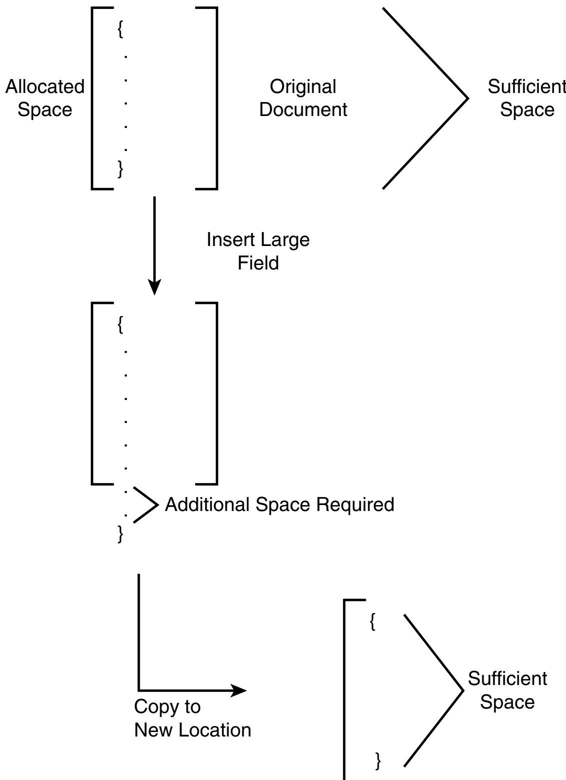


Figure 8.7 *When documents grow larger than the amount of space allocated for them, they may be moved to another location. This puts additional load on the storage systems and can adversely affect performance.*

Avoid Moving Oversized Documents

One way to avoid this problem of moving oversized documents is to allocate sufficient space for the document at the time the document is created. In the case of the truck operations document, you could create the document with an array of 200 embedded documents with the time and other fields specified with default values. When the actual data is transmitted to the database, the corresponding array entry is updated with the actual values (see Figure 8.8).

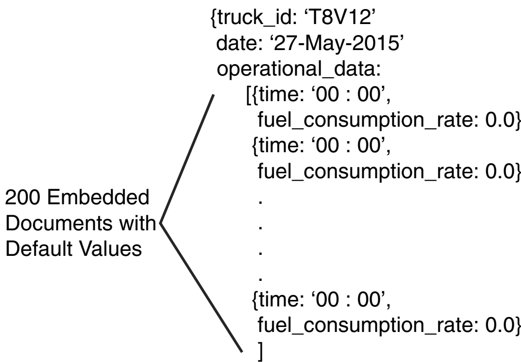


Figure 8.8 *Creating documents with sufficient space for anticipated growth reduces the need to relocate documents.*

Consider the life cycle of a document and when possible plan for anticipated growth. Creating a document with sufficient space for the full life of the document can help to avoid I/O overhead.

The Goldilocks Zone of Indexes

Astronomers have coined the term *Goldilocks Zone* to describe the zone around a star that could sustain a habitable planet. In essence, the zone that is not too close to the sun (too hot) or too far away (too cold) is just right. When you design a document database, you also want to try to identify the right number of indexes. You do not want too few, which could lead to poor read performance, and you do not want too many, which could lead to poor write performance.

Read-Heavy Applications

Some applications have a high percentage of read operations relative to the number of write operations. Business intelligence and other analytic applications can fall into this category. Read-heavy applications should have indexes on virtually all fields used to help filter results. For example, if it was common for users to query documents from a particular sales region or with order items in a certain product category, then the sales region and product category fields should be indexed.

It is sometimes difficult to know which fields will be used to filter results. This can occur in business intelligence applications. An analyst may explore data sets and choose a variety of different fields as filters. Each time he runs a new query, he may learn something new that leads him to issue another query with a different set of filter fields. This iterative process can continue as long as the analyst gains insight from queries.

Read-heavy applications can have a large number of indexes, especially when the query patterns are unknown. It is not unusual to index most fields that could be used to filter results in an analytic application (see Figure 8.9).

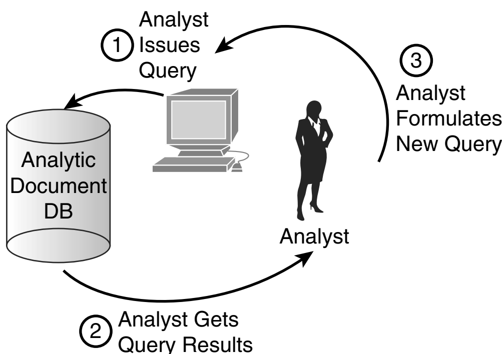


Figure 8.9 *Querying analytic databases is an iterative process. Virtually any field could potentially be used to filter results. In such cases, indexes may be created on most fields.*

Write-Heavy Applications

Write-heavy applications are those with relatively high percentages of write operations relative to read operations. The document database that receives the truck sensor data described previously would likely be a write-heavy database. Because indexes are data structures that must be created and updated, their use will consume CPU, persistent storage, and memory resources and increase the time needed to insert or update a document in the database.

Data modelers tend to try to minimize the number of indexes in write-heavy applications. Essential indexes, such as those created for fields storing the identifiers of related documents, should be in place. As with other design choices, deciding on the number of indexes in a write-heavy application is a matter of balancing competing interests.

Fewer indexes typically correlate with faster updates but potentially slower reads. If users performing read operations can tolerate some delay in receiving results, then minimizing indexes should be considered. If, however, it is important for users to have low-latency queries against a write-heavy database, consider implementing a second database that aggregates the data according to the time-intensive read queries. This is the basic model used in business intelligence.

Transaction processing systems are designed for fast writes and targeted reads. Data is copied from that database using an extraction, transformation, and load (ETL) process and placed in a data mart or data warehouse. The latter two types of databases are usually heavily indexed to improve query response time (see Figure 8.10).

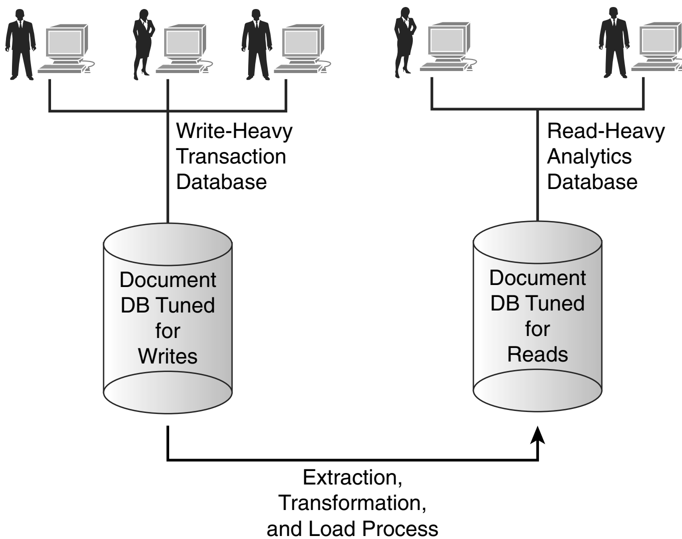


Figure 8.10 *When both write-heavy and read-heavy applications must be supported, a two-database solution may be the best option.*

❖ **Tip** Identifying the right set of indexes for your application can take some experimentation. Start with the queries you expect to support and implement indexes to reduce the time needed to execute the most important and the most frequently executed. If you find the need for both read-heavy and write-heavy applications, consider a two-database solution with one database tuned for each type.

