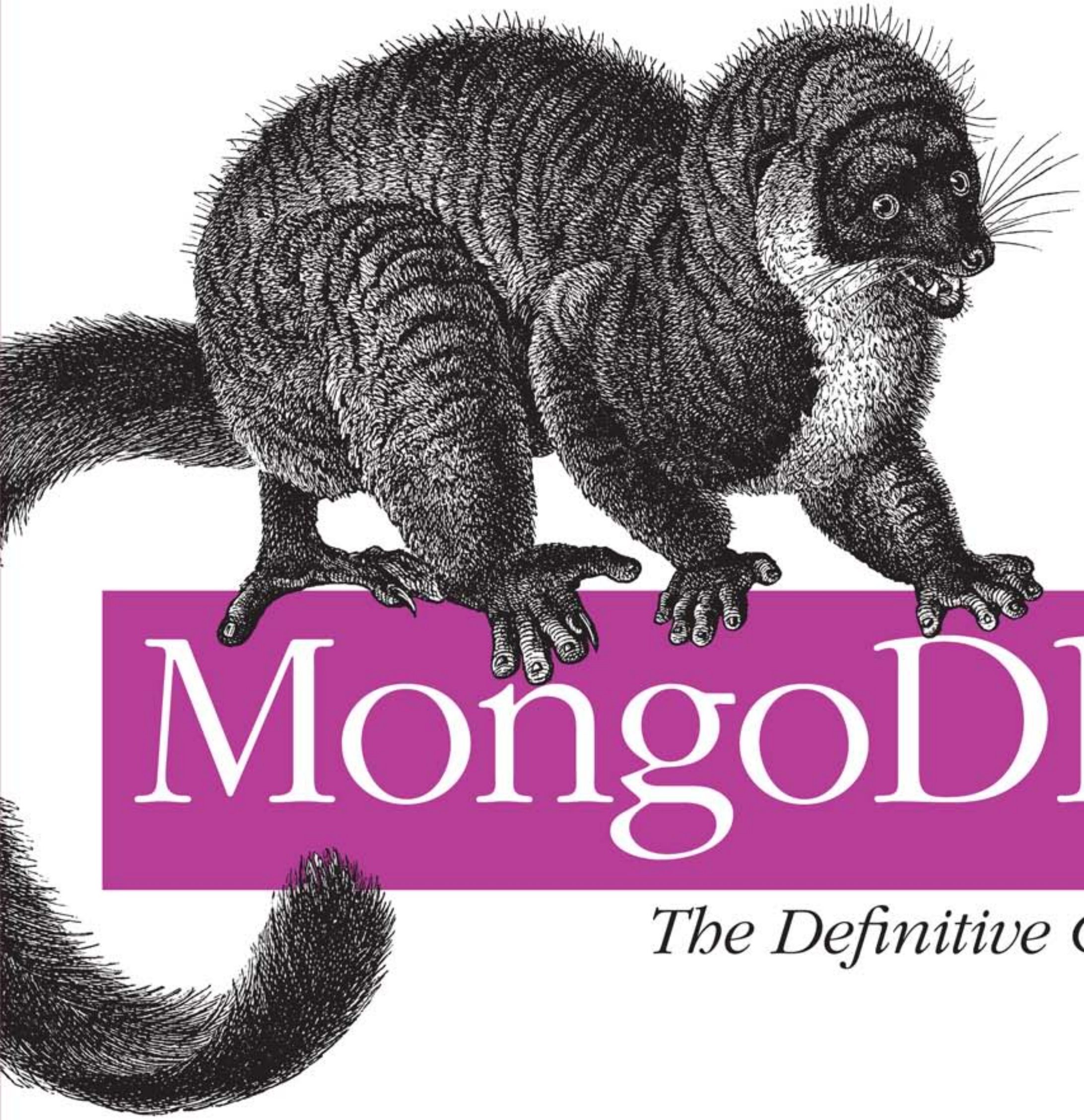
Powerful and Scalable Data Storage



MongoDB

The Definitive Guide

Kristina Chodorow
& Michael Dirolf
Foreword by Jeremy Zawodny

O'REILLY®

Databases/Web Development

MongoDB: The Definitive Guide

How does MongoDB help you manage a huMONGOus amount of data collected through your web application? With this authoritative introduction, you'll learn the many advantages of using document-oriented databases and discover why MongoDB is a reliable, high-performance system that allows for almost infinite horizontal scalability.

Written by engineers from 10gen, the company that develops and supports this open source database, *MongoDB: The Definitive Guide* provides guidance for database developers, advanced configuration for system administrators, and an overview of the concepts and use cases for other people on your project. Learn how easy it is to handle data as self-contained JSON-style documents, rather than as records in a relational database.

- Explore ways that document-oriented storage will work for your project
- Learn how MongoDB's schema-free data model handles documents, collections, and multiple databases
- Execute basic write operations, and create complex queries to find data with any criteria
- Use indexes, aggregation tools, and other advanced query techniques
- Learn about monitoring, security and authentication, backup and repair, and more
- Set up master-slave and automatic failover replication in MongoDB
- Use sharding to scale MongoDB horizontally, and learn how it impacts applications
- Get example applications written in Java, PHP, Python, and Ruby

Previous programming experience is recommended.

> *"Like MongoDB itself, this book is very straightforward and approachable. It's an essential reference to anyone seriously looking at using MongoDB."*
>
> **—Jeremy Zawodny**
> *author of*
> High Performance MySQL

Kristina Chodorow, a software engineer at 10gen, is a core contributor to the MongoDB project and has worked on the database server, PHP driver, Perl driver, and many other areas. She's given talks at conferences around the world, including OSCON, LinuxCon, FOSDEM, and Latinoware.

Michael Dirolf, also a software engineer at 10gen, is the lead maintainer for PyMongo (the MongoDB Python driver), and the former maintainer for the MongoDB Ruby driver. He's given talks about MongoDB at major conferences around the world.

O'REILLY®
oreilly.com

US $39.99 **CAN $45.99**
ISBN: 978-1-449-38156-1

53999
9 781449 381561

Safari Books Online
Free online edition
for 45 days with purchase of this book. Details on last page.

MongoDB: The Definitive Guide

MongoDB: The Definitive Guide

Kristina Chodorow and Michael Dirolf

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

MongoDB: The Definitive Guide
by Kristina Chodorow and Michael Dirolf

Copyright © 2010 Kristina Chodorow and Michael Dirolf. All rights reserved.
Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Julie Steele
Production Editor: Teresa Elsey
Copyeditor: Kim Wimpsett
Proofreader: Apostrophe Editing Services
Production Services: Molly Sharp

Indexer: Ellen Troutman Zaig
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:
September 2010: First Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *MongoDB: The Definitive Guide*, the image of a mongoose lemur, and related trade dress are trademarks of O'Reilly Media, Inc.

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 O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-449-38156-1

[M]

1283534198

Table of Contents

Foreword . . . . **xi**

Preface . . . . **xiii**

1. Introduction **1**
- A Rich Data Model 1
- Easy Scaling 2
- Tons of Features... 2
- ...Without Sacrificing Speed 3
- Simple Administration 3
- But Wait, That's Not All... 4

2. Getting Started **5**
- Documents 5
- Collections 7
  - Schema-Free 7
  - Naming 8
- Databases 8
- Getting and Starting MongoDB 10
- MongoDB Shell 11
  - Running the Shell 11
  - A MongoDB Client 12
  - Basic Operations with the Shell 12
  - Tips for Using the Shell 14
- Data Types 15
  - Basic Data Types 16
  - Numbers 18
  - Dates 19
  - Arrays 19
  - Embedded Documents 20
  - _id and ObjectIds 20

3. Creating, Updating, and Deleting Documents 23
Inserting and Saving Documents 23
  Batch Insert 23
  Inserts: Internals and Implications 24
Removing Documents 25
  Remove Speed 25
Updating Documents 26
  Document Replacement 26
  Using Modifiers 27
  Upserts 36
  Updating Multiple Documents 38
  Returning Updated Documents 39
The Fastest Write This Side of Mississippi 41
  Safe Operations 42
  Catching “Normal” Errors 43
Requests and Connections 43

4. Querying 45
Introduction to find 45
  Specifying Which Keys to Return 46
  Limitations 47
Query Criteria 47
  Query Conditionals 47
  OR Queries 48
  $not 49
  Rules for Conditionals 49
Type-Specific Queries 49
  null 49
  Regular Expressions 50
  Querying Arrays 51
  Querying on Embedded Documents 53
$where Queries 55
Cursors 56
  Limits, Skips, and Sorts 57
  Avoiding Large Skips 58
  Advanced Query Options 60
  Getting Consistent Results 61
Cursor Internals 63

5. Indexing 65
Introduction to Indexing 65
  Scaling Indexes 68
  Indexing Keys in Embedded Documents 68

Indexing for Sorts 69
Uniquely Identifying Indexes 69
Unique Indexes 69
Dropping Duplicates 70
Compound Unique Indexes 70
Using explain and hint 70
Index Administration 75
Changing Indexes 76
Geospatial Indexing 77
Compound Geospatial Indexes 78
The Earth Is Not a 2D Plane 79

6. Aggregation 81
count 81
distinct 81
group 82
Using a Finalizer 84
Using a Function as a Key 86
MapReduce 86
Example 1: Finding All Keys in a Collection 87
Example 2: Categorizing Web Pages 89
MongoDB and MapReduce 90

7. Advanced Topics 93
Database Commands 93
How Commands Work 94
Command Reference 95
Capped Collections 97
Properties and Use Cases 98
Creating Capped Collections 99
Sorting Au Naturel 99
Tailable Cursors 101
GridFS: Storing Files 101
Getting Started with GridFS: mongofiles 102
Working with GridFS from the MongoDB Drivers 102
Under the Hood 103
Server-Side Scripting 104
db.eval 104
Stored JavaScript 105
Security 106
Database References 107
What Is a DBRef? 107
Example Schema 107

Driver Support for DBRefs 108
When Should DBRefs Be Used? 108

8. Administration 111
Starting and Stopping MongoDB 111
  Starting from the Command Line 112
  File-Based Configuration 113
  Stopping MongoDB 114
Monitoring 114
  Using the Admin Interface 115
  serverStatus 116
  mongostat 118
  Third-Party Plug-Ins 118
Security and Authentication 118
  Authentication Basics 118
  How Authentication Works 120
  Other Security Considerations 121
Backup and Repair 121
  Data File Backup 121
  mongodump and mongorestore 122
  fsync and Lock 123
  Slave Backups 124
  Repair 124

9. Replication 127
Master-Slave Replication 127
  Options 128
  Adding and Removing Sources 129
Replica Sets 130
  Initializing a Set 132
  Nodes in a Replica Set 133
  Failover and Primary Election 135
Performing Operations on a Slave 136
  Read Scaling 137
  Using Slaves for Data Processing 137
How It Works 138
  The Oplog 138
  Syncing 139
  Replication State and the Local Database 139
  Blocking for Replication 140
Administration 141
  Diagnostics 141
  Changing the Oplog Size 141

Replication with Authentication 142

10. **Sharding** **143**
Introduction to Sharding 143
Autosharding in MongoDB 143
When to Shard 145
The Key to Sharding: Shard Keys 145
Sharding an Existing Collection 145
Incrementing Shard Keys Versus Random Shard Keys 146
How Shard Keys Affect Operations 146
Setting Up Sharding 147
Starting the Servers 147
Sharding Data 148
Production Configuration 149
A Robust Config 149
Many mongos 149
A Sturdy Shard 150
Physical Servers 150
Sharding Administration 150
config Collections 150
Sharding Commands 152

11. **Example Applications** **155**
Chemical Search Engine: Java 155
Installing the Java Driver 155
Using the Java Driver 155
Schema Design 156
Writing This in Java 158
Issues 159
News Aggregator: PHP 159
Installing the PHP Driver 160
Using the PHP Driver 161
Designing the News Aggregator 162
Trees of Comments 162
Voting 164
Custom Submission Forms: Ruby 164
Installing the Ruby Driver 164
Using the Ruby Driver 165
Custom Form Submission 166
Ruby Object Mappers and Using MongoDB with Rails 167
Real-Time Analytics: Python 168
Installing PyMongo 168
Using PyMongo 168

MongoDB for Real-Time Analytics 169
Schema 169
Handling a Request 170
Using Analytics Data 170
Other Considerations 171

A. Installing MongoDB 173

B. mongo: The Shell 177

C. MongoDB Internals 179

Index 183

Foreword

In the last 10 years, the Internet has challenged relational databases in ways nobody could have foreseen. Having used MySQL at large and growing Internet companies during this time, I've seen this happen firsthand. First you have a single server with a small data set. Then you find yourself setting up replication so you can scale out reads and deal with potential failures. And, before too long, you've added a caching layer, tuned all the queries, and thrown even more hardware at the problem.

Eventually you arrive at the point when you need to shard the data across multiple clusters and rebuild a ton of application logic to deal with it. And soon after that you realize that you're locked into the schema you modeled so many months before.

Why? Because there's so much data in your clusters now that altering the schema will take a long time and involve a lot of precious DBA time. It's easier just to work around it in code. This can keep a small team of developers busy for many months. In the end, you'll always find yourself wondering if there's a better way—or why more of these features are not built into the core database server.

Keeping with tradition, the Open Source community has created a plethora of "better ways" in response to the ballooning data needs of modern web applications. They span the spectrum from simple in-memory key/value stores to complicated SQL-speaking MySQL/InnoDB derivatives. But the sheer number of choices has made finding the right solution more difficult. I've looked at many of them.

I was drawn to MongoDB by its pragmatic approach. MongoDB doesn't try to be everything to everyone. Instead it strikes the right balance between features and complexity, with a clear bias toward making previously difficult tasks far easier. In other words, it has the features that really matter to the vast majority of today's web applications: indexes, replication, sharding, a rich query syntax, and a very flexible data model. All of this comes without sacrificing speed.

Like MongoDB itself, this book is very straightforward and approachable. New MongoDB users can start with Chapter 1 and be up and running in no time. Experienced users will appreciate this book's breadth and authority. It's a solid reference for advanced administrative topics such as replication, backups, and sharding, as well as popular client APIs.

Having recently started to use MongoDB in my day job, I have no doubt that this book will be at my side for the entire journey—from the first install to production deployment of a sharded and replicated cluster. It's an essential reference to anyone seriously looking at using MongoDB.

—Jeremy Zawodny
Craigslist Software Engineer
August 2010

Preface

How This Book Is Organized

Getting Up to Speed with MongoDB

In Chapter 1, *Introduction*, we provide some background about MongoDB: why it was created, the goals it is trying to accomplish, and why you might choose to use it for a project. We go into more detail in Chapter 2, *Getting Started*, which provides an introduction to the core concepts and vocabulary of MongoDB. Chapter 2 also provides a first look at working with MongoDB, getting you started with the database and the shell.

Developing with MongoDB

The next two chapters cover the basic material that developers need to know to work with MongoDB. In Chapter 3, *Creating, Updating, and Deleting Documents*, we describe how to perform those basic write operations, including how to do them with different levels of safety and speed. Chapter 4, *Querying*, explains how to find documents and create complex queries. This chapter also covers how to iterate through results and options for limiting, skipping, and sorting results.

Advanced Usage

The next three chapters go into more complex usage than simply storing and retrieving data. Chapter 5, *Indexing*, explains what indexes are and how to use them with MongoDB. It also covers tools you can use to examine or modify the indexes used to perform a query, and it covers index administration. Chapter 6, *Aggregation*, covers a number of techniques for aggregating data with MongoDB, including counting, finding distinct values, grouping documents, and using MapReduce. Chapter 7, *Advanced Topics*, is a mishmash of important tidbits that didn't fit into any of the previous categories: file storage, server-side JavaScript, database commands, and database references.

Administration

The next three chapters are less about programming and more about the operational aspects of MongoDB. Chapter 8, *Administration*, discusses options for starting the database in different ways, monitoring a MongoDB server, and keeping deployments secure. Chapter 8 also covers how to keep proper backups of the data you've stored in MongoDB. In Chapter 9, *Replication*, we explain how to set up replication with MongoDB, including standard master-slave configuration and setups with automatic failover. This chapter also covers how MongoDB replication works and options for tweaking it. Chapter 10, *Sharding*, describes how to scale MongoDB horizontally: it covers what autosharding is, how to set it up, and the ways in which it impacts applications.

Developing Applications with MongoDB

In Chapter 11, *Example Applications*, we provide example applications using MongoDB, written in Java, PHP, Python, and Ruby. These examples illustrate how to map the concepts described earlier in the book to specific languages and problem domains.

Appendixes

Appendix A, *Installing MongoDB*, explains MongoDB's versioning scheme and how to install it on Windows, OS X, and Linux. Appendix B, *mongo: The Shell*, includes some useful shell tips and tools. Finally, Appendix C, *MongoDB Internals*, details a little about how MongoDB works internally: its storage engine, data format, and wire protocol.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, collection names, database names, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, command-line utilities, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.

This icon indicates a warning or caution.

Using Code Examples

This book can help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*MongoDB: The Definitive Guide* by Kristina Chodorow and Michael Dirolf (O'Reilly). Copyright 2010 Kristina Chodorow and Michael Dirolf, 978-1-449-38156-1."

If you feel your use of code examples falls outside fair use or the permission given here, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that lets you easily search more than 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at *http://my.safaribooksonline.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707 829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/9781449381561

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our website at:

http://www.oreilly.com

Acknowledgments

We would like to thank Eliot Horowitz and Dwight Merriman, who made all of this possible by starting the MongoDB project. We'd also like to thank our tech reviewers: Alberto Lerner, Mathias Stearn, Aaron Staple, James Avery, and John Hornbeck. You guys made this book immeasurably better (and more correct). Thank you, Julie Steele, for being such a terrific editor and for helping us every step of the way. Thanks to everybody else at O'Reilly who helped get this book into production. Finally, a big thanks is owed to the entire MongoDB community, which has supported the project (and this book) from the very beginning.

Acknowledgments from Kristina

Thanks to all of my co-workers at 10gen for sharing your knowledge and advice on MongoDB (as well as your advice on ops, beer, and plane crashes). Also, thank you, Mike, for magically making half of this book appear and correcting some of my more embarrassing errors before Julie saw them. Finally, I would like to thank Andrew, Susan, and Andy for all of their support, patience, and suggestions. I couldn't have done it without you guys.

Acknowledgments from Michael

Thanks to all of my friends, who have put up with me during this process (and in general). Thanks to everyone I've worked with at 10gen for making working on MongoDB a blast. Thank you, Kristina, for being such a great coauthor. Most importantly, I would like to thank my entire family for all of their support with this and everything I undertake.

CHAPTER 1

Introduction

MongoDB is a powerful, flexible, and scalable data store. It combines the ability to scale out with many of the most useful features of relational databases, such as secondary indexes, range queries, and sorting. MongoDB is also incredibly featureful: it has tons of useful features such as built-in support for MapReduce-style aggregation and geospatial indexes.

There is no point in creating a great technology if it's impossible to work with, so a lot of effort has been put into making MongoDB easy to get started with and a pleasure to use. MongoDB has a developer-friendly data model, administrator-friendly configuration options, and natural-feeling language APIs presented by drivers and the database shell. MongoDB tries to get out of your way, letting you program instead of worrying about storing data.

A Rich Data Model

MongoDB is a *document-oriented* database, not a relational one. The primary reason for moving away from the relational model is to make scaling out easier, but there are some other advantages as well.

The basic idea is to replace the concept of a "row" with a more flexible model, the "document." By allowing embedded documents and arrays, the document-oriented approach makes it possible to represent complex hierarchical relationships with a single record. This fits very naturally into the way developers in modern object-oriented languages think about their data.

MongoDB is also schema-free: a document's keys are not predefined or fixed in any way. Without a schema to change, massive data migrations are usually unnecessary. New or missing keys can be dealt with at the application level, instead of forcing all data to have the same shape. This gives developers a lot of flexibility in how they work with evolving data models.

Easy Scaling

Data set sizes for applications are growing at an incredible pace. Advances in sensor technology, increases in available bandwidth, and the popularity of handheld devices that can be connected to the Internet have created an environment where even small-scale applications need to store more data than many databases were meant to handle. A terabyte of data, once an unheard-of amount of information, is now commonplace.

As the amount of data that developers need to store grows, developers face a difficult decision: how should they scale their databases? Scaling a database comes down to the choice between scaling up (getting a bigger machine) or scaling out (partitioning data across more machines). Scaling up is often the path of least resistance, but it has drawbacks: large machines are often very expensive, and eventually a physical limit is reached where a more powerful machine cannot be purchased at any cost. For the type of large web application that most people aspire to build, it is either impossible or not cost-effective to run off of one machine. Alternatively, it is both extensible and economical to scale *out*: to add storage space or increase performance, you can buy another commodity server and add it to your cluster.

MongoDB was designed from the beginning to scale out. Its document-oriented data model allows it to automatically split up data across multiple servers. It can balance data and load across a cluster, redistributing documents automatically. This allows developers to focus on programming the application, not scaling it. When they need more capacity, they can just add new machines to the cluster and let the database figure out how to organize everything.

Tons of Features…

It's difficult to quantify what a feature is: anything above and beyond what a relational database provides? Memcached? Other document-oriented databases? However, no matter what the baseline is, MongoDB has some really nice, unique tools that are not (all) present in any other solution.

Indexing
: MongoDB supports generic secondary indexes, allowing a variety of fast queries, and provides unique, compound, and geospatial indexing capabilities as well.

Stored JavaScript
Instead of stored procedures, developers can store and use JavaScript functions and values on the server side.

Aggregation
MongoDB supports MapReduce and other aggregation tools.

Fixed-size collections
Capped collections are fixed in size and are useful for certain types of data, such as logs.

File storage
MongoDB supports an easy-to-use protocol for storing large files and file metadata.

Some features common to relational databases are not present in MongoDB, notably joins and complex multirow transactions. These are architectural decisions to allow for scalability, because both of those features are difficult to provide efficiently in a distributed system.

…Without Sacrificing Speed

Incredible performance is a major goal for MongoDB and has shaped many design decisions. MongoDB uses a binary wire protocol as the primary mode of interaction with the server (as opposed to a protocol with more overhead, like HTTP/REST). It adds dynamic padding to documents and preallocates data files to trade extra space usage for consistent performance. It uses memory-mapped files in the default storage engine, which pushes the responsibility for memory management to the operating system. It also features a dynamic query optimizer that “remembers” the fastest way to perform a query. In short, almost every aspect of MongoDB was designed to maintain high performance.

Although MongoDB is powerful and attempts to keep many features from relational systems, it is not intended to do everything that a relational database does. Whenever possible, the database server offloads processing and logic to the client side (handled either by the drivers or by a user’s application code). Maintaining this streamlined design is one of the reasons MongoDB can achieve such high performance.

Simple Administration

MongoDB tries to simplify database administration by making servers administrate themselves as much as possible. Aside from starting the database server, very little administration is necessary. If a master server goes down, MongoDB can automatically failover to a backup slave and promote the slave to a master. In a distributed environment, the cluster needs to be told only that a new node exists to automatically integrate and configure it.

MongoDB's administration philosophy is that the server should handle as much of the configuration as possible automatically, allowing (but not requiring) users to tweak their setups if needed.

But Wait, That's Not All…

Throughout the course of the book, we will take the time to note the reasoning or motivation behind particular decisions made in the development of MongoDB. Through those notes we hope to share the philosophy behind MongoDB. The best way to summarize the MongoDB project, however, is through its main focus—to create a full-featured data store that is scalable, flexible, and fast.

CHAPTER 2

Getting Started

MongoDB is very powerful, but it is still easy to get started with. In this chapter we'll introduce some of the basic concepts of MongoDB:

- A *document* is the basic unit of data for MongoDB, roughly equivalent to a row in a relational database management system (but much more expressive).
- Similarly, a *collection* can be thought of as the schema-free equivalent of a table.
- A single instance of MongoDB can host multiple independent *databases*, each of which can have its own collections and permissions.
- MongoDB comes with a simple but powerful JavaScript *shell*, which is useful for the administration of MongoDB instances and data manipulation.
- Every document has a special key, `"_id"`, that is unique across the document's collection.

Documents

At the heart of MongoDB is the concept of a *document*: an ordered set of keys with associated values. The representation of a document differs by programming language, but most languages have a data structure that is a natural fit, such as a map, hash, or dictionary. In JavaScript, for example, documents are represented as objects:

```
{"greeting" : "Hello, world!"}
```

This simple document contains a single key, `"greeting"`, with a value of `"Hello, world!"`. Most documents will be more complex than this simple one and often will contain multiple key/value pairs:

```
{"greeting" : "Hello, world!", "foo" : 3}
```

This example is a good illustration of several important concepts:

- Key/value pairs in documents are ordered—the earlier document is distinct from the following document:

  ```
  {"foo" : 3, "greeting" : "Hello, world!"}
  ```

  > In most cases the ordering of keys in documents is not important. In fact, in some programming languages the default representation of a document does not even maintain ordering (e.g., dictionaries in Python and hashes in Perl or Ruby 1.8). Drivers for those languages usually have some mechanism for specifying documents with ordering for the rare cases when it is necessary. (Those cases will be noted throughout the text.)

- Values in documents are not just “blobs.” They can be one of several different data types (or even an entire embedded document—see “Embedded Documents” on page 20). In this example the value for `"greeting"` is a string, whereas the value for `"foo"` is an integer.

The keys in a document are strings. Any UTF-8 character is allowed in a key, with a few notable exceptions:

- Keys must not contain the character \0 (the `null` character). This character is used to signify the end of a key.
- The . and $ characters have some special properties and should be used only in certain circumstances, as described in later chapters. In general, they should be considered reserved, and drivers will complain if they are used inappropriately.
- Keys starting with _ should be considered reserved; although this is not strictly enforced.

MongoDB is type-sensitive and case-sensitive. For example, these documents are distinct:

```
{"foo" : 3}
{"foo" : "3"}
```

As are as these:

```
{"foo" : 3}
{"Foo" : 3}
```

A final important thing to note is that documents in MongoDB cannot contain duplicate keys. For example, the following is not a legal document:

```
{"greeting" : "Hello, world!", "greeting" : "Hello, MongoDB!"}
```

Collections

A *collection* is a group of documents. If a document is the MongoDB analog of a row in a relational database, then a collection can be thought of as the analog to a table.

Schema-Free

Collections are *schema-free*. This means that the documents within a single collection can have any number of different "shapes." For example, both of the following documents could be stored in a single collection:

```
{"greeting" : "Hello, world!"}
{"foo" : 5}
```

Note that the previous documents not only have different types for their values (string versus integer) but also have entirely different keys. Because any document can be put into any collection, the question often arises: "Why do we need separate collections at all?" It's a good question—with no need for separate schemas for different kinds of documents, why *should* we use more than one collection? There are several good reasons:

- Keeping different kinds of documents in the same collection can be a nightmare for developers and admins. Developers need to make sure that each query is only returning documents of a certain kind or that the application code performing a query can handle documents of different shapes. If we're querying for blog posts, it's a hassle to weed out documents containing author data.
- It is much faster to get a list of collections than to extract a list of the types in a collection. For example, if we had a `type` key in the collection that said whether each document was a "skim," "whole," or "chunky monkey" document, it would be much slower to find those three values in a single collection than to have three separate collections and query for their names (see "Subcollections" on page 8).
- Grouping documents of the same kind together in the same collection allows for data locality. Getting several blog posts from a collection containing only posts will likely require fewer disk seeks than getting the same posts from a collection containing posts and author data.
- We begin to impose some structure on our documents when we create indexes. (This is especially true in the case of unique indexes.) These indexes are defined per collection. By putting only documents of a single type into the same collection, we can index our collections more efficiently.

As you can see, there are sound reasons for creating a schema and for grouping related types of documents together. MongoDB just relaxes this requirement and allows developers more flexibility.

Naming

A collection is identified by its name. Collection names can be any UTF-8 string, with a few restrictions:

- The empty string ("") is not a valid collection name.
- Collection names may not contain the character \0 (the `null` character) because this delineates the end of a collection name.
- You should not create any collections that start with *system.*, a prefix reserved for system collections. For example, the *system.users* collection contains the database's users, and the *system.namespaces* collection contains information about all of the database's collections.
- User-created collections should not contain the reserved character $ in the name. The various drivers available for the database do support using $ in collection names because some system-generated collections contain it. You should not use $ in a name unless you are accessing one of these collections.

Subcollections

One convention for organizing collections is to use namespaced subcollections separated by the . character. For example, an application containing a blog might have a collection named *blog.posts* and a separate collection named *blog.authors*. This is for organizational purposes only—there is no relationship between the *blog* collection (it doesn't even have to exist) and its "children."

Although subcollections do not have any special properties, they are useful and incorporated into many MongoDB tools:

- GridFS, a protocol for storing large files, uses subcollections to store file metadata separately from content chunks (see Chapter 7 for more information about GridFS).
- The MongoDB web console organizes the data in its DBTOP section by subcollection (see Chapter 8 for more information on administration).
- Most drivers provide some syntactic sugar for accessing a subcollection of a given collection. For example, in the database shell, `db.blog` will give you the *blog* collection, and `db.blog.posts` will give you the *blog.posts* collection.

Subcollections are a great way to organize data in MongoDB, and their use is highly recommended.

Databases

In addition to grouping documents by collection, MongoDB groups collections into *databases*. A single instance of MongoDB can host several databases, each of which can be thought of as completely independent. A database has its own permissions, and each database is stored in separate files on disk. A good rule of thumb is to store all data for a single application in the same database. Separate databases are useful when storing data for several application or users on the same MongoDB server.

Like collections, databases are identified by name. Database names can be any UTF-8 string, with the following restrictions:

- The empty string ("") is not a valid database name.
- A database name cannot contain any of these characters: ' ' (a single space), ., $, /, \, or \0 (the null character).
- Database names should be all lowercase.
- Database names are limited to a maximum of 64 bytes.

One thing to remember about database names is that they will actually end up as files on your filesystem. This explains why many of the previous restrictions exist in the first place.

There are also several reserved database names, which you can access directly but have special semantics. These are as follows:

admin
: This is the "root" database, in terms of authentication. If a user is added to the *admin* database, the user automatically inherits permissions for all databases. There are also certain server-wide commands that can be run only from the *admin* database, such as listing all of the databases or shutting down the server.

local
: This database will never be replicated and can be used to store any collections that should be local to a single server (see Chapter 9 for more information about replication and the local database).

config
: When Mongo is being used in a sharded setup (see Chapter 10), the *config* database is used internally to store information about the shards.

By prepending a collection's name with its containing database, you can get a fully qualified collection name called a *namespace*. For instance, if you are using the *blog.posts* collection in the *cms* database, the namespace of that collection would be `cms.blog.posts`. Namespaces are limited to 121 bytes in length and, in practice, should be less than 100 bytes long. For more on namespaces and the internal representation of collections in MongoDB, see Appendix C.

Getting and Starting MongoDB

MongoDB is almost always run as a network server that clients can connect to and perform operations on. To start the server, run the `mongod` executable:

```
$ ./mongod
./mongod --help for help and startup options
Sun Mar 28 12:31:20 Mongo DB : starting : pid = 44978 port = 27017
dbpath = /data/db/ master = 0 slave = 0  64-bit
Sun Mar 28 12:31:20 db version v1.5.0-pre-, pdfile version 4.5
Sun Mar 28 12:31:20 git version: ...
Sun Mar 28 12:31:20 sys info: ...
Sun Mar 28 12:31:20 waiting for connections on port 27017
Sun Mar 28 12:31:20 web admin interface listening on port 28017
```

Or if you're on Windows, run this:

```
$ mongod.exe
```

For detailed information on installing MongoDB on your system, see Appendix A.

When run with no arguments, `mongod` will use the default data directory, */data/db/* (or *C:\data\db\* on Windows), and port 27017. If the data directory does not already exist or is not writable, the server will fail to start. It is important to create the data directory (e.g., *mkdir -p /data/db/*), and to make sure your user has permission to write to the directory, before starting MongoDB. The server will also fail to start if the port is not available—this is often caused by another instance of MongoDB that is already running.

The server will print some version and system information and then begin waiting for connections. By default, MongoDB listens for socket connections on port 27017.

`mongod` also sets up a very basic HTTP server that listens on a port 1,000 higher than the main port, in this case 28017. This means that you can get some administrative information about your database by opening a web browser and going to *http://localhost:28017*.

You can safely stop `mongod` by typing Ctrl-c in the shell that is running the server.

For more information on starting or stopping MongoDB, see "Starting and Stopping MongoDB" on page 111, and for more on the administrative interface, see "Using the Admin Interface" on page 115.

MongoDB Shell

MongoDB comes with a JavaScript shell that allows interaction with a MongoDB instance from the command line. The shell is very useful for performing administrative functions, inspecting a running instance, or just playing around. The `mongo` shell is a crucial tool for using MongoDB and is used extensively throughout the rest of the text.

Running the Shell

To start the shell, run the `mongo` executable:

```
$ ./mongo
MongoDB shell version: 1.6.0
url: test
connecting to: test
type "help" for help
>
```

The shell automatically attempts to connect to a MongoDB server on startup, so make sure you start `mongod` before starting the shell.

The shell is a full-featured JavaScript interpreter, capable of running arbitrary JavaScript programs. To illustrate this, let's perform some basic math:

```
> x = 200
200
> x / 5;
40
```

We can also leverage all of the standard JavaScript libraries:

```
> Math.sin(Math.PI / 2);
1
> new Date("2010/1/1");
"Fri Jan 01 2010 00:00:00 GMT-0500 (EST)"
> "Hello, World!".replace("World", "MongoDB");
Hello, MongoDB!
```

We can even define and call JavaScript functions:

```
> function factorial (n) {
... if (n <= 1) return 1;
... return n * factorial(n - 1);
... }
> factorial(5);
120
```

Note that you can create multiline commands. The shell will detect whether the JavaScript statement is complete when you press Enter and, if it is not, will allow you to continue writing it on the next line.

A MongoDB Client

Although the ability to execute arbitrary JavaScript is cool, the real power of the shell lies in the fact that it is also a stand-alone MongoDB client. On startup, the shell connects to the *test* database on a MongoDB server and assigns this database connection to the global variable `db`. This variable is the primary access point to MongoDB through the shell.

The shell contains some add-ons that are not valid JavaScript syntax but were implemented because of their familiarity to users of SQL shells. The add-ons do not provide any extra functionality, but they are nice syntactic sugar. For instance, one of the most important operations is selecting which database to use:

```
> use foobar
switched to db foobar
```

Now if you look at the `db` variable, you can see that it refers to the *foobar* database:

```
> db
foobar
```

Because this is a JavaScript shell, typing a variable will convert the variable to a string (in this case, the database name) and print it.

Collections can be accessed from the `db` variable. For example, `db.baz` returns the *baz* collection in the current database. Now that we can access a collection in the shell, we can perform almost any database operation.

Basic Operations with the Shell

We can use the four basic operations, create, read, update, and delete (CRUD), to manipulate and view data in the shell.

Create

The `insert` function adds a document to a collection. For example, suppose we want to store a blog post. First, we'll create a local variable called `post` that is a JavaScript object representing our document. It will have the keys `"title"`, `"content"`, and `"date"` (the date that it was published):

```
> post = {"title" : "My Blog Post",
... "content" : "Here's my blog post.",
... "date" : new Date()}
{
    "title" : "My Blog Post",
    "content" : "Here's my blog post.",
    "date" : "Sat Dec 12 2009 11:23:21 GMT-0500 (EST)"
}
```

This object is a valid MongoDB document, so we can save it to the *blog* collection using the `insert` method:

```
> db.blog.insert(post)
```

The blog post has been saved to the database. We can see it by calling `find` on the collection:

```
> db.blog.find()
{
    "_id" : ObjectId("4b23c3ca7525f35f94b60a2d"),
    "title" : "My Blog Post",
    "content" : "Here's my blog post.",
    "date" : "Sat Dec 12 2009 11:23:21 GMT-0500 (EST)"
}
```

You can see that an `"_id"` key was added and that the other key/value pairs were saved as we entered them. The reason for `"_id"`'s sudden appearance is explained at the end of this chapter.

Read

`find` returns all of the documents in a collection. If we just want to see one document from a collection, we can use `findOne`:

```
> db.blog.findOne()
{
    "_id" : ObjectId("4b23c3ca7525f35f94b60a2d"),
    "title" : "My Blog Post",
    "content" : "Here's my blog post.",
    "date" : "Sat Dec 12 2009 11:23:21 GMT-0500 (EST)"
}
```

`find` and `findOne` can also be passed criteria in the form of a query document. This will restrict the documents matched by the query. The shell will automatically display up to 20 documents matching a `find`, but more can be fetched. See Chapter 4 for more information on querying.

Update

If we would like to modify our post, we can use `update`. `update` takes (at least) two parameters: the first is the criteria to find which document to update, and the second is the new document. Suppose we decide to enable comments on the blog post we created earlier. We'll need to add an array of comments as the value for a new key in our document.

The first step is to modify the variable `post` and add a `"comments"` key:

```
> post.comments = []
[  ]
```

Then we perform the update, replacing the post titled "My Blog Post" with our new version of the document:

```
> db.blog.update({title : "My Blog Post"}, post)
```

Now the document has a `"comments"` key. If we call `find` again, we can see the new key:

```
> db.blog.find()
{
    "_id" : ObjectId("4b23c3ca7525f35f94b60a2d"),
    "title" : "My Blog Post",
    "content" : "Here's my blog post.",
    "date" : "Sat Dec 12 2009 11:23:21 GMT-0500 (EST)"
    "comments" : [ ]
}
```

Delete

`remove` deletes documents permanently from the database. Called with no parameters, it removes all documents from a collection. It can also take a document specifying criteria for removal. For example, this would remove the post we just created:

```
> db.blog.remove({title : "My Blog Post"})
```

Now the collection will be empty again.

Tips for Using the Shell

Because `mongo` is simply a JavaScript shell, you can get a great deal of help for it by simply looking up JavaScript documentation online. The shell also includes built-in help that can be accessed by typing `help`:

```
> help
HELP
    show dbs                     show database names
    show collections             show collections in current database
    show users                   show users in current database
    show profile                 show recent system.profile entries w. time >= 1ms
    use <db name>                set current database to <db name>
    db.help()                    help on DB methods
    db.foo.help()                help on collection methods
    db.foo.find()                list objects in collection foo
    db.foo.find( { a : 1 } )     list objects in foo where a == 1
    it                           result of the last line evaluated
```

Help for database-level commands is provided by `db.help();`, and help at the collections can be accessed with `db.foo.help();`.

A good way of figuring out what a function is doing is to type it without the parentheses. This will print the JavaScript source code for the function. For example, if we are curious about how the `update` function works or cannot remember the order of parameters, we can do the following:

```
> db.foo.update
function (query, obj, upsert, multi) {
    assert(query, "need a query");
    assert(obj, "need an object");
    this._validateObject(obj);
    this._mongo.update(this._fullName, query, obj,
                               upsert ? true : false, multi ? true : false);
}
```

There is also an autogenerated API of all the JavaScript functions provided by the shell at *http://api.mongodb.org/js*.

Inconvenient collection names

Fetching a collection with `db.`*`collectionName`* almost always works, unless the collection name actually is a property of the database class. For instance, if we are trying to access the *version* collection, we cannot say `db.version` because `db.version` is a database function. (It returns the version of the running MongoDB server.)

```
> db.version
function () {
    return this.serverBuildInfo().version;
}
```

`db`’s collection-returning behavior is only a fallback for when JavaScript cannot find a matching property. When there is a property with the same name as the desired collection, we can use the `getCollection` function:

```
> db.getCollection("version");
test.version
```

This can also be handy for collections with invalid JavaScript in their names. For example, *foo-bar* is a valid collection name, but it’s variable subtraction in JavaScript. You can get the *foo-bar* collection with `db.getCollection("foo-bar")`.

In JavaScript, `x.y` is identical to `x['y']`. This means that subcollections can be accessed using variables, not just literal names. That is, if you needed to perform some operation on every *blog* subcollection, you could iterate through them with something like this:

```
var collections = ["posts", "comments", "authors"];

for (i in collections) {
    doStuff(db.blog[collections[i]]);
}
```

Instead of this:

```
doStuff(db.blog.posts);
doStuff(db.blog.comments);
doStuff(db.blog.authors);
```

Data Types

The beginning of this chapter covered the basics of what a document is. Now that you are up and running with MongoDB and can try things on the shell, this section will dive a little deeper. MongoDB supports a wide range of data types as values in documents. In this section, we’ll outline all of the supported types.

Basic Data Types

Documents in MongoDB can be thought of as "JSON-like" in that they are conceptually similar to objects in JavaScript. JSON is a simple representation of data: the specification can be described in about one paragraph (*http://www.json.org* proves it) and lists only six data types. This is a good thing in many ways: it's easy to understand, parse, and remember. On the other hand, JSON's expressive capabilities are limited, because the only types are null, boolean, numeric, string, array, and object.

Although these types allow for an impressive amount of expressivity, there are a couple of additional types that are crucial for most applications, especially when working with a database. For example, JSON has no date type, which makes working with dates even more annoying than it usually is. There is a number type, but only one—there is no way to differentiate floats and integers, never mind any distinction between 32-bit and 64-bit numbers. There is no way to represent other commonly used types, either, such as regular expressions or functions.

MongoDB adds support for a number of additional data types while keeping JSON's essential key/value pair nature. Exactly how values of each type are represented varies by language, but this is a list of the commonly supported types and how they are represented as part of a document in the shell:

null

Null can be used to represent both a null value and a nonexistent field:

```
{"x" : null}
```

boolean

There is a boolean type, which will be used for the values `'true'` and `'false'`:

```
{"x" : true}
```

32-bit integer

This cannot be represented on the shell. As mentioned earlier, JavaScript supports only 64-bit floating point numbers, so 32-bit integers will be converted into those.

64-bit integer

Again, the shell cannot represent these. The shell will display them using a special embedded document; see the section "Numbers" on page 18 for details.

64-bit floating point number

All numbers in the shell will be of this type. Thus, this will be a floating-point number:

```
{"x" : 3.14}
```

As will this:

```
{"x" : 3}
```

string

Any string of UTF-8 characters can be represented using the string type:

```
{"x" : "foobar"}
```

symbol
: This type is not supported by the shell. If the shell gets a symbol from the database, it will convert it into a string.

object id
: An object id is a unique 12-byte ID for documents. See the section "_id and ObjectIds" on page 20 for details:

```
{"x" : ObjectId()}
```

date
: Dates are stored as milliseconds since the epoch. The time zone is not stored:

```
{"x" : new Date()}
```

regular expression
: Documents can contain regular expressions, using JavaScript's regular expression syntax:

```
{"x" : /foobar/i}
```

code
: Documents can also contain JavaScript code:

```
{"x" : function() { /* ... */ }}
```

binary data
: Binary data is a string of arbitrary bytes. It cannot be manipulated from the shell.

maximum value
: BSON contains a special type representing the largest possible value. The shell does not have a type for this.

minimum value
: BSON contains a special type representing the smallest possible value. The shell does not have a type for this.

undefined
: Undefined can be used in documents as well (JavaScript has distinct types for null and undefined):

```
{"x" : undefined}
```

array
: Sets or lists of values can be represented as arrays:

```
{"x" : ["a", "b", "c"]}
```

embedded document
: Documents can contain entire documents, embedded as values in a parent document:

```
{"x" : {"foo" : "bar"}}
```

Numbers

JavaScript has one “number” type. Because MongoDB has three number types (4-byte integer, 8-byte integer, and 8-byte float), the shell has to hack around JavaScript’s limitations a bit. By default, any number in the shell is treated as a double by MongoDB. This means that if you retrieve a 4-byte integer from the database, manipulate its document, and save it back to the database *even without changing the integer*, the integer will be resaved as a floating-point number. Thus, it is generally a good idea not to overwrite entire documents from the shell (see Chapter 3 for information on making changes to the values of individual keys).