Modeling Common Relations

As you gather requirements and design a document database, you will likely find the need for one or more of three common relations:

- One-to-many relations
- Many-to-many relations
- Hierarchies

The first two involve relations between two collections, whereas the third can entail an arbitrary number of related documents within a collection. You learned about one-to-one and one-to-many relations previously in the discussion of normalization. At that point, the focus was on the need for joins when normalizing data models. Here, the focus is on how to efficiently implement such relationships in document databases. The following sections discuss design patterns for modeling these three kinds of relations.

One-to-Many Relations in Document Databases

One-to-many relations are the simplest of the three relations. This relation occurs when an instance of an entity has one or more related instances of another entity. The following are some examples:

- One order can have many order items.
- One apartment building can have many apartments.
- One organization can have many departments.
- One product can have many parts.

This is an example in which the typical model of document database differs from that of a relational database. In the case of a one-to-many relation, both entities are modeled using a document embedded within another document. For example:

```
{
   customer_id: 76123,
   name: 'Acme Data Modeling Services',
   person_or_business: 'business',
   address : [
                    { street: '276 North Amber St',
                      city: 'Vancouver',
                      state: 'WA',
                      zip: 99076} ,
```

```
                        { street: '89 Morton St',
                          city: 'Salem',
                          state: 'NH',
                          zip: 01097}
                       ]
   }
```

The basic pattern is that the *one* entity in a one-to-many relation is the primary document, and the *many* entities are represented as an array of embedded documents. The primary document has fields about the *one* entity, and the embedded documents have fields about the *many* entities.

Many-to-Many Relations in Document Databases

A many-to-many relation occurs when instances of two entities can both be related to multiple instances of another entity. The following are some examples:

- Doctors can have many patients and patients can have many doctors.
- Operating system user groups can have many users and users can be in many operating system user groups.
- Students can be enrolled in many courses and courses can have many students enrolled.
- People can join many clubs and clubs can have many members.

Many-to-many relations are modeled using two collections—one for each type of entity. Each collection maintains a list of identifiers that reference related entities. For example, a document with course data would include an array of student IDs, and a student document would include a list of course IDs, as in the following:

Courses:

```
{
  { courseID: 'C1667',
     title: 'Introduction to Anthropology',
     instructor: 'Dr. Margret Austin',
     credits: 3,
     enrolledStudents: ['S1837', 'S3737', 'S9825' …
       'S1847'] },
  { courseID: 'C2873',
     title: 'Algorithms and Data Structures',
     instructor: 'Dr. Susan Johnson',
     credits: 3,
     enrolledStudents: ['S1837','S3737', 'S4321', 'S9825'
       … 'S1847'] },
  { courseID: C3876,
     title: 'Macroeconomics',
     instructor: 'Dr. James Schulen',
     credits: 3,
     enrolledStudents: ['S1837', 'S4321', 'S1470', 'S9825'
       … 'S1847'] },
  ...
```

Students:

```
{
 {studentID:'S1837',
   name: 'Brian Nelson',
   gradYear: 2018,
   courses: ['C1667', C2873,'C3876']},
 {studentID: 'S3737',
   name: 'Yolanda Deltor',
        gradYear: 2017,
        courses: [ 'C1667','C2873']},
    …
}
```

The pattern minimizes duplicate data by referencing related documents with identifiers instead of embedded documents.

Care must be taken when updating many-to-many relationships so that both entities are correctly updated. Also remember that document

databases will not catch referential integrity errors as a relational database will. Document databases will allow you to insert a student document with a courseID that does not correspond to an existing course.

Modeling Hierarchies in Document Databases

Hierarchies describe instances of entities in some kind of parent-child or part-subpart relation. The `product _ category` attribute introduced earlier is an example where a hierarchy could help represent relations between different product categories (see Figure 8.11).

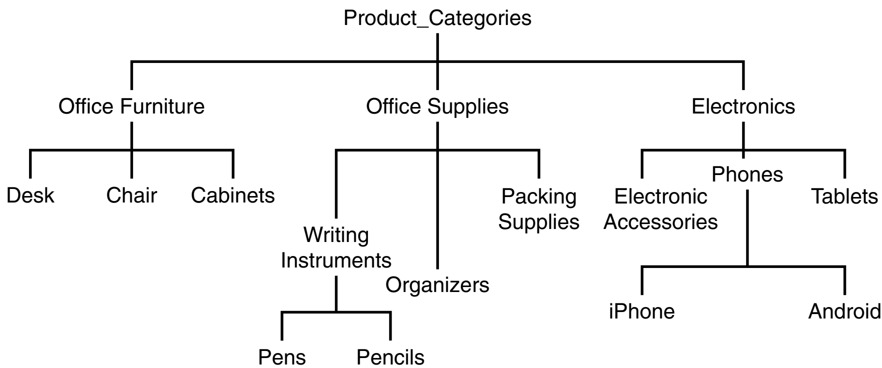


Figure 8.11 *Hierarchies describe parent-child or part-subpart relations.*

There are a few different ways to model hierarchical relations. Each works well with particular types of queries.

Parent or Child References

A simple technique is to keep a reference to either the parent or the children of an entity. Using the data depicted in Figure 8.11, you could model product categories with references to their parents:

```
{
   {productCategoryID: 'PC233', name:'Pencils',
     parentID:'PC72'},
   {productCategoryID: 'PC72', name:'Writing Instruments',
     parentID: 'PC37''},
```

```
   {productCategoryID: 'PC37', name:'Office Supplies',
     parentID: 'P01'},
   {productCategoryID: 'P01', name:'Product Categories' }
}
```

Notice that the root of the hierarchy, `'Product Categories'`, does not have a parent and so has no parent field in its document.

This pattern is useful if you frequently have to show a specific instance and then display the more general type of that category.

A similar pattern works with child references:

```
{
   {productCategoryID: 'P01', name:'Product Categories',
     childrenIDs: ['P37','P39','P41']},
    {productCategoryID: 'PC37', name:'Office Supplies',
      childrenIDs: ['PC72','PC73','PC74'']},
     {productCategoryID: 'PC72', name:'Writing
       Instruments', childrenIDs: ['PC233','PC234']'},
      {productCategoryID: 'PC233', name:'Pencils'}
}
```

The bottom nodes of the hierarchy, such as `'Pencils'`, do not have children and therefore do not have a `childrenIDs` field.