Another problem with every number being represented by a double is that there are some 8-byte integers that cannot be accurately represented by 8-byte floats. Therefore, if you save an 8-byte integer and look at it in the shell, the shell will display it as an embedded document indicating that it might not be exact. For example, if we save a document with a `"myInteger"` key whose value is the 64-bit integer, `3`, and then look at it in the shell, it will look like this:

```
> doc = db.nums.findOne()
{
    "_id" : ObjectId("4c0beecfd096a2580fe6fa08"),
    "myInteger" : {
        "floatApprox" : 3
    }
}
```

The number is not changed in the database (unless you modify and resave the object from the shell, in which case it will turn into a float); the embedded document just indicates that the shell is displaying a floating-point approximation of an 8-byte integer. If this embedded document has only one key, it is, in fact, exact.

If you insert an 8-byte integer that cannot be accurately displayed as a double, the shell will add two keys, `"top"` and `"bottom"`, containing the 32-bit integers representing the 4 high-order bytes and 4 low-order bytes of the integer, respectively. For instance, if we insert `9223372036854775807`, the shell will show us the following:

```
> db.nums.findOne()
{
    "_id" : ObjectId("4c0beecfd096a2580fe6fa09"),
    "myInteger" : {
        "floatApprox" : 9223372036854776000,
        "top" : 2147483647,
        "bottom" : 4294967295
    }
}
```

The `"floatApprox"` embedded documents are special and can be manipulated as numbers as well as documents:

```
> doc.myInteger + 1
4
```

```
> doc.myInteger.floatApprox
3
```

All 4-byte integers can be represented exactly by an 8-byte floating-point number, so they are displayed normally.

Dates

In JavaScript, the `Date` object is used for MongoDB's date type. When creating a new `Date` object, always call `new Date(...)`, not just `Date(...)`. Calling the constructor as a function (that is, not including `new`) returns a string representation of the date, not an actual `Date` object. This is not MongoDB's choice; it is how JavaScript works. If you are not careful to always use the `Date` constructor, you can end up with a mishmash of strings and dates. Strings do not match dates, and vice versa, so this can cause problems with removing, updating, querying...pretty much everything.

For a full explanation of JavaScript's `Date` class and acceptable formats for the constructor, see ECMAScript specification section 15.9 (available for download at *http://www.ecmascript.org*).

Dates in the shell are displayed using local time zone settings. However, dates in the database are just stored as milliseconds since the epoch, so they have no time zone information associated with them. (Time zone information could, of course, be stored as the value for another key.)

Arrays

Arrays are values that can be interchangeably used for both ordered operations (as though they were lists, stacks, or queues) and unordered operations (as though they were sets).

In the following document, the key `"things"` has an array value:

```
{"things" : ["pie", 3.14]}
```

As we can see from the example, arrays can contain different data types as values (in this case, a string and a floating-point number). In fact, array values can be any of the supported values for normal key/value pairs, even nested arrays.

One of the great things about arrays in documents is that MongoDB "understands" their structure and knows how to "reach inside" of arrays to perform operations on their contents. This allows us to query on arrays and build indexes using their contents. For instance, in the previous example, MongoDB can query for all documents where 3.14 is an element of the `"things"` array. If this is a common query, you can even create an index on the `"things"` key to improve the query's speed.

MongoDB also allows atomic updates that modify the contents of arrays, such as reaching into the array and changing the value *pie* to *pi*. We'll see more examples of these types of operations throughout the text.

Embedded Documents

Embedded documents are entire MongoDB documents that are used as the *value* for a key in another document. They can be used to organize data in a more natural way than just a flat structure.

For example, if we have a document representing a person and want to store his address, we can nest this information in an embedded `"address"` document:

```
{
    "name" : "John Doe",
    "address" : {
        "street" : "123 Park Street",
        "city" : "Anytown",
        "state" : "NY"
    }
}
```

The value for the `"address"` key in the previous example is another document with its own values for `"street"`, `"city"`, and `"state"`.

As with arrays, MongoDB "understands" the structure of embedded documents and is able to "reach inside" of them to build indexes, perform queries, or make updates.

We'll discuss schema design in depth later, but even from this basic example, we can begin to see how embedded documents can change the way we work with data. In a relational database, the previous document would probably be modeled as two separate rows in two different tables (one for "people" and one for "addresses"). With MongoDB we can embed the address document directly within the person document. When used properly, embedded documents can provide a more natural (and often more efficient) representation of information.

The flip side of this is that we are basically denormalizing, so there can be more data repetition with MongoDB. Suppose "addresses" were a separate table in a relational database and we needed to fix a typo in an address. When we did a join with "people" and "addresses," we'd get the updated address for everyone who shares it. With MongoDB, we'd need to fix the typo in each person's document.

_id and ObjectIds

Every document stored in MongoDB must have an `"_id"` key. The `"_id"` key's value can be any type, but it defaults to an `ObjectId`. In a single collection, every document must have a unique value for `"_id"`, which ensures that every document in a collection can be uniquely identified. That is, if you had two collections, each one could have a document where the value for `"_id"` was 123. However, neither collection could contain more than one document where `"_id"` was 123.

ObjectIds

`ObjectId` is the default type for `"_id"`. It is designed to be lightweight, while still being easy to generate in a globally unique way across disparate machines. This is the main reason why MongoDB uses `ObjectId`s as opposed to something more traditional, like an autoincrementing primary key: it is difficult and time-consuming to synchronize autoincrementing primary keys across multiple servers. Because MongoDB was designed from the beginning to be a distributed database, dealing with many nodes is an important consideration. The `ObjectId` type, as we'll see, is easy to generate in a sharded environment.

`ObjectId`s use 12 bytes of storage, which gives them a string representation that is 24 hexadecimal digits: 2 digits for each byte. This causes them to appear larger than they are, which makes some people nervous. It's important to note that even though an `ObjectId` is often represented as a giant hexadecimal string, the string is actually twice as long as the data being stored.

If you create multiple new `ObjectId`s in rapid succession, you can see that only the last few digits change each time. In addition, a couple of digits in the middle of the `ObjectId` will change (if you space the creations out by a couple of seconds). This is because of the manner in which `ObjectId`s are created. The 12 bytes of an `ObjectId` are generated as follows:

0	1	2	3	4	5	6	7	8	9	10	11
Timestamp				Machine			PID		Increment		

The first four bytes of an `ObjectId` are a timestamp in seconds since the epoch. This provides a couple of useful properties:

- The timestamp, when combined with the next five bytes (which will be described in a moment), provides uniqueness at the granularity of a second.
- Because the timestamp comes first, it means that `ObjectId`s will sort in *roughly* insertion order. This is not a strong guarantee but does have some nice properties, such as making `ObjectId`s efficient to index.
- In these four bytes exists an implicit timestamp of when each document was created. Most drivers expose a method for extracting this information from an `ObjectId`.

Because the current time is used in `ObjectId`s, some users worry that their servers will need to have synchronized clocks. This is not necessary because the actual value of the timestamp doesn't matter, only that it is often new (once per second) and increasing.

The next three bytes of an `ObjectId` are a unique identifier of the machine on which it was generated. This is usually a hash of the machine's hostname. By including these bytes, we guarantee that different machines will not generate colliding `ObjectId`s.

To provide uniqueness among different processes generating `ObjectId`s concurrently on a single machine, the next two bytes are taken from the process identifier (PID) of the `ObjectId`-generating process.

These first nine bytes of an `ObjectId` guarantee its uniqueness across machines and processes for a single second. The last three bytes are simply an incrementing counter that is responsible for uniqueness within a second in a single process. This allows for up to 256^3 (16,777,216) unique `ObjectId`s to be generated *per process* in a single second.

Autogeneration of _id

As stated previously, if there is no `"_id"` key present when a document is inserted, one will be automatically added to the inserted document. This can be handled by the MongoDB server but will generally be done by the driver on the client side. There are a couple of reasons for that:

- Although `ObjectId`s are designed to be lightweight and easy to generate, there is still some overhead involved in their generation. The decision to generate them on the client side reflects an overall philosophy of MongoDB: work should be pushed out of the server and to the drivers whenever possible. This philosophy reflects the fact that, even with scalable databases like MongoDB, it is easier to scale out at the application layer than at the database layer. Moving work to the client side reduces the burden requiring the database to scale.
- By generating `ObjectId`s on the client side, drivers are capable of providing richer APIs than would be otherwise possible. For example, a driver might have its `insert` method either return the generated `ObjectId` or inject it directly into the document that was inserted. If the driver allowed the server to generate `ObjectId`s, then a separate query would be required to determine the value of `"_id"` for an inserted document.

CHAPTER 3

Creating, Updating, and Deleting Documents

This chapter covers the basics of moving data into and out of the database, including the following:

- Adding new documents to a collection
- Removing documents from a collection
- Updating existing documents
- Choosing the correct level of safety versus speed for all of these operations

Inserting and Saving Documents

Inserts are the basic method for adding data to MongoDB. To insert a document into a collection, use the collection's `insert` method:

```
> db.foo.insert({"bar" : "baz"})
```

This will add an `"_id"` key to the document (if one does not already exist) and save it to MongoDB.

Batch Insert

If you have a situation where you are inserting multiple documents into a collection, you can make the insert faster by using batch inserts. Batch inserts allow you to pass an array of documents to the database.

Sending dozens, hundreds, or even thousands of documents at a time can make inserts significantly faster. A batch insert is a single TCP request, meaning that you do not incur the overhead of doing hundreds of individual requests. It can also cut insert time by eliminating a lot of the header processing that gets done for each message. When an individual document is sent to the database, it is prefixed by a header that tells the database to do an insert operation on a certain collection. By using batch insert, the database doesn’t need to reprocess this information for each document.

Batch inserts are intended to be used in applications, such as for inserting a couple hundred sensor data points into an analytics collection at once. They are useful only if you are inserting multiple documents into a single collection: you cannot use batch inserts to insert into multiple collections with a single request. If you are just importing raw data (for example, from a data feed or MySQL), there are command-line tools like `mongoimport` that can be used instead of batch insert. On the other hand, it is often handy to munge data before saving it to MongoDB (converting dates to the date type or adding a custom `"_id"`) so batch inserts can be used for importing data, as well.

Current versions of MongoDB do not accept messages longer than 16MB, so there is a limit to how much can be inserted in a single batch insert.

Inserts: Internals and Implications

When you perform an insert, the driver you are using converts the data structure into BSON, which it then sends to the database (see Appendix C for more on BSON). The database understands BSON and checks for an `"_id"` key and that the document’s size does not exceed 4MB, but other than that, it doesn’t do data validation; it just saves the document to the database as is. This has a couple of side effects, most notably that you can insert invalid data and that your database is fairly secure from injection attacks.

All of the drivers for major languages (and most of the minor ones, too) check for a variety of invalid data (documents that are too large, contain non-UTF-8 strings, or use unrecognized types) before sending anything to the database. If you are running a driver that you are not sure about, you can start the database server with the `--objcheck` option, and it will examine each document’s structural validity before inserting it (at the cost of slower performance).

> Documents larger than 4MB (when converted to BSON) cannot be saved to the database. This is a somewhat arbitrary limit (and may be raised in the future); it is mostly to prevent bad schema design and ensure consistent performance. To see the BSON size (in bytes) of the document *doc*, run `Object.bsonsize(`*`doc`*`)` from the shell.
>
> To give you an idea of how much 4MB is, the entire text of *War and Peace* is just 3.14MB.

MongoDB does not do any sort of code execution on inserts, so they are not vulnerable to injection attacks. Traditional injection attacks are impossible with MongoDB, and alternative injection-type attacks are easy to guard against in general, but inserts are particularly invulnerable.

Removing Documents

Now that there's data in our database, let's delete it.

```
> db.users.remove()
```

This will remove all of the documents in the *users* collection. This doesn't actually remove the collection, and any indexes created on it will still exist.

The `remove` function optionally takes a query document as a parameter. When it's given, only documents that match the criteria will be removed. Suppose, for instance, that we want to remove everyone from the *mailing.list* collection where the value for `"opt-out"` is `true`:

```
> db.mailing.list.remove({"opt-out" : true})
```

Once data has been removed, it is gone forever. There is no way to undo the remove or recover deleted documents.

Remove Speed

Removing documents is usually a fairly quick operation, but if you want to clear an entire collection, it is faster to *drop* it (and then re-create any indexes).

For example, in Python, suppose we insert a million dummy elements with the following:

```
for i in range(1000000):
    collection.insert({"foo": "bar", "baz": i, "z": 10 - i})
```

Now we'll try to remove all of the documents we just inserted, measuring the time it takes. First, here's a simple `remove`:

```
import time

from pymongo import Connection

db = Connection().foo
collection = db.bar

start = time.time()

collection.remove()
collection.find_one()

total = time.time() - start
print "%d seconds" % total
```

On a MacBook Air, this script prints "46.08 seconds."

If the `remove` and `find_one` are replaced by `db.drop_collection("bar")`, the time drops to .01 seconds! This is obviously a vast improvement, but it comes at the expense of granularity: we cannot specify any criteria. The whole collection is dropped, and all of its indexes are deleted.

Updating Documents

Once a document is stored in the database, it can be changed using the `update` method. `update` takes two parameters: a query document, which locates documents to update, and a modifier document, which describes the changes to make to the documents found.

Updates are atomic: if two updates happen at the same time, whichever one reaches the server first will be applied, and then the next one will be applied. Thus, conflicting updates can safely be sent in rapid-fire succession without any documents being corrupted: the last update will "win."

Document Replacement

The simplest type of update fully replaces a matching document with a new one. This can be useful to do a dramatic schema migration. For example, suppose we are making major changes to a user document, which looks like the following:

```
{
    "_id" : ObjectId("4b2b9f67a1f631733d917a7a"),
    "name" : "joe",
    "friends" : 32,
    "enemies" : 2
}
```

We want to change that document into the following:

```
{
    "_id" : ObjectId("4b2b9f67a1f631733d917a7a"),
    "username" : "joe",
    "relationships" :
        {
            "friends" : 32,
            "enemies" : 2
        }
}
```

We can make this change by replacing the document using an `update`:

```
> var joe = db.users.findOne({"name" : "joe"});
> joe.relationships = {"friends" : joe.friends, "enemies" : joe.enemies};
{
    "friends" : 32,
    "enemies" : 2
}
```

```
> joe.username = joe.name;
"joe"
> delete joe.friends;
true
> delete joe.enemies;
true
> delete joe.name;
true
> db.users.update({"name" : "joe"}, joe);
```

Now, doing a `findOne` shows that the structure of the document has been updated.

A common mistake is matching more than one document with the criteria and then create a duplicate `"_id"` value with the second parameter. The database will throw an error for this, and nothing will be changed.

For example, suppose we create several documents with the same `"name"`, but we don't realize it:

```
> db.people.find()
{"_id" : ObjectId("4b2b9f67a1f631733d917a7b"), "name" : "joe", "age" : 65},
{"_id" : ObjectId("4b2b9f67a1f631733d917a7c"), "name" : "joe", "age" : 20},
{"_id" : ObjectId("4b2b9f67a1f631733d917a7d"), "name" : "joe", "age" : 49},
```

Now, if it's Joe #2's birthday, we want to increment the value of his `"age"` key, so we might say this:

```
> joe = db.people.findOne({"name" : "joe", "age" : 20});
{
    "_id" : ObjectId("4b2b9f67a1f631733d917a7c"),
    "name" : "joe",
    "age" : 20
}
> joe.age++;
> db.people.update({"name" : "joe"}, joe);
E11001 duplicate key on update
```

What happened? When you call update, the database will look for a document matching `{"name" : "joe"}`. The first one it finds will be the 65-year-old Joe. It will attempt to replace that document with the one in the `joe` variable, but there's already a document in this collection with the same `"_id"`. Thus, the update will fail, because `"_id"` values must be unique. The best way to avoid this situation is to make sure that your update always specifies a unique document, perhaps by matching on a key like `"_id"`.

Using Modifiers

Usually only certain portions of a document need to be updated. Partial updates can be done extremely efficiently by using atomic *update modifiers*. Update modifiers are special keys that can be used to specify complex update operations, such as altering, adding, or removing keys, and even manipulating arrays and embedded documents.

Suppose we were keeping website analytics in a collection and wanted to increment a counter each time someone visited a page. We can use update modifiers to do this increment atomically. Each URL and its number of page views is stored in a document that looks like this:

```
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "url" : "www.example.com",
    "pageviews" : 52
}
```

Every time someone visits a page, we can find the page by its URL and use the `"$inc"` modifier to increment the value of the `"pageviews"` key.

```
> db.analytics.update({"url" : "www.example.com"},
... {"$inc" : {"pageviews" : 1}})
```

Now, if we do a `find`, we see that `"pageviews"` has increased by one.

```
> db.analytics.find()
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "url" : "www.example.com",
    "pageviews" : 53
}
```

> Perl and PHP programmers are probably thinking that any character would have been a better choice than $. Both of these languages use $ as a variable prefix and will replace $-prefixed strings with their variable value in double-quoted strings. However, MongoDB started out as a JavaScript database, and $ is a special character that isn't interpreted differently in JavaScript, so it was used. It is an annoying historical relic from MongoDB's primordial soup.
>
> There are several options for Perl and PHP programmers. First, you could just escape the $: `"\$foo"`. You can use single quotes, which don't do variable interpolation: `'$foo'`. Finally, both drivers allow you to define your own character that will be used instead of $. In Perl, set `$MongoDB::BSON::char`, and in PHP set `mongo.cmd_char` in *php.ini* to =, :, ?, or any other character that you would like to use instead of $. Then, if you choose, say, ~, you would use ~inc instead of \$inc and ~gt instead of \$gt.
>
> Good choices for the special character are characters that will not naturally appear in key names (don't use _ or x) and are not characters that have to be escaped themselves, which will gain you nothing and be confusing (such as \ or, in Perl, @).

When using modifiers, the value of `"_id"` cannot be changed. (Note that `"_id"` can be changed by using whole-document replacement.) Values for any other key, including other uniquely indexed keys, can be modified.

Getting started with the "$set" modifier

`"$set"` sets the value of a key. If the key does not yet exist, it will be created. This can be handy for updating schema or adding user-defined keys. For example, suppose you have a simple user profile stored as a document that looks something like the following:

```
> db.users.findOne()
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "name" : "joe",
    "age" : 30,
    "sex" : "male",
    "location" : "Wisconsin"
}
```

This is a pretty bare-bones user profile. If the user wanted to store his favorite book in his profile, he could add it using `"$set"`:

```
> db.users.update({"_id" : ObjectId("4b253b067525f35f94b60a31")},
... {"$set" : {"favorite book" : "war and peace"}})
```

Now the document will have a "favorite book" key:

```
> db.users.findOne()
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "name" : "joe",
    "age" : 30,
    "sex" : "male",
    "location" : "Wisconsin",
    "favorite book" : "war and peace"
}
```

If the user decides that he actually enjoys a different book, `"$set"` can be used again to change the value:

```
> db.users.update({"name" : "joe"},
... {"$set" : {"favorite book" : "green eggs and ham"}})
```

`"$set"` can even change the type of the key it modifies. For instance, if our fickle user decides that he actually likes quite a few books, he can change the value of the "favorite book" key into an array:

```
> db.users.update({"name" : "joe"},
... {"$set" : {"favorite book" :
...     ["cat's cradle", "foundation trilogy", "ender's game"]}})
```

If the user realizes that he actually doesn't like reading, he can remove the key altogether with `"$unset"`:

```
> db.users.update({"name" : "joe"},
... {"$unset" : {"favorite book" : 1}})
```

Now the document will be the same as it was at the beginning of this example.

You can also use `"$set"` to reach in and change embedded documents:

```
> db.blog.posts.findOne()
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "title" : "A Blog Post",
    "content" : "...",
    "author" : {
        "name" : "joe",
        "email" : "joe@example.com"
    }
}
> db.blog.posts.update({"author.name" : "joe"}, {"$set" : {"author.name" : "joe schmoe"}})
> db.blog.posts.findOne()
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "title" : "A Blog Post",
    "content" : "...",
    "author" : {
        "name" : "joe schmoe",
        "email" : "joe@example.com"
    }
}
```

You must always use a $ modifier for adding, changing, or removing keys. A common error people often make when starting out is to try to set the value of `"foo"` to `"bar"` by doing an update that looks like this:

```
> db.coll.update(criteria, {"foo" : "bar"})
```

This will not function as intended. It actually does a full-document replacement, replacing the matched document with `{"foo" : "bar"}`. Always use $ operators for modifying individual key/value pairs.

Incrementing and decrementing

The `"$inc"` modifier can be used to change the value for an existing key or to create a new key if it does not already exist. It is very useful for updating analytics, karma, votes, or anything else that has a changeable, numeric value.

Suppose we are creating a game collection where we want to save games and update scores as they change. When a user starts playing, say, a game of pinball, we can insert a document that identifies the game by name and user playing it:

```
> db.games.insert({"game" : "pinball", "user" : "joe"})
```

When the ball hits a bumper, the game should increment the player's score. As points in pinball are given out pretty freely, let's say that the base unit of points a player can earn is 50. We can use the `"$inc"` modifier to add 50 to the player's score:

```
> db.games.update({"game" : "pinball", "user" : "joe"},
... {"$inc" : {"score" : 50}})
```

If we look at the document after this update, we'll see the following:

```
> db.games.findOne()
{
     "_id" : ObjectId("4b2d75476cc613d5ee930164"),
     "game" : "pinball",
     "name" : "joe",
     "score" : 50
}
```

The score key did not already exist, so it was created by `"$inc"` and set to the increment amount: 50.

If the ball lands in a "bonus" slot, we want to add 10,000 to the score. This can be accomplished by passing a different value to `"$inc"`:

```
> db.games.update({"game" : "pinball", "user" : "joe"},
... {"$inc" : {"score" : 10000}})
```

Now if we look at the game, we'll see the following:

```
> db.games.find()
{
     "_id" : ObjectId("4b2d75476cc613d5ee930164"),
     "game" : "pinball",
     "name" : "joe",
     "score" : 10050
}
```

The `"score"` key existed and had a numeric value, so the server added 10,000 to it.

`"$inc"` is similar to `"$set"`, but it is designed for incrementing (and decrementing) numbers. `"$inc"` can be used only on values of type integer, long, or double. If it is used on any other type of value, it will fail. This includes types that many languages will automatically cast into numbers, like nulls, booleans, or strings of numeric characters:

```
> db.foo.insert({"count" : "1"})
> db.foo.update({}, {$inc : {count : 1}})
Cannot apply $inc modifier to non-number
```

Also, the value of the `"$inc"` key must be a number. You cannot increment by a string, array, or other non-numeric value. Doing so will give a "Modifier `"$inc"` allowed for numbers only" error message. To modify other types, use `"$set"` or one of the array operations described in a moment.

Array modifiers

An extensive class of modifiers exists for manipulating arrays. Arrays are common and powerful data structures: not only are they lists that can be referenced by index, but they can also double as sets.

Array operators can be used only on keys with array values. For example, you cannot push on to an integer or pop off of a string, for example. Use `"$set"` or `"$inc"` to modify scalar values.

`"$push"` adds an element to the end of an array if the specified key already exists and creates a new array if it does not. For example, suppose that we are storing blog posts and want to add a `"comments"` key containing an array. We can push a comment onto the nonexistent `"comments"` array, which will create the array and add the comment:

```
> db.blog.posts.findOne()
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "title" : "A blog post",
    "content" : "..."
}
> db.blog.posts.update({"title" : "A blog post"}, {$push : {"comments" :
... {"name" : "joe", "email" : "joe@example.com", "content" : "nice post."}}})
> db.blog.posts.findOne()
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "title" : "A blog post",
    "content" : "...",
    "comments" : [
        {
            "name" : "joe",
            "email" : "joe@example.com",
            "content" : "nice post."
        }
    ]
}
```

Now, if we want to add another comment, we can simple use `"$push"` again:

```
> db.blog.posts.update({"title" : "A blog post"}, {$push : {"comments" :
... {"name" : "bob", "email" : "bob@example.com", "content" : "good post."}}})
> db.blog.posts.findOne()
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "title" : "A blog post",
    "content" : "...",
    "comments" : [
        {
            "name" : "joe",
            "email" : "joe@example.com",
            "content" : "nice post."
        },
        {
            "name" : "bob",
            "email" : "bob@example.com",
            "content" : "good post."
        }
    ]
}
```

A common use is wanting to add a value to an array only if the value is not already present. This can be done using a `"$ne"` in the query document. For example, to push an author onto a list of citations, but only if he isn't already there, use the following:

```
> db.papers.update({"authors cited" : {"$ne" : "Richie"}},
... {$push : {"authors cited" : "Richie"}})
```

This can also be done with `"$addToSet"`, which is useful for cases where `"$ne"` won't work or where `"$addToSet"` describes what is happening better.

For instance, suppose you have a document that represents a user. You might have a set of email addresses that they have added:

```
> db.users.findOne({"_id" : ObjectId("4b2d75476cc613d5ee930164")})
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "username" : "joe",
    "emails" : [
        "joe@example.com",
        "joe@gmail.com",
        "joe@yahoo.com"
    ]
}
```

When adding another address, you can use `"$addToSet"` to prevent duplicates:

```
> db.users.update({"_id" : ObjectId("4b2d75476cc613d5ee930164")},
... {"$addToSet" : {"emails" : "joe@gmail.com"}})
> db.users.findOne({"_id" : ObjectId("4b2d75476cc613d5ee930164")})
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "username" : "joe",
    "emails" : [
        "joe@example.com",
        "joe@gmail.com",
        "joe@yahoo.com",
    ]
}
> db.users.update({"_id" : ObjectId("4b2d75476cc613d5ee930164")},
... {"$addToSet" : {"emails" : "joe@hotmail.com"}})
> db.users.findOne({"_id" : ObjectId("4b2d75476cc613d5ee930164")})
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "username" : "joe",
    "emails" : [
        "joe@example.com",
        "joe@gmail.com",
        "joe@yahoo.com",
        "joe@hotmail.com"
    ]
}
```

You can also use `"$addToSet"` in conjunction with `"$each"` to add multiple unique values, which cannot be done with the `"$ne"`/`"$push"` combination. For instance, we could use these modifiers if the user wanted to add more than one email address:

```
> db.users.update({"_id" : ObjectId("4b2d75476cc613d5ee930164")}, {"$addToSet" :
... {"emails" : {"$each" : ["joe@php.net", "joe@example.com", "joe@python.org"]}}})
> db.users.findOne({"_id" : ObjectId("4b2d75476cc613d5ee930164")})
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "username" : "joe",
    "emails" : [
        "joe@example.com",
        "joe@gmail.com",
        "joe@yahoo.com",
        "joe@hotmail.com"
        "joe@php.net"
        "joe@python.org"
    ]
}
```

There are a few ways to remove elements from an array. If you want to treat the array like a queue or a stack, you can use `"$pop"`, which can remove elements from either end. `{$pop : {`*`key`*` : 1}}` removes an element from the end of the array. `{$pop : {`*`key`*` : -1}}` removes it from the beginning.

Sometimes an element should be removed based on specific criteria, rather than its position in the array. `"$pull"` is used to remove elements of an array that match the given criteria. For example, suppose we have a list of things that need to be done but not in any specific order:

```
> db.lists.insert({"todo" : ["dishes", "laundry", "dry cleaning"]})
```

If we do the laundry first, we can remove it from the list with the following:

```
> db.lists.update({}, {"$pull" : {"todo" : "laundry"}})
```

Now if we do a find, we'll see that there are only two elements remaining in the array:

```
> db.lists.find()
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "todo" : [
        "dishes",
        "dry cleaning"
    ]
}
```

Pulling removes all matching documents, not just a single match. If you have an array that looks like `[1, 1, 2, 1]` and pull `1`, you'll end up with a single-element array, `[2]`.

Positional array modifications

Array manipulation becomes a little trickier when we have multiple values in an array and want to modify some of them. There are two ways to manipulate values in arrays: by position or by using the position operator (the `"$"` character).

Arrays use 0-based indexing, and elements can be selected as though their index were a document key. For example, suppose we have a document containing an array with a few embedded documents, such as a blog post with comments:

```
> db.blog.posts.findOne()
{
    "_id" : ObjectId("4b329a216cc613d5ee930192"),
    "content" : "...",
    "comments" : [
        {
            "comment" : "good post",
            "author" : "John",
            "votes" : 0
        },
        {
            "comment" : "i thought it was too short",
            "author" : "Claire",
            "votes" : 3
        },
        {
            "comment" : "free watches",
            "author" : "Alice",
            "votes" : -1
        }
    ]
}
```

If we want to increment the number of votes for the first comment, we can say the following:

```
> db.blog.update({"post" : post_id},
... {"$inc" : {"comments.0.votes" : 1}})
```

In many cases, though, we don't know what index of the array to modify without querying for the document first and examining it. To get around this, MongoDB has a positional operator, `"$"`, that figures out which element of the array the query document matched and updates that element. For example, if we have a user named John who updates his name to Jim, we can replace it in the comments by using the positional operator:

```
db.blog.update({"comments.author" : "John"},
... {"$set" : {"comments.$.author" : "Jim"}})
```

The positional operator updates only the first match. Thus, if John had left more than one comment, his name would be changed only for the first comment he left.

Modifier speed

Some modifiers are faster than others. `$inc` modifies a document in place: it does not have to change the size of a document, only the value of a key, so it is very efficient. On the other hand, array modifiers might change the size of a document and can be slow. (`"$set"` can modify documents in place if the size isn't changing but otherwise is subject to the same performance limitations as array operators.)

MongoDB leaves some padding around a document to allow for changes in size (and, in fact, figures out how much documents usually change in size and adjusts the amount of padding it leaves accordingly), but it will eventually have to allocate new space for a document if you make it much larger than it was originally. Compounding this slowdown, as arrays get longer, it takes MongoDB a longer amount of time to traverse the whole array, slowing down each array modification.

A simple program in Python can demonstrate the speed difference. This program inserts a single key and increments its value 100,000 times.

```
from pymongo import Connection

import time

db = Connection().performance_test
db.drop_collection("updates")
collection = db.updates

collection.insert({"x": 1})

# make sure the insert is complete before we start timing
collection.find_one()

start = time.time()

for i in range(100000):
    collection.update({}, {"$inc" : {"x" : 1}})

# make sure the updates are complete before we stop timing
collection.find_one()

print time.time() - start
```

On a MacBook Air this took 7.33 seconds. That's more than 13,000 updates per second (which is pretty good for a fairly anemic machine). Now, let's try it with a document with a single array key, pushing new values onto that array 100,000 times:

```
for i in range(100000):
    collection.update({}, {'$push' : {'x' : 1}})
```

This program took 67.58 seconds to run, which is less than 1,500 updates per second.

Using `"$push"` and other array modifiers is encouraged and often necessary, but it is good to keep in mind the trade-offs of such updates. If `"$push"` becomes a bottleneck, it may be worth pulling an embedded array out into a separate collection.

Upserts

An *upsert* is a special type of update. If no document is found that matches the update criteria, a new document will be created by combining the criteria and update documents. If a matching document is found, it will be updated normally. Upserts can be very handy because they eliminate the need to "seed" your collection: you can have the same code create and update documents.

Let's go back to our example recording the number of views for each page of a website. Without an upsert, we might try to find the URL and increment the number of views or create a new document if the URL doesn't exist. If we were to write this out as a

JavaScript program (instead of a series of shell commands—scripts can be run with `mongo` *`scriptname.js`*), it might look something like the following:

```
// check if we have an entry for this page
blog = db.analytics.findOne({url : "/blog"})

// if we do, add one to the number of views and save
if (blog) {
  blog.pageviews++;
  db.analytics.save(blog);
}
// otherwise, create a new document for this page
else {
  db.analytics.save({url : "/blog", pageviews : 1})
}
```

This means we are making a round-trip to the database, plus sending an update or insert, every time someone visits a page. If we are running this code in multiple processes, we are also subject to a race condition where more than one document can be inserted for a given URL.

We can eliminate the race condition and cut down on the amount of code by just sending an upsert (the third parameter to `update` specifies that this should be an upsert):

```
db.analytics.update({"url" : "/blog"}, {"$inc" : {"visits" : 1}}, true)
```

This line does exactly what the previous code block does, except it's faster and atomic! The new document is created using the criteria document as a base and applying any modifier documents to it. For example, if you do an upsert that matches a key and has an increment to the value of that key, the increment will be applied to the match:

```
> db.math.remove()
> db.math.update({"count" : 25}, {"$inc" : {"count" : 3}}, true)
> db.math.findOne()
{
    "_id" : ObjectId("4b3295f26cc613d5ee93018f"),
    "count" : 28
}
```

The `remove` empties the collection, so there are no documents. The upsert creates a new document with a `"count"` of 25 and then increments that by 3, giving us a document where `"count"` is 28. If the upsert option were not specified, `{"count" : 25}` would not match any documents, so nothing would happen.

If we run the upsert again (with the criteria `{count : 25}`), it will create another new document. This is because the criteria does not match the only document in the collection. (Its `"count"` is 28.)

The save Shell Helper

`save` is a shell function that lets you insert a document if it doesn't exist and update it if it does. It takes one argument: a document. If the document contains an `"_id"` key, `save` will do an upsert. Otherwise, it will do an insert. This is just a convenience function so that programmers can quickly modify documents in the shell:

```
> var x = db.foo.findOne()
> x.num = 42
42
> db.foo.save(x)
```

Without `save`, the last line would have been a more cumbersome `db.foo.update({"_id" : x._id}, x)`.

Updating Multiple Documents

Updates, by default, update only the first document found that matches the criteria. If there are more matching documents, they will remain unchanged. To modify all of the documents matching the criteria, you can pass `true` as the fourth parameter to `update`.

> `update`'s behavior may be changed in the future (the server may update all matching documents by default and update one only if `false` is passed as the fourth parameter), so it is recommended that you always specify whether you want a multiple update.
>
> Not only is it more obvious what the update should be doing, but your program won't break if the default is ever changed.

Multiupdates are a great way of performing schema migrations or rolling out new features to certain users. Suppose, for example, we want to give a gift to every user who has a birthday on a certain day. We can use multiupdate to add a `"gift"` to their account:

```
> db.users.update({birthday : "10/13/1978"},
... {$set : {gift : "Happy Birthday!"}}, false, true)
```

This would add the `"gift"` key to all user documents with birthdays on October 13, 1978.

To see the number of documents updated by a multiple update, you can run the `getLastError` database command (which might be better named `"getLastOpStatus"`). The `"n"` key will contain the number of documents affected by the update:

```
> db.count.update({x : 1}, {$inc : {x : 1}}, false, true)
> db.runCommand({getLastError : 1})
{
    "err" : null,
    "updatedExisting" : true,
    "n" : 5,
    "ok" : true
}
```

`"n"` is `5`, meaning that five documents were affected by the update. `"updatedExisting"` is `true`, meaning that the update modified existing document(s). For more on database commands and their responses, see Chapter 7.

Returning Updated Documents

You can get some limited information about what was updated by calling `getLastError`, but it does not actually return the updated document. For that, you'll need the `findAndModify` command.

`findAndModify` is called differently than a normal update and is a bit slower, because it must wait for a database response. It is handy for manipulating queues and performing other operations that need get-and-set style atomicity.

Suppose we have a collection of processes run in a certain order. Each is represented with a document that has the following form:

```
{
    "_id" : ObjectId(),
    "status" : state,
    "priority" : N
}
```

`"status"` is a string that can be "READY," "RUNNING," or "DONE." We need to find the job with the highest priority in the "READY" state, run the process function, and then update the status to "DONE." We might try querying for the ready processes, sorting by priority, and updating the status of the highest-priority process to mark it is "RUNNING." Once we have processed it, we update the status to "DONE." This looks something like the following:

```
ps = db.processes.find({"status" : "READY").sort({"priority" : -1}).limit(1).next()
db.processes.update({"_id" : ps._id}, {"$set" : {"status" : "RUNNING"}})
do_something(ps);
db.processes.update({"_id" : ps._id}, {"$set" : {"status" : "DONE"}})
```

This algorithm isn't great, because it is subject to a race condition. Suppose we have two threads running. If one thread (call it A) retrieved the document and another thread (call it B) retrieved the same document before A had updated its status to "RUNNING," then both threads would be running the same process. We can avoid this by checking the status as part of the update query, but this becomes complex:

```
var cursor = db.processes.find({"status" : "READY"}).sort({"priority" : -1}).limit(1);
while ((ps = cursor.next()) != null) {
    ps.update({"_id" : ps._id, "status" : "READY"},
              {"$set" : {"status" : "RUNNING"}});

    var lastOp = db.runCommand({getlasterror : 1});
    if (lastOp.n == 1) {
        do_something(ps);
        db.processes.update({"_id" : ps._id}, {"$set" : {"status" : "DONE"}})
        break;
    }
    cursor =  db.processes.find({"status" : "READY"}).sort({"priority" : -1}).limit(1);
}
```

Also, depending on timing, one thread may end up doing all the work while another thread is uselessly trailing it. Thread A could always grab the process, and then B would try to get the same process, fail, and leave A to do all the work. Situations like this are perfect for `findAndModify`. `findAndModify` can return the item and update it in a single operation. In this case, it looks like the following:

```
> ps = db.runCommand({"findAndModify" : "processes",
... "query" : {"status" : "READY"},
... "sort" : {"priority" : -1},
... "update" : {"$set" : {"status" : "RUNNING"}})
{
    "ok" : 1,
    "value" : {
        "_id" : ObjectId("4b3e7a18005cab32be6291f7"),
        "priority" : 1,
        "status" : "READY"
    }
}
```

Notice that the status is still "READY" in the returned document. The document is returned before the modifier document is applied. If you do a find on the collection, though, you will see that the document's `"status"` has been updated to "RUNNING":

```
> db.processes.findOne({"_id" : ps.value._id})
{
    "_id" : ObjectId("4b3e7a18005cab32be6291f7"),
    "priority" : 1,
    "status" : "RUNNING"
}
```

Thus, the program becomes the following:

```
> ps = db.runCommand({"findAndModify" : "processes",
... "query" : {"status" : "READY"},
... "sort" : {"priority" : -1},
... "update" : {"$set" : {"status" : "RUNNING"}}).value
> do_something(ps)
> db.process.update({"_id" : ps._id}, {"$set" : {"status" : "DONE"}})
```

`findAndModify` can have either an `"update"` key or a `"remove"` key. A `"remove"` key indicates that the matching document should be removed from the collection. For instance, if we wanted to simply remove the job instead of updating its status, we could run the following:

```
> ps = db.runCommand({"findAndModify" : "processes",
... "query" : {"status" : "READY"},
... "sort" : {"priority" : -1},
... "remove" : true).value
> do_something(ps)
```

The values for each key in the `findAndModify` command are as follows:

`findAndModify`
: A string, the collection name.

`query`
: A query document, the criteria with which to search for documents.

`sort`
: Criteria by which to sort results.

`update`
: A modifier document, the update to perform on the document found.

`remove`
: Boolean specifying whether the document should be removed.

`new`
: Boolean specifying whether the document returned should be the updated document or the preupdate document. Defaults to the preupdate document.

Either `"update"` or `"remove"` must be included, but not both. If no matching document is found, the command will return an error.

`findAndModify` has a few limitations. First, it can update or remove only one document at a time. There is also no way to use it for an upsert; it can update only existing documents.

The price of using `findAndModify` over a traditional update is speed: it is a bit slower. That said, it is no slower than one might expect: it takes roughly the same amount of time as a `find`, `update`, and `getLastError` command performed in serial.

The Fastest Write This Side of Mississippi

The three operations that this chapter focused on (inserts, removes, and updates) seem instantaneous because none of them waits for a database response. They are not asynchronous; they can be thought of as "fire-and-forget" functions: the client sends the documents to the server and immediately continues. The client never receives an "OK, got that" or a "not OK, could you send that again?" response.

The benefit to this is that the speed at which you can perform these operations is terrific. You are often only limited by the speed at which your client can send them and the speed of your network. This works well most of the time; however, sometimes something goes wrong: a server crashes, a rat chews through a network cable, or a data center is in a flood zone. If the server disappears, the client will happily send some writes to a server that isn't there, entirely unaware of its absence. For some applications, this is acceptable. Losing a couple of seconds of log messages, user clicks, or analytics in a hardware failure is not the end of the world. For others, this is not the behavior the programmer wants (payment-processing systems spring to mind).

Safe Operations

Suppose you are writing an ecommerce application. If someone orders something, the application should probably take a little extra time to make sure the order goes through. That is why you can do a "safe" version of these operations, where you check whether there was an error in execution and attempt to redo them.

> MongoDB developers made unchecked operations the default because of their experience with relational databases. Many applications written on top of relational databases do not care about or check the return codes, yet they incur the performance penalty of their application waiting for them to arrive. MongoDB pushes this option to the user. This way, programs that collect log messages or real-time analytics don't have to wait for return codes that they don't care about.

The safe version of these operations runs a `getLastError` command immediately following the operation to check whether it succeeded (see "Database Commands" on page 93 for more on commands). The driver waits for the database response and then handles errors appropriately, throwing a catchable exception in most cases. This way, developers can catch and handle database errors in whatever way feels "natural" for their language. When an operation is successful, the `getLastError` response also contains some additional information (e.g., for an update or remove, it includes the number of documents affected).

> The same `getLastError` command that powers safe mode also contains functionality for checking that operations have been successfully replicated. For more on this feature, see "Blocking for Replication" on page 140.

The price of performing "safe" operations is performance: waiting for a database response takes an order of magnitude longer than sending the message, ignoring the client-side cost of handling exceptions. (This cost varies by language but is usually fairly heavyweight.) Thus, applications should weigh the importance of their data (and the consequences if some of it is lost) versus the speed needed.

> When in doubt, use safe operations. If they aren't fast enough, start making less important operations fire-and-forget.

More specifically:

- If you live dangerously, use fire-and-forget operations exclusively.

- If you want to live longer, save valuable user input (account sign-ups, credit card numbers, emails) with safe operations and do everything else with fire-and-forget operations.
- If you are cautious, use safe operations exclusively. If your application is automatically generating hundreds of little pieces of information to save (e.g., page, user, or advertising statistics), these can still use the fire-and-forget operation.

Catching "Normal" Errors

Safe operations are also a good way to debug "strange" database behavior, not just for preventing the apocalyptic scenarios described earlier. Safe operations should be used extensively while developing, even if they are later removed before going into production. They can protect against many common database usage errors, most commonly duplicate key errors.

Duplicate key errors often occur when users try to insert a document with a duplicate value for the `"_id"` key. MongoDB does not allow multiple documents with the same `"_id"` in the same collection. If you do a safe insert and a duplicate key error occurs, the server error will be picked up by the safety check, and an exception will be thrown. In unsafe mode, there is no database response, and you might not be aware that the insert failed.

For example, using the shell, you can see that inserting two documents with the same `"_id"` will not work:

```
> db.foo.insert({"_id" : 123, "x" : 1})
> db.foo.insert({"_id" : 123, "x" : 2})
E11000 duplicate key error index: test.foo.$_id_  dup key: { : 123.0 }
```

If we examine the collection, we can see that only the first document was successfully inserted. Note that this error can occur with any unique index, not just the one on `"_id"`. The shell always checks for errors; in the drivers it is optional.

Requests and Connections

For each connection to a MongoDB server, the database creates a queue for that connection's requests. When the client sends a request, it will be placed at the end of its connection's queue. Any subsequent requests on the connection will occur after the enqueued operation is processed. Thus, a single connection has a consistent view of the database and can always read its own writes.

Note that this is a per-connection queue: if we open two shells, we will have two connections to the database. If we perform an insert in one shell, a subsequent query in the other shell might not return the inserted document. However, within a single shell, if we query for the document after inserting, the document will be returned. This behavior can be difficult to duplicate by hand, but on a busy server, interleaved inserts/

queries are very likely to occur. Often developers run into this when they insert data in one thread and then check that it was successfully inserted in another. For a second or two, it looks like the data was not inserted, and then it suddenly appears.

This behavior is especially worth keeping in mind when using the Ruby, Python, and Java drivers, because all three drivers use connection pooling. For efficiency, these drivers open multiple connections (a *pool*) to the server and distribute requests across them. They all, however, have mechanisms to guarantee that a series of requests is processed by a single connection. There is detailed documentation on connection pooling in various languages on the MongoDB wiki.

CHAPTER 4

Querying

This chapter looks at querying in detail. The main areas covered are as follows:

- You can perform ad hoc queries on the database using the `find` or `findOne` functions and a query document.
- You can query for ranges, set inclusion, inequalities, and more by using `$` conditionals.
- Some queries cannot be expressed as query documents, even using $ conditionals. For these types of complex queries, you can use a **`$where`** clause to harness the full expressive power of JavaScript.
- Queries return a database cursor, which lazily returns batches of documents as you need them.
- There are a lot of metaoperations you can perform on a cursor, including skipping a certain number of results, limiting the number of results returned, and sorting results.

Introduction to find

The `find` method is used to perform queries in MongoDB. Querying returns a subset of documents in a collection, from no documents at all to the entire collection. Which documents get returned is determined by the first argument to `find`, which is a document specifying the query to be performed.

An empty query document (i.e., `{}`) matches everything in the collection. If `find` isn't given a query document, it defaults to `{}`. For example, the following:

```
> db.c.find()
```

returns everything in the collection *c*.

When we start adding key/value pairs to the query document, we begin restricting our search. This works in a straightforward way for most types. Integers match integers, booleans match booleans, and strings match strings. Querying for a simple type is as easy as specifying the value that you are looking for. For example, to find all documents where the value for `"age"` is 27, we can add that key/value pair to the query document:

```
> db.users.find({"age" : 27})
```

If we have a string we want to match, such as a `"username"` key with the value `"joe"`, we use that key/value pair instead:

```
> db.users.find({"username" : "joe"})
```

Multiple conditions can be strung together by adding more key/value pairs to the query document, which gets interpreted as "condition1 AND condition2 AND ... AND conditionN." For instance, to get all users who are 27-year-olds with the username "joe," we can query for the following:

```
> db.users.find({"username" : "joe", "age" : 27})
```

Specifying Which Keys to Return

Sometimes, you do not need all of the key/value pairs in a document returned. If this is the case, you can pass a second argument to `find` (or `findOne`) specifying the keys you want. This reduces both the amount of data sent over the wire and the time and memory used to decode documents on the client side.

For example, if you have a user collection and you are interested only in the `"username"` and `"email"` keys, you could return just those keys with the following query:

```
> db.users.find({}, {"username" : 1, "email" : 1})
{
    "_id" : ObjectId("4ba0f0dfd22aa494fd523620"),
    "username" : "joe",
    "email" : "joe@example.com"
}
```

As you can see from the previous output, the `"_id"` key is always returned, even if it isn't specifically listed.

You can also use this second parameter to exclude specific key/value pairs from the results of a query. For instance, you may have documents with a variety of keys, and the only thing you know is that you never want to return the `"fatal_weakness"` key:

```
> db.users.find({}, {"fatal_weakness" : 0})
```

This can even prevent `"_id"` from being returned:

```
> db.users.find({}, {"username" : 1, "_id" : 0})
{
    "username" : "joe",
}
```

Limitations

There are some restrictions on queries. The value of a query document must be a constant as far as the database is concerned. (It can be a normal variable in your own code.) That is, it cannot refer to the value of another key in the document. For example, if we were keeping inventory and we had both `"in_stock"` and `"num_sold"` keys, we could compare their values by querying the following:

```
> db.stock.find({"in_stock" : "this.num_sold"}) // doesn't work
```

There are ways to do this (see "$where Queries" on page 55), but you will usually get better performance by restructuring your document slightly, such that a normal query will suffice. In this example, we could instead use the keys `"initial_stock"` and `"in_stock"`. Then, every time someone buys an item, we decrement the value of the `"in_stock"` key by one. Finally, we can do a simple query to check which items are out of stock:

```
> db.stock.find({"in_stock" : 0})
```

Query Criteria

Queries can go beyond the exact matching described in the previous section; they can match more complex criteria, such as ranges, OR-clauses, and negation.

Query Conditionals

`"$lt"`, `"$lte"`, `"$gt"`, and `"$gte"` are all comparison operators, corresponding to <, <=, >, and >=, respectively. They can be combined to look for a range of values. For example, to look for users who are between the ages of 18 and 30 inclusive, we can do this:

```
> db.users.find({"age" : {"$gte" : 18, "$lte" : 30}})
```

These types of range queries are often useful for dates. For example, to find people who registered before January 1, 2007, we can do this:

```
> start = new Date("01/01/2007")
> db.users.find({"registered" : {"$lt" : start}})
```

An exact match on a date is less useful, because dates are only stored with millisecond precision. Often you want a whole day, week, or month, making a range query necessary.

To query for documents where a key's value is not equal to a certain value, you must use another conditional operator, `"$ne"`, which stands for "not equal." If you want to find all users who do not have the username "joe," you can query for them using this:

```
> db.users.find({"username" : {"$ne" : "joe"}})
```

`"$ne"` can be used with any type.

OR Queries

There are two ways to do an OR query in MongoDB. `"$in"` can be used to query for a variety of values for a single key. `"$or"` is more general; it can be used to query for any of the given values across multiple keys.

If you have more than one possible value to match for a single key, use an array of criteria with `"$in"`. For instance, suppose we were running a raffle and the winning ticket numbers were 725, 542, and 390. To find all three of these documents, we can construct the following query:

```
> db.raffle.find({"ticket_no" : {"$in" : [725, 542, 390]}})
```

`"$in"` is very flexible and allows you to specify criteria of different types as well as values. For example, if we are gradually migrating our schema to use usernames instead of user ID numbers, we can query for either by using this:

```
> db.users.find({"user_id" : {"$in" : [12345, "joe"]})
```

This matches documents with a `"user_id"` equal to 12345, and documents with a `"user_id"` equal to `"joe"`.

If `"$in"` is given an array with a single value, it behaves the same as directly matching the value. For instance, `{ticket_no : {$in : [725]}}` matches the same documents as `{ticket_no : 725}`.

The opposite of `"$in"` is `"$nin"`, which returns documents that don't match any of the criteria in the array. If we want to return all of the people who didn't win anything in the raffle, we can query for them with this:

```
> db.raffle.find({"ticket_no" : {"$nin" : [725, 542, 390]}})
```

This query returns everyone who did not have tickets with those numbers.

`"$in"` gives you an OR query for a single key, but what if we need to find documents where `"ticket_no"` is 725 or `"winner"` is `true`? For this type of query, we'll need to use the `"$or"` conditional. `"$or"` takes an array of possible criteria. In the raffle case, using `"$or"` would look like this:

```
> db.raffle.find({"$or" : [{"ticket_no" : 725}, {"winner" : true}]})
```

`"$or"` can contain other conditionals. If, for example, we want to match any of the three `"ticket_no"` values or the `"winner"` key, we can use this:

```
> db.raffle.find({"$or" : [{"ticket_no" : {"$in" : [725, 542, 390]}},
                           {"winner" : true}]})
```

With a normal AND-type query, you want to narrow your results down as far as possible in as few arguments as possible. OR-type queries are the opposite: they are most efficient if the first arguments match as many documents as possible.

$not

`"$not"` is a metaconditional: it can be applied on top of any other criteria. As an example, let's consider the modulus operator, `"$mod"`. `"$mod"` queries for keys whose values, when divided by the first value given, have a remainder of the second value:

```
> db.users.find({"id_num" : {"$mod" : [5, 1]}})
```

The previous query returns users with `"id_num"`s of 1, 6, 11, 16, and so on. If we want, instead, to return users with `"id_num"`s of 2, 3, 4, 5, 7, 8, 9, 10, 12, and so on, we can use `"$not"`:

```
> db.users.find({"id_num" : {"$not" : {"$mod" : [5, 1]}}})
```

`"$not"` can be particularly useful in conjunction with regular expressions to find all documents that don't match a given pattern (regular expression usage is described in the section "Regular Expressions" on page 50).

Rules for Conditionals

If you look at the update modifiers in the previous chapter and previous query documents, you'll notice that the $-prefixed keys are in different positions. In the query, `"$lt"` is in the inner document; in the update, `"$inc"` is the key for the outer document. This generally holds true: conditionals are an inner document key, and modifiers are always a key in the outer document.

Multiple conditions can be put on a single key. For example, to find all users between the ages of 20 and 30, we can query for both `"$gt"` and `"$lt"` on the `"age"` key:

```
> db.users.find({"age" : {"$lt" : 30, "$gt" : 20}})
```

Any number of conditionals can be used with a single key. Multiple update modifiers *cannot* be used on a single key, however. For example, you cannot have a modifier document such as `{"$inc" : {"age" : 1}, "$set" : {age : 40}}` because it modifies `"age"` twice. With query conditionals, no such rule applies.

Type-Specific Queries

As covered in Chapter 2, MongoDB has a wide variety of types that can be used in a document. Some of these behave specially in queries.

null

`null` behaves a bit strangely. It does match itself, so if we have a collection with the following documents:

```
> db.c.find()
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523621"), "y" : null }
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523622"), "y" : 1 }
{ "_id" : ObjectId("4ba0f148d22aa494fd523623"), "y" : 2 }
```

we can query for documents whose `"y"` key is `null` in the expected way:

```
> db.c.find({"y" : null})
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523621"), "y" : null }
```

However, `null` not only matches itself but also matches "does not exist." Thus, querying for a key with the value `null` will return all documents lacking that key:

```
> db.c.find({"z" : null})
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523621"), "y" : null }
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523622"), "y" : 1 }
{ "_id" : ObjectId("4ba0f148d22aa494fd523623"), "y" : 2 }
```

If we only want to find keys whose value is `null`, we can check that the key is `null` and exists using the `"$exists"` conditional:

```
> db.c.find({"z" : {"$in" : [null], "$exists" : true}})
```

Unfortunately, there is no `"$eq"` operator, which makes this a little awkward, but `"$in"` with one element is equivalent.

Regular Expressions

Regular expressions are useful for flexible string matching. For example, if we want to find all users with the name Joe or joe, we can use a regular expression to do case-insensitive matching:

```
> db.users.find({"name" : /joe/i})
```

Regular expression flags (`i`) are allowed but not required. If we want to match not only various capitalizations of joe, but also joey, we can continue to improve our regular expression:

```
> db.users.find({"name" : /joey?/i})
```

MongoDB uses the Perl Compatible Regular Expression (PCRE) library to match regular expressions; any regular expression syntax allowed by PCRE is allowed in MongoDB. It is a good idea to check your syntax with the JavaScript shell before using it in a query to make sure it matches what you think it matches.

MongoDB can leverage an index for queries on prefix regular expressions (e.g., `/^joey/`), so queries of that kind can be fast.

Regular expressions can also match themselves. Very few people insert regular expressions into the database, but if you insert one, you can match it with itself:

```
> db.foo.insert({"bar" : /baz/})
> db.foo.find({"bar" : /baz/})
```

```
{
    "_id" : ObjectId("4b23c3ca7525f35f94b60a2d"),
    "bar" : /baz/
}
```

Querying Arrays

Querying for elements of an array is simple. An array can mostly be treated as though each element is the value of the overall key. For example, if the array is a list of fruits, like this:

```
> db.food.insert({"fruit" : ["apple", "banana", "peach"]})
```

the following query:

```
> db.food.find({"fruit" : "banana"})
```

will successfully match the document. We can query for it in much the same way as though we had a document that looked like the (illegal) document: `{"fruit" : "apple", "fruit" : "banana", "fruit" : "peach"}`.

$all

If you need to match arrays by more than one element, you can use `"$all"`. This allows you to match a list of elements. For example, suppose we created a collection with three elements:

```
> db.food.insert({"_id" : 1, "fruit" : ["apple", "banana", "peach"]})
> db.food.insert({"_id" : 2, "fruit" : ["apple", "kumquat", "orange"]})
> db.food.insert({"_id" : 3, "fruit" : ["cherry", "banana", "apple"]})
```

Then we can find all documents with both `"apple"` and `"banana"` elements by querying with `"$all"`:

```
> db.food.find({fruit : {$all : ["apple", "banana"]}})
      {"_id" : 1, "fruit" : ["apple", "banana", "peach"]}
      {"_id" : 3, "fruit" : ["cherry", "banana", "apple"]}
```

Order does not matter. Notice `"banana"` comes before `"apple"` in the second result. Using a one-element array with `"$all"` is equivalent to not using `"$all"`. For instance, `{fruit : {$all : ['apple']}` will match the same documents as `{fruit : 'apple'}`.

You can also query by exact match using the entire array. However, exact match will not match a document if any elements are missing or superfluous. For example, this will match the first document shown previously:

```
> db.food.find({"fruit" : ["apple", "banana", "peach"]})
```

But this will not:

```
> db.food.find({"fruit" : ["apple", "banana"]})
```

and neither will this:

```
> db.food.find({"fruit" : ["banana", "apple", "peach"]})
```

If you want to query for a specific element of an array, you can specify an index using the syntax *`key.index`*:

```
> db.food.find({"fruit.2" : "peach"})
```

Arrays are always 0-indexed, so this would match the third array element against the string `"peach"`.

$size

A useful conditional for querying arrays is `"$size"`, which allows you to query for arrays of a given size. Here's an example:

```
> db.food.find({"fruit" : {"$size" : 3}})
```

One common query is to get a range of sizes. `"$size"` cannot be combined with another $ conditional (in this example, `"$gt"`), but this query can be accomplished by adding a `"size"` key to the document. Then, every time you add an element to the array, increment the value of `"size"`. If the original update looked like this:

```
> db.food.update({"$push" : {"fruit" : "strawberry"}})
```

it can simply be changed to this:

```
> db.food.update({"$push" : {"fruit" : "strawberry"}, "$inc" : {"size" : 1}})
```

Incrementing is extremely fast, so any performance penalty is negligible. Storing documents like this allows you to do queries such as this:

```
> db.food.find({"size" : {"$gt" : 3}})
```

Unfortunately, this technique doesn't work as well with the `"$addToSet"` operator.

The $slice operator

As mentioned earlier in this chapter, the optional second argument to `find` specifies the keys to be returned. The special `"$slice"` operator can be used to return a subset of elements for an array key.

For example, suppose we had a blog post document and we wanted to return the first 10 comments:

```
> db.blog.posts.findOne(criteria, {"comments" : {"$slice" : 10}})
```

Alternatively, if we wanted the last 10 comments, we could use -10:

```
> db.blog.posts.findOne(criteria, {"comments" : {"$slice" : -10}})
```

`"$slice"` can also return pages in the middle of the results by taking an offset and the number of elements to return:

```
> db.blog.posts.findOne(criteria, {"comments" : {"$slice" : [23, 10]}})
```

This would skip the first 23 elements and return the 24th through 34th. If there are fewer than 34 elements in the array, it will return as many as possible.

Unless otherwise specified, all keys in a document are returned when `"$slice"` is used. This is unlike the other key specifiers, which suppress unmentioned keys from being returned. For instance, if we had a blog post document that looked like this:

```
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "title" : "A blog post",
    "content" : "...",
    "comments" : [
        {
            "name" : "joe",
            "email" : "joe@example.com",
            "content" : "nice post."
        },
        {
            "name" : "bob",
            "email" : "bob@example.com",
            "content" : "good post."
        }
    ]
}
```

and we did a `"$slice"` to get the last comment, we'd get this:

```
> db.blog.posts.findOne(criteria, {"comments" : {"$slice" : -1}})
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "title" : "A blog post",
    "content" : "...",
    "comments" : [
        {
            "name" : "bob",
            "email" : "bob@example.com",
            "content" : "good post."
        }
    ]
}
```

Both `"title"` and `"content"` are still returned, even though they weren't explicitly included in the key specifier.

Querying on Embedded Documents

There are two ways of querying for an embedded document: querying for the whole document or querying for its individual key/value pairs.

Querying for an entire embedded document works identically to a normal query. For example, if we have a document that looks like this:

```
{
    "name" : {
        "first" : "Joe",
        "last" : "Schmoe"
    },
        "age" : 45
}
```

we can query for someone named Joe Schmoe with the following:

```
> db.people.find({"name" : {"first" : "Joe", "last" : "Schmoe"}})
```

However, if Joe decides to add a middle name key, suddenly this query won't work anymore; it doesn't match the entire embedded document! This type of query is also order-sensitive; `{"last" : "Schmoe", "first" : "Joe"}` would not be a match.

If possible, it's usually a good idea to query for just a specific key or keys of an embedded document. Then, if your schema changes, all of your queries won't suddenly break because they're no longer exact matches. You can query for embedded keys using dot-notation:

```
> db.people.find({"name.first" : "Joe", "name.last" : "Schmoe"})
```

Now, if Joe adds more keys, this query will still match his first and last names.

This dot-notation is the main difference between query documents and other document types. Query documents can contain dots, which mean "reach into an embedded document." Dot-notation is also the reason that documents to be inserted cannot contain the . character. Oftentimes people run into this limitation when trying to save URLs as keys. One way to get around it is to always perform a global replace before inserting or after retrieving, substituting a character that isn't legal in URLs for the dot (.) character.

Embedded document matches can get a little tricky as the document structure gets more complicated. For example, suppose we are storing blog posts and we want to find comments by Joe that were scored at least a five. We could model the post as follows:

```
> db.blog.find()
{
    "content" : "...",
    "comments" : [
        {
            "author" : "joe",
            "score" : 3,
            "comment" : "nice post"
        },
        {
            "author" : "mary",
            "score" : 6,
            "comment" : "terrible post"
        }
    ]
}
```

Now, we can't query using `db.blog.find({"comments" : {"author" : "joe", "score" : {"$gte" : 5}}})`. Embedded document matches have to match the whole document, and this doesn't match the `"comment"` key. It also wouldn't work to do `db.blog.find({"comments.author" : "joe", "comments.score" : {"$gte" : 5}})`, because the author criteria could match a different comment than the score criteria. That is, it would return the document shown earlier; it would match `"author" : "joe"` in the first comment and `"score" : 6` in the second comment.

To correctly group criteria without needing to specify every key, use `"$elemMatch"`. This vaguely named conditional allows you to partially specify criteria to match a single embedded document in an array. The correct query looks like this:

```
> db.blog.find({"comments" : {"$elemMatch" : {"author" : "joe",
                                              "score" : {"$gte" : 5}}}})
```

`"$elemMatch"` allows us to "group" our criteria. As such, it's only needed when you have more than one key you want to match on in an embedded document.

$where Queries

Key/value pairs are a fairly expressive way to query, but there are some queries that they cannot represent. For queries that cannot be done any other way, there are `"$where"` clauses, which allow you to execute arbitrary JavaScript as part of your query. This allows you to do (almost) anything within a query.

The most common case for this is wanting to compare the values for two keys in a document, for instance, if we had a list of items and wanted to return documents where any two of the values are equal. Here's an example:

```
> db.foo.insert({"apple" : 1, "banana" : 6, "peach" : 3})
> db.foo.insert({"apple" : 8, "spinach" : 4, "watermelon" : 4})
```

In the second document, `"spinach"` and `"watermelon"` have the same value, so we'd like that document returned. It's unlikely MongoDB will ever have a $ conditional for this, so we can use a `"$where"` clause to do it with JavaScript:

```
> db.foo.find({"$where" : function () {
... for (var current in this) {
...     for (var other in this) {
...         if (current != other && this[current] == this[other]) {
...             return true;
...         }
...     }
... }
... return false;
... }});
```

If the function returns `true`, the document will be part of the result set; if it returns `false`, it won't be.

We used a function earlier, but you can also use strings to specify a `"$where"` query; the following two `"$where"` queries are equivalent:

```
> db.foo.find({"$where" : "this.x + this.y == 10"})
> db.foo.find({"$where" : "function() { return this.x + this.y == 10; }"})
```

`"$where"` queries should not be used unless strictly necessary: they are much slower than regular queries. Each document has to be converted from BSON to a JavaScript object and then run through the `"$where"` expression. Indexes cannot be used to satisfy a `"$where"`, either. Hence, you should use `"$where"` only when there is no other way of doing the query. You can cut down on the penalty by using other query filters in combination with `"$where"`. If possible, an index will be used to filter based on the non-`$where` clauses; the `"$where"` expression will be used only to fine-tune the results.

Another way of doing complex queries is to use MapReduce, which is covered in the next chapter.

Cursors

The database returns results from `find` using a *cursor*. The client-side implementations of cursors generally allow you to control a great deal about the eventual output of a query. You can limit the number of results, skip over some number of results, sort results by any combination of keys in any direction, and perform a number of other powerful operations.

To create a cursor with the shell, put some documents into a collection, do a query on them, and assign the results to a local variable (variables defined with `"var"` are local). Here, we create a very simple collection and query it, storing the results in the `cursor` variable:

```
> for(i=0; i<100; i++) {
... db.c.insert({x : i});
... }
> var cursor = db.collection.find();
```

The advantage of doing this is that you can look at one result at a time. If you store the results in a global variable or no variable at all, the MongoDB shell will automatically iterate through and display the first couple of documents. This is what we've been seeing up until this point, and it is often the behavior you want for seeing what's in a collection but not for doing actual programming with the shell.

To iterate through the results, you can use the `next` method on the cursor. You can use `hasNext` to check whether there is another result. A typical loop through results looks like the following:

```
> while (cursor.hasNext()) {
... obj = cursor.next();
... // do stuff
... }
```

`cursor.hasNext()` checks that the next result exists, and `cursor.next()` fetches it.

The `cursor` class also implements the iterator interface, so you can use it in a `forEach` loop:

```
> var cursor = db.people.find();
> cursor.forEach(function(x) {
... print(x.name);
... });
adam
matt
zak
```

When you call `find`, the shell does not query the database immediately. It waits until you actually start requesting results to send the query, which allows you to chain additional options onto a query before it is performed. Almost every method on a cursor object returns the cursor itself so that you can chain them in any order. For instance, all of the following are equivalent:

```
> var cursor = db.foo.find().sort({"x" : 1}).limit(1).skip(10);
> var cursor = db.foo.find().limit(1).sort({"x" : 1}).skip(10);
> var cursor = db.foo.find().skip(10).limit(1).sort({"x" : 1});
```

At this point, the query has not been executed yet. All of these functions merely build the query. Now, suppose we call the following:

```
> cursor.hasNext()
```

At this point, the query will be sent to the server. The shell fetches the first 100 results or first 4MB of results (whichever is smaller) at once so that the next calls to `next` or `hasNext` will not have to make trips to the server. After the client has run through the first set of results, the shell will again contact the database and ask for more results. This process continues until the cursor is exhausted and all results have been returned.

Limits, Skips, and Sorts

The most common query options are limiting the number of results returned, skipping a number of results, and sorting. All of these options must be added before a query is sent to the database.

To set a limit, chain the `limit` function onto your call to `find`. For example, to only return three results, use this:

```
> db.c.find().limit(3)
```

If there are fewer than three documents matching your query in the collection, only the number of matching documents will be returned; `limit` sets an upper limit, not a lower limit.

`skip` works similarly to `limit`:

```
> db.c.find().skip(3)
```

This will skip the first three matching documents and return the rest of the matches. If there are less than three documents in your collection, it will not return any documents.

`sort` takes an object: a set of key/value pairs where the keys are key names and the values are the sort directions. Sort direction can be 1 (ascending) or -1 (descending). If multiple keys are given, the results will be sorted in that order. For instance, to sort the results by `"username"` ascending and `"age"` descending, we do the following:

```
> db.c.find().sort({username : 1, age : -1})
```

These three methods can be combined. This is often handy for pagination. For example, suppose that you are running an online store and someone searches for *mp3*. If you want 50 results per page sorted by price from high to low, you can do the following:

```
> db.stock.find({"desc" : "mp3"}).limit(50).sort({"price" : -1})
```

If they click Next Page to see more results, you can simply add a skip to the query, which will skip over the first 50 matches (which the user already saw on page 1):

```
> db.stock.find({"desc" : "mp3"}).limit(50).skip(50).sort({"price" : -1})
```

However, large skips are not very performant, so there are suggestions on avoiding them in a moment.

Comparison order

MongoDB has a hierarchy as to how types compare. Sometimes you will have a single key with multiple types, for instance, integers and booleans, or strings and nulls. If you do a sort on a key with a mix of types, there is a predefined order that they will be sorted in. From least to greatest value, this ordering is as follows:

1. Minimum value
2. `null`
3. Numbers (integers, longs, doubles)
4. Strings
5. Object/document
6. Array
7. Binary data
8. Object ID
9. Boolean
10. Date
11. Timestamp
12. Regular expression
13. Maximum value

Avoiding Large Skips

Using `skip` for a small number of documents is fine. For a large number of results, `skip` can be slow (this is true in nearly every database, not just MongoDB) and should be avoided. Usually you can build criteria into the documents themselves to avoid having to do large skips, or you can calculate the next query based on the result from the previous one.

Paginating results without skip

The easiest way to do pagination is to return the first page of results using limit and then return each subsequent page as an offset from the beginning.

```
> // do not use: slow for large skips
> var page1 = db.foo.find(criteria).limit(100)
> var page2 = db.foo.find(criteria).skip(100).limit(100)
> var page3 = db.foo.find(criteria).skip(200).limit(100)
...
```

However, depending on your query, you can usually find a way to paginate without `skip`s. For example, suppose we want to display documents in descending order based on `"date"`. We can get the first page of results with the following:

```
> var page1 = db.foo.find().sort({"date" : -1}).limit(100)
```

Then, we can use the `"date"` value of the last document as the criteria for fetching the next page:

```
var latest = null;

// display first page
while (page1.hasNext()) {
   latest = page1.next();
   display(latest);
}

// get next page
var page2 = db.foo.find({"date" : {"$gt" : latest.date}});
page2.sort({"date" : -1}).limit(100);
```

Now the query does not need to include a `skip`.

Finding a random document

One fairly common problem is how to get a random document from a collection. The naive (and slow) solution is to count the number of documents and then do a `find`, skipping a random number of documents between 0 and the size of the collection.

```
> // do not use
> var total = db.foo.count()
> var random = Math.floor(Math.random()*total)
> db.foo.find().skip(random).limit(1)
```

It is actually highly inefficient to get a random element this way: you have to do a count (which can be expensive if you are using criteria), and skipping large numbers of elements can be time-consuming.

It takes a little forethought, but if you know you'll be looking up a random element on a collection, there's a much more efficient way to do so. The trick is to add an extra random key to each document when it is inserted. For instance, if we're using the shell, we could use the `Math.random()` function (which creates a random number between 0 and 1):

```
> db.people.insert({"name" : "joe", "random" : Math.random()})
> db.people.insert({"name" : "john", "random" : Math.random()})
> db.people.insert({"name" : "jim", "random" : Math.random()})
```

Now, when we want to find a random document from the collection, we can calculate a random number and use that as query criteria, instead of doing a `skip`:

```
> var random = Math.random()
> result = db.foo.findOne({"random" : {"$gt" : random}})
```

There is a slight chance that `random` will be greater than any of the `"random"` values in the collection, and no results will be returned. We can guard against this by simply returning a document in the other direction:

```
> if (result == null) {
... result = db.foo.findOne({"random" : {"$lt" : random}})
... }
```

If there aren't any documents in the collection, this technique will end up returning `null`, which makes sense.

This technique can be used with arbitrarily complex queries; just make sure to have an index that includes the random key. For example, if we want to find a random plumber in California, we can create an index on `"profession"`, `"state"`, and `"random"`:

```
> db.people.ensureIndex({"profession" : 1, "state" : 1, "random" : 1})
```

This allows us to quickly find a random result (see Chapter 5 for more information on indexing).

Advanced Query Options

There are two types of queries: *wrapped* and *plain*. A plain query is something like this:

```
> var cursor = db.foo.find({"foo" : "bar"})
```

There are a couple options that "wrap" the query. For example, suppose we perform a sort:

```
> var cursor = db.foo.find({"foo" : "bar"}).sort({"x" : 1})
```

Instead of sending `{"foo" : "bar"}` to the database as the query, the query gets wrapped in a larger document. The shell converts the query from `{"foo" : "bar"}` to `{"$query" : {"foo" : "bar"}, "$orderby" : {"x" : 1}}`.

Most drivers provide helpers for adding arbitrary options to queries. Other helpful options include the following:

`$maxscan :` *`integer`*
: Specify the maximum number of documents that should be scanned for the query.

`$min` : *`document`*
: Start criteria for querying.

`$max` : *`document`*
: End criteria for querying.

`$hint` : *`document`*
: Tell the server which index to use for the query.

`$explain` : *`boolean`*
: Get an explanation of how the query will be executed (indexes used, number of results, how long it takes, etc.), instead of actually doing the query.

`$snapshot` : *`boolean`*
: Ensure that the query's results will be a consistent snapshot from the point in time when the query was executed. See the next section for details.

Getting Consistent Results

A fairly common way of processing data is to pull it out of MongoDB, change it in some way, and then save it again:

```
cursor = db.foo.find();

while (cursor.hasNext()) {
    var doc = cursor.next();
    doc = process(doc);
    db.foo.save(doc);
}
```

This is fine for a small number of results, but it breaks down for large numbers of documents. To see why, imagine how the documents are actually being stored. You can picture a collection as a list of documents that looks something like Figure 4-1. Snowflakes represent documents, because every document is beautiful and unique.

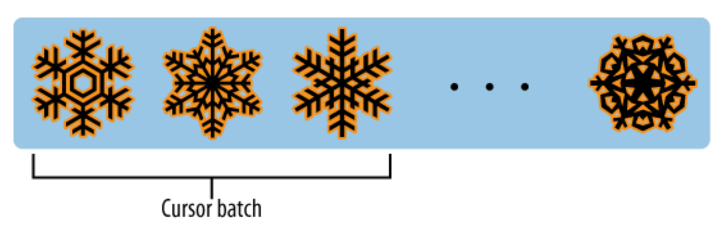


Figure 4-1. A collection being queried

Now, when we do a `find`, it starts returning results from the beginning of the collection and moves right. Your program grabs the first 100 documents and processes them. When you save them back to the database, if a document does not have the padding available to grow to its new size, like in Figure 4-2, it needs to be relocated. Usually, a document will be relocated to the end of a collection (Figure 4-3).

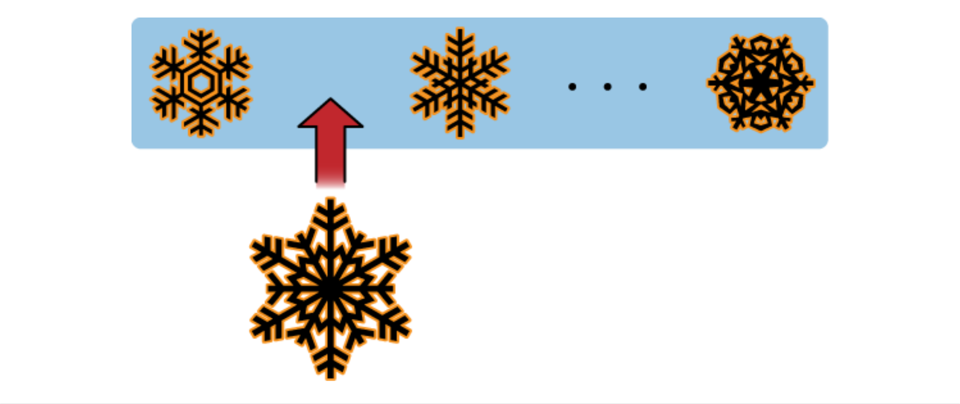


Figure 4-2. An enlarged document may not fit where it did before

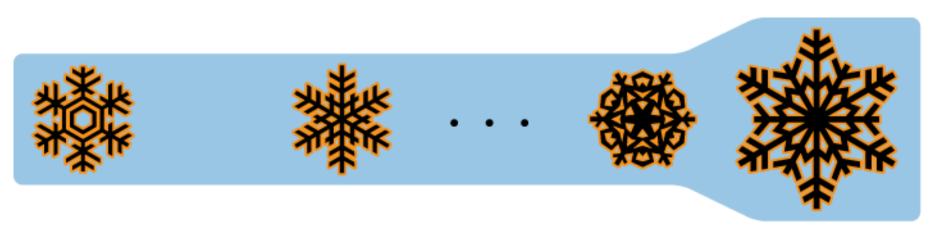


Figure 4-3. MongoDB relocates updated documents that don't fit in their original position

Now our program continues to fetch batches of documents. When it gets toward the end, it will return the relocated documents again (Figure 4-4)!

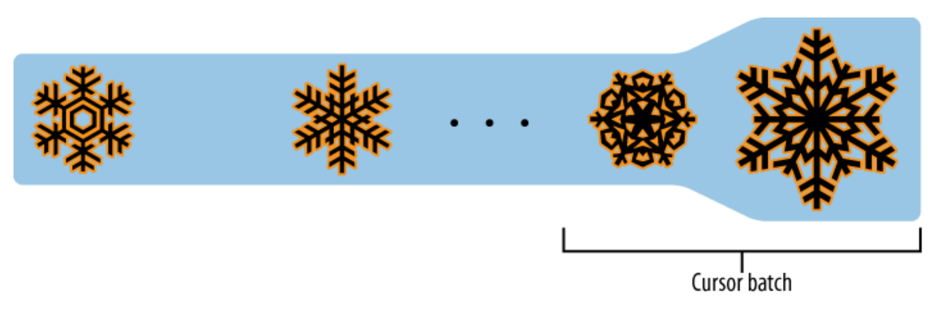


Figure 4-4. A cursor may return these relocated documents again in a later batch

The solution to this problem is to *snapshot* your query. If you add the `"$snapshot"` option, the query will be run against an unchanging view of the collection. All queries that return a single batch of results are effectively snapshotted. Inconsistencies arise only when the collection changes under a cursor while it is waiting to get another batch of results.

Cursor Internals

There are two sides to a cursor: the client-facing cursor and the database cursor that the client-side one represents. We have been talking about the client-side one up until now, but we are going to take a brief look at what's happening on the server side.

On the server side, a cursor takes up memory and resources. Once a cursor runs out of results or the client sends a message telling it to die, the database can free the resources it was using. Freeing these resources lets the database use them for other things, which is good, so we want to make sure that cursors can be freed quickly (within reason).

There are a couple of conditions that can cause the death (and subsequent cleanup) of a cursor. First, when a cursor finishes iterating through the matching results, it will clean itself up. Another way is that, when a cursor goes out of scope on the client side, the drivers send the database a special message to let it know that it can kill that cursor. Finally, even if the user hasn't iterated through all the results and the cursor is still in scope, after 10 minutes of inactivity, a database cursor will automatically "die."

This "death by timeout" is usually the desired behavior: very few applications expect their users to sit around for minutes at a time waiting for results. However, sometimes you might know that you need a cursor to last for a long time. In that case, many drivers have implemented a function called `immortal`, or a similar mechanism, which tells the database not to time out the cursor. If you turn off a cursor's timeout, you must iterate through all of its results or make sure it gets closed. Otherwise, it will sit around in the database hogging resources forever.

CHAPTER 5

Indexing

Indexes are the way to make queries go *vroom*. Database indexes are similar to a book's index: instead of looking through the whole book, the database takes a shortcut and just looks in the index, allowing it to do queries orders of magnitude faster. Once it finds the entry in the index, it can jump right to the location of the desired document.

Extending this metaphor to the breaking point, creating database indexes is like deciding how a book's index will be organized. You have the advantage of knowing what kinds of queries you'll be doing and thus what types of information the database will need to find quickly. If all of your queries involve the `"date"` key, for example, you probably need an index on `"date"` (at least). If you will be querying for usernames, you don't need to index the `"user_num"` key, because you aren't querying on it.

Introduction to Indexing

It can be tricky to figure out what the optimal index for your queries is, but it is well worth the effort. Queries that otherwise take minutes can be instantaneous with the proper indexes.

> MongoDB's indexes work almost identically to typical relational database indexes, so if you are familiar with those, you can skim this section for syntax specifics. We'll go over some indexing basics, but keep in mind that it's an extensive topic and most of the material out there on index optimization for MySQL/Oracle/SQLite will apply equally well to MongoDB.

Suppose that you are querying for a single key, such as the following:

```
> db.people.find({"username" : "mark"})
```

When only a single key is used in the query, that key can be indexed to improve the query's speed. In this case, you would create an index on `"username"`. To create the index, use the `ensureIndex` method:

```
> db.people.ensureIndex({"username" : 1})
```

An index needs to be created only once for a collection. If you try to create the same index again, nothing will happen.

An index on a key will make queries on that key fast. However, it may not make other queries fast, even if they contain the indexed key. For example, this wouldn't be very performant with the previous index:

```
> db.people.find({"date" : date1}).sort({"date" : 1, "username" : 1})
```

The server has to "look through the whole book" to find the desired dates. This process is called a *table scan*, which is basically what you'd do if you were looking for information in a book without an index: you start at page 1 and read through the whole thing. In general, you want to avoid making the server do table scans, because it is very slow for large collections.

As a rule of thumb, you should create an index that contains all of the keys in your query. For instance, to optimize the previous query, you should have an index on `date` and `username`:

```
> db.ensureIndex({"date" : 1, "username" : 1})
```

The document you pass to `ensureIndex` is of the same form as the document passed to the `sort` function: a set of keys with value 1 or -1, depending on the direction you want the index to go. If you have only a single key in the index, direction is irrelevant. A single key index is analogous to a book's index that is organized in alphabetical order: whether it goes from A–Z or Z–A, it's going to be fairly obvious where to find entries starting with *M*.

If you have more than one key, you need to start thinking about index direction. For example, suppose we have a collection of users:

```
{ "_id" : ..., "username" : "smith", "age" : 48, "user_id" : 0 }
{ "_id" : ..., "username" : "smith", "age" : 30, "user_id" : 1 }
{ "_id" : ..., "username" : "john", "age" : 36, "user_id" : 2 }
{ "_id" : ..., "username" : "john", "age" : 18, "user_id" : 3 }
{ "_id" : ..., "username" : "joe", "age" : 36, "user_id" : 4 }
{ "_id" : ..., "username" : "john", "age" : 7, "user_id" : 5 }
{ "_id" : ..., "username" : "simon", "age" : 3, "user_id" : 6 }
{ "_id" : ..., "username" : "joe", "age" : 27, "user_id" : 7 }
{ "_id" : ..., "username" : "jacob", "age" : 17, "user_id" : 8 }
{ "_id" : ..., "username" : "sally", "age" : 52, "user_id" : 9 }
{ "_id" : ..., "username" : "simon", "age" : 59, "user_id" : 10 }
```

Let's say we index them with `{"username" : 1, "age" : -1}`. MongoDB will organize the users as follows:

```
{ "_id" : ..., "username" : "jacob", "age" : 17, "user_id" : 8 }
{ "_id" : ..., "username" : "joe", "age" : 36, "user_id" : 4 }
{ "_id" : ..., "username" : "joe", "age" : 27, "user_id" : 7 }
{ "_id" : ..., "username" : "john", "age" : 36, "user_id" : 2 }
{ "_id" : ..., "username" : "john", "age" : 18, "user_id" : 3 }
{ "_id" : ..., "username" : "john", "age" : 7, "user_id" : 5 }
{ "_id" : ..., "username" : "sally", "age" : 52, "user_id" : 9 }
{ "_id" : ..., "username" : "simon", "age" : 59, "user_id" : 10 }
{ "_id" : ..., "username" : "simon", "age" : 3, "user_id" : 6 }
{ "_id" : ..., "username" : "smith", "age" : 48, "user_id" : 0 }
{ "_id" : ..., "username" : "smith", "age" : 30, "user_id" : 1 }
```

The usernames are in strictly increasing alphabetical order, and within each name group the ages are in decreasing order. This optimizes sorting by `{"username" : 1, "age" : -1}` but is less efficient at sorting by `{"username" : 1, "age" : 1}`. If we wanted to optimize `{"username" : 1, "age" : 1}`, we would create an index on `{"username" : 1, "age" : 1}` to organize ages in ascending order.

The index on `username` and `age` also makes queries on `username` fast. In general, if an index has N keys, it will make queries on any prefix of those keys fast. For instance, if we have an index that looks like `{"a" : 1, "b" : 1, "c" : 1, ..., "z" : 1}`, we effectively have an index on `{"a" : 1}`, `{"a" : 1, "b" : 1}`, `{"a" : 1, "b" : 1, "c" : 1}`, and so on. Queries that would use the index `{"b" : 1}`, `{"a" : 1, "c" :1}`, and so on will not be optimized: only queries that can use a prefix of the index can take advantage of it.

The MongoDB query optimizer will reorder query terms to take advantage of indexes: if you query for `{"x" : "foo", "y" : "bar"}` and you have an index on `{"y" : 1, "x" : 1}`, MongoDB will figure it out.

The disadvantage to creating an index is that it puts a little bit of overhead on every insert, update, and remove. This is because the database not only needs to do the operation but also needs to make a note of it in any indexes on the collection. Thus, the absolute minimum number of indexes should be created. There is a built-in maximum of 64 indexes per collection, which is more than almost any application should need.

Do not index every key. This will make inserts slow, take up lots of space, and probably not speed up your queries very much. Figure out what queries you are running, what the best indexes are for these queries, and make sure that the server is using the indexes you've created using the `explain` and `hint` tools described in the next section.

Sometimes the most efficient solution is actually not to use an index. In general, if a query is returning a half or more of the collection, it will be more efficient for the database to just do a table scan instead of having to look up the index and then the value for almost every single document. Thus, for queries such as checking whether a key exists or determining whether a boolean value is true or false, it may actually be better to not use an index at all.