This pattern is useful if you routinely need to retrieve the children or subparts of the instance modeled in the document. For example, if you had to support a user interface that allowed users to drill down, you could use this pattern to fetch all the children or subparts of the current level of the hierarchy displayed in the interface.

Listing All Ancestors

Instead of just listing the parent in a child document, you could keep a list of all ancestors. For example, the `'Pencils'` category could be structured in a document as

```
{productCategoryID: 'PC233', name:'Pencils',
  ancestors:['PC72', 'PC37', 'P01']}
```

This pattern is useful when you have to know the full path from any point in the hierarchy back to the root.

An advantage of this pattern is that you can retrieve the full path to the root in a single read operation. Using a parent or child reference requires multiple reads, one for each additional level of the hierarchy.

A disadvantage of this approach is that changes to the hierarchy may require many write operations. The higher up in the hierarchy the change is, the more documents will have to be updated. For example, if a new level was introduced between `'Product Category'` and `'Office Supplies'`, all documents below the new entry would have to be updated. If you added a new level to the bottom of the hierarchy—for example, below `'Pencils'` you add `'Mechanical Pencils'` and `'Non-mechanical Pencils'`—then no existing documents would have to change.

❖ **Note** One-to-many, many-to-many, and hierarchies are common patterns in document databases. The patterns described here are useful in many situations, but you should always evaluate the utility of a pattern with reference to the kinds of queries you will execute and the expected changes that will occur over the lives of the documents. Patterns should support the way you will query and maintain documents by making those operations faster or less complicated than other options.

Summary

This chapter concludes the examination of document databases by considering several key issues you should consider when modeling for document databases.

Normalization and denormalization are both useful practices. Normalization helps to reduce the chance of data anomalies while denormalization is introduced to improve performance. Denormalization is a common practice in document database modeling. One of the advantages of denormalization is that it reduces or eliminates the need for joins. Joins can be complex and/or resource-intensive operations. It helps to avoid them when you can, but there will likely be times you will have to implement joins in your applications. Document databases, as a rule, do not support joins.

In addition to considering the logical aspects of modeling, you should consider the physical implementation of your design. Mutable documents, in particular, can adversely affect performance. Mutable documents that grow in size beyond the storage allocated for them may have to be moved in persistent storage, such as on disks. This need for additional writing of data can slow down your applications' update operations.

Indexes are another important implementation topic. The goal is to have the right number of indexes for your application. All instances should help improve query performance. Indexes that would help with query performance may be avoided if they would adversely impact write performance in a noticeable way. You will have to balance benefits of faster query response with the cost of slower inserts and updates when indexes are in place.

Finally, it helps to use design patterns when modeling common relations such as one-to-many, many-to-many, and hierarchies. Sometimes embedded documents are called for, whereas in other cases, references to other document identifiers are a better option when modeling these relations.

Part IV, "Column Family Databases," introduces wide column databases. These are another important type of NoSQL database and are especially important for managing large data sets with potentially billions of rows and millions of columns.

Case Study: Customer Manifests

Chapter 1, "Different Databases for Different Requirements," introduced TransGlobal Transport and Shipping (TGTS), a fictitious transportation company that coordinates the movement of goods around the globe for businesses of all sizes. As business has grown, TGTS is transporting and tracking more complicated and varied shipments. Analysts have gathered requirements and some basic estimates about the number of containers that will be shipped. They found a mix of common fields for all containers and specialized fields for different types of containers.

All containers will require a core set of fields such as customer name, origination facility, destination facility, summary of contents, number of items in container, a hazardous material indicator, an expiration date for perishable items such as fruit, a destination facility, and a delivery point of contact and contact information.

In addition, some containers will require specialized information. Hazardous materials must be accompanied by a material safety data sheet (MSDS), which includes information for emergency responders who may have to handle the hazardous materials. Perishable foods must also have details about food inspections, such as the name of the person who performed the inspection, the agency responsible for the inspection, and contact information of the agency.

The analyst found that 70%–80% of the queries would return a single manifest record. These are typically searched for by a manifest identifier or by customer name, date of shipment, and originating facility. The remaining 20%–30% would be mostly summary reports by customers showing a subset of common information. Occasionally, managers will run summary reports by type of shipment (for example, hazardous materials, perishable foods), but this is rarely needed.

Executives inform the analysts that the company has plans to substantially grow the business in the next 12 to 18 months. The analysts realize that they may have many different types of cargo in the future with specialized information, just as hazardous materials and perishable foods have specialized fields. They also realize they must plan for future scaling up and the need to support new fields in the database. They concluded that a document database that supports horizontal scaling and a flexible schema is required.

The analysts start the document and collection design process by considering fields that are common to most manifests. They decided on a collection called Manifests with the following fields:

- Customer name
- Customer contact person's name
- Customer address
- Customer phone number
- Customer fax
- Customer email
- Origination facility
- Destination facility
- Shipping date
- Expected delivery date
- Number of items in container

They also determine fields they should track for perishable foods and hazardous materials. They decide that both sets of specialized fields should be grouped into their own documents. The next question they have to decide is, should those documents be embedded with manifest documents or should they be in a separate collection?

Embed or Not Embed?

The analysts review sample reports that managers have asked for and realize that the perishable foods fields are routinely reported along with the common fields in the manifest. They decide to embed the perishable foods within the manifest document.

They review sample reports and find no reference to the MSDS for hazardous materials. They ask a number of managers and executives about this apparent oversight. They are eventually directed to a compliance officer. She explains that the MSDS is required for all hazardous materials shipments. The company must demonstrate to regulators that their database includes MSDSs and must make the information available in the event of an emergency. The compliance officer and analyst conclude they need to define an additional report for facility managers who will run the report and print MSDS information in the event of an emergency.

Because the MSDS information is infrequently used, they decide to store it in a separate collection. The Manifest collection will include a field called msdsID that will reference the corresponding MSDS document. This approach has the added benefit that the compliance officer can easily run a report listing any hazardous material shipments that do not have an msdsID. This allows her to catch any missing MSDSs and continue to comply with regulations.

Choosing Indexes

The analysts anticipate a mix of read and write operations with approximately 60%–65% reads and 35%–40% writes. They would like to maximize the speed of both reads and writes, so they carefully weigh the set of indexes to create.

Because most of the reads will be looks for single manifests, they decide to focus on that report first. The manifest identifier is a logical choice for index field because it is used to retrieve manifest doccuments.