Scaling Indexes

Suppose we have a collection of status messages from users. We want to query by user and date to pull up all of a user's recent statuses. Using what we've learned so far, we might create an index that looks like the following:

```
> db.status.ensureIndex({user : 1, date : -1})
```

This will make the query for user and date efficient, but it is not actually the best index choice.

Imagine this as a book index again. We would have a list of documents sorted by user and then subsorted by date, so it would look something like the following:

```
User 123 on March 13, 2010
User 123 on March 12, 2010
User 123 on March 11, 2010
User 123 on March 5, 2010
User 123 on March 4, 2010
User 124 on March 12, 2010
User 124 on March 11, 2010
...
```

This looks fine at this scale, but imagine if the application has millions of users who have dozens of status updates per day. If the index entries for each user's status messages take up a page's worth of space on disk, then for every "latest statuses" query, the database will have to load a different page into memory. This will be very slow if the site becomes popular enough that not all of the index fits into memory.

If we flip the index order to `{date : -1, user : 1}`, the database can keep the last couple days of the index in memory, swap less, and thus query for the latest statuses for any user much more quickly.

Thus, there are several questions to keep in mind when deciding what indexes to create:

1. What are the queries you are doing? Some of these keys will need to be indexed.
2. What is the correct direction for each key?
3. How is this going to scale? Is there a different ordering of keys that would keep more of the frequently used portions of the index in memory?

If you can answer these questions, you are ready to index your data.

Indexing Keys in Embedded Documents

Indexes can be created on keys in embedded documents in the same way that they are created on normal keys. For example, if we want to be able to search blog post comments by date, we can create an index on the `"date"` key in the array of embedded `"comments"` documents:

```
> db.blog.ensureIndex({"comments.date" : 1})
```

Indexes on keys in embedded documents are identical to those on top-level keys, and the two can be combined in compound indexes.

Indexing for Sorts

As your collection grows, you'll need to create indexes for any large sorts your queries are doing. If you call `sort` on a key that is not indexed, MongoDB needs to pull all of that data into memory to sort it. Thus, there's a limit on the amount you can sort without an index: you can't sort a terabyte of data in memory. Once your collection is too big to sort in memory, MongoDB will just return an error for queries that try.

Indexing the sort allows MongoDB to pull the sorted data in order, allowing you to sort any amount of data without running out of memory.

Uniquely Identifying Indexes

Each index in a collection has a string name that uniquely identifies the index and is used by the server to delete or manipulate it. Index names are, by default, ***`keyname1_dir1_keyname2_dir2_..._keynameN_dirN`***, where `keynameX` is the index's key and `dirX` is the index's direction (1 or -1). This can get unwieldy if indexes contain more than a couple keys, so you can specify your own name as one of the options to `ensureIndex`:

```
> db.foo.ensureIndex({"a" : 1, "b" : 1, "c" : 1, ..., "z" : 1}, {"name" : "alphabet"})
```

There is a limit to the number of characters in an index name, so complex indexes may need custom names to be created. A call to `getLastError` will show whether the index creation succeeded or why it didn't.

Unique Indexes

Unique indexes guarantee that, for a given key, every document in the collection will have a unique value. For example, if you want to make sure no two documents can have the same value in the `"username"` key, you can create a unique index:

```
> db.people.ensureIndex({"username" : 1}, {"unique" : true})
```

Keep in mind that `insert`, by default, does not check whether the document was actually inserted. Therefore, you may want to use safe inserts if you are inserting documents that might have a duplicate value for a unique key. This way, you will get a duplicate key error when you attempt to insert such a document.

A unique index that you are probably already familiar with is the index on `"_id"`, which is created whenever you create a normal collection. This is a normal unique index, aside from the fact that it cannot be deleted.

If a key does not exist, the index stores its value as `null`. Therefore, if you create a unique index on a key and try to insert more than one document that is missing the indexed key, the inserts will fail because you already have a document with a value of `null`.

Dropping Duplicates

When building unique indexes for an existing collection, some values may be duplicates. If there are any duplicates, this will cause the index building to fail. In some cases, you may just want to drop all of the documents with duplicate values. The **`dropDups`** option will save the first document found and remove any subsequent documents with duplicate values:

```
> db.people.ensureIndex({"username" : 1}, {"unique" : true, "dropDups" : true})
```

This is a bit of a drastic option; it might be better to create a script to preprocess the data if it is important.

Compound Unique Indexes

You can also create a compound unique index. If you do this, individual keys can have the same values, but the combination of values for all keys must be unique.

GridFS, the standard method for storing large files in MongoDB (see Chapter 7), uses a compound unique index. The collection that holds the file content has a unique index on `{files_id : 1, n : 1}`, which allows documents that look like (in part) the following:

```
{files_id : ObjectId("4b23c3ca7525f35f94b60a2d"), n : 1}
{files_id : ObjectId("4b23c3ca7525f35f94b60a2d"), n : 2}
{files_id : ObjectId("4b23c3ca7525f35f94b60a2d"), n : 3}
{files_id : ObjectId("4b23c3ca7525f35f94b60a2d"), n : 4}
```

Note that all of the values for `"files_id"` are the same, but `"n"` is different. If you attempted to insert `{files_id : ObjectId("4b23c3ca7525f35f94b60a2d"), n : 1}` again, the database would complain about the duplicate key.

Using explain and hint

`explain` is an incredibly handy tool that will give you lots of information about your queries. You can run it on any query by tacking it on to a cursor. `explain` returns a document, not the cursor itself, unlike most cursor methods:

```
> db.foo.find().explain()
```

`explain` will return information about the indexes used for the query (if any) and stats about timing and the number of documents scanned.

Let's take an example. The index `{"username" : 1}` works well to speed up a simple key/value pair lookup, but many queries are more complicated. For example, suppose you are querying and sorting the following:

```
> db.people.find({"age" : 18}).sort({"username" : 1})
```

You may be unsure if the database is using the index you created or how effective it is. If you run `explain` on the query, it will return the index currently being used for the query, how long it took, and how many documents the database needed to scan to find the results.

For a very simple query (`{}`) on a database with no indexes (other than the index on `"_id"`) and 64 documents, the output for `explain` looks like this:

```
> db.people.find().explain()
{
    "cursor" : "BasicCursor",
    "indexBounds" : [ ],
    "nscanned" : 64,
    "nscannedObjects" : 64,
    "n" : 64,
    "millis" : 0,
    "allPlans" : [
        {
            "cursor" : "BasicCursor",
            "indexBounds" : [ ]
        }
    ]
}
```

The important parts of this result are as follows:

`"cursor" : "BasicCursor"`
: This means that the query did not use an index (unsurprisingly, because there was no query criteria). We'll see what this value looks like for an indexed query in a moment.

`"nscanned" : 64`
: This is the number of documents that the database looked through. You want to make sure this is as close to the number returned as possible.

`"n" : 64`
: This is the number of documents returned. We're doing pretty well here, because the number of documents scanned exactly matches the number returned. Of course, given that we're returning the entire collection, it would be difficult to do otherwise.

`"millis" : 0`
: The number of milliseconds it took the database to execute the query. 0 is a good time to shoot for.

Now, let's suppose we have an index on the `"age"` key and we're querying for users in their 20s. If we run `explain` on this query, we'll see the following:

```
> db.c.find({age : {$gt : 20, $lt : 30}}).explain()
{
    "cursor" : "BtreeCursor age_1",
    "indexBounds" : [
        [
            {
                "age" : 20
            },
            {
                "age" : 30
            }
        ]
    ],
    "nscanned" : 14,
    "nscannedObjects" : 12,
    "n" : 12,
    "millis" : 1,
    "allPlans" : [
        {
            "cursor" : "BtreeCursor age_1",
            "indexBounds" : [
                [
                    {
                        "age" : 20
                    },
                    {
                        "age" : 30
                    }
                ]
            ]
        }
    ]
}
```

There are some differences here, because this query uses an index. Thus, some of `explain`'s outputted keys have changed:

`"cursor" : "BtreeCursor age_1"`

This query is not using a `BasicCursor`, like the first query was. Indexes are stored in data structures called B-trees, so, when a query uses an index, it uses a special type of cursor called a BtreeCursor.

This value also identifies the name of the index being used: `age_1`. Using this name, we could query the *system.indexes* collection to find out more about this index (e.g., whether it is unique or what keys it includes):

```
> db.system.indexes.find({"ns" : "test.c", "name" : "age_1"})
{
    "_id" : ObjectId("4c0d211478b4eaaf7fb28565"),
    "ns" : "test.c",
    "key" : {
        "age" : 1
        },
        "name" : "age_1"
    }
```

`"allPlans" : [ ... ]`
: This key lists all of the plans MongoDB considered for the query. The choice here was fairly obvious, because we had an index on `"age"` and we were querying for `"age"`. If we had multiple overlapping indexes and a more complex query, `"allPlans"` would contain information about all of the possible plans that could have been used.

Let's take a slightly more complicated example: suppose you have an index on `{"username" : 1, "age" : 1}` and an index on `{"age" : 1, "username" : 1}`. What happens if you query for `username` and `age`? Well, it depends on the query:

```
> db.c.find({age : {$gt : 10}, username : "sally"}).explain()
{
        "cursor" : "BtreeCursor username_1_age_1",
        "indexBounds" : [
                [
                        {
                                "username" : "sally",
                                "age" : 10
                        },
                        {
                                "username" : "sally",
                                "age" : 1.7976931348623157e+308
                        }
                ]
        ],
        "nscanned" : 13,
        "nscannedObjects" : 13,
        "n" : 13,
        "millis" : 5,
        "allPlans" : [
                {
                        "cursor" : "BtreeCursor username_1_age_1",
                        "indexBounds" : [
                                [
                                        {
                                                "username" : "sally",
                                                "age" : 10
                                        },
                                        {
                                                "username" : "sally",
                                                "age" : 1.7976931348623157e+308
                                        }
                                ]
                        ]
                }
        ],
        "oldPlan" : {
                "cursor" : "BtreeCursor username_1_age_1",
                "indexBounds" : [
                        [
                                {
                                        "username" : "sally",
                                        "age" : 10
                                },
                                {
                                        "username" : "sally",
                                        "age" : 1.7976931348623157e+308
                                }
                        ]
                ]
        }
}
```

We are querying for an exact match on `"username"` and a range of values for `"age"`, so the database chooses to use the `{"username" : 1, "age" : 1}` index, reversing the terms of the query. If, on the other hand, we query for an exact age and a range of names, MongoDB will use the other index:

```
> db.c.find({"age" : 14, "username" : /.*/}).explain()
{
        "cursor" : "BtreeCursor age_1_username_1 multi",
        "indexBounds" : [
                [
                        {
                                "age" : 14,
                                "username" : ""
                        },
                        {
                                "age" : 14,
                                "username" : {

                                }
                        }
                ],
                [
                        {
                                "age" : 14,
                                "username" : /.*/
                        },
                        {
                                "age" : 14,
                                "username" : /.*/
                        }
                ]
        ],
        "nscanned" : 2,
        "nscannedObjects" : 2,
        "n" : 2,
        "millis" : 2,
        "allPlans" : [
                {
                        "cursor" : "BtreeCursor age_1_username_1 multi",
                        "indexBounds" : [
                                [
                                        {
                                                "age" : 14,
                                                "username" : ""
                                        },
                                        {
                                                "age" : 14,
                                                "username" : {

                                                }
                                        }
                                ],
                                [
                                        {
                                                "age" : 14,
                                                "username" : /.*/
                                        },
                                        {
                                                "age" : 14,
                                                "username" : /.*/
                                        }
                                ]
                        ]
                }
        ]
}
```

If you find that Mongo is using different indexes than you want it to for a query, you can force it to use a certain index by using `hint`. For instance, if you want to make sure MongoDB uses the `{"username" : 1, "age" : 1}` index on the previous query, you could say the following:

```
> db.c.find({"age" : 14, "username" : /.*/}).hint({"username" : 1, "age" : 1})
```

Hinting is usually unnecessary. MongoDB has a query optimizer and is very clever about choosing which index to use. When you first do a query, the query optimizer tries out a number of query plans concurrently. The first one to finish will be used, and the rest of the query executions are terminated. That query plan will be remembered for future queries on the same keys. The query optimizer periodically retries other plans, in case you've added new data and the previously chosen plan is no longer best. The only part you should need to worry about is giving the query optimizer useful indexes to choose from.

Index Administration

Metainformation about indexes is stored in the *system.indexes* collection of each database. This is a reserved collection, so you cannot insert documents into it or remove documents from it. You can manipulate its documents only through `ensureIndex` and the `dropIndexes` database command.

The *system.indexes* collection has detailed information about each index, but the *system.namespaces* collection also lists their names. If you look at this collection, you can see that there are at least two documents for each collection: one for the collection itself and one for each index it contains. For a collection with just the standard `"_id"` index, its *system.namespaces* entries would look like this:

```
{ "name" : "test.foo" }
{ "name" : "test.foo.$_id_"}
```

If we add a compound index on the name and age keys, a new document is added to *system.namespaces* with the following form:

```
{ "name" : "test.foo.$name_1_age_1" }
```

In Chapter 2, we mentioned that collection names are limited to 121 bytes in length. This somewhat strange limit is because the `"_id"` index's namespace needs 6 extra bytes (`".$_id_"`), bringing the namespace length for that index up to a more sensical 127 bytes.

It is important to remember that the size of the collection name plus the index name cannot exceed 127 bytes. If you approach the maximum namespace size or have large index names, you may have to use custom index names to avoid creating namespaces that are too long. It's generally easier just to keep database, collection, and key names to reasonable lengths.

Changing Indexes

As you and your application grow old together, you may find that your data or queries have changed and that the indexes that used to work well no longer do. You can add new indexes to existing collections at any time with `ensureIndex`:

```
> db.people.ensureIndex({"username" : 1}, {"background" : true})
```

Building indexes is time-consuming and resource-intensive. Using the `{"background" : true}` option builds the index in the background, while handling incoming requests. If you do not include the `background` option, the database will block all other requests while the index is being built.

Blocking lets index creation go faster but means your application will be unresponsive during the build. Even building an index in the background can take a toll on normal operations, so it is best to do it during an "off" time; building indexes in the background will still add extra load to your database, but it will not bring it grinding to a halt.

Creating indexes on existing documents is slightly faster than creating the index first and then inserting all of the documents. Of course, creating an index beforehand is an option only if you're populating a new collection from scratch.

If an index is no longer necessary, you can remove it with the `dropIndexes` command and the index name. Often, you may have to look at the *system.indexes* collection to figure out what the index name is, because even the autogenerated names vary somewhat from driver to driver:

```
> db.runCommand({"dropIndexes" : "foo", "index" : "alphabet"})
```

You can drop all indexes on a collection by passing in * as the value for the `index` key:

```
> db.runCommand({"dropIndexes" : "foo", "index" : "*"})
```

Another way to delete all indexes is to drop the collection. This will also delete the index on `_id` (and all of the documents in the collection). Removing all of the documents in a collection (with `remove`) does not affect the indexes; they will be repopulated when new documents are inserted.

Geospatial Indexing

There is another type of query that is becoming increasingly common (especially with the emergence of mobile devices): finding the nearest N things to a current location. MongoDB provides a special type of index for coordinate plane queries, called a *geospatial index*.

Suppose we wanted to find the nearest coffee shop to where we are now, given a latitude and longitude. We need to create a special index to do this type of query efficiently, because it needs to search in two dimensions. A geospatial index can be created using the `ensureIndex` function, but by passing `"2d"` as a value instead of 1 or -1:

```
> db.map.ensureIndex({"gps" : "2d"})
```

The `"gps"` key must be some type of pair value, that is, a two-element array or embedded document with two keys. These would all be valid:

```
{ "gps" : [ 0, 100 ] }
{ "gps" : { "x" : -30, "y" : 30 } }
{ "gps" : { "latitude" : -180, "longitude" : 180 } }
```

The keys can be anything; for example, `{"gps" : {"foo" : 0, "bar" : 1}}` is valid.

By default, geospatial indexing assumes that your values are going to range from -180 to 180 (which is convenient for latitude and longitude). If you are using it for other values, you can specify what the minimum and maximum values will be as options to `ensureIndex`:

```
> db.star.trek.ensureIndex({"light-years" : "2d"}, {"min" : -1000, "max" : 1000})
```

This will create a spatial index calibrated for a 2,000 × 2,000 light-year square.

Geospatial queries can be performed in two ways: as a normal query (using `find`) or as a database command. The `find` approach is similar to performing a nongeospatial query, except the conditional `"$near"` is used. It takes an array of the two target values:

```
> db.map.find({"gps" : {"$near" : [40, -73]}})
```

This finds all of the documents in the *map* collection, in order by distance from the point (40, -73). A default limit of 100 documents is applied if no limit is specified. If you don't need that many results, you should set a limit to conserve server resources. For example, the following code returns the 10 nearest documents to (40, -73):

```
> db.map.find({"gps" : {"$near" : [40, -73]}}).limit(10)
```

This same type of query can be performed using the `geoNear` command:

```
> db.runCommand({geoNear : "map", near : [40, -73], num : 10});
```

`geoNear` also returns the distance of each result document from the query point. The distance is in whatever units you inserted the data in; if you inserted degrees of longitude and latitude, the distance is in degrees. `find` with `"$near"` doesn't give you the distance for each point but must be used if you have more than 4MB of results.

MongoDB also allows you to find all of the documents within a shape, as well as near a point. To find points in a shape, we use the `"$within"` conditional instead of the `"$near"` conditional. `"$within"` can take an increasing number of shapes; check the online documentation for geospatial indexing to see the most up-to-date list of what's supported (*http://www.mongodb.org/display/DOCS/Geospatial+Indexing*). As of this writing, there are two options: you can query for all points within a rectangle or a circle. To use a rectangle, use the `"$box"` option:

```
> db.map.find({"gps" : {"$within" : {"$box" : [[10, 20], [15, 30]]}}})
```

`"$box"` takes a two-element array: the first element specifies the coordinates of the lower-left corner, the second element the upper right.

Also, you can find all points within a circle with `"$center"`, which takes an array with the center point and then a radius:

```
> db.map.find({"gps" : {"$within" : {"$center" : [[12, 25], 5]}}})
```

Compound Geospatial Indexes

Applications often need to search for more complex criteria than just a location. For example, a user might want to find all coffee shops or pizza parlors near where they are. To facilitate this type of query, you can combine geospatial indexes with normal indexes. In this situation, for instance, we might want to query on both the `"location"` key and a `"desc"` key, so we'd create a compound index:

```
> db.ensureIndex({"location" : "2d", "desc" : 1})
```

Then we can quickly find the nearest coffee shop:

```
> db.map.find({"location" : {"$near" : [-70, 30]}, "desc" : "coffeeshop"}).limit(1)
{
    "_id" : ObjectId("4c0d1348928a815a720a0000"),
    "name" : "Mud",
    "location" : [x, y],
    "desc" : ["coffee", "coffeeshop", "muffins", "espresso"]
}
```

Note that creating an array of keywords is a good way to perform user-defined searches.

The Earth Is Not a 2D Plane

MongoDB's geospatial indexes assumes that whatever you're indexing is a flat plane. This means that results aren't perfect for spherical shapes, like the earth, especially near the poles. The problem is that the distance between lines of longitude in the Yukon is much shorter than the distance between them at the equator. There are various projections that can be used to convert the earth to a 2D plane with varying levels of accuracy and corresponding levels of complexity.

CHAPTER 6

Aggregation

MongoDB provides a number of aggregation tools that go beyond basic query functionality. These range from simply counting the number of documents in a collection to using MapReduce to do complex data analysis.

count

The simplest aggregation tool is `count`, which returns the number of documents in the collection:

```
> db.foo.count()
0
> db.foo.insert({"x" : 1})
> db.foo.count()
1
```

Counting the total number of documents in a collection is fast regardless of collection size.

You can also pass in a query, and Mongo will count the number of results for that query:

```
> db.foo.insert({"x" : 2})
> db.foo.count()
2
> db.foo.count({"x" : 1})
1
```

This can be useful for getting a total for pagination: "displaying results 0–10 of 439." Adding criteria does make the count slower, however.

distinct

The `distinct` command finds all of the distinct values for a given key. You must specify a collection and key:

```
> db.runCommand({"distinct" : "people", "key" : "age"})
```

For example, suppose we had the following documents in our collection:

```
{"name" : "Ada", "age" : 20}
{"name" : "Fred", "age" : 35}
{"name" : "Susan", "age" : 60}
{"name" : "Andy", "age" : 35}
```

If you call distinct on the `"age"` key, you will get back all of the distinct ages:

```
> db.runCommand({"distinct" : "people", "key" : "age"})
{"values" : [20, 35, 60], "ok" : 1}
```

A common question at this point is if there's a way to get all of the distinct *keys* in a collection. There is no built-in way of doing this, although you can write something to do it yourself using MapReduce (described in a moment).

group

`group` allows you to perform more complex aggregation. You choose a key to group by, and MongoDB divides the collection into separate groups for each value of the chosen key. For each group, you can create a result document by aggregating the documents that are members of that group.

> If you are familiar with SQL, `group` is similar to SQL's `GROUP BY`.

Suppose we have a site that keeps track of stock prices. Every few minutes from 10 a.m. to 4 p.m., it gets the latest price for a stock, which it stores in MongoDB. Now, as part of a reporting application, we want to find the closing price for the past 30 days. This can be easily accomplished using `group`.

The collection of stock prices contains thousands of documents with the following form:

```
{"day" : "2010/10/03", "time" : "10/3/2010 03:57:01 GMT-400", "price" : 4.23}
{"day" : "2010/10/04", "time" : "10/4/2010 11:28:39 GMT-400", "price" : 4.27}
{"day" : "2010/10/03", "time" : "10/3/2010 05:00:23 GMT-400", "price" : 4.10}
{"day" : "2010/10/06", "time" : "10/6/2010 05:27:58 GMT-400", "price" : 4.30}
{"day" : "2010/10/04", "time" : "10/4/2010 08:34:50 GMT-400", "price" : 4.01}
```

> You should never store money amounts as floating-point numbers because of inexactness concerns, but we'll do it anyway in this example for simplicity.

We want our results to be a list of the latest time and price for each day, something like this:

```
[
    {"time" : "10/3/2010 05:00:23 GMT-400", "price" : 4.10},
    {"time" : "10/4/2010 11:28:39 GMT-400", "price" : 4.27},
    {"time" : "10/6/2010 05:27:58 GMT-400", "price" : 4.30}
]
```

We can accomplish this by splitting the collection into sets of documents grouped by day then finding the document with the latest timestamp for each day and adding it to the result set. The whole function might look something like this:

```
> db.runCommand({"group" : {
... "ns" : "stocks",
... "key" : "day",
... "initial" : {"time" : 0},
... "$reduce" : function(doc, prev) {
...     if (doc.time > prev.time) {
...         prev.price = doc.price;
...         prev.time = doc.time;
...     }
... }}})
```

Let's break this command down into its component keys:

`"ns" : "stocks"`
: This determines which collection we'll be running the group on.

`"key" : "day"`
: This specifies the key on which to group the documents in the collection. In this case, that would be the `"day"` key. All of the documents with a `"day"` key of a given value will be grouped together.

`"initial" : {"time" : 0}`
: The first time the `reduce` function is called for a given group, it will be passed the initialization document. This same accumulator will be used for each member of a given group, so any changes made to it can be persisted.

`"$reduce" : function(doc, prev) { ... }`
: This will be called once for each document in the collection. It is passed the current document and an accumulator document: the result so far for that group. In this example, we want the `reduce` function to compare the current document's time with the accumulator's time. If the current document has a later time, we'll set the accumulator's day and price to be the current document's values. Remember that there is a separate accumulator for each group, so there is no need to worry about different days using the same accumulator.

In the initial statement of the problem, we said that we wanted only the last 30 days worth of prices. Our current solution is iterating over the entire collection, however. This is why you can include a `"condition"` that documents must satisfy in order to be processed by the group command at all:

```
> db.runCommand({"group" : {
... "ns" : "stocks",
... "key" : "day",
... "initial" : {"time" : 0},
... "$reduce" : function(doc, prev) {
...     if (doc.time > prev.time) {
...         prev.price = doc.price;
...         prev.time = doc.time;
...     }},
... "condition" : {"day" : {"$gt" : "2010/09/30"}}
... }})
```

Some documentation refers to a `"cond"` or `"q"` key, both of which are identical to the `"condition"` key (just less descriptive).

Now the command will return an array of 30 documents, each of which is a group. Each group has the key on which the group was based (in this case, `"day" : string`) and the final value of `prev` for that group. If some of the documents do not contain the key, these will be grouped into a single group with a `day : null` element. You can eliminate this group by adding `"day" : {"$exists" : true}` to the `"condition"`. The `group` command also returns the total number of documents used and the number of distinct values for `"key"`:

```
> db.runCommand({"group" : {...}})
{
    "retval" :
        [
            {
                "day" : "2010/10/04",
                "time" : "Mon Oct 04 2010 11:28:39 GMT-0400 (EST)"
                "price" : 4.27
            },
            ...
        ],
    "count" : 734,
    "keys" : 30,
    "ok" : 1
}
```

We explicitly set the `"price"` for each group, and the `"time"` was set by the initializer and then updated. The `"day"` is included because the key being grouped by is included by default in each `"retval"` embedded document. If you don't want to return this key, you can use a finalizer to change the final accumulator document into anything, even a nondocument (e.g., a number or string).

Using a Finalizer

Finalizers can be used to minimize the amount of data that needs to be transferred from the database to the user, which is important, because the `group` command's output needs to fit in a single database response. To demonstrate this, we'll take the example of a blog where each post has tags. We want to find the most popular tag for each day. We can group by day (again) and keep a count for each tag. This might look something like this:

```
> db.posts.group({
... "key" : {"tags" : true},
... "initial" : {"tags" : {}},
... "$reduce" : function(doc, prev) {
...     for (i in doc.tags) {
...         if (doc.tags[i] in prev.tags) {
...             prev.tags[doc.tags[i]]++;
...         } else {
...             prev.tags[doc.tags[i]] = 1;
...         }
...     }
... }})
```

This will return something like this:

```
[
    {"day" : "2010/01/12", "tags" : {"nosql" : 4, "winter" : 10, "sledding" : 2}},
    {"day" : "2010/01/13", "tags" : {"soda" : 5, "php" : 2}},
    {"day" : "2010/01/14", "tags" : {"python" : 6, "winter" : 4, "nosql": 15}}
]
```

Then we could find the largest value in the `"tags"` document on the client side. However, sending the entire tags document for every day is a lot of extra overhead to send to the client: an entire set of key/value pairs for each day, when all we want is a single string. This is why `group` takes an optional `"finalize"` key. `"finalize"` can contain a function that is run on each group once, right before the result is sent back to the client. We can use a `"finalize"` function to trim out all of the cruft from our results:

```
> db.runCommand({"group" : {
... "ns" : "posts",
... "key" : {"tags" : true},
... "initial" : {"tags" : {}},
... "$reduce" : function(doc, prev) {
...     for (i in doc.tags) {
...         if (doc.tags[i] in prev.tags) {
...             prev.tags[doc.tags[i]]++;
...         } else {
...             prev.tags[doc.tags[i]] = 1;
...         }
...     },
... "finalize" : function(prev) {
...     var mostPopular = 0;
...     for (i in prev.tags) {
...         if (prev.tags[i] > mostPopular) {
...             prev.tag = i;
...             mostPopular = prev.tags[i];
...         }
...     }
...     delete prev.tags
... }}})
```

Now, we're only getting the information we want; the server will send back something like this:

```
[
    {"day" : "2010/01/12", "tag" : "winter"},
    {"day" : "2010/01/13", "tag" : "soda"},
    {"day" : "2010/01/14", "tag" : "nosql"}
]
```

`finalize` can either modify the argument passed in or return a new value.

Using a Function as a Key

Sometimes you may have more complicated criteria that you want to group by, not just a single key. Suppose you are using `group` to count how many blog posts are in each category. (Each blog post is in a single category.) Post authors were inconsistent, though, and categorized posts with haphazard capitalization. So, if you group by category name, you'll end up with separate groups for "MongoDB" and "mongodb." To make sure any variation of capitalization is treated as the same key, you can define a function to determine documents' grouping key.

To define a grouping function, you must use a `$keyf` key (instead of `"key"`). Using `"$keyf"` makes the `group` command look something like this:

```
> db.posts.group({"ns" : "posts",
... "$keyf" : function(x) { return x.category.toLowerCase(); },
... "initializer" : ... })
```

`"$keyf"` allows you can group by arbitrarily complex criteria.

MapReduce

MapReduce is the Uzi of aggregation tools. Everything described with `count`, `distinct`, and `group` can be done with MapReduce, and more. It is a method of aggregation that can be easily parallelized across multiple servers. It splits up a problem, sends chunks of it to different machines, and lets each machine solve its part of the problem. When all of the machines are finished, they merge all of the pieces of the solution back into a full solution.

MapReduce has a couple of steps. It starts with the map step, which *maps* an operation onto every document in a collection. That operation could be either "do nothing" or "emit these keys with *X* values." There is then an intermediary stage called the shuffle step: keys are grouped and lists of emitted values are created for each key. The reduce takes this list of values and *reduces* it to a single element. This element is returned to the shuffle step until each key has a list containing a single value: the result.

The price of using MapReduce is speed: `group` is not particularly speedy, but MapReduce is slower and is not supposed to be used in "real time." You run

MapReduce as a background job, it creates a collection of results, and then you can query that collection in real time.

We'll go through a couple of MapReduce examples because it is incredibly useful and powerful but also a somewhat complex tool.

Example 1: Finding All Keys in a Collection

Using MapReduce for this problem might be overkill, but it is a good way to get familiar with how MapReduce works. If you already understand MapReduce, feel free to skip ahead to the last part of this section, where we cover MongoDB-specific MapReduce considerations.

MongoDB is schemaless, so it does not keep track of the keys in each document. The best way, in general, to find all the keys across all the documents in a collection is to use MapReduce. In this example, we'll also get a count of how many times each key appears in the collection. This example doesn't include keys for embedded documents, but it would be a simple addition to the `map` function to do so.

For the mapping step, we want to get every key of every document in the collection. The `map` function uses a special function to "return" values that we want to process later: `emit`. `emit` gives MapReduce a key (like the one used by `group` earlier) and a value. In this case, we emit a count of how many times a given key appeared in the document (once: `{count : 1}`). We want a separate count for each key, so we'll call emit for every key in the document. `this` is a reference to the current document we are mapping:

```
> map = function() {
... for (var key in this) {
...     emit(key, {count : 1});
... }};
```

Now we have a ton of little `{count : 1}` documents floating around, each associated with a key from the collection. An array of one or more of these `{count : 1}` documents will be passed to the `reduce` function. The `reduce` function is passed two arguments: `key`, which is the first argument from `emit`, and an array of one or more `{count : 1}` documents that were emitted for that key:

```
> reduce = function(key, emits) {
... total = 0;
... for (var i in emits) {
...     total += emits[i].count;
... }
... return {"count" : total};
... }
```

`reduce` must be able to be called repeatedly on results from either the map phase or previous reduce phases. Therefore, `reduce` must return a document that can be re-sent to `reduce` as an element of its second argument. For example, say we have the key `x` mapped to three documents: `{count : 1, id : 1}`, `{count : 1, id : 2}`, and `{count :`

`1, id : 3}`. (The ID keys are just for identification purposes.) MongoDB might call `reduce` in the following pattern:

```
> r1 = reduce("x", [{count : 1, id : 1}, {count : 1, id : 2}])
{count : 2}
> r2 = reduce("x", [{count : 1, id : 3}])
{count : 1}
> reduce("x", [r1, r2])
{count : 3}
```

You cannot depend on the second argument always holding one of the initial documents (`{count : 1}` in this case) or being a certain length. `reduce` should be able to be run on any combination of `emit` documents and `reduce` return values.

Altogether, this MapReduce function would look like this:

```
> mr = db.runCommand({"mapreduce" : "foo", "map" : map, "reduce" : reduce})
{
    "result" : "tmp.mr.mapreduce_1266787811_1",
    "timeMillis" : 12,
    "counts" : {
        "input" : 6
        "emit" : 14
        "output" : 5
    },
    "ok" : true
}
```

The document MapReduce returns gives you a bunch of metainformation about the operation:

`"result" : "tmp.mr.mapreduce_1266787811_1"`
: This is the name of the collection the MapReduce results were stored in. This is a temporary collection that will be deleted when the connection that did the MapReduce is closed. We will go over how to specify a nicer name and make the collection permanent in a later part of this chapter.

`"timeMillis" : 12`
: How long the operation took, in milliseconds.

`"counts" : { ... }`
: This embedded document contains three keys:

  `"input" : 6`
  : The number of documents sent to the `map` function.

  `"emit" : 14`
  : The number of times `emit` was called in the `map` function.

  `"output" : 5`
  : The number of documents created in the result collection.

  `"counts"` is mostly useful for debugging.

If we do a find on the resulting collection, we can see all of the keys and their counts from our original collection:

```
> db[mr.result].find()
{ "_id" : "_id", "value" : { "count" : 6 } }
{ "_id" : "a", "value" : { "count" : 4 } }
{ "_id" : "b", "value" : { "count" : 2 } }
{ "_id" : "x", "value" : { "count" : 1 } }
{ "_id" : "y", "value" : { "count" : 1 } }
```

Each of the key values becomes an `"_id"`, and the final result of the reduce step(s) becomes the `"value"`.

Example 2: Categorizing Web Pages

Suppose we have a site where people can submit links to other pages, such as reddit.com. Submitters can tag a link as related to certain popular topics, e.g., "politics," "geek," or "icanhascheezburger." We can use MapReduce to figure out which topics are the most popular, as a combination of recent and most-voted-for.

First, we need a `map` function that emits tags with a value based on the popularity and recency of a document:

```
map = function() {
    for (var i in this.tags) {
        var recency = 1/(new Date() - this.date);
        var score = recency * this.score;

        emit(this.tags[i], {"urls" : [this.url], "score" : score});
    }
};
```

Now we need to reduce all of the emitted values for a tag into a single score for that tag:

```
reduce = function(key, emits) {
    var total = {urls : [], score : 0}
    for (var i in emits) {
        emits[i].urls.forEach(function(url) {
            total.urls.push(url);
        }
        total.score += emits[i].score;
    }
    return total;
};
```

The final collection will end up with a full list of URLs for each tag and a score showing how popular that particular tag is.

MongoDB and MapReduce

Both of the previous examples used only the `mapreduce`, `map`, and `reduce` keys. These three keys are required, but there are many optional keys that can be passed to the MapReduce command.

`"finalize" :` *`function`*
: A final step to send `reduce`'s output to.

`"keeptemp" :` *`boolean`*
: If the temporary result collection should be saved when the connection is closed.

`"output" :` *`string`*
: Name for the output collection. Setting this option implies `keeptemp : true`.

`"query" :` *`document`*
: Query to filter documents by before sending to the `map` function.

`"sort" :` *`document`*
: Sort to use on documents before sending to the map (useful in conjunction with the `limit` option).

`"limit" :` *`integer`*
: Maximum number of documents to send to the `map` function.

`"scope" :` *`document`*
: Variables that can be used in any of the JavaScript code.

`"verbose" :` *`boolean`*
: Whether to use more verbose output in the server logs.

The finalize function

As with the previous `group` command, MapReduce can be passed a `finalize` function that will be run on the last `reduce`'s output before it is saved to a temporary collection.

Returning large result sets is less critical with MapReduce than `group`, because the whole result doesn't have to fit in 4MB. However, the information will be passed over the wire eventually, so `finalize` is a good chance to take averages, chomp arrays, and remove extra information in general.

Keeping output collections

By default, Mongo creates a temporary collection while it is processing the MapReduce with a name that you are unlikely to choose for a collection: a dot-separated string containing *mr*, the name of the collection you're MapReducing, a timestamp, and the job's ID with the database. It ends up looking something like *mr.stuff.18234210220.2*. MongoDB will automatically destroy this collection when the connection that did the MapReduce is closed. (You can also drop it manually when you're done with it.) If you want to persist this collection even after disconnecting, you can specify `keeptemp : true` as an option.

If you'll be using the temporary collection regularly, you may want to give it a better name. You can specify a more human-readable name with the `out` option, which takes a string. If you specify `out`, you need not specify `keeptemp : true`, because it is implied. Even if you specify a "pretty" name for the collection, MongoDB will use the autogenerated collection name for intermediate steps of the MapReduce. When it has finished, it will automatically and atomically rename the collection from the autogenerated name to your chosen name. This means that if you run MapReduce multiple times with the same target collection, you will never be using an incomplete collection for operations.

The output collection created by MapReduce is a normal collection, which means that there is no problem with doing a MapReduce on it, or a MapReduce on the results from that MapReduce, ad infinitum!

MapReduce on a subset of documents

Sometimes you need to run MapReduce on only part of a collection. You can add a query to filter the documents before they are passed to the `map` function.

Every document passed to the `map` function needs to be deserialized from BSON into a JavaScript object, which is a fairly expensive operation. If you know that you will need to run MapReduce only on a subset of the documents in the collection, adding a filter can greatly speed up the command. The filter is specified by the `"query"`, `"limit"`, and `"sort"` keys.

The `"query"` key takes a query document as a value. Any documents that would ordinarily be returned by that query will be passed to the `map` function. For example, if we have an application tracking analytics and want a summary for the last week, we can use MapReduce on only the most recent week's documents with the following command:

```
> db.runCommand({"mapreduce" : "analytics", "map" : map, "reduce" : reduce,
                 "query" : {"date" : {"$gt" : week_ago}}})
```

The `sort` option is mostly useful in conjunction with `limit`. `limit` can be used on its own, as well, to simply provide a cutoff on the number of documents sent to the `map` function.

If, in the previous example, we wanted an analysis of the last 10,000 page views (instead of the last week), we could use `limit` and `sort`:

```
> db.runCommand({"mapreduce" : "analytics", "map" : map, "reduce" : reduce,
                 "limit" : 10000, "sort" : {"date" : -1}})
```

`query`, `limit`, and `sort` can be used in any combination, but `sort` isn't useful if `limit` isn't present.

Using a scope

MapReduce can take a code type for the `map`, `reduce`, and `finalize` functions, and, in most languages, you can specify a scope to be passed with code. However, MapReduce ignores this scope. It has its own scope key, `"scope"`, and you must use that if there are client-side values you want to use in your MapReduce. You can set them using a plain document of the form `variable_name : value`, and they will be available in your `map`, `reduce`, and `finalize` functions. The scope is immutable from within these functions.

For instance, in the example in the previous section, we calculated the recency of a page using `1/(new Date() - this.date)`. We could, instead, pass in the current date as part of the scope with the following code:

```
> db.runCommand({"mapreduce" : "webpages", "map" : map, "reduce" : reduce,
                 "scope" : {now : new Date()}})
```

Then, in the `map` function, we could say `1/(now - this.date)`.

Getting more output

There is also a verbose option for debugging. If you would like to see the progress of your MapReduce as it runs, you can specify `"verbose" : true`.

You can also use `print` to see what's happening in the `map`, `reduce`, and `finalize` functions. `print` will print to the server log.

CHAPTER 7

Advanced Topics

MongoDB supports some advanced functionality that goes well beyond the capabilities discussed so far. When you want to become a power user, this chapter has you covered; in it we'll discuss the following:

- Using database commands to take advantage of advanced features
- Working with capped collections, a special type of collection
- Leveraging GridFS for storing large files
- Taking advantage of MongoDB's support for server-side JavaScript
- Understanding what database references are and when you should consider using them

Database Commands

In the previous chapters we've seen how to create, read, update, and delete documents in MongoDB. In addition to these basic operations, MongoDB supports a wide range of advanced operations that are implemented as *commands*. Commands implement all of the functionality that doesn't fit neatly into "create, read, update, delete."

We've already seen a couple of commands in the previous chapters; for instance, we used the `getLastError` command in Chapter 3 to check the number of documents affected by an update:

```
> db.count.update({x : 1}, {$inc : {x : 1}}, false, true)
> db.runCommand({getLastError : 1})
{
    "err" : null,
    "updatedExisting" : true,
    "n" : 5,
    "ok" : true
}
```

In this section, we'll take a closer look at commands to see exactly what they are and how they're implemented. We'll also describe some of the most useful commands that are supported by MongoDB.

How Commands Work

One example of a database command that you are probably familiar with is `drop`: to drop a collection from the shell, we run `db.test.drop()`. Under the hood, this function is actually running the `drop` command—we can perform the exact same operation using `runCommand`:

```
> db.runCommand({"drop" : "test"});
{
    "nIndexesWas" : 1,
    "msg" : "indexes dropped for collection",
    "ns" : "test.test",
    "ok" : true
}
```

The document we get as a result is the *command response*, which contains information about whether the command was successful, as well as any other information that the command might provide. The command response will always contain the key `"ok"`. If `"ok"` is `true`, the command was successful, and if it is `false`, the command failed for some reason.

In version 1.5 and earlier, the value of `"ok"` was `1.0` or `0.0` instead of `true` or `false`, respectively.

If `"ok"` is `false`, then an additional key will be present, `"errmsg"`. The value of `"errmsg"` is a string explaining why the command failed. As an example, let's try running the `drop` command again, on the collection that we just dropped:

```
> db.runCommand({"drop" : "test"});
{ "errmsg" : "ns not found", "ok" : false }
```

Commands in MongoDB are actually implemented as a special type of query that gets performed on the *$cmd* collection. `runCommand` just takes a command document and performs the equivalent query, so our drop call becomes the following:

```
db.$cmd.findOne({"drop" : "test"});
```

When the MongoDB server gets a query on the *$cmd* collection, it handles it using special logic, rather than the normal code for handling queries. Almost all MongoDB drivers provide a helper method like `runCommand` for running commands, but commands can always be run using a simple query if necessary.

Some commands require administrator access and must be run on the *admin* database. If such a command is run on any other database, it will return an "access denied" error.

Command Reference

At the time of this writing, MongoDB supports more than 75 different commands, and more commands are being added all the time. There are two ways to get an up-to-date list of all of the commands supported by a MongoDB server:

- Run `db.listCommands()` from the shell, or run the equivalent `listCommands` command from any other driver.
- Browse to the *http://localhost:28017/_commands* URL on the MongoDB admin interface (for more on the admin interface see Chapter 8).

The following list contains some of the most frequently used MongoDB commands along with example documents showing how each command should be represented:

buildInfo

```
{"buildInfo" : 1}
```

Admin-only command that returns information about the MongoDB server's version number and host operating system.

collStats

```
{"collStats" : collection}
```

Gives some stats about a given collection, including its data size, the amount of storage space allocated for it, and the size of its indexes.

distinct

```
{"distinct" : collection, "key": key, "query": query}
```

Gets a list of distinct values for ***`key`*** in documents matching ***`query`***, across a given collection.

drop

```
{"drop" : collection}
```

Removes all data for ***`collection`***.

dropDatabase

```
{"dropDatabase" : 1}
```

Removes all data for the current database.

dropIndexes

```
{"dropIndexes" : collection, "index" : name}
```

Deletes the index named ***`name`*** from ***`collection`***, or all indexes if ***`name`*** is `"*"`.

findAndModify

See Chapter 3 for a full reference on using the `findAndModify` command.

getLastError

```
{"getLastError" : 1[, "w" : w[, "wtimeout" : timeout]]}
```

Checks for errors or other status information about the last operation performed on this connection. The command will optionally block until `w` slaves have replicated the last operation on this connection (or until `timeout` milliseconds have gone by).

isMaster

```
{"isMaster" : 1}
```

Checks if this server is a master or slave.

listCommands

```
{"listCommands" : 1}
```

Returns a list of all database commands available on this server, as well as some information about each command.

listDatabases

```
{"listDatabases" : 1}
```

Admin-only command listing all databases on this server.

ping

```
{"ping" : 1}
```

Checks if a server is alive. This command will return immediately even if the server is in a lock.

renameCollection

```
{"renameCollection" : a, "to" : b}
```

Renames collection `a` to `b`, where both `a` and `b` are *full collection namespaces* (e.g., `"foo.bar"` for the collection *bar* in the *foo* database).

repairDatabase

```
{"repairDatabase" : 1}
```

Repairs and compacts the current database, which can be a long-running operation. See "Repair" on page 124 for more information.

serverStatus

```
{"serverStatus" : 1}
```

Gets administrative statistics for this server. See "Monitoring" on page 114 for more information.

Remember, there are far more supported commands than just those listed earlier. Others are documented as appropriate throughout the rest of the book, and for the full list, just run `listCommands`.

Capped Collections

We've already seen how normal collections in MongoDB are created dynamically and automatically grow in size to fit additional data. MongoDB also supports a different type of collection, called a *capped collection*, which is created in advance and is fixed in size (see Figure 7-1). Having fixed-size collections brings up an interesting question: what happens when we try to insert into a capped collection that is already full? The answer is that capped collections behave like circular queues: if we're out of space, the oldest document(s) will be deleted, and the new one will take its place (see Figure 7-2). This means that capped collections automatically age-out the oldest documents as new documents are inserted.

Certain operations are not allowed on capped collections. Documents cannot be removed or deleted (aside from the automatic age-out described earlier), and updates that would cause documents to move (in general updates that cause documents to grow in size) are disallowed. By preventing these two operations, we guarantee that documents in a capped collection are stored in insertion order and that there is no need to maintain a free list for space from removed documents.

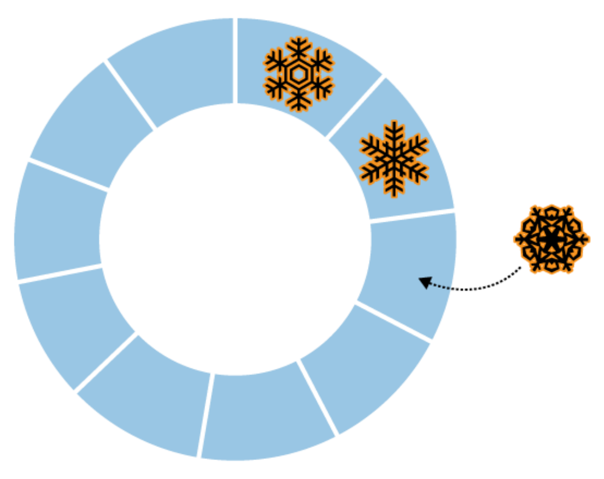


Figure 7-1. New documents are inserted at the end of the queue

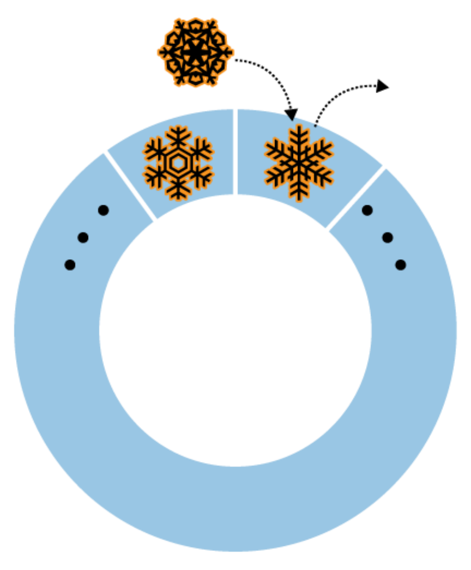


Figure 7-2. When the queue is full, the oldest element will be replaced by the newest

A final difference between capped and normal collections is that in a capped collection, there are no indexes by default, not even an index on `"_id"`.

Properties and Use Cases

The set of features and limitations possessed by capped collections combine to give them some interesting properties. First, inserts into a capped collection are extremely fast. When doing an insert, there is never a need to allocate additional space, and the server never needs to search through a free list to find the right place to put a document. The inserted document can always be placed directly at the "tail" of the collection, overwriting old documents if needed. By default, there are also no indexes to update on an insert, so an insert is essentially a single `memcpy`.

Another interesting property of capped collections is that queries retrieving documents in insertion order are very fast. Because documents are always stored in insertion order, queries for documents in that order just walk over the collection, returning documents in the exact order that they appear on disk. By default, any `find` performed on a capped collection will always return results in insertion order.

Finally, capped collections have the useful property of automatically aging-out old data as new data is inserted. The combination of fast inserts, fast queries for documents sorted by insertion order, and automatic age-out makes capped collections ideal for use cases like logging. In fact, the primary motivation for including capped collections in MongoDB is so that they can be used to store an internal replication log, the *oplog* (for more on replication and the oplog, see Chapter 9). Another good use case to consider for capped collections is caching of small numbers of documents. In general, capped collections are good for any case where the auto age-out property is helpful as opposed to undesirable and the limitations on available operations are not prohibitive.

Creating Capped Collections

Unlike normal collections, capped collections must be explicitly created before they are used. To create a capped collection, use the `create` command. From the shell, this can be done using `createCollection`:

```
> db.createCollection("my_collection", {capped: true, size: 100000});
{ "ok" : true }
```

The previous command creates a capped collection, *my_collection*, that is a fixed size of 100,000 bytes. `createCollection` has a couple of other options as well. We can specify a limit on the number of documents in a capped collection in addition to the limit on total collection size:

```
> db.createCollection("my_collection", {capped: true, size: 100000, max: 100});
{ "ok" : true }
```

When limiting the number of documents in a capped collection, you must specify a size limit as well. Age-out will be based on the number of documents in the collection, *unless* the collection runs out of space before the limit is reached. In that case, age-out will be based on collection size, as in any other capped collection.

Another option for creating a capped collection is to convert an existing, regular collection into a capped collection. This can be done using the `convertToCapped` command—in the following example, we convert the *test* collection to a capped collection of 10,000 bytes:

```
> db.runCommand({convertToCapped: "test", size: 10000});
{ "ok" : true }
```

Sorting Au Naturel

There is a special type of sort that you can do with capped collections, called a *natural sort*. Natural order is just the order that documents appear on disk (see Figure 7-3).

Because documents in a capped collection are always kept in insertion order, natural order is the same as insertion order. As mentioned earlier, queries on a capped collection return documents in insertion order by default. You can also sort in reverse insertion order with a natural sort (see Figure 7-4):

```
> db.my_collection.find().sort({"$natural" : -1})
```

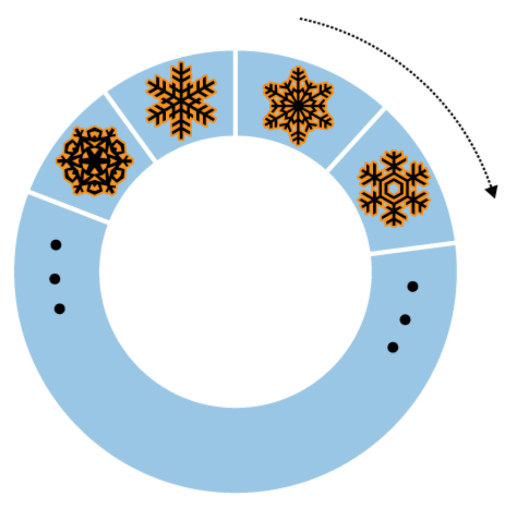


Figure 7-3. Sort by {"$natural" : 1}

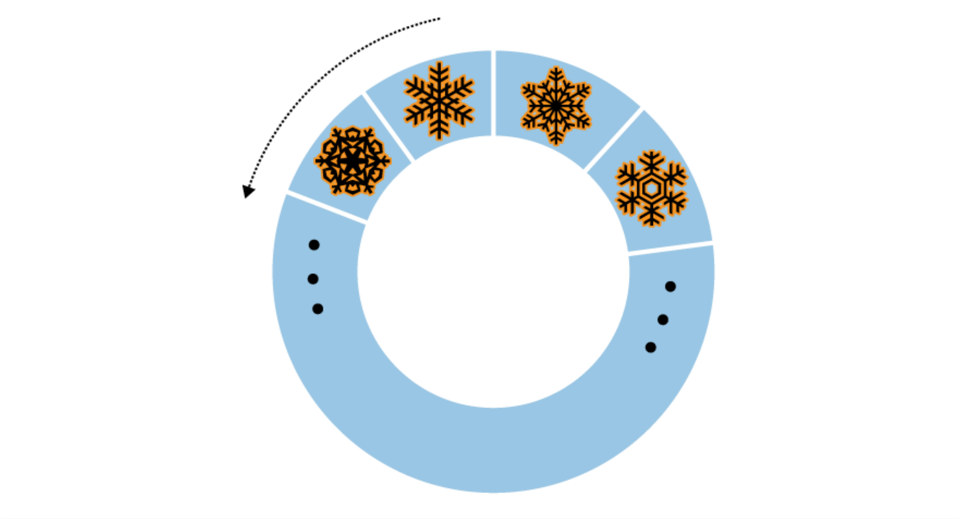


Figure 7-4. Sort by {"$natural" : -1}

Sorting by `{"$natural" : 1}` is identical to the default sort. Noncapped collections do not guarantee that documents are stored in any particular order, so their natural ordering is not as significant.

Tailable Cursors

Tailable cursors are a very special type of persistent cursor that are not closed when their results are exhausted. They were inspired by the *tail -f* command and, similar to the command, will continue fetching output for as long as possible. Because the cursors do not die when they runs out of results, they can continue to fetch new results as they are added to the collection. Tailable cursors can be used only on capped collections.

Again, the Mongo shell does not allow you to use tailable cursors, but using one in PHP looks something like the following:

```
$cursor = $collection->find()->tailable();

while (true) {
    if (!$cursor->hasNext()) {
        if ($cursor->dead()) {
            break;
        }
        sleep(1);
    }
    else {
        while (cursor->hasNext()) {
            do_stuff(cursor->getNext());
        }
    }
}
```

Although the cursor has not died, it will be either processing results or waiting for more results to arrive.

GridFS: Storing Files

GridFS is a mechanism for storing large binary files in MongoDB. There are several reasons why you might consider using GridFS for file storage:

- Using GridFS can simplify your stack. If you're already using MongoDB, GridFS obviates the need for a separate file storage architecture.
- GridFS will leverage any existing replication or autosharding that you've set up for MongoDB, so getting failover and scale-out for file storage is easy.
- GridFS can alleviate some of the issues that certain filesystems can exhibit when being used to store user uploads. For example, GridFS does not have issues with storing large numbers of files in the same directory.
- You can get great disk locality with GridFS, because MongoDB allocates data files in 2GB chunks.

Getting Started with GridFS: mongofiles

The easiest way to get up and running with GridFS is by using the `mongofiles` utility. `mongofiles` is included with all MongoDB distributions and can be used to upload, download, list, search for, or delete files in GridFS. As with any of the other command-line tools, run `mongofiles --help` to see the options available for `mongofiles`. The following session shows how to use `mongofiles` to upload a file from the filesystem to GridFS, list all of the files in GridFS, and download a file that we've previously uploaded:

```
$ echo "Hello, world" > foo.txt
$ ./mongofiles put foo.txt
connected to: 127.0.0.1
added file: { _id: ObjectId('4c0d2a6c3052c25545139b88'),
              filename: "foo.txt", length: 13, chunkSize: 262144,
              uploadDate: new Date(1275931244818),
              md5: "a7966bf58e23583c9a5a4059383ff850" }
done!
$ ./mongofiles list
connected to: 127.0.0.1
foo.txt 13
$ rm foo.txt
$ ./mongofiles get foo.txt
connected to: 127.0.0.1
done write to: foo.txt
$ cat foo.txt
Hello, world
```

In the previous example, we perform three basic operations using `mongofiles`: `put`, `list`, and `get`. The `put` operation takes a file in the filesystem and adds it to GridFS, `list` will list any files that have been added to GridFS, and `get` does the inverse of `put`: it takes a file from GridFS and writes it to the filesystem. `mongofiles` also supports two other operations: `search` for finding files in GridFS by filename and `delete` for removing a file from GridFS.

Working with GridFS from the MongoDB Drivers

We've seen how easy it is to work with GridFS from the command line, and it's equally easy to work with from the MongoDB drivers. For example, we can use PyMongo, the Python driver for MongoDB, to perform the same series of operations as we did with `mongofiles`:

```
>>> from pymongo import Connection
>>> import gridfs
>>> db = Connection().test
>>> fs = gridfs.GridFS(db)
>>> file_id = fs.put("Hello, world", filename="foo.txt")
>>> fs.list()
[u'foo.txt']
>>> fs.get(file_id).read()
'Hello, world'
```

The API for working with GridFS from PyMongo is very similar to that of `mongofiles`: we can easily perform the basic `put`, `get`, and `list` operations. Almost all of the MongoDB drivers follow this basic pattern for working with GridFS, while often exposing more advanced functionality as well. For driver-specific information on GridFS, please check out the documentation for the specific driver you're using.

Under the Hood

GridFS is a lightweight specification for storing files that is built on top of normal MongoDB documents. The MongoDB server actually does almost nothing to "special-case" the handling of GridFS requests; all of the work is handled by the client-side drivers and tools.

The basic idea behind GridFS is that we can store large files by splitting them up into *chunks* and storing each chunk as a separate document. Because MongoDB supports storing binary data in documents, we can keep storage overhead for chunks to a minimum. In addition to storing each chunk of a file, we store a single document that groups the chunks together and contains metadata about the file.

The chunks for GridFS are stored in their own collection. By default chunks will use the collection *fs.chunks*, but this can be overridden if needed. Within the chunks collection the structure of the individual documents is pretty simple:

```
{
    "_id" : ObjectId("..."),
    "n" : 0,
    "data" : BinData("..."),
    "files_id" : ObjectId("...")
}
```

Like any other MongoDB document, the chunk has its own unique `"_id"`. In addition, it has a couple of other keys. `"files_id"` is the `"_id"` of the file document that contains the metadata for this chunk. `"n"` is the chunk number; this attribute tracks the order that chunks were present in the original file. Finally, `"data"` contains the binary data that makes up this chunk of the file.

The metadata for each file is stored in a separate collection, which defaults to *fs.files*. Each document in the files collection represents a single file in GridFS and can contain any custom metadata that should be associated with that file. In addition to any user-defined keys, there are a couple of keys that are mandated by the GridFS specification:

_id
: A unique id for the file—this is what will be stored in each chunk as the value for the `"files_id"` key.

length
: The total number of bytes making up the content of the file.

chunkSize
: The size of each chunk comprising the file, in bytes. The default is 256K, but this can be adjusted if needed.

uploadDate
: A timestamp representing when this file was stored in GridFS.

md5
: An md5 checksum of this file's contents, generated on the server side.

Of all of the required keys, perhaps the most interesting (or least self-explanatory) is `"md5"`. The value for `"md5"` is generated by the MongoDB server using the `filemd5` command, which computes the md5 checksum of the uploaded chunks. This means that users can check the value of the `"md5"` key to ensure that a file was uploaded correctly.

When we understand the underlying GridFS specification, it becomes trivial to implement features that the driver we're using might not implement for us. For example, we can use the `distinct` command to get a list of unique filenames stored in GridFS:

```
> db.fs.files.distinct("filename")
[ "foo.txt" ]
```

Server-Side Scripting

JavaScript can be executed on the server using the `db.eval` function. It can also be stored in the database and is used in some database commands.

db.eval

`db.eval` is a function that allows you to execute arbitrary JavaScript on the MongoDB server. It takes a string of JavaScript, sends it to MongoDB (which executes it), and returns the result.

`db.eval` can be used to imitate multidocument transactions: `db.eval` locks the database, executes the JavaScript, and unlocks the database. There's no built-in rollback, but this does give you a guarantee of a series of operations occurring in a certain order (unless an error occurs).

There are two options for sending code: enclosing it in a function or not. The following two lines are equivalent:

```
> db.eval("return 1;")
1
> db.eval("function() { return 1; }")
1
```

Defining an enclosing function is necessary only if you are passing in arguments. These can be passed using `db.eval`'s second argument, which is an array of values. For example, if we wanted to pass the username as an argument to a function, we could say the following:

```
> db.eval("function(u) { print('Hello, '+u+'!'); }", [username])
```

You can pass in as many arguments as necessary. For instance, if we want a sum of three numbers, we can do the following:

```
> db.eval("function(x,y,z) { return x + y + z; }", [num1, num2, num3])
```

`num1` becomes `x`, `num2` becomes `y`, and `num3` becomes `z`. If you would like to use a variable number of parameters, arguments in JavaScript are stored in an `arguments` array when a function is called.

As a `db.eval` expression becomes more complex, debugging can be tricky. The JavaScript code is run by the database and often doesn't have useful line numbers in error messages. A good way of debugging is printing to the database log, which you can do with the print function:

```
> db.eval("print('Hello, world');");
```

Stored JavaScript

MongoDB has a special collection for each database called *system.js*, which can store JavaScript variables. These variables can then be used in any of MongoDB's JavaScript contexts, including `"$where"` clauses, `db.eval` calls, and MapReduce jobs. You can add variables to *system.js* with a simple insert:

```
> db.system.js.insert({"_id" : "x", "value" : 1})
> db.system.js.insert({"_id" : "y", "value" : 2})
> db.system.js.insert({"_id" : "z", "value" : 3})
```

This defines variables `x`, `y`, and `z` in the global scope. Now, if we want to find their sum, we can execute the following:

```
> db.eval("return x+y+z;")
6
```

system.js can be used to store JavaScript code as well as simple values. This can be handy for defining your own utilities. For example, if you want to create a logging function to use in JavaScript code, you can store it in *system.js*:

```
> db.system.js.insert({"_id" : "log", "value" :
... function(msg, level) {
...     var levels = ["DEBUG", "WARN", "ERROR", "FATAL"];
...     level = level ? level : 0; // check if level is defined
...     var now = new Date();
...     print(now + " " + levels[level] + msg);
... }})
```

Now, in any JavaScript context, you can call this log function:

```
> db.eval("x = 1; log('x is '+x); x = 2; log('x is greater than 1', 1);");
```

The database log will then contain something like this:

```
Fri Jun 11 2010 11:12:39 GMT-0400 (EST) DEBUG x is 1
Fri Jun 11 2010 11:12:40 GMT-0400 (EST) WARN x is greater than 1
```

There are downsides to using stored JavaScript: it keeps portions of your code out of source control, and it can obfuscate JavaScript sent from the client.

The best reason for storing JavaScript is if you have multiple parts of your code (or code in different programs or languages) using a single JavaScript function. Keeping such functions in a central location means they do not need to be updated in multiple places if changes are required. Stored JavaScript can also be useful if your JavaScript code is long and executed frequently, because storing it once can cut down on network transfer time.

Security

Executing JavaScript is one of the few times you must be careful about security with MongoDB. If done incorrectly, server-side JavaScript is susceptible to injection attacks similar to those that occur in a relational database. Luckily, it is very easy to prevent these attacks and use JavaScript safely.

Suppose you want to print "Hello, *`username`*!" to the user. If the username is in a variable called `username`, you could write a JavaScript function such as the following:

```
> func = "function() { print('Hello, "+username+"!'); }"
```

If `username` is a user-defined variable, it could contain the string `"'); db.dropDatabase(); print('"`, which would turn the code into this:

```
> func = "function() { print('Hello, '); db.dropDatabase(); print('!'); }"
```

Now your entire database has been dropped!

To prevent this, you should use a *scope* to pass in the username. In PHP, for example, this looks like this:

```
$func = new MongoCode("function() { print('Hello, "+username+"!'); }",
... array("username" => $username));
```

Now the database will harmlessly print this:

```
Hello, '); db.dropDatabase(); print('!
```

Most drivers have a special type for sending code to the database, since code can actually be a composite of a string and a scope. A scope is just a document mapping variable names to values. This mapping becomes a local scope for the JavaScript function being executed.

> The shell does not have a code type that includes scope; you can only use strings or JavaScript functions with it.

Database References

Perhaps one of the least understood features of MongoDB is its support for *database references*, or *DBRefs*. DBRefs are like URLs: they are simply a specification for uniquely identifying a reference to document. They do not automatically load the document any more than a URL automatically loads a web page into a site with a link.

What Is a DBRef?

A DBRef is an embedded document, just like any other embedded document in MongoDB. A DBRef, however, has specific keys that must be present. A simple example looks like the following:

```
{"$ref" : collection, "$id" : id_value}
```

The DBRef references a specific `collection` and an `id_value` that we can use to find a single document by its `"_id"` within that collection. These two pieces of information allow us to use a DBRef to uniquely identify and reference any document within a MongoDB database. If we want to reference a document in a different database, DBRefs support an optional third key that we can use, `"$db"`:

```
{"$ref" : collection, "$id" : id_value, "$db" : database}
```

> DBRefs are one place in MongoDB where the order of keys in a document matters. The first key in a DBRef must be `"$ref"`, followed by `"$id"`, and then (optionally) `"$db"`.

Example Schema

Let's look at an example schema that uses DBRefs to reference documents across collections. The schema consists of two collections, *users* and *notes*. Users can create notes, which can reference users or other notes. Here are a couple of user documents, each with a unique username as its `"_id"` and a separate free-form `"display_name"`:

```
{"_id" : "mike", "display_name" : "Mike D"}
{"_id" : "kristina", "display_name" : "Kristina C"}
```

Notes are a little more complex. Each has a unique `"_id"`. Normally this `"_id"` would probably be an `ObjectId`, but we use an integer here to keep the example concise. Notes also have an `"author"`, some `"text"`, and an optional set of `"references"` to other notes or users:

```
{"_id" : 5, "author" : "mike", "text" : "MongoDB is fun!"}
{"_id" : 20, "author" : "kristina", "text" : "... and DBRefs are easy, too",
 "references": [{"$ref" : "users", "$id" : "mike"}, {"$ref" : "notes", "$id" : 5}]}
```

The second note contains some references to other documents, each stored as a DBRef. Our application code can use those DBRefs to get the documents for the "mike" user and the “MongoDB is fun!” note, both of which are associated with Kristina’s note. This dereferencing is easy to implement; we use the value of the `"$ref"` key to get the collection to query on, and we use the value of the `"$id"` key to get the `"_id"` to query for:

```
> var note = db.notes.findOne({"_id" : 20});
> note.references.forEach(function(ref) {
... printjson(db[ref.$ref].findOne({"_id" : ref.$id}));
... });
{ "_id" : "mike", "display_name" : "Mike D" }
{ "_id" : 5, "author" : "mike", "text" : "MongoDB is fun!" }
```

Driver Support for DBRefs

One thing that can be confusing about DBRefs is that not all drivers treat them as normal embedded documents. Some provide a special type for DBRefs that will be automatically translated to and from the normal document representation. This is mainly provided as a convenience for developers, because it can make working with DBRefs a little less verbose. As an example, here we represent the same note as earlier using PyMongo and its DBRef type:

```
>>> note = {"_id": 20, "author": "kristina",
...         "text": "... and DBRefs are easy, too",
...         "references": [DBRef("users", "mike"), DBRef("notes", 5)]}
```

When the note is saved, the DBRef instances will automatically be translated to the equivalent embedded documents. When the note is returned from a query, the opposite will happen, and we’ll get DBRef instances back.

Some drivers also add other helpers for working with DBRefs, like methods to handle dereferencing or even mechanisms for automatically dereferencing DBRefs as they are returned in query results. This helper functionality tends to vary from driver to driver, so for up-to-date information on what’s supported, you should reference driver-specific documentation.

When Should DBRefs Be Used?

DBRefs are not essential for representing references to other documents in MongoDB. In fact, even the previous example does some referencing using a different mechanism: the `"author"` key in each note just stores the value of the author document’s `"_id"` key. We don’t need to use a DBRef because we know that each author is a document in the *users* collection. We’ve seen another example of this type of referencing as well: the `"files_id"` key in GridFS chunk documents is just an `"_id"` reference to a file document. With this option in mind, we have a decision to make each time we need to store a reference: should we use a DBRef or just store an `"_id"`?

Storing `"_id"`s is nice because they are more compact than DBRefs and also can be a little more lightweight for developers to work with. DBRefs, on the other hand, are capable of referencing documents in any collection (or even database) without the developer having to know or remember what collection the referenced document might reside in. DBRefs also enable drivers and tools to provide some more enhanced functionality (e.g., automatic dereferencing) and could allow for more advanced support on the server side in the future.

In short, the best times to use DBRefs are when you're storing heterogeneous references to documents in different collections, like in the previous example or when you want to take advantage of some additional DBRef-specific functionality in a driver or tool. Otherwise, it's generally best to just store an `"_id"` and use that as a reference, because that representation tends to be more compact and easier to work with.

CHAPTER 8

Administration

Administering MongoDB is usually a simple task. From taking backups to setting up multinode systems with replication, most administrative tasks are quick and painless. This reflects a general philosophy of MongoDB, which is to minimize the number of dials in the system. Whenever possible, configuration is done automatically by the system rather than forcing users and administrators to tweak configuration settings. That said, there are still some administrative tasks that require manual intervention.

In this chapter we'll be switching gears from the developer perspective and discussing what you need to know to work with MongoDB from the operations or administration side. Whether you're working for a startup where you are both the engineering *and* ops teams or you're a DBA looking to work with MongoDB, this is the chapter for you. Here's the big picture:

- MongoDB is run as a normal command-line program using the `mongod` executable.
- MongoDB features a built-in admin interface and monitoring functionality that is easy to integrate with third-party monitoring packages.
- MongoDB supports basic, database-level authentication including read-only users and a separate level of authentication for admin access.
- There are several different ways of backing up a MongoDB system, the choice of which depends on a couple of key considerations.

Starting and Stopping MongoDB

In Chapter 2, we covered the basics of starting MongoDB. This chapter will go into more detail about what administrators need to know to deploy Mongo robustly in production.

Starting from the Command Line

The MongoDB server is started with the `mongod` executable. `mongod` has many configurable startup options; to view all of them, run `mongod --help` from the command line. A couple of the options are widely used and important to be aware of:

--dbpath

Specify an alternate directory to use as the data directory; the default is */data/db/* (or *C:\data\db\* on Windows). Each `mongod` process on a machine needs its own data directory, so if you are running three instances of `mongod`, you'll need three separate data directories. When `mongod` starts up, it creates a *mongod.lock* file in its data directory, which prevents any other `mongod` process from using that directory. If you attempt to start another MongoDB server using the same data directory, it will give an error:

```
"Unable to acquire lock for lockfilepath: /data/db/mongod.lock."
```

--port

Specify the port number for the server to listen on. By default, `mongod` uses port 27017, which is unlikely to be used by another process (besides other `mongod` processes). If you would like to run more than one `mongod` process, you'll need to specify different ports for each one. If you try to start `mongod` on a port that is already being used, it will give an error:

```
"Address already in use for socket: 0.0.0.0:27017"
```

--fork

Fork the server process, running MongoDB as a daemon.

--logpath

Send all output to the specified file rather than outputting on the command line. This will create the file if it does not exist, assuming you have write permissions to the directory. It will also overwrite the log file if it already exists, erasing any older log entries. If you'd like to keep old logs around, use the `--logappend` option in addition to `--logpath`.

--config

Use a configuration file for additional options not specified on the command line. See "File-Based Configuration" on page 113 for details.

So, to start the server as a daemon listening on port 5586 and sending all output to *mongodb.log*, we could run this:

```
$ ./mongod --port 5586 --fork --logpath mongodb.log
forked process: 45082
all output going to: mongodb.log
```

When you first install and start MongoDB, it is a good idea to look at the log. This might be an easy thing to miss, especially if MongoDB is being started from an init script, but the log often contains important warnings that prevent later errors from occurring. If you don't see any warnings in the MongoDB log on startup, then you are all set. However, you might see something like this:

```
$ ./mongod
Sat Apr 24 11:53:49 Mongo DB : starting : pid = 18417 port = 27017
dbpath = /data/db/ master = 0 slave = 0  32-bit
****
WARNING: This is development version of MongoDB.
         Not recommended for production.
****

** NOTE: when using MongoDB 32 bit, you are limited to about
**       2 gigabytes of data see
**       http://blog.mongodb.org/post/137788967/32-bit-limitations
**       for more

Sat Apr 24 11:53:49 db version v1.5.1-pre-, pdfile version 4.5
Sat Apr 24 11:53:49 git version: f86d93fd949777d5fbe00bf9784ec0947d6e75b9
Sat Apr 24 11:53:49 sys info: Linux ubuntu 2.6.31-15-generic ...
Sat Apr 24 11:53:49 waiting for connections on port 27017
Sat Apr 24 11:53:49 web admin interface listening on port 28017
```

The MongoDB being run here is a development version—if you download a stable release, it will not have the first warning. The second warning occurs because we are running a 32-bit build of MongoDB. We are limited to about 2GB of data when running 32 bit, because MongoDB uses a memory-mapped file-based storage engine (see Appendix C for details on MongoDB's storage engine). If you are using a stable release on a 64-bit machine, you won't get either of these messages, but it's a good idea to understand how MongoDB logs work and get used to how they look.

The log preamble won't change when you restart the database, so feel free to run MongoDB from an init script and ignore the logs, once you know what they say. However, it's a good idea to check again each time you do an install, upgrade, or recover from a crash, just to make sure MongoDB and your system are on the same page.

File-Based Configuration

MongoDB supports reading configuration information from a file. This can be useful if you have a large set of options you want to use or are automating the task of starting up MongoDB. To tell the server to get options from a configuration file, use the `-f` or `--config` flags. For example, run `mongod --config ~/.mongodb.conf` to use *~/.mongodb.conf* as a configuration file.

The options supported in a configuration file are exactly the same as those accepted at the command line. Here's an example configuration file:

```
# Start MongoDB as a daemon on port 5586

port = 5586
fork = true # daemonize it!
```

```
logpath = mongodb.log
```

This configuration file specifies the same options we used earlier when starting with regular command-line arguments. It also highlights most of the interesting aspects of MongoDB configuration files:

- Any text on a line that follows the # character is ignored as a comment.
- The syntax for specifying options is *`option = value`*, where *`option`* is case-sensitive.
- For command-line switches like `--fork`, the value `true` should be used.

Stopping MongoDB

Being able to safely stop a running MongoDB server is at least as important as being able to start one. There are a couple of different options for doing this effectively.

The most basic way to stop a running MongoDB server is to send it a SIGINT or SIGTERM signal. If the server is running as the foreground process in a terminal, this can be done by pressing Ctrl-C. Otherwise, a command like `kill` can be used to send the signal. If `mongod` has 10014 as its PID, the command would be `kill -2 10014` (SIGINT) or `kill 10014` (SIGTERM).

When `mongod` receives a SIGINT or SIGTERM, it will do a clean shutdown. This means it will wait for any currently running operations or file preallocations to finish (this could take a moment), close all open connections, flush all data to disk, and halt.

> It is important not to send a SIGKILL message (`kill -9`) to a running MongoDB server. Doing so will cause the database to shut down without going through the steps outlined earlier and could lead to corrupt data files. If this happens, the database should be repaired (see "Repair" on page 124) before being started back up.

Another way to cleanly shut down a running server is to use the `shutdown` command, `{"shutdown" : 1}`. This is an admin command and must be run on the *admin* database. The shell features a helper function to make this easier:

```
> use admin
switched to db admin
> db.shutdownServer();
server should be down...
```

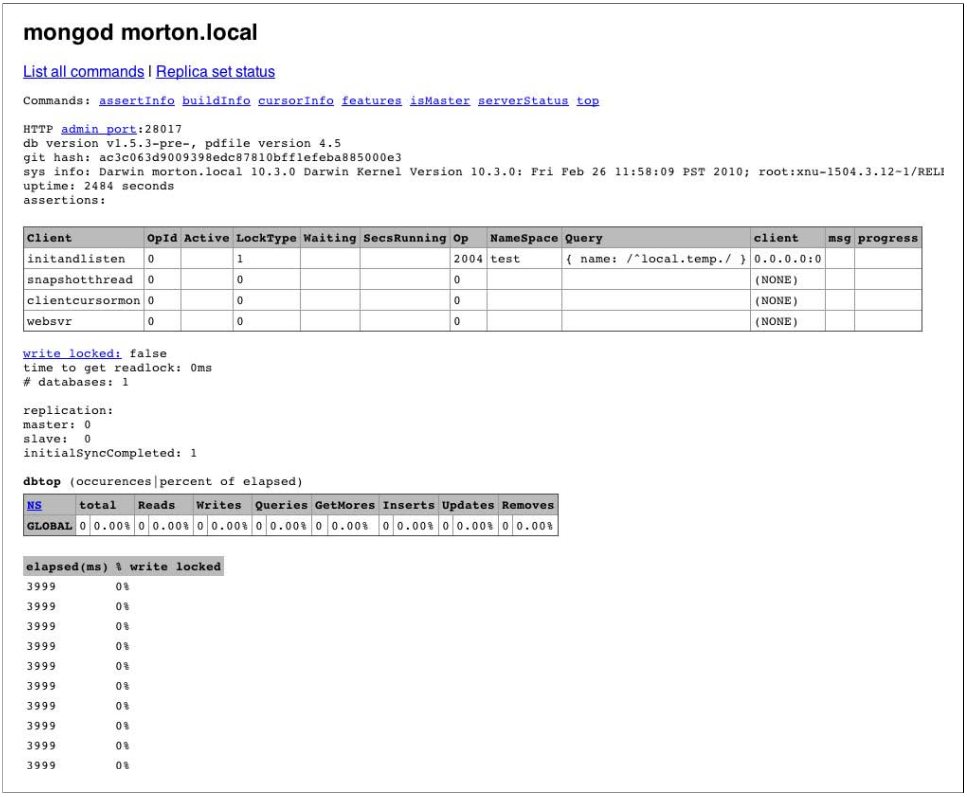
Monitoring

As the administrator of a MongoDB server, it's important to monitor the health and performance of your system. Fortunately, MongoDB has functionality that makes monitoring easy.

Using the Admin Interface

By default, starting `mongod` also starts up a (very) basic HTTP server that listens on a port 1,000 higher than the native driver port. This server provides an HTTP interface that can be used to see basic information about the MongoDB server. All of the information presented can also be seen through the shell, but the HTTP interface gives a nice, easy-to-read overview.

To see the admin interface, start the database and go to *http://localhost:28017* in a web browser. (Use 1,000 higher than the port you specified, if you used the `--port` option when starting MongoDB.) You'll see a page that looks like Figure 8-1.



mongod morton.local

List all commands | Replica set status

```
Commands: assertInfo buildInfo cursorInfo features isMaster serverStatus top

HTTP admin port:28017
db version v1.5.3-pre-, pdfile version 4.5
git hash: ac3c063d9009398edc87810bff1efeba885000e3
sys info: Darwin morton.local 10.3.0 Darwin Kernel Version 10.3.0: Fri Feb 26 11:58:09 PST 2010; root:xnu-1504.3.12~1/RELI
uptime: 2484 seconds
assertions:
```

Client	OpId	Active	LockType	Waiting	SecsRunning	Op	NameSpace	Query	client	msg	progress
initandlisten	0		1			2004	test	{ name: /^local.temp./ }	0.0.0.0:0		
snapshotthread	0		0			0			(NONE)		
clientcursormon	0		0			0			(NONE)		
websvr	0		0			0			(NONE)		

```
write locked: false
time to get readlock: 0ms
# databases: 1

replication:
master: 0
slave:  0
initialSyncCompleted: 1
```

dbtop (occurences|percent of elapsed)

NS	total		Reads		Writes		Queries		GetMores		Inserts		Updates		Removes	
GLOBAL	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%

elapsed(ms)	% write locked
3999	0%
3999	0%
3999	0%
3999	0%
3999	0%
3999	0%
3999	0%
3999	0%
3999	0%
3999	0%

Figure 8-1. The admin interface

As you can see, this interface gives access to assertion, locking, indexing, and replication information about the MongoDB server. It also gives some more general information, like the log preamble and access to a list of available database commands.

To make full use of the admin interface (e.g., to access the command list), you'll need to turn on REST support with `--rest`. You can also turn off the admin interface altogether by starting `mongod` with the `--nohttpinterface` option.

Do not attempt to connect a driver to the HTTP interface port, and do not try to connect to the native driver port via HTTP. The driver port handles only the native MongoDB wire protocol; it will not handle HTTP requests. For example, if you go to *http://localhost:27017* in a web browser, you will see:

```
You are trying to access MongoDB on the native driver port.
 For http diagnostic access, add 1000 to the port number
```

Similarly, you cannot use the native MongoDB wire protocol when connecting on the admin interface's port.

serverStatus

The most basic tool for getting statistics about a running MongoDB server is the `serverStatus` command, which has the following output (exact keys present may vary by platform/server version):

```
> db.runCommand({"serverStatus" : 1})
{
    "version" : "1.5.3",
    "uptime" : 166,
    "localTime" : "Thu Jun 10 2010 15:47:40 GMT-0400 (EDT)",
    "globalLock" : {
        "totalTime" : 165984675,
        "lockTime" : 91471425,
        "ratio" : 0.551083556358441
    },
    "mem" : {
        "bits" : 64,
        "resident" : 101,
        "virtual" : 2824,
        "supported" : true,
        "mapped" : 336
    },
    "connections" : {
        "current" : 141,
        "available" : 19859
    },
    "extra_info" : {
        "note" : "fields vary by platform"
    },
    "indexCounters" : {
        "btree" : {
            "accesses" : 1563,
            "hits" : 1563,
            "misses" : 0,
            "resets" : 0,
            "missRatio" : 0
        }
    },
    "backgroundFlushing" : {
        "flushes" : 2,
        "total_ms" : 44,
        "average_ms" : 22,
        "last_ms" : 36,
        "last_finished" : "Thu Jun 10 2010 15:46:54 GMT-0400 (EDT)"
    },
    "opcounters" : {
        "insert" : 38195,
        "query" : 8874,
        "update" : 4058,
        "delete" : 389,
        "getmore" : 888,
        "command" : 17731
    },
    "asserts" : {
        "regular" : 0,
        "warning" : 0,
        "msg" : 0,
        "user" : 5054,
        "rollovers" : 0
    },
    "ok" : true
}
```

Raw status information can also be retrieved as JSON using the MongoDB HTTP interface, at the */_status* (*http://localhost:28017/_status*) URL: this includes the output of `serverStatus`, as well as the output of some other useful commands. See "Using the Admin Interface" on page 115 for more on the admin interface.

`serverStatus` provides a detailed look at what is going on inside a MongoDB server. Information such as the current server version, uptime (in seconds), and current number of connections is easily available. Some of the other information in the `serverStatus` response might need some explaining, however.

The value for `"globalLock"` gives a quick look at how much time a global write lock has been held on the server (the times are given in microseconds). `"mem"` contains information on how much data the server has memory mapped and what the virtual and resident memory sizes are for the server process (all in megabytes). `"indexCounters"` gives information on the number of B-Tree lookups that have had to go to disk (`"misses"`) versus successful lookups from memory (`"hits"`)—if this ratio starts to increase you should consider adding more RAM, or system performance might suffer. `"background Flushing"` tells us how many background `fsyncs` have been performed and how long they've taken. One of the most important pieces of the response is the `"opcounters"` document, which contains counters for each of the major operation types. Finally, `"asserts"` counts any assertions that have occurred on the server.

All of the counters in the `serverStatus` output are tracked from the time the server was started and will eventually roll over if the counts get high enough. When a rollover occurs for any counter, all counters will roll over, and the value of `"rollovers"` in the `"asserts"` document will increment.

mongostat

Although powerful, `serverStatus` is not exactly a user-friendly mechanism for monitoring server health and performance. Fortunately, MongoDB distributions also ship with `mongostat`, which puts a friendly face on the output of `serverStatus`.

`mongostat` prints some of the most important information available from `serverStatus`. It prints a new line every second, which gives a more real-time view to the static counters we saw previously. The columns printed by `mongostat` have names like *inserts/s*, *commands/s*, *vsize*, and *% locked*, each of which corresponds exactly to data available in `serverStatus`.

Third-Party Plug-Ins

Most administrators are probably already using monitoring packages to keep track of their servers, and the presence of `serverStatus` and the */_status* URL make it pretty easy to write a MongoDB plug-in for any such tool. At the time of this writing, MongoDB plug-ins exist for at least Nagios, Munin, Ganglia, and Cacti. For an up-to-date list of third-party plug-ins, check out the MongoDB documentation on monitoring tools.

Security and Authentication

One of the first priorities for any systems administrator is to ensure their systems are secure. The best way to handle security with MongoDB is to run it in a trusted environment, ensuring that only trusted machines are able to connect to the server. That said, MongoDB supports per connection authentication, albeit with a pretty coarse-grained permissions scheme.

Authentication Basics

Each database in a MongoDB instance can have any number of users. When security is enabled, only authenticated users of a database are able to perform read or write operations on it. In the context of authentication, MongoDB treats one database as special: *admin*. A user in the *admin* database can be thought of as a superuser. After authenticating, admin users are able to read or write from *any* database and are able to perform certain admin-only commands, like `listDatabases` or `shutdown`.

Before starting the database with security turned on, it's important that at least one admin user has been added. Let's run through a quick example, starting from a shell connected to a server without security turned on:

```
> use admin
switched to db admin
> db.addUser("root", "abcd");
{
    "user" : "root",
    "readOnly" : false,
    "pwd" : "1a0f1c3c3aa1d592f490a2addc559383"
}
> use test
switched to db test
> db.addUser("test_user", "efgh");
{
    "user" : "test_user",
    "readOnly" : false,
    "pwd" : "6076b96fc3fe6002c810268702646eec"
}
> db.addUser("read_only", "ijkl", true);
{
    "user" : "read_only",
    "readOnly" : true,
    "pwd" : "f497e180c9dc0655292fee5893c162f1"
}
```

Here we've added an admin user, root, and two users on the *test* database. One of those users, read_only, has read permissions only and cannot write to the database. From the shell, a read-only user is created by passing `true` as the third argument to `addUser`. To call `addUser`, you must have write permissions for the database in question; in this case we can call `addUser` on any database because we have not enabled security yet.

> The `addUser` method is useful for more than just adding new users: it can be used to change a user's password or read-only status. Just call `addUser` with the username and a new password or read-only setting for the user.