Analysts can also look up manifests by customer name, shipment date, and origination facility. The analysts consider creating three indexes: one for each field. They realize, however, that they will rarely need to list all shipments by date or by origination facility, so they decide against separate indexes for those fields.

Instead, they create a single index on all three fields: customer name, shipment date, and origination facility. With this index, the database can determine if a manifest exists for a particular customer, shipping date, and origination facility by checking the index only; there is no need to check the actual collection of documents, thus reducing the number of read operations that have to be performed.

Separate Collections by Type?

The analysts realize that they are working with a small number of manifest types, but there may be many more in the future. For example, the company does not ship frozen goods now, but there has been discussion about providing that service. The analysts know that if you frequently filter documents by type, it can be an indicator that they should use separate collections for each type.

They soon realize they are the exception to that rule because they do not know all the types they may have. The number of types can grow quite large, and managing a large number of collections is less preferable to managing types within a single collection.

By using requirements for reports and keeping in mind some basic design principles, the analysts are able to quickly create an initial schema for tracking a complex set of shipment manifests.

Review Questions

1. What are the advantages of normalization?
2. What are the advantages of denormalization?
3. Why are joins such costly operations?
4. How do document database modelers avoid costly joins?
5. How can adding data to a document cause more work for the I/O subsystem in addition to adding the data to a document?
6. How can you, as a document database modeler, help avoid that extra work mentioned in Question 5?
7. Describe a situation where it would make sense to have many indexes on your document collections.
8. What would cause you to minimize the number of indexes on your document collection?
9. Describe how to model a many-to-many relationship.
10. Describe three ways to model hierarchies in a document database.

References

Apache Foundation. Apache CouchDB 1.6 Documentation: http://docs.couchdb.org/en/1.6.1/.

Brewer, Eric. “CAP Twelve Years Later: How the ‘Rules’ Have Changed.” Computer vol. 45, no. 2 (Feb 2012): 23–29.

Brewer, Eric A. “Towards Robust Distributed Systems.” *PODC*. vol. 7. 2000.

Chodorow, Kristina. *50 Tips and Tricks for MongoDB Developers*. Sebastopol, CA: O’Reilly Media, Inc., 2011.

Chodorow, Kristina. *MongoDB: The Definitive Guide*. Sebastopol, CA: O’Reilly Media, Inc., 2013.

Copeland, Rick. *MongoDB Applied Design Patterns*. Sebastopol, CA: O'Reilly Media, Inc., 2013.

Couchbase. Couchbase Documentation: http://docs.couchbase.com/.

Han, Jing, et al. "Survey on NoSQL Database." Pervasive computing and applications (ICPCA), 2011 6th International Conference on IEEE, 2011.

MongoDB. MongoDB 2.6 Manual: http://docs.mongodb.org/manual/.

O'Higgins, Niall. *MongoDB and Python: Patterns and Processes for the Popular Document-Oriented Database*. Sebastopol, CA: O'Reilly Media, Inc., 2011.

OrientDB. OrientDB Manual, version 2.0: http://www.orientechnologies.com/docs/last/.

Index

A

abstract/concrete entities, modeling, 369

abstract entity types, avoiding, 191-193

abstraction, 120

access, random, 9

ACID (atomicity, consistency, isolation, and durability), 54, 124, 169-170, 429, 435

addition, 118

addQueryResultsToCache function, 88

advantages of graph databases, 372-376

Aerospike, 477

aggregation, 166-169

algorithms
- compression, 140
- Dijkstra, 395
- graphs, 407
- hash functions, 137-138
- partitioning, 230

AllegroGraph, 477

Amazon Web Services, 477

analyzing
- big data, 351, 354-355
- graphs, 388
- predictions, 351-352

ancestor lists, 266

anomalies, 233, 254

anti-entropy, 299-300, 323-324

Apache
- Accumulo, 477
- Cassandra, 295, 300-302
- CouchDB, 477
- Giraph, 478
- HBase, 478

applications
- e-commerce, 5-6, 433
- RDBMSs, 26-27
- read-heavy, 259
- write-heavy, 260-261

applying
- column family databases, 303-304
- dynamic control over columns, 280
- graph databases, 385
  - *intersections, 386*
  - *traversal, 387*
  - *unions, 385*
- modelers, 248-254
- relational databases with NoSQL, 434-436

secondary indexes, 345-347
valueless columns, 334
architecture
column family databases, 293
Cassandra, 295-302
distributed databases, 299-300
gossip protocols, 296-299
HBase, 293-294
key-value databases, 131
clusters, 131-133
replication, 135-136
rings, 133
arcs. *See* edges
ArrangoDB, 478
arrays, 118
associative, 84-85
key-value databases, 82-84
atomic aggregates, 169-170. *See also* aggregation
atomic rows, reading/writing, 283-284
attributes, 120, 244
aggregation, 166-169
keys, 170-171
naming conventions, 145
automatic indexes, 341
availability
BASE, 56-59
CAP theorem, 51-54
of data, 44-48
of databases, 32-33
avoiding
abstract entity types, 191-193
complex data structures, 339-340
explicit schema definitions, 199-201
hotspotting, 337-338
joins, 372-375
moving oversized documents, 258
multirow transactions, 290-291
subqueries, 291-292
write problems, 107-110

B

bags, 215
BASE (basically available, soft state, eventually consistent), 56-59
benefits of denormalization, 249-250
betweenness, 391
big data tools, 348-356
bigraphs, 394
BigTable (Google), 279-285
bipartite graphs, 394
BLOBs (binary large objects), 123
Bloom filters, 319-320
breadth, traversing graphs, 412
Brewer's theorem. *See* CAP theorem

C

caches
key-value databases, 85-88
TTL, 163-164
CAP theorem, 51-54
case studies, key-value databases, 174-177
Cassandra, 295, 300-302, 310, 418, 478
Cassandra's Query Language. *See* CQL
child records, 12
child references, 265
closeness, 390-391