Now let's restart the server, this time adding the `--auth` command-line option to enable security. After enabling security, we can reconnect from the shell and try it:

```
> use test
switched to db test
> db.test.find();
error: { "$err" : "unauthorized for db [test] lock type: -1 " }
> db.auth("read_only", "ijkl");
1
> db.test.find();
{ "_id" : ObjectId("4bb007f53e8424663ea6848a"), "x" : 1 }
> db.test.insert({"x" : 2});
unauthorized
> db.auth("test_user", "efgh");
1
> db.test.insert({"x": 2});
> db.test.find();
{ "_id" : ObjectId("4bb007f53e8424663ea6848a"), "x" : 1 }
{ "_id" : ObjectId("4bb0088cbe17157d7b9cac07"), "x" : 2 }
> show dbs
assert: assert failed : listDatabases failed:{
    "assertion" : "unauthorized for db [admin] lock type: 1
",
    "errmsg" : "db assertion failure",
    "ok" : 0
}
> use admin
switched to db admin
> db.auth("root", "abcd");
1
> show dbs
admin
local
test
```

When we first connect, we are unable to perform any operations (read or write) on the *test* database. After authenticating as the read_only user, however, we are able to perform a simple `find`. When we try to insert data, we are again met with a failure because of the lack of authorization. test_user, which was not created as read-only, is able to insert data normally. As a nonadmin user, though, test_user is not able to list all of the available databases using the `show dbs` helper. The final step is to authenticate as an admin user, root, who is able to perform operations of any kind on any particular database.

How Authentication Works

Users of a given database are stored as documents in its *system.users* collection. The structure of a user document is `{"user" : `*`username`*`, "readOnly" : `*`true`*`, "pwd" : `*`password hash`*`}`. The *`password hash`* is a hash based on the *`username`* and password chosen.

Knowing where and how user information is stored makes performing some common administration tasks trivial. For example, to remove a user, simply remove the user document from the *system.users* collection:

```
> db.auth("test_user", "efgh");
1
> db.system.users.remove({"user" : "test_user"});
> db.auth("test_user", "efgh");
0
```

When a user authenticates, the server keeps track of that authentication by tying it to the connection used for the `authenticate` command. This means that if a driver or tool is employing connection pooling or fails over to another node, any authenticated users will need to reauthenticate on any new connections. Some drivers may be capable of handling this transparently, but if not, it will need to be done manually. If that is the case, then it might be best to avoid using `--auth` altogether (again, by deploying MongoDB in a trusted environment and handling authentication on the client side).

Other Security Considerations

There are a couple of options besides authentication that should be considered when locking down a MongoDB instance. First, even when using authentication, the MongoDB wire protocol is not encrypted. If that is a requirement, consider using SSH tunneling or another similar mechanism to encrypt traffic between clients and the MongoDB server.

We suggest always running your MongoDB servers behind a firewall or on a network accessible only through your application servers. If you do have MongoDB on a machine accessible to the outside world, however, it is recommended that you start it with the `--bindip` option, which allows you to specify a local IP address that `mongod` will be bound to. For instance, to only allow connections from an application server running on the same machine, you could run `mongod --bindip localhost`.

As documented in the section "Using the Admin Interface" on page 115, by default MongoDB starts up a very simple HTTP server that allows you to see information about current operations, locking, and replication from your browser. If you don't want this information exposed, you can turn off the admin interface by using the `--nohttpinterface` option.

Finally, you can entirely disallow server-side JavaScript execution by starting the database with `--noscripting`.

Backup and Repair

Taking backups is an important administrative task with any data storage system. Often, doing backups properly can be tricky, and the only thing worse than not taking backups at all is taking them incorrectly. Luckily, MongoDB has several different options that make taking backups a painless process.

Data File Backup

MongoDB stores all of its data in a *data directory*. By default, this directory is */data/db/* (or *C:\data\db\* on Windows). The directory to use as the data directory is configurable through the `--dbpath` option when starting MongoDB. Regardless of where the data directory is, its contents form a complete representation of the data stored in MongoDB. This suggests that making a backup of MongoDB is as simple as creating a copy of all of the files in the data directory.

It is not safe to create a copy of the data directory while MongoDB is running unless the server has done a full `fsync` and is not allowing writes. Such a backup will likely turn out to be corrupt and need repairing (see the section "Repair" on page 124).

Because it is not safe in general to copy the data directory while MongoDB is running, one option for taking a backup is to shut down the MongoDB server and then copy the data directory. Assuming the server is shut down safely (see the section "Starting and Stopping MongoDB" on page 111), the data directory will represent a safe snapshot of the data stored when it was shut down. That directory can be copied as a backup before restarting the server.

Although shutting down the server and copying the data directory is an effective and safe method of taking backups, it is not ideal. In the remainder of this chapter, we'll look at techniques for backing up MongoDB without requiring any downtime.

mongodump and mongorestore

One method for backing up a running instance of MongoDB is to use the `mongodump` utility that is included with all MongoDB distributions. `mongodump` works by querying against a running MongoDB server and writing all of the documents it contains to disk. Because `mongodump` is just a regular client, it can be run against a live instance of MongoDB, even one handling other requests and performing writes.

> Because `mongodump` operates using the normal MongoDB query mechanism, the backups it produces are not necessarily point-in-time snapshots of the server's data. This is especially evident if the server is actively handling writes during the course of the backup.
>
> Another consequence of the fact that `mongodump` acts through the normal query mechanism is that it can cause some performance degradation for other clients throughout the duration of the backup.

Like most of the command-line tools included with MongoDB, we can see the options available for `mongodump` by running with the `--help` option:

```
$ ./mongodump --help
options:
  --help                     produce help message
  -v [ --verbose ]           be more verbose (include multiple times for more
                             verbosity e.g. -vvvvv)
  -h [ --host ] arg          mongo host to connect to ("left,right" for pairs)
  -d [ --db ] arg            database to use
  -c [ --collection ] arg    collection to use (some commands)
  -u [ --username ] arg      username
  -p [ --password ] arg      password
  --dbpath arg               directly access mongod data files in the given path,
                             instead of connecting to a mongod instance - needs
                             to lock the data directory, so cannot be used if a
                             mongod is currently accessing the same path
  --directoryperdb           if dbpath specified, each db is in a separate
                             directory
  -o [ --out ] arg (=dump) output directory
```

Along with `mongodump`, MongoDB distributions include a corresponding tool for restoring data from a backup, `mongorestore`. `mongorestore` takes the output from running `mongodump` and inserts the backed-up data into a running instance of MongoDB. The following example session shows a hot backup of the database *test* to the *backup* directory, followed by a separate call to `mongorestore`:

```
$ ./mongodump -d test -o backup
connected to: 127.0.0.1
DATABASE: test   to     backup/test
        test.x to backup/test/x.bson
                1 objects
$ ./mongorestore -d foo --drop backup/test/
connected to: 127.0.0.1
backup/test/x.bson
          going into namespace [foo.x]
          dropping
          1 objects
```

In the previous example, we use `-d` to specify a database to restore to, in this case *foo*. This option allows us to restore a backup to a database with a different name than the original. We also use the `--drop` option, which will drop the collection (if it exists) before restoring data to it. Otherwise, the data will be merged into any existing collection, possibly overwriting some documents. Again, for a complete list of options, run `mongorestore --help`.

fsync and Lock

Although `mongodump` and `mongorestore` allow us to take backups without shutting down the MongoDB server, we lose the ability to get a point-in-time view of the data. MongoDB's `fsync` command allows us to copy the data directory of a running MongoDB server without risking any corruption.

The `fsync` command will force the MongoDB server to flush all pending writes to disk. It will also, optionally, hold a lock preventing any further writes to the database until the server is unlocked. This write lock is what allows the `fsync` command to be useful for backups. Here is an example of how to run the command from the shell, forcing an `fsync` and acquiring a write lock:

```
> use admin
switched to db admin
> db.runCommand({"fsync" : 1, "lock" : 1});
{
        "info" : "now locked against writes, use db.$cmd.sys.unlock.findOne() to unlock",
        "ok" : 1
}
```

At this point, the data directory represents a consistent, point-in-time snapshot of our data. Because the server is locked for writes, we can safely make a copy of the data directory to use as a backup. This is especially useful when running on a snapshotting filesystem, like LVM[*] or EBS[†], where taking a snapshot of the data directory is a fast operation.

After performing the backup, we need to unlock the database again:

```
> db.$cmd.sys.unlock.findOne();
{ "ok" : 1, "info" : "unlock requested" }
> db.currentOp();
{ "inprog" : [ ] }
```

Here we run the `currentOp` command to ensure that the lock has been released. (It may take a moment after the unlock is first requested.)

The `fsync` command allows us to take very flexible backups, without shutting down the server or sacrificing the point-in-time nature of the backup. The price we've paid, however, is a momentary block against write operations. The only way to have a point-in-time snapshot without any downtime for reads *or* writes is to backup from a slave.

Slave Backups

Although the options discussed earlier provide a wide range of flexibility in terms of backups, none is as flexible as backing up from a slave server. When running MongoDB with replication (see Chapter 9), any of the previously mentioned backup techniques can be applied to a slave server rather than the master. The slave will always have a copy of the data that is nearly in sync with the master. Because we're not depending on the performance of the slave or its availability for reads or writes, we are free to use any of the three options above: shutting down, the dump and restore tools, or the `fsync` command. Backing up from a slave is the recommended way to handle data backups with MongoDB.

Repair

We take backups so that when a disaster occurs, which could be anything from a power failure to an elephant on the loose in the data center, our data is safe. There will unfortunately always be cases when a server with no backups (or slaves to failover to) fails. In the case of a power failure or a software crash, the disk will be fine when the machine comes back up. Because of the way MongoDB stores data, however, we are not guaranteed that the data on the disk is OK to use: corruption might have occurred (see Appendix C for more on MongoDB's storage engine). Luckily, MongoDB has built-in repairing functionality to attempt to recover corrupt data files.

---

[*] A logical volume manager for Linux

[†] Amazon's Elastic Block Store

A repair should be run after any unclean shutdown of MongoDB. If an unclean shutdown has occurred, you'll be greeted with the following warning when trying to start the server back up:

```
**************
old lock file: /data/db/mongod.lock. probably means unclean shutdown
recommend removing file and running --repair
see: http://dochub.mongodb.org/core/repair for more information
*************
```

The easiest way to repair all of the databases for a given server is to start up a server with `--repair`: `mongod --repair`. The underlying process of repairing a database is actually pretty easy to understand: all of the documents in the database are exported and then immediately imported, ignoring any that are invalid. After that is complete, all indexes are rebuilt. Understanding this mechanism explains some of the properties of repair. It can take a long time for large data sets, because all of the data is validated and all indexes are rebuilt. Repairing can also leave a database with fewer documents than it had before the corruption originally occurred, because any corrupt documents are simply ignored.

> Repairing a database will also perform a compaction. Any extra free space (which might exist after dropping large collections or removing large number of documents, for example) will be reclaimed after a repair.

To repair a single database on a running server, you can use the `repairDatabase` method from the shell. If we wanted to repair the database *test*, we would do the following:

```
> use test
switched to db test
> db.repairDatabase()
{ "ok" : 1 }
```

To do the same from a driver rather than the shell, issue the `repairDatabase` command, `{"repairDatabase" : 1}`.

Repairing to eliminate corruption should be treated as a last resort. The most effective way to manage data is to always stop the MongoDB server cleanly, use replication for failover, and take regular backups.

CHAPTER 9
Replication

Perhaps the most important job of any MongoDB administrator is making sure that replication is set up and functioning correctly. Use of MongoDB's replication functionality is always recommended in production settings, especially since the current storage engine does not provide single-server durability (see Appendix C for details). Replicas can be used purely for failover and data integrity, or they can be used in more advanced ways, such as for scaling out reads, taking hot backups, or as a data source for offline batch processing. In this chapter, we'll cover everything you need to know about replication.

Master-Slave Replication

Master-slave replication is the most general replication mode supported by MongoDB. This mode is very flexible and can be used for backup, failover, read scaling, and more (see Figures 9-1 and 9-2).

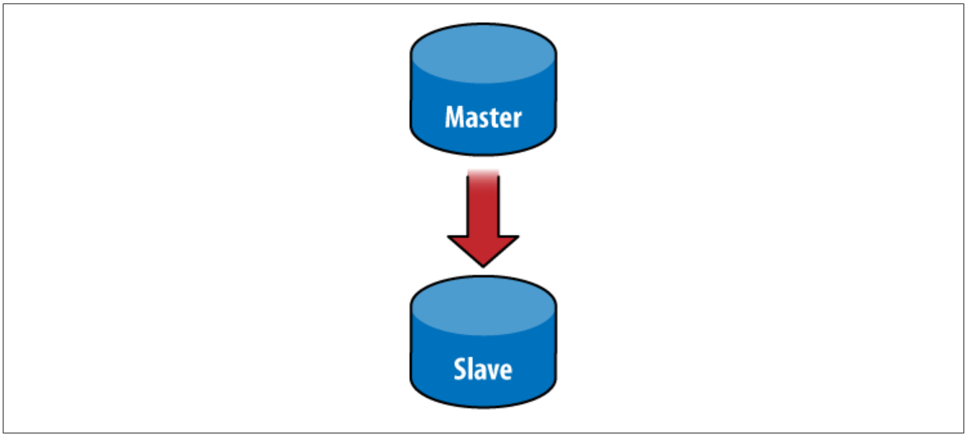


Figure 9-1. A master with one slave

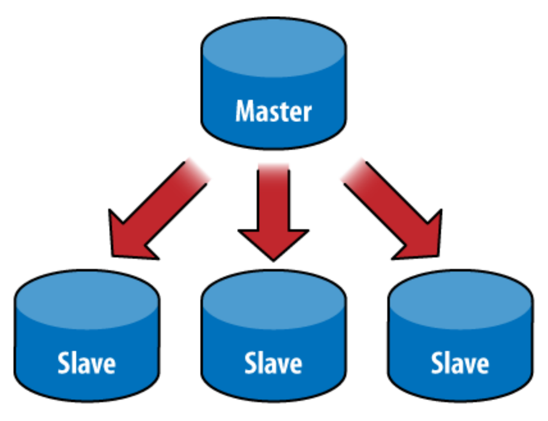


Figure 9-2. A master with three slaves

The basic setup is to start a master node and one or more slave nodes, each of which knows the address of the master. To start the master, run `mongod --master`. To start a slave, run `mongod --slave --source` ***`master_address`***, where ***`master_address`*** is the address of the master node that was just started.

It is simple to try this on a single machine, although in production you would use multiple servers. First, create a directory for the master to store data in and choose a port (10000):

```
$ mkdir -p ~/dbs/master
$ ./mongod --dbpath ~/dbs/master --port 10000 --master
```

Now set up the slave, choosing a different data directory and port. For a slave, you also need to tell it who its master is with the `--source` option:

```
$ mkdir -p ~/dbs/slave
$ ./mongod --dbpath ~/dbs/slave --port 10001 --slave --source localhost:10000
```

All slaves must be replicated from a master node. There is currently no mechanism for replicating from a slave (*daisy chaining*), because slaves do not keep their own oplog (see "How It Works" on page 138 for more on the oplog).

There is no explicit limit on the number of slaves in a cluster, but having a thousand slaves querying a single master will likely overwhelm the master node. In practice, clusters with less than a dozen slaves tend to work well.

Options

There are a few other useful options in conjunction with master-slave replication:

--only
: Use on a slave node to specify only a single database to replicate. (The default is to replicate all databases.)

--slavedelay
: Use on a slave node to add a delay (in seconds) to be used when applying operations from the master. This makes it easy to set up delayed slaves, which can be useful in case a user accidentally deletes important documents or inserts bad data. Either of those operations will be replicated to all slaves. By delaying the application of operations, you have a window in which recovery from the bad operation is possible.

--fastsync
: Start a slave from a snapshot of the master node. This option allows a slave to bootstrap much faster than doing a full sync, if its data directory is initialized with a snapshot of the master's data.

--autoresync
: Automatically perform a full resync if this slave gets out of sync with the master (see "How It Works" on page 138).

--oplogSize
: Size (in megabytes) for the master's oplog (see "How It Works" on page 138 for more on the oplog).

Adding and Removing Sources

You can specify a master by starting your slave with the `--source` option, but you can also configure its source(s) from the shell.

Suppose we have a master at `localhost:27017`. We could start a slave without any source and then add the master to the *sources* collection:

```
$ ./mongod --slave --dbpath ~/dbs/slave --port 27018
```

Now we can add `localhost:27017` as a source for our slave by starting the shell and running the following:

```
> use local
> db.sources.insert({"host" : "localhost:27017"})
```

If you watch the slave's log, you can see it sync to `localhost:27017`.

If we do a `find` on the *sources* collection immediately after inserting the source, it will show us the document we inserted:

```
> db.sources.find()
{
    "_id" : ObjectId("4c1650c2d26b84cc1a31781f"),
    "host" : "localhost:27017"
}
```

Once the slave's log shows that it has finished syncing, the document will be updated to reflect this:

```
> db.sources.find()
{
    "_id" : ObjectId("4c1650c2d26b84cc1a31781f"),
    "host" : "localhost:27017",
    "source" : "main",
    "syncedTo" : {
        "t" : 1276530906000,
        "i" : 1
    },
    "localLogTs" : {
        "t" : 0,
        "i" : 0
    },
    "dbsNextPass" : {
        "test_db" : true
    }
}
```

Now, suppose we are going into production and we want to change the slave's configuration such that it slaves off of `prod.example.com`. We can change the source for the slave using `insert` and `remove`:

```
> db.sources.insert({"host" : "prod.example.com:27017"})
> db.sources.remove({"host" : "localhost:27017"})
```

As you can see, *sources* can be manipulated like a normal collection and provides a great deal of flexibility for managing slaves.

If you slave off of two different masters with the same collections, MongoDB will attempt to merge them, but correctly doing so is not guaranteed. If you are using a single slave with multiple different masters, it is best to make sure the masters use different namespaces.

Replica Sets

A *replica set* is basically a master-slave cluster with automatic failover. The biggest difference between a master-slave cluster and a replica set is that a replica set does not have a single master: one is elected by the cluster and may change to another node if the current master goes down. However, they look very similar: a replica set always has a single master node (called a *primary*) and one or more slaves (called *secondaries*). See Figures 9-3, 9-4, and 9-5.

The nice thing about replica sets is how automatic everything is. First, the set itself does a lot of the administration for you, promoting slaves automatically and making sure you won't run into inconsistencies. For a developer, they are easy to use: you specify a few servers in a set, and the driver will automatically figure out all of the servers in the set and handle failover if the current master dies.

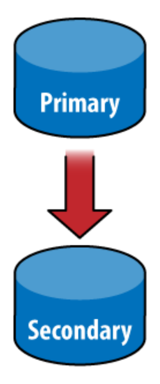

Primary
Secondary

Figure 9-3. A replica set with two members

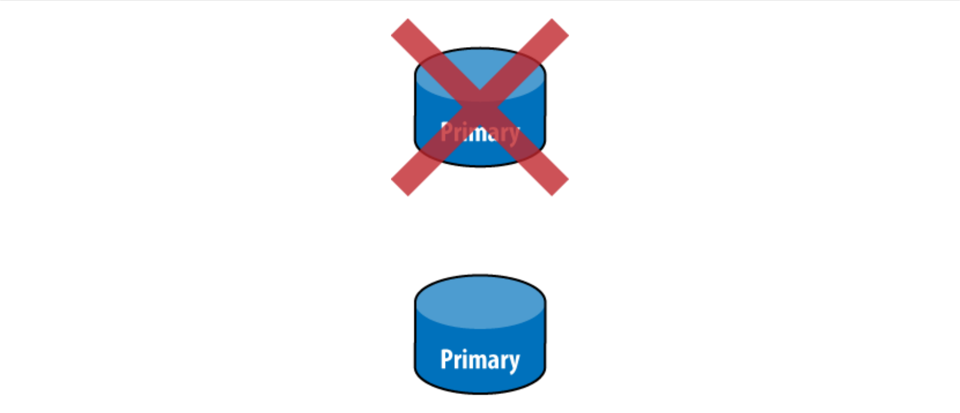

Primary
Primary

Figure 9-4. When the primary server goes down, the secondary server will become master

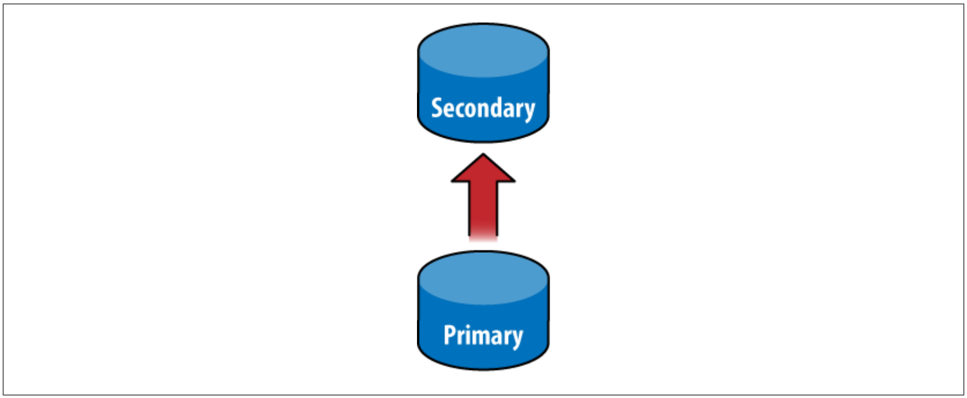

Secondary
Primary

Figure 9-5. If the original primary comes back up, it will begin slaving off of the new primary

Initializing a Set

Setting up a replica set is a little more involved than setting up a master-slave cluster. We'll just start out by setting up the smallest set possible: two servers.

> You cannot specify `localhost` addresses as members, so you need to figure out what the hostname is of your machine. On *NIX, this can be done with the following:
>
> ```
> $ cat /etc/hostname
> morton
> ```

First, we create our data directories and choose ports for each server:

```
$ mkdir -p ~/dbs/node1 ~/dbs/node2
```

We have one more decision to make before we start up the servers: we must choose a *name* for this replica set. This name makes it easy to refer to the set as a whole and distinguish between sets. We'll call our replica set `"blort"`.

Now we can actually start up the servers. The only new option is `--replSet`, which lets the server know that it's a member of the `replSet` `"blort"` that contains another member at `morton:10002` (which hasn't been started yet):

```
$ ./mongod --dbpath ~/dbs/node1 --port 10001 --replSet blort/morton:10002
```

We start up the other server in the same way:

```
$ ./mongod --dbpath ~/dbs/node2 --port 10002 --replSet blort/morton:10001
```

If we wanted to add a third server, we could do so with either of these commands:

```
$ ./mongod --dbpath ~/dbs/node3 --port 10003 --replSet blort/morton:10001
$ ./mongod --dbpath ~/dbs/node3 --port 10003 --replSet blort/morton:10001,morton:10002
```

One of the nice things about replica sets is that they are self-detecting: you can specify a single server in the set, and MongoDB will figure out and connect to the rest of the nodes automatically.

Once you have a few servers up, you'll notice that the server logs are complaining about the replica set not being initialized. This is because there's one more step: initializing the set in the shell.

Connect to one of the servers (we use `morton:10001` in the following example) with the shell. Initializing the set is a database command that has to be run only once:

```
$ ./mongo morton:10001/admin
MongoDB shell version: 1.5.3
connecting to localhost:10001/admin
type "help" for help
> db.runCommand({"replSetInitiate" : {
... "_id" : "blort",
... "members" : [
...     {
```

```
...        "_id" : 1,
...        "host" : "morton:10001"
...    },
...    {
...        "_id" : 2,
...        "host" : "morton:10002"
...    }
... ]}})
{
        "info" : "Config now saved locally.  Should come online in about a minute.",
        "ok" : true
}
```

The initialization document is a bit complicated, but going through it key by key should make sense:

`"_id" : "blort"`
: The name of this set.

`"members" : [...]`
: A list of servers in the set. You can add more later. Each server document has (at least) two keys:

  `"_id" :` *`N`*
  : Each server needs a unique ID.

  `"host" :` *`hostname`*
  : This is the key that actually specifies the host.

Now you should see some log messages about which server is being elected primary.

If we connect to the other server and do a `find` on the *local.system.replset* namespace, you can see that the configuration has been propagated to the other server in the set.

> At the time of this writing, replica sets are still under development and have not yet been released in a production version of MongoDB. As such, the information here is subject to change. For the most up-to-date documentation on replica sets, see the MongoDB wiki.

Nodes in a Replica Set

At any point in time, one node in the cluster is *primary*, and the rest are *secondary*. The primary node is essentially the master, the difference being that which node is designated as primary can vary over time.

There are several different types of nodes that can coexist in a replica set:

standard
: This is a regular replica set node. It stores a full copy of the data being replicated, takes part in voting when a new primary is being elected, and is capable of becoming the primary node in the set.

passive
: Passive nodes store a full copy of the data and participate in voting but will never become the primary node for the set.

arbiter
: An arbiter node participates only in voting; it does not receive any of the data being replicated and cannot become the primary node.

The difference between a standard node and a passive node is actually more of a sliding scale; each *participating node* (nonarbiter) has a *priority* setting. A node with priority `0` is passive and will never be selected as primary. Nodes with nonzero priority will be selected in order of decreasing priority, using freshness of data to break ties between nodes with the same priority. So, in a set with two priority `1` nodes and a priority `0.5` node, the third node will be elected primary only if neither of the priority `1` nodes are available.

Standard and passive nodes can be configured as part of a node's description, using the `priority` key:

```
> members.push({
... "_id" : 3,
... "host" : "morton:10003",
... "priority" : 40
... });
```

The default priority is 1, and priorities must be between 0 and 1000 (inclusive).

Arbiters are specified using the `"arbiterOnly"` key.

```
> members.push({
... "_id" : 4,
... "host" : "morton:10004",
... "arbiterOnly" : true
... });
```

There is more information about arbiters in the next section.

Secondary nodes will pull from the primary node's oplog and apply operations, just like a slave in a master-slave system. A secondary node will also write the operation to its own local oplog, however, so that it is capable of becoming the primary. Operations in the oplog also include a monotonically increasing ordinal. This ordinal is used to determine how up-to-date the data is on any node in the cluster.

Failover and Primary Election

If the current primary fails, the rest of the nodes in the set will attempt to elect a new primary node. This election process will be initiated by any node that cannot reach the primary. The new primary must be elected by a *majority* of the nodes in the set. Arbiter nodes participate in voting as well and are useful for breaking ties (e.g., when the participating nodes are split into two halves separated by a network partition). The new primary will be the node with the highest priority, using freshness of data to break ties between nodes with the same priority (see Figures 9-6, 9-7, and 9-8).

The primary node uses a heartbeat to track how many nodes in the cluster are visible to it. If this falls below a majority, the primary will automatically fall back to secondary status. This prevents the primary from continuing to function as such when it is separated from the cluster by a network partition.

Whenever the primary changes, the data on the new primary is assumed to be the most up-to-date data in the system. Any operations that have been applied on any other nodes (i.e., the former primary node) will be rolled back, even if the former primary comes back online. To accomplish this rollback, all nodes go through a resync process when connecting to a new primary. They look through their oplog for operations that have not been applied on the primary and query the new primary to get an up-to-date copy of any documents affected by such operations. Nodes that are currently in the process of resyncing are said to be *recovering* and will not be eligible for primary election until the process is complete.

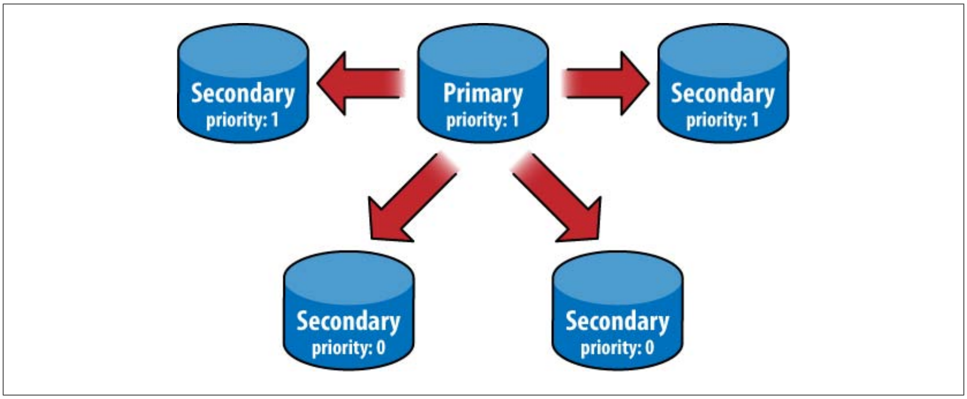


Figure 9-6. A replica set can have several servers of different priority levels

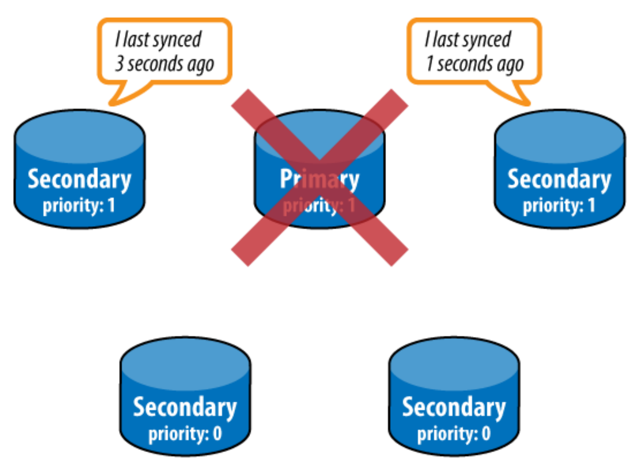

I last synced 3 seconds ago
I last synced 1 seconds ago
Secondary
priority: 1
Primary
priority: 1
Secondary
priority: 1
Secondary
priority: 0
Secondary
priority: 0

Figure 9-7. If the primary goes down, the highest-priority servers will compare how up-to-date they are

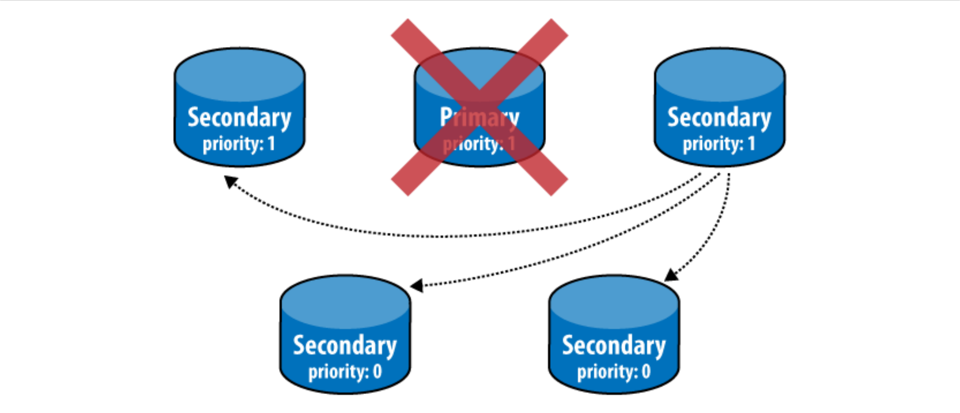

Secondary
priority: 1
Primary
priority: 1
Secondary
priority: 1
Secondary
priority: 0
Secondary
priority: 0

Figure 9-8. The highest-priority most-up-to-date server will become the new primary

Performing Operations on a Slave

The primary purpose and most common use case of a MongoDB slave is to function as failover mechanism in the case of data loss or downtime on the master node. There are other valid use cases for a MongoDB slave, however. A slave can be used as a source for taking backups (see Chapter 8). It can also be used for scaling out reads or for performing data processing jobs on.

Read Scaling

One way to scale reads with MongoDB is to issue queries against slave nodes. By issuing queries on slaves, the workload for the master is reduced. In general, this is a good approach to scaling when your workload is read heavy—if you have a more write-intensive workload, see Chapter 10 to learn how to scale with autosharding.

> One important note about using slaves to scale reads in MongoDB is that replication is asynchronous. This means that when data is inserted or updated on the master, the data on the slave will be out-of-date momentarily. This is important to consider if you are serving some requests using queries to slaves.

Scaling out reads with slaves is easy: just set up master-slave replication like usual, and make connections directly to the slave servers to handle queries. The only trick is that there is a special query option to tell a slave server that it is allowed to handle a query. (By default, queries will not be executed on a slave.) This option is called `slaveOkay`, and all MongoDB drivers provide a mechanism for setting it. Some drivers also provide facilities to automate the process of distributing queries to slaves—this varies on a per-driver basis, however.

Using Slaves for Data Processing

Another interesting technique is to use slaves as a mechanism for offloading intensive processing or aggregation to avoid degrading performance on the master. To do this, start a normal slave, but with the addition of the `--master` command-line argument. Starting with both `--slave` and `--master` may seem like a bit of a paradox. What it means, however, is that you'll be able to write to the slave, query on it like usual, and basically treat it like you would a normal MongoDB master node. In addition, the slave will continue to replicate data from the actual master. This way, you can perform blocking operations on the slave without ever affecting the performance of the master node.

> When using this technique, you should be sure never to write to any database on the slave that is being replicated from the master. The slave will not revert any such writes in order to properly mirror the master.
>
> The slave should also not have any of the databases that are being replicated when it first starts up. If it does, those databases will not ever be fully synced but will just update with new operations.

How It Works

At a very high level, a replicated MongoDB setup always consists of at least two servers, or nodes. One node is the master and is responsible for handling normal client requests. The other node(s) is a slave and is responsible for mirroring the data stored on the master. The master keeps a record of all operations that have been performed on it. The slave periodically polls the master for any new operations and then performs them on its copy of the data. By performing all of the same operations that have been performed on the master node, the slave keeps its copy of the data up-to-date with the master's.

The Oplog

The record of operations kept by the master is called the *oplog*, short for operation log. The oplog is stored in a special database called *local*, in the *oplog.$main* collection. Each document in the oplog represents a single operation performed on the master server. The documents contain several keys, including the following:

ts

Timestamp for the operation. The *timestamp* type is an internal type used to track when operations are performed. It is composed of a 4-byte timestamp and a 4-byte incrementing counter.

op

Type of operation performed as a 1-byte code (e.g., "i" for an insert).

ns

Namespace (collection name) where the operation was performed.

o

Document further specifying the operation to perform. For an insert, this would be the document to insert.

One important note about the oplog is that it stores only operations that change the state of the database. A query, for example, would not be stored in the oplog. This makes sense because the oplog is intended only as a mechanism for keeping the data on slaves in sync with the master.

The operations stored in the oplog are also not exactly those that were performed on the master server itself. The operations are transformed before being stored such that they are idempotent. This means that operations can be applied multiple times on a slave with no ill effects, so long as the operations are applied in the correct order (e.g., an incrementing update, using `"$inc"`, will be transformed to a `"$set"` operation).

A final important note about the oplog is that it is stored in a capped collection (see "Capped Collections" on page 97). As new operations are stored in the oplog, they will automatically replace the oldest operations. This guarantees that the oplog does not grow beyond a preset bound. That bound is configurable using the `--oplogSize` option when starting the server, which allows you to specify the size of the oplog in megabytes. By default, 64-bit instances will use 5 percent of available free space for the oplog. This space will be allocated in the *local* database and will be preallocated when the server starts.

Syncing

When a slave first starts up, it will do a full sync of the data on the master node. The slave will copy every document from the master node, which is obviously an expensive operation. After the initial sync is complete, the slave will begin querying the master's oplog and applying operations in order to stay up-to-date.

If the application of operations on the slave gets too far behind the actual operations being performed on the master, the slave will fall *out of sync*. An out-of-sync slave is unable to continue to apply operations to catch up to the master, because every operation in the master's oplog is too "new." This could happen if the slave has had downtime or is busy handling reads. It can also happen following a full sync, if the sync takes long enough that the oplog has rolled over by the time it is finished.

When a slave gets out of sync, replication will halt, and the slave will need to be fully resynced from the master. This resync can be performed manually by running the command `{"resync" : 1}` on the slave's *admin* database or automatically by starting the slave with the `--autoresync` option. Either way, doing a resync is a very expensive operation, and it's a situation that is best avoided by choosing a large enough oplog size.

To avoid out of sync slaves, it's important to have a large oplog so that the master can store a long history of operations. A larger oplog will obviously use up more disk space, but in general this is a good trade-off to make (hence the default oplog size of 5 percent of free space). For more information on sizing the oplog, see "Administration" on page 141.

Replication State and the Local Database

The *local database* is used for all internal replication state, on both the master and the slave. The local database's name is ***`local`***, and its contents will never be replicated. Thus, the local database is guaranteed to be local to a single MongoDB server.

Use of the local database isn't limited to MongoDB internals. If you have documents that you don't want to replicate, just store them in a collection in the local database.

Other replication state stored on the master includes a list of its slaves. (Slaves perform a handshake using the `handshake` command when they connect to the master.) This list is stored in the *slaves* collection:

```
> db.slaves.find()
{ "_id" : ObjectId("4c1287178e00e93d1858567c"), "host" : "127.0.0.1",
  "ns" : "local.oplog.$main", "syncedTo" : { "t" : 1276282710000, "i" : 1 } }
{ "_id" : ObjectId("4c128730e6e5c3096f40e0de"), "host" : "127.0.0.1",
  "ns" : "local.oplog.$main", "syncedTo" : { "t" : 1276282710000, "i" : 1 } }
```

Slaves also store state in the local database. They store a unique slave identifier in the `me` collection, and a list of *sources*, or nodes, that they are slaving from, in the `sources` collection:

```
> db.sources.find()
{ "_id" : ObjectId("4c1287178e00e93d1858567b"), "host" : "localhost:27017",
  "source" : "main", "syncedTo" : { "t" : 1276283096000, "i" : 1 },
  "localLogTs" : { "t" : 0, "i" : 0 } }
```

Both the master and slave keep track of how up-to-date a slave is, using the timestamp stored in `"syncedTo"`. Each time the slave queries the oplog for new operations, it uses `"syncedTo"` to specify which new operations it needs to apply or to find out if it is out of sync.

Blocking for Replication

MongoDB's `getLastError` command allows developers to enforce guarantees about how up-to-date replication is by using the optional `"w"` parameter. Here we run a `getLastError` that will block until at least *`N`* servers have replicated the last write operation:

```
> db.runCommand({getLastError: 1, w: N});
```

If *`N`* is not present or is less than two, the command will return immediately. If *`N`* is two, the master won't respond to the command until at least one slave has replicated the last operation. (The master itself is included toward *`N`*.) The master uses the `"syncedTo"` information stored in *local.slaves* to track how up-to-date each slave is.

When specifying `"w"`, `getLastError` takes an additional parameter, `"wtimeout"`, which is a timeout in milliseconds. This allows the `getLastError` to time out and return an error before the last operation has replicated to *`N`* servers. (By default the command has no timeout.)

Blocking for replication will cause write operations to slow down significantly, particularly for large values of `"w"`. In practice, setting `"w"` to two or three for important operations will yield a good combination of efficiency and safety.

Administration

In this section, we'll introduce some administration concepts that are specific to replication.

Diagnostics

MongoDB includes a couple of useful administrative helpers for inspecting the status of replication. When connected to the master, use the `db.printReplicationInfo` function:

```
> db.printReplicationInfo();
    configured oplog size:   10.48576MB
    log length start to end: 34secs (0.01hrs)
    oplog first event time:  Tue Mar 30 2010 16:42:57 GMT-0400 (EDT)
    oplog last event time:   Tue Mar 30 2010 16:43:31 GMT-0400 (EDT)
    now:                     Tue Mar 30 2010 16:43:37 GMT-0400 (EDT)
```

This gives information about the size of the oplog and the date ranges of operations contained in the oplog. In this example, the oplog is about 10MB and is only able to fit about 30 seconds of operations. This is almost certainly a case where we should increase the size of the oplog (see the next section). We want the log length to be *at least* as long as the time it takes to do a full resync—that way, we don't run into a case where a slave is already out of sync by the time its initial sync (or resync) is finished.

> The log length is computed by taking the time difference between the first and last operation in the oplog. If the server has just started, then the first operation will be relatively recent. In that case, the log length will be small, even though the oplog probably still has free space available. The log length is a more useful metric for servers that have been operational long enough for the oplog to "roll over."

We can also get some information when connected to the slave, using the `db.printSlaveReplicationInfo` function:

```
> db.printSlaveReplicationInfo();
    source:   localhost:27017
    syncedTo: Tue Mar 30 2010 16:44:01 GMT-0400 (EDT)
    = 12secs ago (0hrs)
```

This will show a list of sources for the slave, each with information about how far behind the master it is. In this case, we are only 12 seconds behind the master.

Changing the Oplog Size

If we find that the oplog size needs to be changed, the simplest way to do so is to stop the master, delete the files for the *local* database, and restart with a new setting for

`--oplogSize`. To change the oplog size to *`size`*, we shut down the master and run the following:

```
$ rm /data/db/local.*
$ ./mongod --master --oplogSize size
```

`size` is specified in megabytes.

> Preallocating space for a large oplog can be time-consuming and might cause too much downtime for the master node. It is possible to manually preallocate data files for MongoDB if that is the case; see the MongoDB documentation on halted replication for more detailed information.

After restarting the master, any slaves should either be restarted with the `--autoresync` or have a manual resync performed.

Replication with Authentication

If you are using replication in tandem with MongoDB's support for authentication (see "Authentication Basics" on page 118), there is some additional configuration that needs to be performed to allow the slave to access the data on the master. On both the master and the slave, a user needs to be added to the *local* database, with the same username and password on each node. Users on the *local* database are similar to users on *admin*; they have full read and write permissions on the server.

When the slave attempts to connect to the master, it will authenticate using a user stored in *`local.system.users`*. The first username it will try is "repl," but if no such user is found, it will just use the first available user in *`local.system.users`*. So, to set up replication with authentication, run the following code on both the master *and* any slaves, replacing *`password`* with a secure password:

```
> use local
switched to db local
> db.add User("repl", password);
{
    "user" : "repl",
    "readOnly" : false,
    "pwd" : "..."
}
```

The slave will then be able to replicate from the master.

CHAPTER 10
Sharding

Sharding is MongoDB's approach to scaling out. Sharding allows you to add more machines to handle increasing load and data size without affecting your application.

Introduction to Sharding

Sharding refers to the process of splitting data up and storing different portions of the data on different machines; the term *partitioning* is also sometimes used to describe this concept. By splitting data up across machines, it becomes possible to store more data and handle more load without requiring large or powerful machines.

Manual sharding can be done with almost any database software. It is when an application maintains connections to several different database servers, each of which are completely independent. The application code manages storing different data on different servers and querying against the appropriate server to get data back. This approach can work well but becomes difficult to maintain when adding or removing nodes from the cluster or in the face of changing data distributions or load patterns.

MongoDB supports *autosharding*, which eliminates some of the administrative headaches of manual sharding. The cluster handles splitting up data and rebalancing automatically. Throughout the rest of this book (and most MongoDB documentation in general), the terms *sharding* and *autosharding* are used interchangeably, but it's important to note the difference between that and manual sharding in an application.

Autosharding in MongoDB

The basic concept behind MongoDB's sharding is to break up collections into smaller *chunks*. These chunks can be distributed across *shards* so that each shard is responsible for a subset of the total data set. We don't want our application to have to know what shard has what data, or even that our data is broken up across multiple shards, so we run a routing process called `mongos` in front of the shards. This router knows where all of the data is located, so applications can connect to it and issue requests normally. As far as the application knows, it's connected to a normal `mongod`. The router, knowing what data is on which shard, is able to forward the requests to the appropriate shard(s). If there are responses to the request, the router collects them and sends them back to the application.

In a nonsharded MongoDB setup, you would have a client connecting to a `mongod` process, like in Figure 10-1. In a sharded setup, like Figure 10-2, the client connects to a `mongos` process, which abstracts the sharding away from the application. From the application's point of view, a sharded setup looks just like a nonsharded setup. There is no need to change application code when you need to scale.

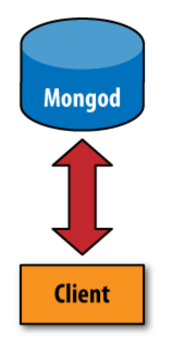


Figure 10-1. Nonsharded client connection

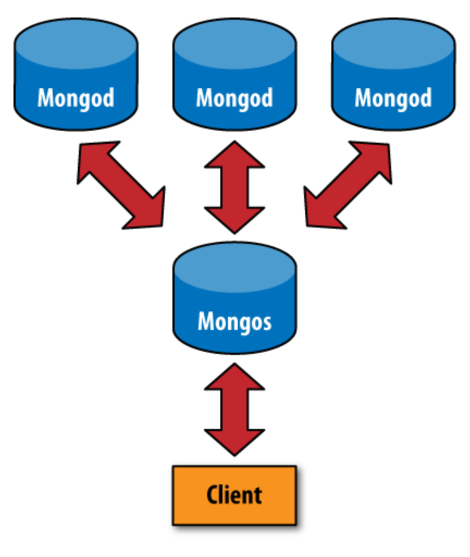


Figure 10-2. Sharded client connection

If you are using an old version of MongoDB, you should upgrade to at least 1.6.0 before using sharding. Sharding has been around for a while, but the first production-ready release is 1.6.0.

When to Shard

One question people often have is when to start sharding. There are a couple of signs that sharding might be a good idea:

- You've run out of disk space on your current machine.
- You want to write data faster than a single `mongod` can handle.
- You want to keep a larger proportion of data in memory to improve performance.

In general, you should start with a nonsharded setup and convert it to a sharded one, if and when you need.

The Key to Sharding: Shard Keys

When you set up sharding, you choose a key from a collection and use that key's values to split up the data. This key is called a *shard key*.

Let's look at an example to see how this works: suppose we had a collection of documents representing people. If we chose `"name"` as our shard key, one shard could hold documents where the `"name"` started with A–F, the next shard could hold names from G–P, and the final shard would hold names from Q–Z. As you added (or removed) shards, MongoDB would rebalance this data so that each shard was getting a balanced amount of traffic and a sensible amount of data (e.g., if a shard is getting a lot of traffic, it might have less data than a shard with data that is less "hot").

Sharding an Existing Collection

Suppose we have an existing collection of logs and we want to shard it. If we enable sharding and tell MongoDB to use `"timestamp"` as the shard key, we'll have a single shard with all of our data. We can insert any data we'd like, and it will all go to that one shard.

Now, suppose we add a new shard. Once this shard is up and running, MongoDB will break up the collection into two pieces, called *chunks*. A chunk contains all of the documents for a range of values for the shard key, so one chunk would have documents with a timestamp value between -∞ and, say, June 26, 2003, and the other chunk would have timestamps between June 27, 2003 and ∞. One of these chunks would then be moved to the new shard.

If we get a new document with a timestamp value before June 27, 2003, we'll add that to the first chunk; otherwise, we'll add the document to the second chunk.

Incrementing Shard Keys Versus Random Shard Keys

The distribution of inserts across shards is very dependent on which key we're sharding on.

If we choose to shard on something like `"timestamp"`, where the value is probably going to increase and not jump around a lot, we'll be sending all of the inserts to one shard (the one with the [June 27, 2003, ∞] chunk). Notice that, if we add a new shard and it splits the data again, we'll still be inserting on just one server. If we add a new shard, MongoDB might split [June 27, 2003, ∞] into [June 27, 2003, December 12, 2010) and [December 12, 2010, ∞]. We'll always have a chunk that will be "some date through infinity," which is where our inserts will be going. This isn't good for a very high write load, but it will make queries on the shard key very efficient.

If we have a high write load and want to evenly distribute writes across multiple shards, we should pick a shard key that will jump around more. This could be a hash of the timestamp in the log example or a key like `"logMessage"`, which won't have any particular pattern to it.

Whether your shard key jumps around or increases steadily, it is important to choose a key that will vary somewhat. If, for example, we had a `"logLevel"` key that had only values `"DEBUG"`, `"WARN"`, or `"ERROR"`, MongoDB won't be able to break up your data into more than three chunks (because there are only three different values). If you have a key with very little variation and want to use it as a shard key anyway, you can do so by creating a compound shard key on that key and a key that varies more, like `"logLevel"` and `"timestamp"`.

Determining which key to shard on and creating shard keys should be reminiscent of indexing, because the two concepts are similar. In fact, often your shard key will just be the index you use most often.

How Shard Keys Affect Operations

To the end user, a sharded setup should be indistinguishable from a nonsharded one. However, it can be useful, especially when setting up sharding, to understand how different queries will be done depending on the shard key chosen.

Suppose we have the collection described in the previous section, which is sharded on the `"name"` key and has three shards with names ranging from A to Z. Different queries will be executed in different ways:

`db.people.find({"name" : "Susan"})`

`mongos` will send this query directly to the Q–Z shard, receive a response from that shard, and send it to the client.

`db.people.find({"name" : {"$lt" : "L"}})`

`mongos` will send the query to the A–F and G–P shards in serial. It will forward their responses to the client.

`db.people.find().sort({"email" : 1})`
: `mongos` will query all of the shards and do a merge sort when it gets the results to make sure it is returning them in the correct order. `mongos` uses cursors to retrieve data from each server, so it does not need to get the entire data set in order to start sending batches of results to the client.

`db.people.find({"email" : "joe@example.com"})`
: `mongos` does not keep track of the `"email"` key, so it doesn't know which shard to send this to. Thus, it sends the query to all of the shards in serial.

If we insert a new document, `mongos` will send that document to the appropriate shard, based on the value of its `"name"` key.

Setting Up Sharding

There are two steps to setting up sharding: starting the actual servers and then deciding how to shard your data.

Sharding basically involves three different components working together:

shard
: A shard is a container that holds a subset of a collection's data. A shard is either a single `mongod` server (for development/testing) or a replica set (for production). Thus, even if there are many servers in a shard, there is only one master, and all of the servers contain the same data.

`mongos`
: This is the router process and comes with all MongoDB distributions. It basically just routes requests and aggregates responses. It doesn't store any data or configuration information. (Although it does cache information from the config servers.)

config server
: Config servers store the configuration of the cluster: which data is on which shard. Because `mongos` doesn't store anything permanently, it needs somewhere to get the shard configuration. It syncs this data from the config servers.

If you are working with MongoDB already, you probably have a shard ready to go. (Your current `mongod` can become your first shard.) The following section shows how to create a new shard from scratch, but feel free to use your existing database instead.

Starting the Servers

First we need to start up our config server and `mongos`. The config server needs to be started first, because `mongos` uses it to get its configuration. The config server can be started like any other `mongod` process:

```
$ mkdir -p ~/dbs/config
$ ./mongod --dbpath ~/dbs/config --port 20000
```

A config server does not need much space or resources. (A generous estimate is 1KB of config server space per 200MB of actual data.)

Now you need a `mongos` process for your application to connect to. Routing servers don't even need a data directory, but they need to know where the config server is:

```
$ ./mongos --port 30000 --configdb localhost:20000
```

Shard administration is always done through a `mongos`.

Adding a shard

A shard is just a normal `mongod` instance (or replica set):

```
$ mkdir -p ~/dbs/shard1
$ ./mongod --dbpath ~/dbs/shard1 --port 10000
```

Now we'll connect to the `mongos` process we started and add the shard to the cluster. Start up a shell connected to your `mongos`:

```
$ ./mongo localhost:30000/admin
MongoDB shell version: 1.6.0
url: localhost:30000/admin
connecting to localhost:30000/admin
type "help" for help
>
```

Make sure you're connected to `mongos`, not a `mongod`. Now you can add this shard with the `addshard` database command:

```
> db.runCommand({addshard : "localhost:10000", allowLocal : true})
{
    "added" : "localhost:10000",
    "ok" : true
}
```

The `"allowLocal"` key is necessary only if you are running the shard on `localhost`. MongoDB doesn't want to let you accidentally set up a cluster locally, so this lets it know that you're just in development and know what you're doing. If you're in production, you should have shards on different machines (although there can be some overlap; see the next section for details).

Whenever we want to add a new shard, we can run the `addshard` database command. MongoDB will take care of integrating it into the cluster.

Sharding Data

MongoDB won't just distribute every piece of data you've ever stored: you have to explicitly turn sharding on at both the database and collection levels. Let's look at an example: we'll shard the *bar* collection in the *foo* database on the `"_id"` key. First, we enable sharding for *foo*:

```
> db.runCommand({"enablesharding" : "foo"})
```

Sharding a database results in its collections being stored on different shards and is a prerequisite to sharding one of its collections.

Once you've enabled sharding on the database level, you can shard a collection by running the `shardcollection` command:

```
> db.runCommand({"shardcollection" : "foo.bar", "key" : {"_id" : 1}})
```

Now the collection will be sharded by the `"_id"` key. When we start adding data, it will automatically distribute itself across our shards based on the values of `"_id"`.

Production Configuration

The example in the previous section is fine for trying sharding or for development. However, when you move an application into production, you'll want a more robust setup. To set up sharding with no points of failure, you'll need the following:

- Multiple config servers
- Multiple `mongos` servers
- Replica sets for each shard
- `w` set correctly (see the previous chapter for information on `w` and replication)

A Robust Config

Setting up multiple config servers is simple. As of this writing, you can have one config server (for development) or three config servers (for production).

Setting up multiple config servers is the same as setting up one; you just do it three times:

```
$ mkdir -p ~/dbs/config1 ~/dbs/config2 ~/dbs/config3
$ ./mongod --dbpath ~/dbs/config1 --port 20001
$ ./mongod --dbpath ~/dbs/config2 --port 20002
$ ./mongod --dbpath ~/dbs/config3 --port 20003
```

Then, when you start a `mongos`, you should connect it to all three config servers:

```
$ ./mongos --configdb localhost:20001,localhost:20002,localhost:20003
```

Config servers use two-phase commit, not the normal MongoDB asynchronous replication, to maintain separate copies of the cluster's configuration. This ensures that they always have a consistent view of the cluster's state. It also means that if a single config server is down, the cluster's configuration information will go read-only. Clients are still able to do both reads and writes, but no rebalancing will happen until all of the config servers are back up.

Many mongos

You can also run as many `mongos` processes as you want. One recommended setup is to run a `mongos` process for every application server. That way, each application server can talk to `mongos` locally, and if the server goes down, no one will be trying to talk to a `mongos` that isn't there.

A Sturdy Shard

In production, each shard should be a replica set. That way, an individual server can fail without bringing down the whole shard. To add a replica set as a shard, pass its name and a seed to the `addshard` command.

For example, say we have a replica set named `"foo"` containing a server at *prod.example.com:27017* (among other servers). We could add this set to the cluster with the following:

```
> db.runCommand({"addshard" : "foo/prod.example.com:27017"})
```

If *prod.example.com* goes down, `mongos` will know that it is connected to a replica set and use the new primary for that set.

Physical Servers

This may seem like an overwhelming number of machines: three config servers, at least two `mongod`s per shard, and as many `mongos` processes as you want. However, not everything has to have its own machine. The main thing to avoid is putting an entire component on one machine. For example, avoid putting all three config servers, all of your `mongos` processes, or an entire replica set on one machine. However, a config server and `mongos` processes can happily share a box with a member of a replica set.

Sharding Administration

Sharding information is mostly stored in the *config* database, which can be accessed from any connection to a `mongos` process.

config Collections

All of the code in the following sections assume that you are running a shell connected to a `mongos` process and have already run `use config`.

Shards

You can find a list of shards in the *shards* collection:

```
> db.shards.find()
{ "_id" : "shard0", "host" : "localhost:10000" }
{ "_id" : "shard1", "host" : "localhost:10001" }
```

Each shard is assigned a unique, human-readable `_id`.

Databases

The *databases* collection contains a list of databases that exist on the shards and information about them:

```
> db.databases.find()
{ "_id" : "admin", "partitioned" : false, "primary" : "config" }
{ "_id" : "foo", "partitioned" : false, "primary" : "shard1" }
{ "_id" : "x", "partitioned" : false, "primary" : "shard0" }
{
    "_id" : "test",
    "partitioned" : true,
    "primary" : "shard0",
    "sharded" : {
        "test.foo" : {
            "key" : {"x" : 1},
            "unique" : false
        }
    }
}
```

Every database available is listed here and has some basic information available.

`"_id" : string`
: The `_id` is the database's name.

`"partitioned" : boolean`
: If `"partitioned"` is true, then the `enablesharding` command has been run on this database.

`"primary" : string`
: The value corresponds to a shard `"_id"` and indicates where this database's "home" is. A database always has a home, whether it is sharded. In a sharded setup, a new database will be created on a random shard. This home is where it will start creating data files. If it is sharded, it will use other servers as well, but it will start out on this shard.

Chunks

Chunk information is stored in the *chunks* collection. This is where things get more interesting; you can actually see how your data has been divided up across the cluster:

```
> db.chunks.find()
{
    "_id" : "test.foo-x_MinKey",
    "lastmod" : { "t" : 1276636243000, "i" : 1 },
    "ns" : "test.foo",
    "min" : {
        "x" : { $minKey : 1 }
    },
    "max" : {
        "x" : { $maxKey : 1 }
    },
        "shard" : "shard0"
    }
```

This is what a collection with a single chunk will look like: the chunk range goes from -∞ (`MinKey`) to ∞ (`MaxKey`).

Sharding Commands

We've already covered some of the basic commands, such as adding chunks and enabling sharding on a collection. There are a couple more commands that are useful for administering a cluster.

Getting a summary

The `printShardingStatus` function will give you a quick summary of the previous collections:

```
> db.printShardingStatus()
--- Sharding Status ---
  sharding version: { "_id" : 1, "version" : 3 }
  shards:
      { "_id" : "shard0", "host" : "localhost:10000" }
      { "_id" : "shard1", "host" : "localhost:10001" }
  databases:
      { "_id" : "admin", "partitioned" : false, "primary" : "config" }
      { "_id" : "foo", "partitioned" : false, "primary" : "shard1" }
      { "_id" : "x", "partitioned" : false, "primary" : "shard0" }
      { "_id" : "test", "partitioned" : true, "primary" : "shard0",
          "sharded" : { "test.foo" : { "key" : { "x" : 1 }, "unique" : false } } }
              test.foo chunks:
                  { "x" : { $minKey : 1 } } -->> { "x" : { $maxKey : 1 } } on : shard0
                  { "t" : 1276636243000, "i" : 1 }
```

Removing a shard

Shards can be removed from a cluster with the `removeshard` command. `removeshard` drains all of the chunks on a given shard to the other shards.

```
> db.runCommand({"removeshard" : "localhost:10000"});
{
    "started draining" : "localhost:10000",
    "ok" : 1
}
```

As the shard is drained, `removeshard` will give you the status of how much remains on the shard.

```
> db.runCommand({"removeshard" : "localhost:10000"});
{
    "msg" : "already draining...",
    "remaining" : {
        "chunks" : 39,
        "dbs" : 2
    },
    "ok" : 1
}
```

Finally, when the shard has finished draining, `removeshard` shows that the shard has been successfully removed.

As of version 1.6.0, if a removed shard was the primary shard for a database, the database has to be manually moved (using the `moveprimary` command):

```
> db.runCommand({"moveprimary" : "test", "to" : "localhost:10001"})
{
    "primary" : "localhost:10001",
    "ok" : 1
}
```

This will likely be automated in future releases.

CHAPTER 11

Example Applications

Throughout this text, almost all of the examples have been in JavaScript. This chapter explores using MongoDB with languages that are more likely to be used in a real application.

Chemical Search Engine: Java

The Java driver is the oldest MongoDB driver. It has been used in production for years and is stable and a popular choice for enterprise developers.

We'll be using the Java driver to build a search engine for chemical compounds, heavily inspired by *http://www.chemeo.com*. This search engine has the chemical and physical properties of thousands of compounds on file, and its goal is to make this information fully searchable.

Installing the Java Driver

The Java driver comes as a JAR file that can be downloaded from Github. To install, add the JAR to your classpath.

All of the Java classes you will probably need to use in a normal application are in the `com.mongodb` and `com.mongodb.gridfs` packages. There are a number of other packages included in the .JAR that are useful if you are planning on manipulating the driver's internals or expanding its functionality, but most applications can ignore them.

Using the Java Driver

Like most things in Java, the API is a bit verbose (especially compared to the other languages' APIs). However, all of the concepts are similar to using the shell, and almost all of the method names are identical.

The `com.mongodb.Mongo` class creates a connection to a MongoDB server. You can access a database from the connection and then get a collection from the database:

```
import com.mongodb.Mongo;
import com.mongodb.DB;
import com.mongodb.DBCollection;

class ChemSearch {

    public static void main(String[] args) {
        Mongo connection = new Mongo();
        DB db = connection.getDB("search");
        DBCollection chemicals = db.getCollection("chemicals");

        /* ... */
    }
}
```

This will connect to `localhost:27017` and get the `search.chemicals` namespace.

Documents in Java must be instances of `org.bson.DBObject`, an interface that is basically an ordered `java.util.Map`. While there are a few ways to create a document in Java, the simplest one is to use the `com.mongodb.BasicDBObject` class. Thus, creating the document that could be represented by the shell as `{"x" : 1, "y" : "foo"}` would look like this:

```
BasicDBObject doc = new BasicDBObject();
doc.put("x", 1);
doc.put("y", "foo");
```

If we wanted to add an embedded document, such as `"z" : {"hello" : "world"}`, we would create another BasicDBObject and then `put` it in the top-level one:

```
BasicDBObject z = new BasicDBObject();
z.put("hello", "world");

doc.put("z", z);
```

Then we would have the document `{"x" : 1, "y" : "foo", "z" : {"hello" : "world"}}`.

From there, all of the other methods implemented by the Java driver are similar to the shell. For instance, we could say `chemicals.insert(doc)` or `chemicals.find(doc)`. There is full API documentation for the Java driver at *http://api.mongodb.org/java* and some articles on specific areas of interest (concurrency, data types, etc.) at the MongoDB Java Language Center.

Schema Design

The interesting thing about this problem is that there are thousands of possible properties for each chemical, and we want to be able to search for any of them efficiently. Take two simple examples: silicon and silicon nitride. A document representing silicon might look something like this:

```
{
    "name" : "silicon",
    "mw" : 32.1173
}
```

`mw` stands for “molecular weight.”

Silicon nitride might have a couple other properties, so its document would look like this:

```
{
    "name" : "silicon nitride",
    "mw" : 42.0922,
    "ΔfH°gas" : {
        "value" : 372.38,
        "units" : "kJ/mol"
    },
    "S°gas" : {
        "value" : 216.81,
        "units" : "J/mol×K"
    }
}
```

MongoDB lets us store chemicals with any number or structure of properties, which makes this application nicely extensible, but there’s no efficient way to index it in its current form. To be able to quickly search for any property, we would need to index almost every key! As we learned in Chapter 5, this is a bad idea.

There is a solution. We can take advantage of the fact that MongoDB indexes every element of an array, so we can store all of the properties we want to search for in an array with common key names. For example, with silicon nitride we can add an array just for indexing containing each property of the given chemical:

```
{
    "name" : "silicon nitride",
    "mw" : 42.0922,
    "ΔfH°gas" : {
        "value" : 372.38,
        "units" : "kJ/mol"
    },
    "S°gas" : {
        "value" : 216.81,
        "units" : "J/mol×K"
    },
    "index" : [
        {"name" : "mw", "value" : 42.0922},
        {"name" : "ΔfH°gas", "value" : 372.38},
        {"name" : "S°gas", "value" : 216.81}
    ]
}
```

Silicon, on the other hand, would have a single-element array with just the molecular weight:

```
{
    "name" : "silicon",
    "mw" : 32.1173,
        "index" : [
            {"name" : "mw", "value" : 32.1173}
        ]
    }
```

Now, all we need to do is create a compound index on the `"index.name"` and `"index.value"` keys. Then we'll be able to do a fairly quick search through the chemical compounds for any attribute.

Writing This in Java

Going back to our original Java code snippet, we'll create a compound index with the `ensureIndex` function:

```
BasicDBObject index = new BasicDBObject();
index.put("index.name", 1);
index.put("index.value", 1);

chemicals.ensureIndex(index);
```

Creating a document for, say, silicon nitride is not difficult, but it is verbose:

```
public static DBObject createSiliconNitride() {
    BasicDBObject sn = new BasicDBObject();
    sn.put("name", "silicon nitride");
    sn.put("mw", 42.0922);

    BasicDBObject deltafHgas = new BasicDBObject();
    deltafHgas.put("value", 372.38);
    deltafHgas.put("units", "kJ/mol");

    sn.put("ΔfH°gas", deltafHgas);

    BasicDBObject sgas = new BasicDBObject();
    sgas.put("value", 216.81);
    sgas.put("units", "J/mol×K");

    sn.put("S°gas", sgas);

    ArrayList<BasicDBObject> index = new ArrayList<BasicDBObject>();
    index.add(BasicDBObjectBuilder.start()
              .add("name", "mw").add("value", 42.0922).get());
    index.add(BasicDBObjectBuilder.start()
              .add("name", "ΔfH°gas").add("value", 372.38).get());
    index.add(BasicDBObjectBuilder.start()
              .add("name", "S°gas").add("value", 216.81).get());

    sn.put("index", index);

    return sn;
}
```

Arrays can be represented by anything that implements `java.util.List`, so we create a `java.util.ArrayList` of embedded documents for the chemical's properties.

Issues

One issue with this structure is that, if we are querying for multiple criteria, search order matters. For example, suppose we are looking for all documents with a molecular weight of less than 1000, a boiling point greater than 0°, and a freezing point of -20°. Naively, we could do this query by concatenating the criteria in an `$all` conditional:

```
BasicDBObject criteria = new BasicDBObject();

BasicDBObject all = new BasicDBObject();

BasicDBObject mw = new BasicDBObject("name", "mw");
mw.put("value", new BasicDBObject("$lt", 1000));

BasicDBObject bp = new BasicDBObject("name", "bp");
bp.put("value", new BasicDBObject("$gt", 0));

BasicDBObject fp = new BasicDBObject("name", "fp");
fp.put("value", -20);

all.put("$elemMatch", mw);
all.put("$elemMatch", bp);
all.put("$elemMatch", fp);
criteria.put("index", new BasicDBObject("$all", all));

chemicals.find(criteria);
```

The problem with this approach is that MongoDB can use an index only for the first item in an `$all` conditional. Suppose there are 1 million documents with a `"mw"` key whose value is less than 1,000. MongoDB can use the index for that part of the query, but then it will have to scan for the boiling and freezing points, which will take a long time.

If we know some of the characteristics of our data, for instance, that there are only 43 chemicals with a freezing point of -20°, we can rearrange the `$all` to do that query first:

```
all.put("$elemMatch", fp);
all.put("$elemMatch", mw);
all.put("$elemMatch", bp);
criteria.put("index", new BasicDBObject("$all", all));
```

Now the database can quickly find those 43 elements and, for the subsequent clauses, has to scan only 43 elements (instead of 1 million). Figuring out a good ordering for arbitrary searches is the real trick of course, of course. This could be done with pattern recognition and data aggregation algorithms that are beyond the scope of this book.

News Aggregator: PHP

We will be creating a basic news aggregation application: users submit links to interesting sites, and other users can comment and vote on the quality of the links (and other comments). This will involve creating a tree of comments and implementing a voting system.

Installing the PHP Driver

The MongoDB PHP driver is a PHP extension. It is easy to install on almost any system. It should work on any system with PHP 5.1 or newer installed.

Windows install

Look at the output of `phpinfo()` and determine the version of PHP you are running (PHP 5.2 and 5.3 are supported on Windows; 5.1 is not), including VC version, if shown. If you are using Apache, you should use VC6; otherwise, you're probably running a VC9 build. Some obscure Zend installs use VC8. Also notice whether it is thread-safe (usually abbreviated "ts").

While you're looking at `phpinfo()`, make a note of the `extension_dir` value, which is where we'll need to put the extension.

Now that you know what you're looking for, go to Github. Download the package that matches your PHP version, VC version, and thread safety. Unzip the package, and move *php_mongo.dll* to the `extension_dir` directory.

Finally, add the following line to your *php.ini* file:

```
extension=php_mongo.dll
```

If you are running an application server (Apache, WAMPP, and so on), restart it. The next time you start PHP, it will automatically load the Mongo extension.

Mac OS X Install

It is easiest to install the extension through PECL, if you have it available. Try running the following:

```
$ pecl install mongo
```

Some Macs do not, however, come with PECL or the correct PHP libraries to install extensions.

If PECL does not work, you can download binary builds for OS X, available at Github (*http://www.github.com/mongodb/mongo-php-driver/downloads*). Run `php -i` to see what version of PHP you are running and what the value of `extension_dir` is, and then download the correct version. (It will have "osx" in the filename.) Unarchive the extension, and move *mongo.so* to the directory specified by `extension_dir`.

After the extension is installed via either method, add the following line to your *php.ini* file:

```
extension=mongo.so
```

Restart any application server you might have running, and the Mongo extension will be loaded the next time PHP starts.

Linux and Unix install

Run the following:

```
$ pecl install mongo
```

Then add the following line to your *php.ini* file:

```
extension=mongo.so
```

Restart any application server you might have running, and the Mongo extension will be loaded the next time PHP is started.

Using the PHP Driver

The `Mongo` class is a connection to the database. By default, the constructor attempts to connect to a database server running locally on the default port.

You can use the `__get` function to get a database from the connection and a collection from the database (even a subcollection from a collection). For example, this connects to MongoDB and gets the `bar` collection in the `foo` database:

```
<?php

$connection = new Mongo();

$collection = $connection->foo->bar;

?>
```

You can continue chaining getters to access subcollections. For example, to get the `bar.baz` collection, you can say the following:

```
$collection = $connection->foo->bar->baz;
```

Documents are represented by associative arrays in PHP. Thus, something like `{"foo" : "bar"}` in JavaScript could be represented as `array("foo" => "bar")` in PHP. Arrays are also represented as arrays in PHP, which sometimes leads to confusion: `["foo", "bar", "baz"]` in JavaScript is equivalent to `array("foo", "bar", "baz")`.

The PHP driver uses PHP's native types for null, booleans, numbers, strings, and arrays. For all other types, there is a Mongo-prefixed type: `MongoCollection` is a collection, `MongoDB` is a database, and `MongoRegex` is a regular expression. There is extensive documentation in the PHP manual for all of these classes.

Designing the News Aggregator

We'll be creating a simple news aggregator, where users can submit links to interesting stories and other users can vote and comment on them. We will just be covering two aspects of it: creating a tree of comments and handling votes.

To store the submissions and comments, we need only a single collection, *`posts`*. The initial posts linking to some article will look something like the following:

```
{
    "_id" : ObjectId(),
    "title" : "A Witty Title",
    "url" : "http://www.example.com",
    "date" : new Date(),
    "votes" : 0,
    "author" : {
        "name" : "joe",
        "_id" : ObjectId(),
    }
}
```

The comments will be almost identical, but they need a `"content"` key instead of a `"url"` key.

Trees of Comments

There are several different ways to represent a tree in MongoDB; the choice of which representation to use depends on the types of query being performed.

We'll be storing an *array of ancestors* tree: each node will contain an array of its parent, grandparent, and so on. So, if we had the following comment structure:

```
original link
|- comment 1
|   |- comment 3 (reply to comment 1)
|   |- comment 4 (reply to comment 1)
|       |- comment 5 (reply to comment 4)
|- comment 2
|   |- comment 6 (reply to comment 2)
```

then comment 5's array of ancestors would contain the original link's `_id`, comment 1's `_id`, and comment 4's `_id`. Comment 2's ancestors would be the original link's `_id` and comment 2's `_id`. This allows us to easily search for "all comments for link *`X`*" or "the subtree of comment 2's replies."

This method of storing comments assumes that we are going to have a lot of them and that we might be interested in seeing just parts of a comment thread. If we knew that we always wanted to display all of the comments and there weren't going to be thousands, we could store the entire tree of comments as an embedded document in the submitted link's document.

Using the array of ancestors approach, when someone wants to create a new comment, we need to add a new document to the collection. To create this document, we create a leaf document by linking it to the parent's `"_id"` value and its array of ancestors.

```
function createLeaf($parent, $replyInfo) {
    $child = array(
        "_id" => new MongoId(),
        "content" => $replyInfo['content'],
        "date" => new MongoDate(),
        "votes" => 0,
        "author" => array(
            "name" => $replyInfo['name'],
            "name" => $replyInfo['name'],
        ),
        "ancestors" => $parent['ancestors'],
        "parent" => $parent['_id']
    );

    // add the parent's _id to the ancestors array
    $child['ancestors'][] = $parent['_id'];

    return $child;
}
```

Then we can add the new comment to the *posts* collection:

```
$comment = createLeaf($parent, $replyInfo);

$posts = $connection->news->posts;
$posts->insert($comment);
```

We can get a list of the latest submissions (sans comments) with the following:

```
$cursor = $posts->find(array("ancestors" => array('$size' => 0)));
$cursor = $cursor->sort(array("date" => -1));
```

If someone wants to see the comments for a given post, we can find them all with the following:

```
$cursor = $posts->find(array("ancestors" => $postId));
```

In fact, we can use this query to access any subtree of comments. If the root of the subtree is passed in as `$postId`, every child will contain `$postId` in its ancestor's array and be returned.

To make these queries fast, we should index the `"date"` and `"ancestors"` keys:

```
$pageOfComments = $posts->ensureIndex(array("date" => -1, "ancestors" => 1));
```

Now we can quickly query for the main page, a tree of comments, or a subtree of comments.

Voting

There are many ways of implementing voting, depending on the functionality and information you want: do you allow up and down votes? Will you prevent users from voting more than once? Will you allow them to switch their vote? Do you care *when* people voted, to see if a link is trending? Each of these requires a different solution with far more complex coding than the simplest way of doing it: using `"$inc"`:

```
$posts->update(array("_id" => $postId), array('$inc' => array("votes", 1)));
```

For a controversial or popular link, we wouldn't want people to be able to vote hundreds of times, so we want to limit users to one vote each. A simple way to do this is to add a `"voters"` array to keep track of who has voted on this post, keeping an array of user `"_id"` values. When someone tries to vote, we do an update that checks the user `"_id"` against the array of `"_id"` values:

```
$posts->update(array("_id" => $postId, "voters" => array('$ne' => $userId)),
    array('$inc' => array("votes", 1), '$push' => array("voters" => $userId)));
```

This will work for up to a couple million users. For larger voting pools, we would hit the 4MB limit, and we would have to special-case the most popular links by putting spillover votes into a new document.

Custom Submission Forms: Ruby

MongoDB is a popular choice for Ruby developers, likely because the document-oriented approach meshes well with Ruby's dynamism and flexibility. In this example we'll use the MongoDB Ruby driver to build a framework for custom form submissions, inspired by a *New York Times* blog post about how it uses MongoDB to handle submission forms (*http://open.blogs.nytimes.com/2010/05/25/building-a-better-submission-form/*). For even more documentation on using MongoDB from Ruby, check out the Ruby Language Center.

Installing the Ruby Driver

The Ruby driver is available as a RubyGem, hosted at *http://rubygems.org*. Installation using the gem is the easiest way to get up and running. Make sure you're using an up-to-date version of RubyGems (with `gem update --system`) and then install the *mongo* gem:

```
$ gem install mongo
Successfully installed bson-1.0.2
Successfully installed mongo-1.0.2
2 gems installed
Installing ri documentation for bson-1.0.2...
Building YARD (yri) index for bson-1.0.2...
Installing ri documentation for mongo-1.0.2...
Building YARD (yri) index for mongo-1.0.2...
```

```
Installing RDoc documentation for bson-1.0.2...
Installing RDoc documentation for mongo-1.0.2...
```

Installing the *mongo* gem will also install the *bson* gem on which it depends. The *bson* gem handles all of the BSON encoding and decoding for the driver (for more on BSON, see "BSON" on page 179). The *bson* gem will also make use of C extensions available in the *bson_ext* gem to improve performance, if that gem has been installed. For maximum performance, be sure to install *bson_ext*:

```
$ gem install bson_ext
Building native extensions.  This could take a while...
Successfully installed bson_ext-1.0.1
1 gem installed
```

If *bson_ext* is on the load path, it will be used automatically.

Using the Ruby Driver

To connect to an instance of MongoDB, use the `Mongo::Connection` class. Once we have an instance of `Mongo::Connection`, we can get an individual database (here we use the *stuffy* database) using bracket notation:

```
> require 'rubygems'
 => true
> require 'mongo'
 => true
> db = Mongo::Connection.new["stuffy"]
```

The Ruby driver uses hashes to represent documents. Aside from that, the API is similar to that of the shell with most method names being the same. (Although the Ruby driver uses underscore_naming, whereas the shell often uses camelCase.) To insert the document `{"x" : 1}` into the *bar* collection and query for the result, we would do the following:

```
> db["bar"].insert :x => 1
 => BSON::ObjectID('4c168343e6fb1b106f000001')
> db["bar"].find_one
 => {"_id"=>BSON::ObjectID('4c168343e6fb1b106f000001'), "x"=>1}
```

There are some important gotchas about documents in Ruby that you need to be aware of:

- Hashes are ordered in Ruby 1.9, which matches how documents work in MongoDB. In Ruby 1.8, however, hashes are unordered. The driver provides a special type, `BSON::OrderedHash`, which must be used instead of a regular hash whenever key order is important.
- Hashes being saved to MongoDB can have symbols as either keys or values. Hashes returned from MongoDB will have symbol values wherever they were present in the input, but any symbol keys will be returned as strings. So, `{:x => :y}` will become `{"x" => :y}`. This is a side effect of the way documents are represented in BSON (see Appendix C for more on BSON).

Custom Form Submission

The problem at hand is to generate custom forms for user-submitted data and to handle user submissions for those forms. Forms are created by editors and can contain arbitrary fields, each with different types and rules for validation. Here we'll leverage the ability to embed documents and store each field as a separate document within a form. A form document for a comment submission form might look like this:

```
comment_form = {
    :_id => "comments",
    :fields => [
        {
            :name => "name",
            :label => "Your Name",
            :help_text => "Required",
            :required => true,
            :type => "string",
            :max_length => 200
        },
        {
            :name => "email",
            :label => "Your E-mail Address",
            :help_text => "Required, but will not be displayed",
            :required => true,
            :type => "email"
        },
        {
            :name => "comment",
            :label => "Your Comment",
            :help_text => "Comments will be moderated",
            :required => true,
            :type => "string",
            :word_limit => 200
        }
    ]
}
```

This form shows some of the benefits of working with a document-oriented database like MongoDB. First, we're able to embed the form's fields directly within the form document. We don't need to store them separately and do a join—we can get the entire representation for a form by querying for a single document. We're also able to specify different keys for different types of fields. In the previous example, the name field has a `:max_length`, key and the comment field has a `:word_limit` key, while the email field has neither.

In this example we use `"_id"` to store a human-readable name for our form. This works well because we need to index on the form name anyway to make queries efficient. Because the `"_id"` index is a unique index, we're also guaranteed that form names will be unique across the system.

When an editor adds a new form, we simply save the resultant document. To save the `comment_form` document that we created, we'd do the following:

```
db["forms"].save comment_form
```

Each time we want to render a page with the comment form, we can query for the form document by its name:

```
db["forms"].find_one :_id => "comments"
```

The single document returned contains all the information we need in order to render the form, including the name, label, and type for each input field that needs to be rendered. When a form needs to be changed, editors can easily add a field or specify additional constraints for an existing field.

When we get a user submission for a form, we can run the same query as earlier to get the relevant form document. We'll need this in order to validate that the user's submission includes values for all required fields and meets any other requirements specified in our form. After validation, we can save the submission as a separate document in a *submissions* collection. A submission for our comment form might look like this:

```
comment_submission = {
    :form_id => "comments",
    :name => "Mike D.",
    :email => "mike@example.com",
    :comment => "MongoDB is flexible!"
}
```

We're again leveraging the document model by including custom keys for each submission (here we use `:name`, `:email`, and `:comment`). The only key that we require in each submission is `:form_id`. This allows us to efficiently retrieve all submissions for a certain form:

```
db["submissions"].find :form_id => "comments"
```

To perform this query, we should have an index on `:form_id`:

```
db["submissions"].create_index :form_id
```

We can also use `:form_id` to retrieve the form document for a given submission.

Ruby Object Mappers and Using MongoDB with Rails

There are several libraries written on top of the basic Ruby driver to provide things like models, validations, and associations for MongoDB documents. The most popular of these tools seem to be MongoMapper and Mongoid. If you're used to working with tools like ActiveRecord or DataMapper, you might consider using one of these object mappers in addition to the basic Ruby driver.

MongoDB also works nicely with Ruby on Rails, especially when working with one of the previously mentioned mappers. There are up-to-date instructions on integrating MongoDB with Rails on the MongoDB site.

Real-Time Analytics: Python

The Python driver for MongoDB is called *PyMongo*. In this section, we'll use PyMongo to implement some real-time tracking of metrics for a web application. The most up-to-date documentation on PyMongo is available at *http://api.mongodb.org/python*.

Installing PyMongo

PyMongo is available in the Python Package Index and can be installed using `easy_install` (*http://pypi.python.org/pypi/setuptools*):

```
$ easy_install pymongo
Searching for pymongo
Reading http://pypi.python.org/simple/pymongo/
Reading http://github.com/mongodb/mongo-python-driver
Best match: pymongo 1.6
Downloading ...
Processing pymongo-1.6-py2.6-macosx-10.6-x86_64.egg
Moving ...
Adding pymongo 1.6 to easy-install.pth file

Installed ...
Processing dependencies for pymongo
Finished processing dependencies for pymongo
```

This will install PyMongo and will attempt to install an optional C extension as well. If the C extension fails to build or install, everything will continue to work, but performance will suffer. An error message will be printed during install in that case.

As an alternative to `easy_install`, PyMongo can also be installed by running `python setup.py install` from a source checkout.

Using PyMongo

We use the `pymongo.connection.Connection` class to connect to a MongoDB server. Here we create a new `Connection` and use attribute-style access to get the *analytics* database:

```
from pymongo import Connection
db = Connection().analytics
```

The rest of the API for PyMongo is similar to the API of the MongoDB shell; like the Ruby driver, PyMongo uses underscore_naming instead of camelCase, however. Documents are represented using dictionaries in PyMongo, so to insert and retrieve the document `{"a" : [1, 2, 3]}`, we do the following:

```
db.test.insert({"a": [1, 2, 3]})
db.test.find_one()
```

Dictionaries in Python are unordered, so PyMongo provides an ordered subclass of `dict`, `pymongo.son.SON`. In most places where ordering is required, PyMongo provides

APIs that hide it from the user. If not, applications can use `SON` instances instead of dictionaries to ensure their documents maintain key order.

MongoDB for Real-Time Analytics

MongoDB is a great tool for tracking metrics in real time for a couple of reasons:

- Upsert operations (see Chapter 3) allow us to send a single message to either create a new tracking document or increment the counters on an existing document.
- The upsert we send does not wait for a response; it's fire-and-forget. This allows our application code to avoid blocking on each analytics update. We don't need to wait and see whether the operation is successful, because an error in analytics code wouldn't get reported to a user anyway.
- We can use an `$inc` update to increment a counter without having to do a separate query and update operation. We also eliminate any contention issues if multiple updates are happening simultaneously.
- MongoDB's update performance is very good, so doing one or more updates per request for analytics is reasonable.

Schema

In our example we will be tracking page views for our site, with hourly roll-ups. We'll track both total page views as well as page views for each individual URL. The goal is to end up with a collection, *hourly*, containing documents like this:

```
{ "hour" : "Tue Jun 15 2010 9:00:00 GMT-0400 (EDT)", "url" : "/foo", "views" : 5 }
{ "hour" : "Tue Jun 15 2010 9:00:00 GMT-0400 (EDT)", "url" : "/bar", "views" : 5 }
{ "hour" : "Tue Jun 15 2010 10:00:00 GMT-0400 (EDT)", "url" : "/", "views" : 12 }
{ "hour" : "Tue Jun 15 2010 10:00:00 GMT-0400 (EDT)", "url" : "/bar", "views" : 3 }
{ "hour" : "Tue Jun 15 2010 10:00:00 GMT-0400 (EDT)", "url" : "/foo", "views" : 10 }
{ "hour" : "Tue Jun 15 2010 11:00:00 GMT-0400 (EDT)", "url" : "/foo", "views" : 21 }
{ "hour" : "Tue Jun 15 2010 11:00:00 GMT-0400 (EDT)", "url" : "/", "views" : 3 }
...
```

Each document represents all of the page views for a single URL in a given hour. If a URL gets no page views in an hour, there is no document for it. To track total page views for the entire site, we'll use a separate collection, *hourly_totals*, which has the following documents:

```
{ "hour" : "Tue Jun 15 2010 9:00:00 GMT-0400 (EDT)", "views" : 10 }
{ "hour" : "Tue Jun 15 2010 10:00:00 GMT-0400 (EDT)", "views" : 25 }
{ "hour" : "Tue Jun 15 2010 11:00:00 GMT-0400 (EDT)", "views" : 24 }
...
```

The difference here is just that we don't need a `"url"` key, because we're doing site-wide tracking. If our entire site doesn't get any page views during an hour, there will be no document for that hour.

Handling a Request

Each time our application receives a request, we need to update our analytics collections appropriately. We need to add a page view both to the *hourly* collection for the specific URL requested and to the *hourly_totals* collection. Let's define a function that takes a URL and updates our analytics appropriately:

```
from datetime import datetime

def track(url):
    hour = datetime.utcnow().replace(minute=0, second=0, microsecond=0)
    db.hourly.update({"hour": hour, "url": url},
                     {"$inc": {"views": 1}}, upsert=True)
    db.hourly_totals.update({"hour": hour},
                            {"$inc": {"views": 1}}, upsert=True)
```

We'll also want to make sure that we have indexes in place to be able to perform these updates efficiently:

```
from pymongo import ASCENDING

db.hourly.create_index([("url", ASCENDING), ("hour", ASCENDING)], unique=True)
db.hourly_totals.create_index("hour", unique=True)
```

For the *hourly* collection, we create a compound index on `"url"` and `"hour"`, while for *hourly_totals* we just index on `"hour"`. Both of the indexes are created as unique, because we want only one document for each of our roll-ups.

Now, each time we get a request, we just call `track` a single time with the requested URL. It will perform two upserts; each will create a new roll-up document if necessary or increment the `"views"` for an existing roll-up.