Cloudant, 478
clusters
  column family databases, 314-316
  definition of, 131-133
CODASYL (Conference on Data Systems Languages) Consortium, 17
Codd, E. F., 19
code
  keys, 145-147
  sharing, 195-198
  validation, 222
collections
  document databases
    deleting, 204-206
    inserting, 202-204
    managing in, 188-198
    retrieving, 208-209
    updating, 206-208
  indexes, 217
  multiple, 194
  terminology, 214-219
collisions, definition of, 138
column family databases, 69-71
  anti-entropy, 323-324
  applying, 303-304
  architecture, 293
    Cassandra, 295, 300-302
    distributed databases, 299-300
    gossip protocols, 296-299
    HBase, 293-294
  atomic rows, 283-284
  clusters, 314-316
  comparing to other databases, 286-292
  design
    big data tools, 348-356
    indexing, 340-348
    tables, 332-340
  dynamic control over columns, 280
  Google BigTable, 279-285
  gossip protocols, 324-325
  hinted handoffs, 325-326
  implementing, 313-322
  indexing, 281
  locations of data, 282-283
  partitions, 316
  replication, 322
  rows, 284-285
  selecting, 431-432
  terminology, 308
    columns, 310-313
    keyspaces, 309
    row keys, 309-310
columns, 121, 244, 310-312. *See also* tables
  families, 312-313
  names, 281
  storage, 334
  valueless, 334
  values
    avoiding complex data structures, 339-340
    versions, 338
commands
  remove, 204
  update, 207
commit logs, 317-318
common relations, modeling, 261
comparing
  column family databases, 286-292

graphs, 388
replicas, 323
components of RDBMSs, 20
compression, definition of, 139-140
concrete/abstract entities, modeling, 369
Conference on Data Systems Languages. *See* CODASYL Consortium
configuration
arrays, 83
collections, 191-193
column family databases
big data tools, 348-356
indexing, 340-348
tables, 332-340
databases, 29
availability, 32-33
costs, 31
early systems, 6, 17-18
flat file systems, 7-11
flexibility, 31-32
hierarchical data model systems, 12-14
network data management systems, 14-17
scalability, 29-31
document databases, 182
avoiding explicit schema definitions, 199-201
basic operations, 201
collections, 218
deleting from collections, 204-206
denormalization, 235
embedded documents, 218-219
horizontal partitions, 227-231
HTML, 182-187
inserting into collections, 202-204
key-value pairs, 187
managing in collections, 188-198
normalization, 233-234
partitions, 224
polymorphic schemas, 223
query processors, 235-236
retrieving from collections, 208-209
schemaless, 220-222
terminology, 214-217
updating in collections, 206-208
vertical partitions, 225-227
graph databases, 363-364, 400-401
advantages of, 372-376
intersections, 386
network modeling, 365-371
operations, 385
optimizing, 415-419
queries, 405-415
social networks, 401-404
traversal, 387
unions, 385
key-value databases
limitations, 159-162
partitioning, 144-151
patterns, 162-173
keys, 103
constructing, 103-104
locating values, 105-110
mobile applications, 174-177
parameters, 313
relational databases, 4-5
e-commerce applications, 5-6
history of, 19-29

secondary indexes, 345-347
structured values, 151
optimizing values, 155-159
reducing latency, 152-155
values, 110-113
consistency, 49-51
ACID, 54
BASE, 56-59
CAP theorem, 51-54
of data, 42-48
eventual, 57-59
levels, 321-322
monotonic read, 58
sessions, 58
constraints, 24
constructing keys, 103-104
conventions, naming, 145
costs, 31
Couchbase, 478
CQL (Cassandra's Query Language), 311
create function, 90
cycles, traversing graphs, 417
Cypher, 408-415

D

Data Definition Language. *See* DDL
data dictionaries, 22-23
data management. *See* management
Data Manipulation Language. *See* DML
data models, 92
data types
keys, 216-217
values, 216-217
databases
column families
anti-entropy, 323-324
applying, 303-304
architecture, 293-302
big data tools, 348-356
clusters, 314-316
columns, 310-313
comparing to other databases, 286-292
dynamic control over columns, 280
Google BigTable, 279-285
gossip protocols, 324-325
hinted handoffs, 325-326
implementing, 313-322
indexing, 281, 340-348
keyspaces, 309
locations of data, 282-283
maintaining rows in sorted order, 284-285
partitions, 316
reading/writing atomic rows, 283-284
replication, 322
row keys, 309-310
selecting, 431-432
tables, 332-340
terminology, 308
design, 4-5, 29
availability, 32-33
costs, 31
e-commerce applications, 5-6
flexibility, 31-32
scalability, 29-31
distributed, 299-300

document, 182
applying modelers, 248-254
avoiding explicit schema definitions, 199-201
balancing denormalization/ normalization, 241
basic operations, 201
collections, 218
deleting from collections, 204-206
denormalization, 235
embedded documents, 218-219
executing joins, 245-248
Goldilocks Zone of indexes, 258-260
hierarchies, 265-266
horizontal partitions, 227-231
HTML, 182-187
inserting into collections, 202-204
joins, 243-245
key-value pairs, 187
managing in collections, 188-198
many-to-many relationships, 243, 263-264
modeling common relations, 261
normalization, 233-234
one-to-many relationships, 242-263
partitions, 224
planning mutable documents, 255-258
polymorphic schemas, 223
query processors, 235-236
retrieving from collections, 208-209
schemaless, 220-222
selecting, 430
terminology, 214-217
updating in collections, 206-208
vertical partitions, 225-227

graph, 363-364
advantages of, 372-376
betweenness, 391
bigraphs, 394
closeness, 390-391
degrees, 390
design, 400-401
directed/undirected, 392-393
edges, 381-382
flow networks, 393
intersections, 386
isomorphism, 388-389
loops, 384
multigraphs, 395
network modeling, 365-371
operations, 385
optimizing, 415-419
order/size, 389
paths, 383
properties, 388
queries, 405-415
selecting, 433
social networks, 401-404
terminology, 380
traversal, 387
types of, 392
unions, 385
vertices, 380-381
weighted graphs, 395-396

key-value
architecture, 131-136
arrays, 82-84
associative arrays, 84-85

caches, 85-88
features, 91-95
implementing, 137-140
in-memory, 89-90
keys, 103-110
limitations, 159-162
models, 118-131
on-disk, 89-90
partitioning, 144-151
patterns, 162-173
scalability, 95-102
selecting, 429
values, 110-113
key-values case study, 174-177
management
early systems, 6, 17-18
flat file systems, 7-11
hierarchical data model systems, 12-14
network data management systems, 14-17
relational
history of, 19-29
using with NoSQL, 434-436
selecting, 428
types of, 59, 477-480
column family databases, 69-71
distributed databases, 41-54
document databases, 66-68
graph databases, 71-75
key-value pair databases, 60-65
DDL (Data Definition Language), 24-25
degrees, 390
delete function, 90
DELETE statements, 27
deleting documents from collections, 204-206
denormalization, 28, 155, 235
benefits of, 249-250
document database design, 241-243
overusing, 251-253
tables, 333
depth, traversing graphs, 412
design. *See also* configuration
collections, 191-193
column family databases
big data tools, 348-356
indexing, 340-348
tables, 332-340
databases, 29
availability, 32-33
costs, 31
early systems, 6, 17-18
flat file systems, 7-11
flexibility, 31-32
hierarchical data model systems, 12-14
network data management systems, 14-17
scalability, 29-31
document databases, 182
applying modelers, 248-254
avoiding explicit schema definitions, 199-201
balancing denormalization/ normalization, 241
basic operations, 201
collections, 218
deleting from collections, 204-206
denormalization, 235
embedded documents, 218-219

executing joins, 245-248
Goldilocks Zone of indexes, 258-260
hierarchies, 265-266
horizontal partitions, 227-231
HTML, 182-187
inserting into collections, 202-204
joins, 243-245
key-value pairs, 187
managing in collections, 188-198
many-to-many relationships, 243, 263-264
modeling common relations, 261
normalization, 233-234
one-to-many relationships, 242, 262-263
partitions, 224
planning mutable documents, 255-258
polymorphic schemas, 223
query processes, 235-236
retrieving from collections, 208-209
schemaless, 220-222
terminology, 214-217
updating in collections, 206-208
vertical partitions, 225-227
graph databases, 363-364, 400-401
advantages of, 372-376
intersections, 386
network modeling, 365-371
operations, 385
optimizing, 415-419
queries, 405-415
social networks, 401-404
traversal, 387
unions, 385
key-value databases
limitations, 159-162
partitioning, 144-151
patterns, 162-173
mobile applications, 174-177
relational databases, 4-5
e-commerce applications, 5-6
history of, 19-29
secondary indexes, 345-347
structured values, 151
optimizing values, 155-159
reducing latency, 152-155
Design Patterns: Elements of Reusable Object-Oriented Software, 162
dictionaries, 22-23
Dijkstra algorithms, 395
Dijkstra, Edsger, 395
directed edges, 382. *See also* edges
directed graphs, 392-393
diseases, infectious, 366-368
distributed databases, 41, 299-300
availability, 44-48
CAP theorem, 51-54
consistency, 42-48
persistent storage, 41-42
quorums, 49-51
distributing data, 230
division, 119
DML (Data Manipulation Language), 25-26
document databases, 66-68, 182
avoiding explicit schema definitions, 199-201

basic operations, 201
collections
- *deleting from, 204-206*
- *inserting into, 202-204*
- *managing, 188-198*
- *retrieving from, 208-209*
- *updating in, 206-208*

column family databases, 286-292
design
- *applying modelers, 248-254*
- *balancing denormalization/ normalization, 241*
- *executing joins, 245-248*
- *Goldilocks Zone of indexes, 258-260*
- *hierarchies, 265-266*
- *joins, 243-245*
- *many-to-many relationships, 243, 263-264*
- *modeling common relations, 261*
- *one-to-many relationships, 242, 262-263*
- *planning mutable documents, 255-258*

HTML, 182-187
key-value pairs, 187
selecting, 430
terminology, 214-217
- *collections, 218*
- *denormalization, 235*
- *embedded documents, 218-219*
- *horizontal partitions, 227-231*
- *normalization, 233-234*
- *partitions, 224*
- *polymorphic schemas, 223*
- *query processors, 235-236*
- *schemaless, 220-222*
- *vertical partitions, 225-227*

duplicating data, 155
durability, 49, 51, 54
dynamic control over columns, 280

E

early systems, database management, 6, 17-18
- flat file systems, 7-11
- hierarchical data model systems, 12-14
- network data management systems, 14-17

e-commerce, 5-6, 433
edges, 381-382
- degrees, 390
- querying, 411
- selecting, 416

elements of graphs, 380
- edges, 381-382
- loops, 384
- paths, 383
- vertices, 380-381

embedded documents, 218-219
entities, 120-121, 244
- abstract/concrete, 369
- aggregation, 166-169
- multiple between relations, 375-376
- naming conventions, 145
- single rows, 335

enumeration
- keys, 170-171
- tables, 165-166

epidemiology, 389

errors, write problems, 107-110
ETL (extracting, transforming, and loading data), 350-351
eventual consistency, types of, 57-59
executing joins, 245-248
explicit schema definitions, avoiding, 199-201
Extensible Markup Language. *See* XML
extracting, transforming, and loading data. *See* ETL

F

Facebook, 370. *See also* social media
features
  column family databases, 286
  key-value databases, 91
    keys, 103-110
    scalability, 95-102
    simplicity, 91-92
    speed, 93-95
    values, 110-113
files, flat file data management systems, 7-11
filters, Bloom, 319-320
find method, 208
flat file data management systems, 7-11
flexibility
  document databases, 190
  schemaless databases, 221
flexibility of databases, 31-32
flow networks, 393
for loops, 253
formatting. *See also* configuration
  code, 145-147
  document databases
    HTML, 182-187
    key-value pairs, 187
  secondary indexes, 345-347
  strings, 123
  values, optimizing, 155-159
FoundationDB, 478
functions
  addQueryResultsToCache, 88
  create, 90
  delete, 90
  hash, 106-107, 137-138
  indexes. *See* indexes

G

Gamma, Erich, 162
Ganglia, 355
geographic locations, modeling, 365
Global Positioning System. *See* GPS
Goldilocks Zone of indexes, 258-260
Google
  BigTable, 279-285
  Cloud Datastore, 478
gossip protocols, 296-299, 324-325
GPS (Global Positioning System), 435
graph databases, 71-75, 363-364
  advantages of, 372-376
  design, 400-401
    queries, 405-410
    social networks, 401-404
  network modeling, 365-371
  operations, 385
    intersections, 386
    traversal, 387
    unions, 385

properties, 388
betweenness, 391
closeness, 390-391
degrees, 390
order/size, 389
selecting, 433
terminology, 380
edges, 381-382
loops, 384
paths, 383
vertices, 380-381
types of, 392
bigraphs, 394
directed/undirected, 392-393
flow networks, 393
multigraphs, 395
weighted graphs, 395-396
graphs, traversal, 410-417
Gremlin, 410-418
groups, column family databases, 279
guidelines
column family databases, 431-432
databases, 428
document databases, 430
graph databases, 433
indexing, 340-348
key-value databases, 429
table design, 332-340

H

Hadoop, 285
Hadoop File System. *See* HDFS
handoffs, hinted, 300-302, 325-326
hashes, 122, 150
functions, 106-107, 137-138
partitions, 230
HBase, 285, 293-294, 478
HDFS (Hadoop File System), 293
Helm, Richard, 162
Hernandez, Michael J., 121
hierarchies
data model systems, 12-14
document databases, 265-266
hinted handoffs, 300-302, 325-326
history
early database management systems, 6, 17-18
flat file systems, 7-11
hierarchical data model systems, 12-14
network data management systems, 14-17
of relational databases, 19-29
horizontal partitioning, 227-231
hotspotting, avoiding, 337-338
HTML (Hypertext Markup Language), document databases, 182-187
Hypertable, 479

I

identifiers, keys, 104. *See also* keys
if statements, caches, 88
implementation
column family databases, 313-322
key-value databases, 137
collisions, 138
compression, 139-140
hash functions, 137-138
limitations, 149
indexes, 23, 171-173
collections, 217

column family databases, 281, 340-348
Goldilocks Zone of, 258-260
retrieval time, 415
infectious diseases, modeling, 366-368
Infinispan, 479
in-memory caches, 86. *See also* arrays, associative
in-memory key-value databases, 89-90
INSERT statements, 27
inserting documents into collections, 202-204
instances, 121, 145
internal structures, 313
intersections of graphs, 386
isolation, ACID, 54
isomorphism, 388-389

J

Jackson, Ralph, 162
JavaScript Object Notation. *See* JSON
joins
avoiding, 372-375
executing, 245-248
need for, 243-245
tables, 333
JSON (JavaScript Object Notation), 66, 123, 161

K

key-value databases, 60-65
architecture, 131
clusters, 131-133
replication, 135-136
rings, 133
arrays, 82-84
associative arrays, 84-85
caches, 85-88
case study, 174-177
column family databases, 286-292
design
partitioning, 144-151
patterns, 162-173
features, 91
scalability, 95-102
simplicity, 91-92
speed, 93-95
implementing, 137-140, 149
in-memory, 89-90
keys, 103
constructing, 103-104
locating values, 105-110
limitations, 159-162
models, 118-121
keys, 121-123
namespaces, 124-126
partitions, 126-129
schemaless, 129-131
values, 123-124
on-disk, 89-90
selecting, 429
values, 110-113
key-value pairs, 5
document databases, 187
ordered sets of, 215
keys, 60
constructing, 103-104
data types, 216-217
definition of, 121-123
enumerating, 170-171
indexes, 171-173

key-value databases, 103
naming conventions, 145
partitioning, 129, 150-151
rows, 309-310, 337-338
shard, 229
TTL, 163-164
values
locating, 105-110
searching, 160-161
keyspaces, 287, 309

L

languages
Cypher, 408-415
query (SQL), 24
DDL, 24-25
DML, 25-26
standard query, 161-162
latency, reducing, 152-155
laws of thermodynamics, 299-300
layers, abstraction, 120
least recently used. *See* LRU
LevelDB library, 140, 479
levels, consistency, 321-322
licenses, cost of, 31
limitations
of arrays, 84
of flat file data management systems, 9-11
of hierarchical data management systems, 14
of key-value databases, 159-162
of network data management systems, 17
of relational databases, 27-29
of values, 112-113
LinkedIn, 370. *See also* social media
linking records, 15
links. *See* edges
list-based partitioning, 231
lists, 122, 266
locating values, 105-110
location of data, 282-283
locations, modeling, 365
logs, commit, 317-318
loops, 384
for, 253
while, 148
LRU (least recently used), 94
Lucene, 162

M

machine learning, searching patterns, 353
magnetic tape, 7. *See also* storage
maintenance
availability of data, 44-48
consistency of data, 42-48
management
applications, 26-27
databases
design, 4-5
document databases in collections, 188-198
early systems, 6, 17-18
e-commerce applications, 5-6
flat file systems, 7-11
hierarchical data model systems, 12-14
network data management systems, 14-17
distributed databases, 41
availability, 44-48

CAP theorem, 51-54
consistency, 42-48
persistent storage, 41-42
quorums, 49-51
memory programs, 22
schemaless databases, 222
secondary indexes, 341-344
storage programs, 20-21
many-to-many relationships, 243, 263-264
mapping queries, 406
MapReduce, 354
MapReduce programs, 355
Marklogic, 479
master-slave replication, scalability, 95
masterless replication, 98-102
MATCH operation, 409
media, social, 370
memory
caches, 86. *See also* caches
management programs, 22
TTL, 163-164
methods, find, 208
Microsoft Azure DocumentDB, 479
mobile applications, configuring, 174-177
modelers, applying, 248-254
models
common relations, 261
entities, 335
hierarchies, 265-266
key-value databases, 92, 118-121
keys, 121-123
namespaces, 124-126
partition keys, 129
partitions, 126-127
schemaless, 129-131
values, 123-124
master-slave, 97
networks, 365-371
abstract/concrete entities, 369
geographic locations, 365
infectious diseases, 366-368
social media, 370
simplified, 375
MongoDB, 479
monitoring big data, 355-356
monotonic read consistency, 58
moving oversized documents, avoiding, 258
multigraphs, 395
multiple collections, 194
multiple relations between entities, 375-376
multiplication, 119
multirow transactions, avoiding, 290-291
mutable documents, planning, 255-258

N

N documents, 253
names, columns, 281, 334
namespaces
definition of, 124-126
naming conventions, 146
naming conventions, keys, 145
Neo4j, 479
networks
data management systems, 14-17
flow, 393

modeling, 365-371
abstract/concrete entities, 369
geographic location, 365
infectious diseases, 366-368
social media, 370, 401-404
nodes, 72, 363, 380
HBase, 293-294
properties, 388-389
normalization, 233-234, 241-243
NoSQL databases. *See* databases

O

on-disk key-value databases, 89-90
one-to-many relationships, 242, 262-263
operations
graph databases, 385, 388-389
betweenness, 391
closeness, 390-391
degrees, 390
intersections, 386
order/size, 389
properties, 388
traversal, 387
unions, 385
MATCH, 409
OpsCenter, 356
optimizing
graph database design, 415-419
key-value databases, 93-102
keys, 103
constructing, 103-104
locating values, 105-110
queries, 372-375
values, 110-113, 155-159
oracle Berkeley DB, 479
Oracle Real Applications Clusters. *See* RACs
ordered lists, arrays, 84. *See also* arrays
ordered sets of key-value pairs, 215
organization. *See* management; storage
OrientDB, 480
oversized documents, avoiding moving, 258
overusing denormalization, 251-253

P

parameters, configuring, 313
parent-child relationships, 15
parent references, 265
partitioning
algorithms, 230
CAP theorem, 51-54
column family databases, 314-316
definition of, 126-127
key-value databases, 144-151
keys, 129
ranges, 150
types of, 224-231
paths, 383
patterns
code, 145-147
key-value databases, 162-173
searching, 353
peer-to-peer servers, Cassandra, 295-302
performance
caches. *See* caches
duplicating data, 155
graph databases, 415-419

key-value databases, 93-102
keys, 145-147
queries, avoiding joins, 372-375
persistent data storage, 41-42
planning mutable documents, 255-258
polymorphic schemas, 223
populations, 351
predicting with statistics, 351-352
primary indexes, 341. *See also* indexes
primary keys, 104. *See also* keys
processes, column family databases
anti-entropy, 323-324
gossip protocols, 324-325
hinted handoffs, 325-326
implementing, 313-322
replication, 322
processors, queries, 235-236
programs
caches. *See* caches
memory management, 22
RDBMSs, 20
storage management, 20-21
properties, graph databases
betweenness, 391
closeness, 390-391
degrees, 390
isomorphism, 388-389
order/size, 389
traversal, 388
protocols
column family databases
anti-entropy, 323-324
replication, 322
gossip, 296-299, 324, 325

Q

queries
caches. *See* caches
Cypher, 408-415
documents, 67
graph databases, 400, 405-415
normalization, 234
processors, 235-236
ranges, 161
subqueries, avoiding, 291-292
query languages, 24
SQL DDL, 24-25
SQL DML, 25-26
quorums, 49-51

R

RACs (Oracle Real Applications Clusters), 30
random access of data, 9
ranges
key-value database design, 147-148
partitioning, 150, 230
queries, 161
RavenDB, 480
RDBMSs (relational database management systems), 19-29
read-heavy applications, 259
read/writer operations, troubleshooting, 155-159
reading
from arrays, 83
atomic rows, 283-284
records
hierarchical data management systems, 12
linking, 15

Redis, 124, 480
reducing
  anomalies, 254
  latency, 152-155
relational database management systems. *See* RDBMSs
relational databases
  column family databases, 289-292
  design, 4-6
  history of, 19-29
  NoSQL, using with, 434-436
relationships, 15, 72
  common, 261
  many-to-many, 243, 263-264
  multiple between entities, 375-376
  one-to-many, 242, 262-263
remove command, 204
replicas, comparing, 323
replication
  column family databases, 322
  definition of, 135-136
  masterless, 98-102
  master-slave, 95
response times, 49-51
retrieval time, optimizing, 415
retrieving documents from collections, 208-209
Riak, 480
rings, definition of, 133
root nodes, 12
rows, 121. *See also* column family databases
  atomic, 283-284
  indexing, 281
  keys, 309-310, 337-338
rules
  constraints, 24
  Third Normal Form, 234

S

scalability, 29-31
  of graph databases, 418-419
  key-value databases, 95-102
  keys, 123
  master-slave replication, 95
  masterless replication, 98-102
schemaless, 129-131, 220-222
schemas, 23
  explicit definitions, 199-201
  polymorphic, 223
searching. *See also* queries
  indexes, 171-173
  patterns, 353
  values, 105-113, 160-161
secondary indexes. *See also* indexes
  applying, 345-347
  managing, 341-344
SELECT statements, 27
selecting
  databases, 428
    column family, 431-432
    document, 430
    graph, 433
    key-value, 429
  edges, 416
separating data, 229
sequential access to data, 7
sessions, consistency, 58
sharding, 227-231
sharing code, 195-198

simplicity of key-value databases, 91-92
simplified modeling, 375
single rows, 335
sink vertices, 393
sizing
  graphs, 389
  values, 155-159
SKUs (stock keeping units), 336
social media, modeling, 370
social network designs, 401-404
Solr, 162
sorted sets, 122
sorting rows, 284-285
source vertices, 393
Spark, 354-355
Sparksee, 480
speed, key-value databases, 93-95
SQL (Structured Query Language)
  DDL, 24-25
  DML, 25-26
Sqrrl, 480
standard query languages, 161-162
state, BASE, 56-59
statements
  DELETE, 27
  if, 88
  INSERT, 27
  SELECT, 27
  UPDATE, 27
statistics, predicting with, 351-352
stock keeping units. *See* SKUs
storage
  caches. *See* caches
  column family databases, 282-283, 334
  flat file data management systems, 7
  keys, 104. *See also* keys
  management programs, 20-21
  persistent data, 41-42
  rows, 310
  values, 110-113
strings, formatting, 123
strong typing values, 110-111
structured value design, 151-159
structures
  column family databases, 313-322
  columns, 310-313
  key-value databases, 91-92
  keyspaces, 309
subqueries, avoiding, 291-292
subtraction, 118
subtypes
  aggregation, 166-169
  code sharing, 195-198
support, range queries, 161

T

tables, 23, 121, 244
  column family databases, 332-340
  emulating, 165-166
  secondary indexes, 345-347
terminology
  column family databases, 308-313
    keyspaces, 309
    row keys, 309-310
  document databases, 214-217
    collections, 218
    denormalization, 235
    embedded documents, 218-219

horizontal partitions, 227-231
normalization, 233-234
partitions, 224
polymorphic schemas, 223
query processors, 235-236
schemaless, 220-222
vertical partitions, 225-227
graph databases, 380
edges, 381-382
loops, 384
paths, 383
vertices, 380-381
key-value database architecture, 131, 137-140
clusters, 131-133
replication, 135-136
rings, 133
key-value database modeling, 118-121
keys, 121-123
namespaces, 124-126
partition keys, 129
partitions, 126-127
schemaless, 129-131
values, 123-124
thermodynamics, laws of, 299-300
Third Normal Form, 234
time, optimizing retrieval, 415
time stamps, indexing, 281
Time to Live. *See* TTL
TinkerPop, 418
Titan, 418, 480
tools, big data, 348-356
transactions, 45
ACID, 429. *See also* ACID
atomic aggregation, 169-170
consistency of, 47-48
multirow, avoiding, 290-291
transportation networks, 393
traversal, graphs, 387, 410-417
troubleshooting
read/write operations, 155-159
write problems, 107-110
TTL (Time to Live) keys, 163-164
types
data. *See* data types
of databases, 59, 477-480
distributed databases, 41-54
document databases, 66-68
graph databases, 71-75
key-value pair databases, 60-65
of edges, 382
of eventual consistency, 57-59
of graphs, 392
bigraphs, 394
directed/undirected, 392-393
flow networks, 393
multigraphs, 395
weighted graphs, 395-396
of partitions, 224
horizontal, 227-231
vertical, 225-227

U

undirected edges, 382. *See also* edges
undirected graphs, 392-393
unions of graphs, 385
update command, 207
UPDATE statements, 27
updating documents in collections, 206-208

V

validation of code, 222
valueless columns, 334
values, 64, 110-113
- arrays. *See* arrays
- atomic aggregation, 169-170
- columns
  - *avoiding complex data structures, 339-340*
  - *storage, 334*
  - *versions, 338*
- data types, 216-217
- definition of, 123-124
- indexes, 171-173
- key-value databases
  - *architecture terms, 131-136*
  - *design, 147-148*
  - *modeling terms, 118-131*
- keys, 105-110, 215
- optimizing, 155-159
- searching, 112-113, 160-161
- structured design, 151-159

versions, column values, 338
vertical partitioning, 225-227
vertices, 380-381, 363. *See also* nodes
- betweenness, 391
- closeness, 390-391
- degrees, 390
- graph traversal, 387

views, 23
Vlissides, John, 162

W

weight of edges, 382. *See also* edges
weighted graphs, 395-396
while loops, 148
write-heavy applications, 260-261
write problems, avoiding, 107-110
writing atomic rows, 283-284

X

XML (Extensible Markup Language), 66

Z

zones, Goldilocks Zone of indexes, 258-260
Zookeeper, 293-294