Using Analytics Data

Now that we're tracking page views, we need a way to query that data and put it to use. Here we print the hourly page view totals for the last 10 hours:

```
from pymongo import DESCENDING

for rollup in db.hourly_totals.find().sort("hour", DESCENDING).limit(10):
    pretty_date = rollup["hour"].strftime("%Y/%m/%d %H")
    print "%s - %d" % (pretty_date, rollup["views"])
```

This query will be able to leverage the index we've already created on `"hour"`. We can perform a similar operation for an individual *`url`*:

```
for rollup in db.hourly.find({"url": url}).sort("hour", DESCENDING).limit(10):
    pretty_date = rollup["hour"].strftime("%Y/%m/%d %H")
    print "%s - %d" % (pretty_date, rollup["views"])
```

The only difference is that here we add a query document for selecting an individual `"url"`. Again, this will leverage the compound index we've already created on `"url"`, and `"hour"`.

Other Considerations

One thing we might want to consider is running a periodic cleaning task to remove old analytics documents. If we're displaying only the last 10 hours of data, then we can conserve space by not keeping around a month's worth of documents. To remove all documents older than 24 hours, we can do the following, which could be run using *cron* or a similar mechanism:

```
from datetime import timedelta

remove_before = datetime.utcnow() - timedelta(hours=24)

db.hourly.remove({"hour": {"$lt": remove_before}})
db.hourly_totals.remove({"hour": {"$lt": remove_before}})
```

In this example, the first `remove` will actually need to do a table scan because we haven't defined an index on `"hour"`. If we need to perform this operation efficiently (or any other operation querying by `"hour"` for all URLs), we should consider adding a second index on `"hour"` to the *hourly* collection.

Another important note about this example is that it would be easy to add tracking for other metrics besides page views or to do roll-ups on a window other than hourly (or even to do roll-ups on multiple windows at once). All we need to do is to tweak the `track` function to perform upserts tracking whatever metric we're interested in, at whatever granularity we want.

APPENDIX A

Installing MongoDB

Installing MongoDB is a simple process on most platforms. Precompiled binaries are available for Linux, Mac OS X, Windows, and Solaris. This means that, on most platforms, you can download the archive from *http://www.mongodb.org*, inflate it, and run the binary. The MongoDB server requires a directory it can write database files to and a port it can listen for connections on. This section covers the entire install on the two variants of system: Windows and everything else (Linux, Max, Solaris).

When we speak of "installing MongoDB," generally what we are talking about is setting up `mongod`, the core database server. `mongod` is used in a single-server setup as either master or slave, as a member of a replica sets, and as a shard. Most of the time, this will be the MongoDB process you are using. Other binaries that come with the download are covered in Chapter 8.

Choosing a Version

MongoDB uses a fairly simple versioning scheme: even-point releases are stable, and odd-point releases are development versions. For example, anything starting with 1.6 is a stable release, such as 1.6.0, 1.6.1, and 1.6.15. Anything starting with 1.7 is a development release, such as 1.7.0, 1.7.2, or 1.7.10. Let's take the 1.6/1.7 release as a sample case to demonstrate how the versioning timeline works:

1. Developers release 1.6.0. This is a major release and will have an extensive changelog. Everyone in production is advised to upgrade as soon as possible.
2. After the developers start working on the milestones for 1.8, they release 1.7.0. This is the new development branch that is fairly similar to 1.6.0, but probably with an extra feature or two and maybe some bugs.
3. As the developers continue to add features, they will release 1.7.1, 1.7.2, and so on.
4. Any bug fixes or "nonrisky" features will be backported to the 1.6 branch, and 1.6.1, 1.6.2, and so on, will be released. Developers are conservative about what

is backported; few new features are ever added to a stable release. Generally, only bug fixes are ported.

5. After all of the major milestones have been reached for 1.8.0, developers will release something like, say, 1.7.5.
6. After extensive testing of 1.7.5, usually there are a couple minor bugs that need to be fixed. Developers fix these bugs and release 1.7.6.
7. Developers repeat step 6 until no new bugs are apparent, and then 1.7.6 (or whatever the latest release ended up being) is renamed 1.8.0. That is, the last development release in a series becomes the new stable release.
8. Start over from step 1, incrementing all versions by .2.

Thus, the initial releases in a development branch may be highly unstable (x.y.0, x.y.1, x.y.2), but usually, by the time a branch gets to x.y.5, it's fairly close to production-ready. You can see how close a production release is by browsing the core server roadmap on the MongoDB bug tracker.

If you are running in production, you should use a stable release unless there are features in the development release that you need. Even if you need certain features from a development release, it is worth first getting in touch with the developers through the mailing list and IRC to let them know you are planning on going into production with a development release and get advice about keeping your data safe. (Of course, this is always a good idea.)

If you are just starting development on a project, using a development release may be a better choice. By the time you deploy to production, there will probably be a stable release (MongoDB keeps a regular cycle of stable releases every couple of months), and you'll get to use the latest features. However, you must balance this against the possibility that you would run into server bugs, which could be confusing or discouraging to most new users.

Windows Install

To install MongoDB on Windows, download the Windows zip from the MongoDB downloads page. Use the advice in the previous section to choose the correct version of MongoDB. There are 32-bit and 64-bit releases for Windows, so select whichever version you're running. When you click the link, it will download the *.zip*. Use your favorite extraction tool to unzip the archive.

Now you need to make a directory in which MongoDB can write database files. By default, MongoDB tries to use *C:\data\db* as its data directory. You can create this directory or any other empty directory anywhere on the filesystem. If you chose to use a directory other than *C:\data\db*, you'll need to specify the path when you start MongoDB, which is covered in a moment.

Now that you have a data directory, open the command prompt (*cmd.exe*). Navigate to the directory where you unzipped the MongoDB binaries and run the following:

```
$ bin\mongod.exe
```

If you chose a directory other than *C:\data\db*, you'll have to specify it here, with the `--dbpath` argument:

```
$ bin\mongod.exe --dbpath C:\Documents and Settings\Username\My Documents\db
```

See the Chapter 8 section for more common options, or run `mongod.exe --help` to see all options.

Installing as a Service

MongoDB can also be installed as a service on Windows. To install, simply run with the full path, escape any spaces, and use the `--install` option. For example:

```
$ C:\mongodb-windows-32bit-1.6.0\bin\mongod.exe
     --dbpath "\"C:\Documents and Settings\Username\My Documents\db\"" --install
```

It can then be started and stopped from the Control Panel.

POSIX (Linux, Mac OS X, and Solaris) Install

Choose a version of MongoDB, based on the advice in the section "Choosing a Version" on page 173. Go to the MongoDB downloads page, and select the correct version for your OS.

> If you are using a Mac, check whether you're running 32-bit or 64-bit. Macs are especially picky that you choose the correct build and will refuse to start MongoDB and give confusing error messages if you choose the wrong build. You can check what you're running by clicking the apple in the upper-left corner and selecting the About This Mac option.

You must create a directory for the database to put its files. By default, the database will use */data/db*, although you can specify any other directory. If you create the default directory, make sure it has the correct write permissions. You can create the directory and set the permissions by running the following:

```
$ mkdir -p /data/db
$ chown -R $USER:$USER /data/db
```

`mkdir -p` creates the directory and all its parents, if necessary (i.e., if the */data* directory didn't exist, it will create the `/data` directory and then the */data/db* directory). `chown` changes the ownership of */data/db* so that your user can write to it. Of course, you can also just create a directory in your home folder and specify that MongoDB should use that when you start the database, to avoid any permissions issues.

Decompress the *.tar.gz* file you downloaded from *http://www.mongodb.org*.

```
$ tar zxf mongodb-linux-i686-1.6.0.tar.gz
$ cd mongodb-linux-i686-1.6.0
```

Now you can start the database:

```
$ bin/mongod
```

Or if you'd like to use an alternate database path, specify it with the `--dbpath` option:

```
$ bin/mongod --dbpath ~/db
```

See Chapter 8 for a summary of the most common options, or run `mongod` with `--help` to see all the possible options.

Installing from a Package Manager

On these systems, there are many package managers that can also be used to install MongoDB. If you prefer using one of these, there are official packages for Debian and Ubuntu and unofficial ones for Red Hat, Gentoo, and FreeBSD. If you use an unofficial version, make sure to check the logs when you start the database; sometimes these packages are not built with UTF-8 support.

On Mac, there are also unofficial packages in Homebrew and MacPorts. If you go for the MacPorts version, be forewarned: it takes hours to compile all the Boost libraries, which are MongoDB prerequisites. Start the download, and leave it overnight.

Regardless of the package manager you use, it is a good idea to figure out where it is putting the MongoDB log files before you have a problem and need to find them. It's important to make sure they're being saved properly in advance of any possible issues.

APPENDIX B

mongo: The Shell

Throughout this text, we use the `mongo` binary, which is the database shell. We generally assume that you are running it on the same machine as `mongod` and that you are running `mongod` on the default port, but if you are not, you can specify this on startup and have the shell connect to another server:

```
$ bin/mongo staging.example.com:20000
```

This would connect to a `mongod` running at *staging.example.com* on port 20000.

The shell also, by default, starts out connected to the `test` database. If you'd like `db` to refer to a different database, you can use `/dbname` after the server address:

```
$ bin/mongo localhost:27017/admin
```

This connects to *mongod* running locally on the default port, but `db` will immediately refer to the `admin` database.

You can also start the shell without connecting to any database by using the `--nodb` option. This is useful if you'd like to just play around with JavaScript or connect later:

```
$ bin/mongo --nodb
MongoDB shell version: 1.5.3
type "help" for help
>
```

Keep in mind that *db* isn't the only database connection you can have. You can connect to as many databases as you would like from the shell, which can be handy in multi-server environments. Simply use the `connect()` method, and assign the resulting connection to any variable you'd like. For instance, with sharding, we might want `mongos` to refer to the `mongos` server and also have a connection to each shard:

```
> mongos = connect("localhost:27017")
connecting to: localhost:27017
localhost:27017
> shard0 = connect("localhost:30000")
connecting to: localhost:30000
localhost:30000
> shard1 = connect("localhost:30001")
```

```
connecting to: localhost:30001
localhost:30001
```

Then, we can use `mongos`, `shard0`, or `shard1` as the `db` variable is usually used. (Although special helpers, such as `use foo` or `show collections`, will not work.)

Shell Utilities

There are a number of useful shell functions that were not covered earlier.

For administrating multiple databases, it can be useful to have multiple database variables, not `db`. For example, with sharding, you may want to maintain a separate variable pointing to your `config` database:

```
> config = db.getSisterDB("config")
config
> config.shards.find()
...
```

You can even connect to multiple servers within a single shell using the `connect` function:

```
> shard_db = connect("shard.example.com:27017/mydb")
connecting to shard.example.com:27017/mydb
mydb
>
```

The shell can also run shell commands:

```
> runProgram("echo", "Hello", "world")
shell: started mongo program echo Hello world
0
> sh6487| Hello world
```

(The output looks strange because the shell is running.)

APPENDIX C

MongoDB Internals

For the most part, users of MongoDB can treat it as a black box. When trying to understand performance characteristics or looking to get a deeper understanding of the system, it helps to know a little bit about the internals of MongoDB, though.

BSON

Documents in MongoDB are an abstract concept—the concrete representation of a document varies depending on the driver/language being used. Because documents are used extensively for communication in MongoDB, there also needs to be a representation of documents that is shared by all drivers, tools, and processes in the MongoDB ecosystem. That representation is called Binary JSON (BSON).

BSON is a lightweight binary format capable of representing any MongoDB document as a string of bytes. The database understands BSON, and BSON is the format in which documents are saved to disk.

When a driver is given a document to insert, use as a query, and so on, it will encode that document to BSON before sending it to the server. Likewise, documents being returned to the client from the server are sent as BSON strings. This BSON data is decoded by the driver to its native document representation before being returned to the client.

The BSON format has three primary goals:

Efficiency

BSON is designed to represent data efficiently, without using much extra space. In the worst case BSON is slightly less efficient than JSON and in the best case (e.g., when storing binary data or large numerics), it is much more efficient.

Traversability

In some cases, BSON does sacrifice space efficiency to make the format easier to traverse. For example, string values are prefixed with a length rather than relying on a terminator to signify the end of a string. This traversability is useful when the MongoDB server needs to introspect documents.

Performance
: Finally, BSON is designed to be fast to encode to and decode from. It uses C-style representations for types, which are fast to work with in most programming languages.

For the exact BSON specification, see *http://www.bsonspec.org*.

Wire Protocol

Drivers access the MongoDB server using a lightweight TCP/IP wire protocol. The protocol is documented on the MongoDB wiki but basically consists of a thin wrapper around BSON data. For example, an insert message consists of 20 bytes of header data (which includes a code telling the server to perform an insert and the message length), the collection name to insert into, and a list of BSON documents to insert.

Data Files

Inside of the MongoDB data directory, which is */data/db/* by default, there are separate files for each database. Each database has a single *.ns* file and several data files, which have monotonically increasing numeric extensions. So, the database *foo* would be stored in the files *foo.ns*, *foo.0*, *foo.1*, *foo.2*, and so on.

The numeric data files for a database will double in size for each new file, up to a maximum file size of 2GB. This behavior allows small databases to not waste too much space on disk, while keeping large databases in mostly contiguous regions on disk.

MongoDB also preallocates data files to ensure consistent performance. (This behavior can be disabled using the `--noprealloc` option.) Preallocation happens in the background and is initiated every time that a data file is filled. This means that the MongoDB server will always attempt to keep an extra, empty data file for each database to avoid blocking on file allocation.

Namespaces and Extents

Within its data files, each database is organized into *namespaces*, each storing a specific type of data. The documents for each collection have their own namespace, as does each index. Metadata for namespaces is stored in the database's *.ns* file.

The data for each namespace is grouped on disk into sections of the data files, called *extents*. In Figure C-1 the *foo* database has three data files, the third of which has been preallocated and is empty. The first two data files have been divided up into extents belonging to several different namespaces.

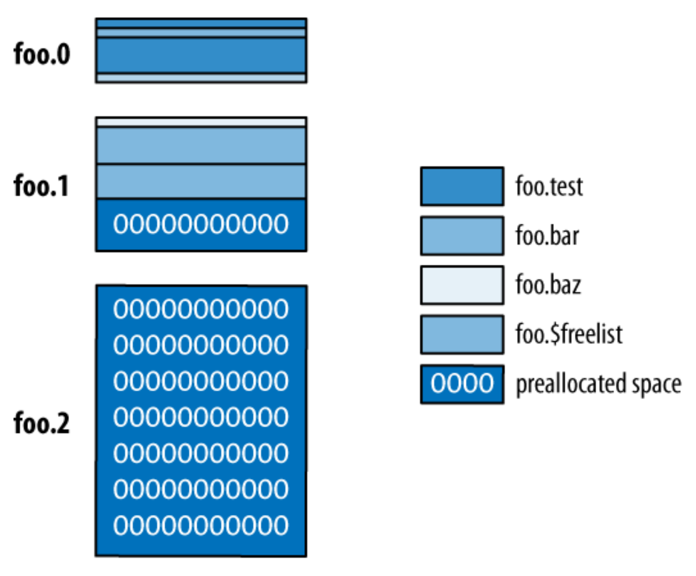


Figure C-1. Namespaces and extents

Figure C-1 shows us several interesting things about namespaces and extents. Each namespace can have several different extents, which are not (necessarily) contiguous on disk. Like data files for a database, extents for a namespace grow in size with each new allocation. This is done to balance wasted space used by a namespace versus the desire to keep data for a namespace mostly contiguous on disk. The figure also shows a special namespace, *$freelist*, which keeps track of extents that are no longer in use (e.g., extents from a dropped collection or index). When a namespace allocates a new extent, it will first search the freelist to see whether an appropriately sized extent is available.

Memory-Mapped Storage Engine

The default storage engine (and only supported storage engine at the time of this writing) for MongoDB is a memory-mapped engine. When the server starts up, it memory maps all its data files. It is then the responsibility of the operating system to manage flushing data to disk and paging data in and out. This storage engine has several important properties:

- MongoDB's code for managing memory is small and clean, because most of that work is pushed to the operating system.
- The virtual size of a MongoDB server process is often very large, exceeding the size of the entire data set. This is OK, because the operating system will handle keeping the amount of data resident in memory contained.

- MongoDB cannot control the order that data is written to disk, which makes it impossible to use a writeahead log to provide single-server durability. Work is ongoing on an alternative storage engine for MongoDB to provide single-server durability.
- 32-bit MongoDB servers are limited to a total of about 2GB of data per ***mongod***. This is because all of the data must be addressable using only 32 bits.

Index

Symbols

32-bit integer type, 16
64-bit floating point number type, 16
64-bit integer type, 16
$ (dollar sign) reserved character, 8
  in conditionals, 45
  key names and, 6
  in update modifiers, 28
    using to add, change, or remove key/value pairs, 30
. (dot)
  dot notation in queries for embedded keys, 54
  in namespaced subcollection names, 8
  key names and, 6
\0 (null character), 8
" " (quotes, double), using in strings, 28
' ' (quotes, single), using with strings, 28
_ (underscore), key names starting with, 6

A

addShard command, 148
$addToSet array modifier, 33
addUser( ) method, 118
admin database, 9
  switching to, 114
  users as superusers, 118
admin interface, 115
administration, 3, 111–125
  authentication basics, 118
  backup and repair, 121–125
    repairing corrupted data files, 124
    slave backups, 124
    using fsync and lock, 123
    using mongodump and mongorestore, 122
  file-based configuration, 113
  monitoring, 114–118
  of replication, 141
    authentication, 142
    changing oplog size, 142
    diagnostics, 141
  security and authentication, 118–121
    how authentication works, 120
    other security considerations, 121
  of sharding, 150–153
    config collections, 150
    printShardingStatus( ) function, 152
    removeshard commands, 152
  starting MongoDB from command line, 112–113
  stopping MongoDB, 114
age-out (automatic), of data in capped collections, 99
aggregation, 3
aggregation tools, 81–92
  count( ) function, 81
  distinct command, 81
  finalizers, 85
  function as grouping key, 86
  group command, 82
  MapReduce, 86–92
    categorizing web pages (example), 89
    finding all keys in collection (example), 87–89
    MongoDB and MapReduce, 90
$all conditional, 51, 159
allowLocal key, 148

We'd like to hear your suggestions for improving our indexes. Send email to *index@oreilly.com*.

analytics application (real-time), using Python, 168–171
application examples, 155–171
  custom submission forms, using Ruby driver, 164–167
    custom form submission, 166
    installing Ruby driver, 164
    object mappers and MongoDB with Rails, 167
    using Ruby driver, 165
  news aggregator using PHP driver, 160–164
    designing news aggregator, 162
    installing PHP driver, 160
    trees of comments, 162
    using PHP driver, 161
    voting, 164
  real-time analytics using Python driver, 168
    handling a request, 170
    installing PyMongo, 168
    other considerations, 171
    schema, 169
    using analytics data, 170
    using PyMongo, 168
  search engine for chemicals, using Java driver, 155–159
    coding in Java, 158
    installing Java driver, 155
    issues with, 159
    schema design, 156
    using Java driver, 155
arbiter nodes, 134
arbiterOnly key, 134
array modifiers, 31–34
  $ positional operator, 34
  $addToSet, 33
    using with $each, 33
  $ne, 32
  $pop, 34
  $pull, 34
  $push, 32
  positional array modifications, 34
  speed of, 35
array of ancestors tree, 162
array type, 17
ArrayList class (Java), 158
arrays, 19
  indexing of elements in MongoDB, 157
  querying, 51–53
    $all conditional, 51
    $size conditional, 52
    $slice operator, 52
ascending sorts, 58
authentication, 118–120
  basics of, 118
  how it works, 120
  replication with, 142
autosharding, 143, 144
  (see also sharding)

B

background fsyncs, 117
background option for building indexes, 76
backups, 121–124
  restores from, using mongorestore, 123
  slave server, 124
  using fsync and lock commands, 123
  using mongodump, 122
BasicDBObject class (Java), 156
batch inserts, 23
binary data type, 17
binary files, storing (see GridFS)
blocking
  during index creation, 76
  for replication, 140
boolean type, 16
$box option, finding points within rectangle, 78
BSON, 179
bson gem and bson_ext gem, 165
buildInfo command, 95

C

capped collections, 97–101
  creating, 99
  natural sorting, 99
  properties and use cases, 98
  tailable cursors, 101
case-sensitivity in MongoDB, 6
$center option, finding points within a circle, 78
characters
  not allowed in database names, 9
  not allowed in keys, 6
chunks, 144
  distribution over shards, 145
  information about, 151

- splitting documents into (GridFS), 103
- chunkSize key (GridFS), 104
- client, shell as stand-alone MongoDB client, 12
- $cmd collection, 94
- code examples from this book, xv
- code type, 17
- collections, 5, 7
  - capped (see capped collections)
  - dropping all indexes on, 77
  - enabling sharding on, 148
  - fetching collections with inconvenient names, 15
  - fixed-size, 3
  - fully qualified names (namespaces), 9
  - grouped into databases by MongoDB, 9
  - help for, 14
  - inserting documents into, 23
  - MongoCollection, 161
  - names of, and index name size, limiting for namespaces, 76
  - naming, 8
  - reasons for separate collections, 7
  - schema-free, 7
  - subcollections, 8
  - system.indexes, 75
- collStats command, 95
- com.mongodb and com.mongodb.gridfs packages, 155
- com.mongodb.BasicDBObject class, 156
- com.mongodb.Mongo class, 155
- command line, starting MongoDB, 112
- command response, 94
- commands (see database commands)
- comparison operators, 47
- comparison order for types, 58
- compound geospatial indexes, 78
- compound unique indexes, 70
- condition, including for group command, 83
- conditionals, 47–49
  - $not metaconditional, 49
  - in OR queries, 48
  - rules for, 49
- config database, 9
- config servers, 147
  - setting up multiple, 149
  - starting, 147
- configuration file, 113
- connect( ) method, 177
- Connection class (PyMongo), 168
- Connection class (Ruby), 165
- connection pooling, 44
- connections, client requests and, 43
- convertToCapped command, 99
- count( ) function, 81
- create operations, MongoDB shell, 12
- createCollection command, 99
- Ctrl-C to stop MongoDB, 114
- currentOp command, 124
- cursor.hasNext( ) method, 56
- cursor.next( ) method, 56
- cursors, 56–63
  - advanced query options, 60
  - avoiding large skips, 59–60
  - client-side and database, 63
  - death and cleanup of, 63
  - getting consistent results, 61
  - limits, skips, and sorts, 57
  - tacking explain onto, 70
  - tailable, 101

D

- data directory, 10, 121
  - creating on Linux, Mac OS X, and Solaris, 175
  - creating on Windows, 174
- data files, 180
- data model, document-oriented, 1
- data processing, using slave servers, 137
- data types, 15–22
  - arrays, 19
  - basic, 16
  - changing key type with $set update modifier, 29
  - comparison order, 58
  - dates, 19
  - embedded documents, 20
  - _id and ObjectIds, 20
  - numbers, 18
  - type-sensitivity in MongoDB, 6
  - type-specific queries, 49–53
  - used by PHP driver, 161
  - values in documents, 6
- database commands, 93–97
  - how they work in MongoDB, 94
  - listing commands supported by MongoDB server, 95

  most frequently used MongoDB commands, 95
database references (DBRefs), 107–109
  definition of DBRef, 107
  driver support for, 108
  example schema, 107
  when to use, 108
databases
  enabling sharding on, 148
  help for database-level commands, 14
  listing for shards, 151
  in MongoDB, 9, 161
dates
  data types for, 19
  date type, 17
db variable, 12
db.addUser( ) method, 118
db.drop_collection( ) function, 26
db.eval( ) function, 104
db.getCollection( ) function, 15
db.listCommands( ) function, 95
db.printReplicationInfo( ) function, 141
db.runCommand( ) function, 93
db.version function, 15
DBObject interface, 156
decrementing, using $inc update modifier, 30
deletes
  all indexes on a collection, 77
  delete operations, MongoDB shell, 14
descending sorts, 58
diagnostic tools for replication, 141
dictionaries in Python, 169
distinct command, 81, 95
  listing unique filenames stored in GridFS, 104
document-oriented databases, 1
documents
  in Java, 156
  in MongoDB, 5
  in PHP, 161
  in Python, 168
  real-time analytics application using PyMongo, 169
  removing outdated, in analytics application, 171
  in Ruby, gotchas with, 165
dot notation (.), querying for embedded keys, 54
doubles, representing numbers in MongoDB, 18
drop command, 94, 95
dropDatabase command, 95
dropIndexes command, 76, 95
drop_collection( ) function, 26
duplicates
  dropping when creating indexes, 70
  duplicate keys, not allowed in MongoDB, 6
dynamic query optimizer (see query optimizer)

E

$each array modifier, 33
$elemMatch conditional, 55
embedded document type, 17
embedded documents, 20
  indexing keys in, 68
  querying on, 53
emit function, 87
encrypting traffic between clients and server, 121
errors
  catching normal errors with safe operations, 43
  getLastError command, 38, 93, 96
escaping special characters, 28
eval( ) function, 104
example applications (see application examples)
$exists conditional, 50
explain tool, 70–75
  output from query using index on two keys, 73
  output from query with index on single key, 72
  output from simple query on database without indexes, 71
extents, 180

F

features in MongoDB, 2
file storage, 3
files, storing (see GridFS)
files_id keys, 103
finalizers, 85
  passing finalize function to MapReduce, 90
find( ) method, 13, 45
  chaining limit( ) method to, 57

  chaining skip method to, 57
  geospatial queries, 77
  sorting returns, 58
  specifying keys to return, 46
findAndModify command, 39–41, 96
  values for keys, 40
findOne( ) method, 13
fire-and-forget functions, 41
fixed-size collections, 3
floating-point numbers, 16
  representation of integers as doubles in MongoDB, 18
forEach loop, using cursor class in, 56
fs.chunks collection, 103
fs.files collection, 103
fsync command, 123
functions
  JavaScript, defining and calling, 11
  printing JavaScript source code for, 14
  using as keys, 86

G

geoNear command, 78
geospatial indexes, 77–79
  assumption of a flat plane, spherical earth and, 79
  compound, 78
  values range, 77
getLastError command, 38, 93, 96
Github, 160
gps key, 77
GridFS, 101–104
  file storage, 101
  getting started with, using mongofiles, 102
  how it works, 103
  use of compound unique index, 70
  working with, from MongoDB drivers, 102
group command, 82
  component keys, 83
  condition for documents to be processed, 83
  using a finalizer, 85
  using function as key, 86
$gt (greater than) conditional, 47
$gte (greater than or equal to) conditional, 47

H

handshake command, 140
hardware failures, 41
hashes in Ruby, 165
hasNext( ) method, 56
help command, 14
hint tool, 75
HTTP admin interface, 10, 115

I

_id keys, 20
  autogeneration of, 22
  DBRefs versus, 108
  GridFS, 103
  unique indexes on, 69
immortal function, 63
importing data, using batch inserts, 24
$in conditional
  type-specific queries, 50
  using in OR queries, 48
$inc update modifier, 28
  incrementing a counter, 169
  incrementing and decrementing with, 30
indexes
  administration of, 75
    adding, removing, and dropping all indexes, 76
  $all conditional and, 159
  for collections of documents in MongoDB, 7
  compound geospatial indexes, 78
  dropIndexes command, 95
  forcing Mongo to use indexes you want for a query, 75
  real-time analytics using PyMongo (example), 170
  unique, 69
    compound unique indexes, 70
    dropping duplicates, 70
  uniquely identifying, 69
indexing, 2, 65–79
  on all keys in your query, 66
  arrays, selecting elements by index, 34
  chemical search engine using Java driver (example), 157
    creating compound index, 158
  disadvantage of indexes, 67
  geospatial, 77–79
  keys in embedded documents, 68
  on multiple keys, considering index direction, 66

    - questions to consider when creating indexes, 68
    - scaling indexes, 68
    - on single key used in query, 65
    - for sorts, 69
    - using explain, 70–75
    - using hint, 75
- insert( ) method, 12
- inserts, 23
    - batch inserts, 23
    - into capped collections, 98
    - insert( ) method, 12
    - interleaved inserts/queries, 44
    - internals and implications, 24
    - safe inserts for documents with duplicate value for unique key, 69
    - safe operations, 42
    - upserts, 36
- installation, MongoDB, 173–176
    - choosing a version, 173
    - POSIX install on Linux, Mac OS X, and Solaris, 175
    - Windows install, 174
- integers
    - 32- and 64-bit, 16
    - basic data types and, 16
    - representation as floating-point numbers, 18
- isMaster command, 96

J

- Java
    - documentation for Java driver and articles, 156
    - search engine for chemical compounds, 155–159
- java.util.ArrayList, 158
- JavaScript
    - Date class, 19
    - executing as part of query with $where clause, 55
    - MongoDB shell, 11
    - server-side execution, disallowing, 121
    - server-side scripting, 104–107
        - db.eval( ) function, 104
        - security and, 106
        - stored JavaScript, 105
    - stored, in MongoDB, 3
- JSON (JavaScript Object Notation), 16
    - data types, 16

K

- $keyf key, 86
- key/value pairs
    - functions as keys, 86
    - keys in MapReduce operations, 90
    - in MongoDB documents, 6
    - specifying keys to return with find method, 46
- keys
    - removing with $unset update modifier, 29
    - setting value with $set update modifier, 29
- kill command, 114

L

- latitude and longitude in geospatial indexing, 77
- length key (GridFS), 103
- libraries (JavaScript), leveraging in MongoDB shell, 11
- limits for query results, 57
- Linux
    - installing MongoDB, 175
    - installing PHP driver, 161
- listCommands command, 95, 96
- listDatabases command, 96
- local database, 9, 139
    - user for slave and master server, 142
- local.system.replset namespace, 133
- local.system.users namespace, 142
- localhost, running shard on, 148
- locking
    - fsync command, holding a lock, 123
    - information about, 117
- logging
    - creating function for JavaScript code, 105
    - inspecting MongoDB log after installation, 113
    - use of capped collections for, 99
- $lt (less than) conditional, 47

M

- Mac OS X
    - installing MongoDB, 175
    - installing PHP driver, 160
- manual sharding, 143

map collection, finding all documents in order by distance from a point, 77
map step (MapReduce), 86
  getting all keys of all documents in collection, 87
MapReduce, 3, 86–92
  finalize function passed to, 90
  finding all keys in a collection, 87
    MapReduce function in MongoDB, 88
    metainformation in document returned, 88
  getting more output from, 92
  keeping output collections from, 90
  optional keys that can be passed to, 90
  running on subset of documents, 91
  using scope with, 92
master-slave replication, 127–130
  adding and removing sources, 129
  options, 128
Math.random function, 60
maximum value type, 17
md5 key (GridFS), 104
memory management, 3
  information on memory from serverStatus, 117
  keeping index in memory, 68
memory-mapped storage engine, 181
metadata for namespaces, 180
minimum value type, 17
modifiers, update (see update modifiers)
mongo (shell), 177–178, 177
  (see also shell)
  --nodb option, 177
  utilities, 178
Mongo class (Java), 155
mongo gem, installing, 164
Mongo::Connection class (Ruby), 165
mongod executable
  --master option, 128
  --replSet option, 132
  --rest option, 115
  running, 10
  --slave option, 128
  startup options, 112
    --bindip, 121
    --config, 112
    --dbpath, 112, 176
    --fork, 112
    --logpath, 112
    --nohttpinterface, 115
    --noscripting, 121
    --oplogSize, 139
    --port, 112, 115
    --repair, 125
  stopping, 10, 114
mongod.exe, installation options, 175
MongoDB
  advantages offered by, 1–4
  data types, 15
  getting and starting, 10
  installing, 173–176
  shell, 11–15
MongoDB Java Language Center, 156
mongodump utility, 122
mongofiles utility, 102
mongorestore utility, 123
mongos routing process, 144, 147
  connecting to, 148
  running multiple, 150
mongostat utility, 118
monitoring, 114–118
  server health and performance, using serverStatus, 116
  third-party plug-ins for, 118
  using admin interface, 115
  using mongostat for serverStatus output, 118

N

namespaced subcollections, 8
namespaces, 9
  and extents, 180
  for indexes, 76
naming
  collections, 8
  databases, 9
  indexes, 69
natural sorts, 99
$ne (not equal) conditional, 47
$near conditional, 77
news aggregator using PHP (example), 160–164
next( ) method, using on cursor, 56
$nin (not in) conditional, 48
nodes
  master and slave (see master-slave replication)
  types in replica sets, 133

$not conditional, 49
null character (\0), 8
null type, 16
  queries for, 49
numbers, data types for, 18

O

object id type, 17
ObjectIds, 20–22
oplog, 99, 138
  changing size of, 142
  getting information about, 141
OR queries, 48
org.bson.DBObject interface, 156

P

package manager, installing MongoDB from, 176
pagination
  combining find, limit, and sort methods for, 58
  query results without skip, 59
partitioning, 143
passive nodes, 134
performance, 3
  index creation and, 76
  indexes and, 67
  price of safe operations, 42
  speed of update modifiers, 35
Perl
  $ (dollar sign) in MongoDB update modifiers, 28
  Perl Compatible Regular Expression (PCRE) library, 50
PHP
  $ (dollar sign) in MongoDB update modifiers, 28
  news aggregator application (example), 160–164
  using tailable cursor in, 101
PID (process identifier) for ObjectId-generating process, 22
ping command, 96
plain queries, 60
point-in-time data snapshots, 123
points
  finding document in map collection by order of distance from, 77
  finding in a shape, 78
$pop array modifier, 34
positional operator ($), 34
POSIX install (MongoDB), 175
preallocation of data files, 180
primary, 130
primary node, 133
  failure of, and election of new primary, 135
printReplicationInfo( ) function, 141
printShardingStatus( ) function, 152
priority key, 134
processes
  PID for ObjectId-generating process, 22
  status of, 39
$pull array modifier, 34
$push array modifier, 32
PyMongo
  DBRef type, 108
  real-time analytics application (example), 168–171
pymongo.connection.Connection class, 168
Python Package Index, 168

Q

queries, 45–63
  commands implemented as, 94
  cursors, 56–63
    advanced query options, 60
    avoiding large skips, 59–60
    getting consistent results, 61
    limits, skips, and sorts, 57
  find( ) method, 45
    specifying keys to return, 46
  geospatial, 77–79
  handling on slave servers, 137
  matching more complex criteria, 47
    conditionals, 47
    $not conditional, 49
    OR queries, 48
    rules for conditionals, 49
  querying on embedded documents, 53
  restrictions on, 47
  type-specific, 49–53
    arrays, 51–53
    null type, 49
    regular expressions, 50
  $where clauses in, 55
query optimizer, 3
  choosing index to use, 75

  - reordering query terms to take advantage of indexes, 67
- quotation marks in strings, 28

R

- random function, 60
- range queries, using conditionals, 47
- read scaling with slave servers, 137
- real-time analytics, MongoDB for, 169
- reduce step (MapReduce), 86
  - calling reduce in MongoDB (example), 87
- references to documents, uniquely identifying (see database references)
- regular expressions, 50
  - MongoRegex, 161
  - regular expression type, 17
- relational databases
  - document-oriented databases versus, 1
  - features not available in MongoDB, 3
- remove( ) function, 14
  - query document as parameter, 25
- removes, 25
  - safe operations, 42
  - speed of, 25
- removeshard command, 152
- renameCollection command, 96
- repair of corrupt data files, 124
- repairDatabase command, 96
- repairDatabase( ) method, 125
- replica sets, 130
  - failover and primary election, 135
  - initializing, 132
    - keys in initialization document, 133
  - nodes in, 133
  - shards as, 150
- replication, 127–142
  - administration of, 141
    - authentication, 142
    - changing oplog size, 142
    - diagnostics, 141
  - blocking for, 140
  - master-slave, 127–130
  - oplog, 138
  - performing operations on a slave, 136
  - replica sets (see replica sets)
  - replication state and local database, 139
  - syncing slave to master node, 139
- replSetInitiate command, 132
- requests
  - batch inserts and, 24
  - connections and, 43
- reserved database names, 9
- REST support, 115
- restores, 121
  - (see also administration, backup and repair)
  - using mongorestore utility, 123
- resync command, 139
- retrieve operations, MongoDB shell, 13
- routing process (mongos), 144, 147, 148
- Ruby
  - custom submission forms application (example), 164–167
  - object mappers and using MongoDB with Rails, 167
- RubyGems, 164
- runCommand( ) function, 93

S

- safe operations, 42
  - catching normal errors, 43
- save function, 38
- scaling with MongoDB, 2
- schema-free collections, 7
- schema-free MongoDB, 2
- schemas
  - chemical search engine using Java driver (example), 156
  - example schema using DBRefs, 107
  - real-time analytics application using PyMongo, 169
- scope, using with MapReduce, 92
- search engine for chemicals, using Java driver (example), 155–159
- secondaries, 130
- secondary nodes, 133
- security
  - authentication, 118–120
  - execution of server-side JavaScript, 106
  - other considerations, 121
- server-side scripting, 104–107
  - disallowing server-side JavaScript execution, 121
- servers
  - database server offloading processing and logic to client side, 3
  - for production sharding, 150
- serverStatus command, 96, 116–118

  - information from, printing with mongostat, 118
- $set update modifier, 29
- shapes, finding all documents within, 78
- shardCollection command, 149
- sharding, 143–153
  - administration, 150–153
    - config collections, 150
    - printShardingStatus command, 152
    - removeshard command, 152
  - autosharding in MongoDB, 144
  - database variable pointing to config database, 178
  - defined, 143
  - production configuration, 149
  - setting up, 147
    - sharding data, 148
    - starting servers, 147
  - shard keys, 145
    - effects of shard keys on operations, 146
    - existing collection, 145
    - incrementing, versus random shard keys, 146
  - when to shard, 145
- shards, 144
  - adding a shard, 148
  - defined, 147
  - listing in shards collection, 150
  - replica sets as, 150
- shell, 5, 11–15, 177–178
  - connecting to database, 177
  - create operations, 12
  - creating a cursor, 56
  - delete operations, 14
  - figuring out what functions are doing, 14
  - help with, 14
  - JavaScript functions provided by, autogenerated API, 15
  - MongoDB client, 12
  - repairing single database on running server, 125
  - retrieve operations, 13
  - running, 11
  - running scripts, 37
  - save function, 38
  - starting without connecting to database, 177
  - update operations, 13
  - utilities, 178
- shutdown command, 114
- SIGINT or SIGTERM signal, 114
- skips
  - avoiding large skips, 59–60
    - finding a random document, 59
  - skipping query results, 57
- slave nodes
  - adding and removing sources, 129
  - secondaries, in replica sets, 130
  - setting up, 128
  - syncing to master node, 139
- slave servers
  - backups, 124
  - performing operations on, 136
- $slice operator, 52
- snapshots of data, 123
- Solaris, installing MongoDB, 175
- sorting
  - find( ) method results, 58
  - indexing for, 69
  - natural sorts, 99
- sources collection, 140
- sources for slave nodes, 129
- standard nodes, 134
- starting MongoDB, 111–113
  - file-based configuration, 113
  - from command line, 112
- status of processes, 39
- stopping MongoDB, 114
- storage engine, 113, 181
- strings
  - matching with regular expressions, 50
  - string type, 16
- subcollections, 8
  - accessing using variables, 15
- submission forms (custom), using Ruby, 164–167
- symbol type, 17
- syncedTo, 140
- syncing slave to master node, 139
- system prefix, collection names, 8
- system.indexes collection, 75
- system.js collection, 105
- system.namespaces collection, lists of index names, 76

T

- table scans, 66
- tailable cursors, 101

third-party plug-ins for monitoring, 118
timestamps
  in ObjectIds, 21
  stored in syncedTo, 140
  uploadDate in GridFS, 104
trees of comments (news aggregator example), 162
type-sensitivity in MongoDB, 6

U

undefined type, 17
Unix, installing PHP driver, 161
update modifiers, 27–36
  $ positional operator, 34
  $inc, 28, 30
  $set, 29
  $unset, 29
  array modifiers, 31–34
    $addToSet, 33
    $ne, 32
    $pop, 34
    $pull, 34
    $push, 32
  speed of, 35
updates, 26
  replacing a document, 26
  returning updated documents, 39–41
  safe operations, 42
  update operations, MongoDB shell, 13
  updating multiple documents, 38
  upserts, 36
  using modifiers, 27–36
uploadDate key (GridFS), 104
upserts
  real-time analytics application (example), 170
  real-time analytics using MongoDB, 169
  save shell helper function, 38

V

values in documents, 6
variables
  JavaScript, in shell, 12
  using to access subcollections, 15
versions, MongoDB, 173
voting, implementing, 164

W

web page for this book, xvi
web pages
  categorizing, using MapReduce, 89
  tracking views with analytics application, 168–170
$where clauses in queries, 55
Windows systems
  installing MongoDB, 174
  installing PHP driver, 160
  running mongod executable, 10
wire protocol, 180
$within conditional, 78
wrapped queries, 60

Download from Wow! eBook <www.wowebook.com>

About the Authors

Kristina Chodorow is a core contributor to the MongoDB project. She has worked on the database server, PHP driver, Perl driver, and many other MongoDB-related projects. She has given talks at conferences around the world, including OSCON, LinuxCon, FOSDEM, and Latinoware, and maintains a website about MongoDB and other topics at *http://www.snailinaturtleneck.com*. She works as a software engineer for 10gen and lives in New York City.

Michael Dirolf, also a software engineer at 10gen, is the lead maintainer for PyMongo (the MongoDB Python driver), and the former maintainer for the MongoDB Ruby driver. He has also contributed to the MongoDB server and worked on several other related libraries and tools. He has given talks about MongoDB at major conferences around the world, but currently resides in New York City.

Colophon

The animal on the cover of *MongoDB: The Definitive Guide* is a mongoose lemur, a member of a highly diverse group of primates endemic to Madagascar. Ancestral lemurs are believed to have inadvertently traveled to Madagascar from Africa (a trip of at least 350 miles) by raft some 65 million years ago. Freed from competition with other African species (such as monkeys and squirrels), lemurs adapted to fill a wide variety of ecological niches, branching into the almost 100 species known today. These animals' otherworldly calls, nocturnal activity, and glowing eyes earned them their name, which comes from the *lemures* (specters) of Roman myth. Malagasy culture also associates lemurs with the supernatural, variously considering them the souls of ancestors, the source of taboo, or spirits bent on revenge. Some villages identify a particular species of lemur as the ancestor of their group.

Mongoose lemurs (*Eulemur mongoz*) are medium-sized lemurs, about 12 to 18 inches long and 3 to 4 pounds. The bushy tail adds an additional 16 to 25 inches. Females and young lemurs have white beards, while males have red beards and cheeks. Mongoose lemurs eat fruit and flowers and they act as pollinators for some plants; they are particularly fond of the nectar of the kapok tree. They may also eat leaves and insects.

Mongoose lemurs inhabit the dry forests of northwestern Madagascar. One of the two species of lemur found outside of Madagascar, they also live in the Comoros Islands (where they are believed to have been introduced by humans). They have the unusual quality of being cathemeral (alternately wakeful during the day and at night), changing their activity patterns to suit the wet and dry seasons. Mongoose lemurs are threatened by habitat loss and they are classified as a vulnerable species.

The cover image is from Lydekker's *Royal Natural History*. The cover font is Adobe ITC Garamond. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSansMonoCondensed.