

O'REILLY®

3rd Edition

Early Release

RAW & UNEDITED

# MongoDB

## The Definitive Guide

POWERFUL AND SCALABLE DATA STORAGE



Shannon Bradshaw &  
Kristina Chodorow

# MongoDB: The Definitive Guide

THIRD EDITION

**Shannon Bradshaw and Kristina Chodorow**



Beijing • Boston • Farnham • Sebastopol • Tokyo



## **MongoDB: The Definitive Guide**

by Shannon Bradshaw and Kristina Chodorow

Copyright © 2019 Shannon Bradshaw and Kristina Chodorow. 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://oreilly.com/safari>). For more information, contact our corporate/institutional sales department: 800-998-9938 or [corporate@oreilly.com](mailto:corporate@oreilly.com).

- Editor: Michele Cronin
- Production Editor: Kristen Brown
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Rebecca Demarest
- April 2019: Third Edition



### **Revision History for the Third Edition**

- 2017-09-29: Tenth Early Release
- 2017-10-20: Eleventh Early Release
- 2017-12-06: Twelfth Early Release
- 2018-01-17: Thirteenth Early Release
- 2018-03-27: Fourteenth Early Release
- 2018-09-06: Fifteenth Early Release
- 2018-10-09: Sixteenth Early Release
- 2018-10-15: Seventeenth Early Release

- 2018-11-28: Eighteenth Early Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781449344689> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *MongoDB: The Definitive Guide*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-95446-1



# **Part I. Introduction to MongoDB**

---

# Chapter 1. Introduction

---

MongoDB is a powerful, flexible, and scalable general-purpose database. It combines the ability to scale out with features such as secondary indexes, range queries, sorting, aggregations, and geospatial indexes. This chapter covers the major design decisions that made MongoDB what it is.

## Ease of Use

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.

A document-oriented database replaces 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 naturally into the way developers in modern object-oriented languages think about their data.

There are also no predefined schemas: a document’s keys and values are not of fixed types or sizes. Without a fixed schema, adding or removing fields as needed becomes easier. Generally, this makes development faster as developers can quickly iterate. It is also easier to experiment. Developers can try dozens of models for the data and then choose the best one to pursue.

## Designed to Scale

Data set sizes for applications are growing at an incredible pace. Increases in available bandwidth and cheap storage 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. The alternative is to scale *out*: to add storage space or increase throughput for read and write operations, buy additional servers and add them to your cluster. This is both cheaper and more scalable; however, it is more difficult to administer a thousand machines than it is to care for one.

MongoDB was designed to scale out. The document-oriented data model makes it easier to split data across multiple servers. MongoDB automatically takes care of balancing data and load across a cluster, redistributing documents automatically and routing reads and writes to the correct machines.

The topology of a MongoDB cluster or whether, in fact, there is a cluster versus a single node at the other end of a database connection is transparent to the application. This allows developers to focus on programming the application, not scaling it. Likewise if the topology of an existing deployment needs to change in order to, for example, scale to support greater load, the application logic can remain the same.

## **Rich with Features...**

MongoDB is a general-purpose database, so aside from creating, reading, updating, and deleting data, it provides most of the features you would expect from a DBMS and many others that set it apart:

### Indexing

MongoDB supports generic secondary indexes and provides unique, compound, geospatial, and full-text indexing capabilities as well. Secondary indexes on hierarchical structures such as nested documents and arrays are also supported and enable developers to take full advantage of the ability to model in ways that best suit their applications.

### Aggregation

MongoDB provides an aggregation framework based on the concept of data processing pipelines. Aggregation pipelines allow you to build complex analytics engines by processing data through a series of relatively simple stages on the server side and with the full advantage of database optimizations.

### Special collection and index types

MongoDB supports time-to-live (TTL) collections for data that should expire at a certain time, such as sessions and fixed-size (capped) collections, for holding recent data, such as

logs. MongoDB also supports partial indexes limited to only those documents matching a criteria filter in order to increase efficiency and reduce the amount of storage space required.

## 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 complex multirow transactions. MongoDB also only supports joins in a very limited way through use of the \$lookup aggregation operator introduced in the 3.2 release. MongoDB's treatment of multirow transactions and joins were architectural decisions to allow for greater scalability, because both of those features are difficult to provide efficiently in a distributed system.

## **...Without Sacrificing Speed**

Performance is a driving objective for MongoDB which has shaped much of its design. It uses opportunistic locking in its WiredTiger storage engine to maximize concurrency and throughput. It uses as much of RAM as it can as its cache and attempts to automatically choose the correct indexes for queries. In short, almost every aspect of MongoDB was designed to maintain high performance.

Although MongoDB is powerful, incorporating many features from relational systems, it is not intended to do everything that a relational database does. For some functionality, 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.

## **Let's Get Started**

Throughout this 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 powerful but 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 and is roughly equivalent to a row in a relational database management system (but much more expressive).
- Similarly, a *collection* can be thought of as a table with a dynamic schema.
- A single instance of MongoDB can host multiple independent *databases*, each of which can have its own collections.
- Every document has a special key, "`_id`", that is unique within a collection.
- MongoDB is distributed with a simple but powerful tool called the `mongo shell`. The `mongo shell` provides built-in support for administering MongoDB instances and manipulating data using the MongoDB query language. It is also a fully-functional JavaScript interpreter which enables users to create and load their own scripts for a variety of purposes.

## Documents

At the heart of MongoDB is the *document*: an ordered set of keys with associated values. The representation of a document varies 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!", "views" : 3}
```

As you can see from the example above, 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](#)”). In this example the value for "greeting" is a string, whereas the value for "views" 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.

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

```
{ "count" : 5 }
{ "count" : "5" }
```

as are as these:

```
{ "count" : 5 }
{ "Count" : 5 }
```

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!"}
```

Key/value pairs in documents are ordered: { "x" : 1, "y" : 2 } is not the same as { "y" : 2, "x" : 1 }. Field order does not usually matter and you should not design your schema to depend on a certain ordering of fields (MongoDB may reorder them). This text will note the special cases where field order is important.

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, when necessary.

## 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.

### Dynamic Schemas

Collections have *dynamic schemas*. 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!", "views": 3}  
{"signoff": "Good night, and good luck"}
```

Note that the previous documents have different keys, different numbers of keys, and values of different types. Because any document can be put into any collection, the question often arises: “Why do we need separate collections at all?” 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 type 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`” field in each document that specified whether the document was a “skim,” “whole,” or “chunky monkey,” it would be much slower to find those three values in a single collection than to have three separate collections and query the correct collection.
- 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.

There are sound reasons for creating a schema and for grouping related types of documents together. While not required by default, defining schemas for your application is good practice and can be enforced through the use of MongoDB’s documentation validation functionality and object-document mapping libraries available for many programming languages.

## 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 internal 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 6](#) for more information about GridFS).
- 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 good way to organize data in MongoDB for many use cases.

## Databases

In addition to grouping documents by collection, MongoDB groups collections into *databases*. A single instance of MongoDB can host several databases, each grouping together zero or more collections. 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). Basically, stick with alphanumeric ASCII.
- Database names are case-sensitive, even on non-case-sensitive filesystems. To keep things simple, try to just use lowercase characters.
- 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 but which have special semantics. These are as follows:

### *admin*

The *admin* database plays a role in authentication and authorization. In addition, access to this database is required for some administrative operations.

### *local*

This database stores data specific to a single server. In replica sets, *local* stores data used in the replication process. The local database itself is never replicated. See [Chapter 8](#) for more

information about replication and the local database).

### config

Sharded MongoDB clusters (see Chapter 12), use the *config* database to store information about each shard.

By concatenating a database name with a collection in that 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 fewer than 100 bytes long.

## Getting and Starting MongoDB

MongoDB is almost always run as a network server that clients can connect to and perform operations on. [Download MongoDB](#) and decompress it. To start the server, run the `mongod` executable:

```
$ mongod
2016-04-27T22:15:55.871-0400 I CONTROL  [initandlisten] MongoDB starting :
2016-04-27T22:15:55.872-0400 I CONTROL  [initandlisten] db version v3.3.5
2016-04-27T22:15:55.872-0400 I CONTROL  [initandlisten] git version:
34e65e5383f7ea1726332cb175b73077ec4a1b02
2016-04-27T22:15:55.872-0400 I CONTROL  [initandlisten] allocator: system
2016-04-27T22:15:55.872-0400 I CONTROL  [initandlisten] modules: none
2016-04-27T22:15:55.872-0400 I CONTROL  [initandlisten] build environment:
2016-04-27T22:15:55.872-0400 I CONTROL  [initandlisten]   distarch: x86_
2016-04-27T22:15:55.872-0400 I CONTROL  [initandlisten]   target_arch: x86_
2016-04-27T22:15:55.872-0400 I CONTROL  [initandlisten] options: {}
2016-04-27T22:15:55.889-0400 I JOURNAL  [initandlisten] journal dir=/data/
2016-04-27T22:15:55.889-0400 I JOURNAL  [initandlisten] recover : no journal
present, no recovery needed
2016-04-27T22:15:55.909-0400 I JOURNAL  [durability] Durability thread sta
2016-04-27T22:15:55.909-0400 I JOURNAL  [journal writer] Journal writer th
2016-04-27T22:15:55.909-0400 I CONTROL  [initandlisten]
2016-04-27T22:15:56.777-0400 I NETWORK  [HostnameCanonicalizationWorker] S
canonicalization worker
2016-04-27T22:15:56.778-0400 I FTDC     [initandlisten] Initializing full-
data capture with directory '/data/db/diagnostic.data'
2016-04-27T22:15:56.779-0400 I NETWORK  [initandlisten] waiting for connec
27017
```

Or if you're on Windows, run this:

```
> mongod.exe
```

## TIP

For detailed information on installing MongoDB on your system, see the appropriate [installation tutorial](#) in the MongoDB Documentation.

When run with no arguments, `mongod` will use the default data directory, `/data/db/` (or `\data\db\` on the current volume on Windows). 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.

On startup, 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. The server will fail to start if the port is not available—the most common cause of this is another instance of MongoDB that is already running. You should always secure your `mongod` instances.

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

## Introduction to the MongoDB Shell

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

### Running the Shell

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

```
$ mongo
MongoDB shell version: 3.3.5
connecting to: test
>
```

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. If the statement is not complete, the shell will allow you to continue writing it on the next line. Pressing Enter three times in a row will cancel the half-formed command and get you back to the >-prompt.

## 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 standalone 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 your MongoDB server through the shell.

To see the database `db` is currently assigned to, type in `db` and hit Enter:

```
> db
test
```

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 video
switched to db video
```

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

```
> db
movies
```

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.

Access collections from the `db` variable. For example, `db.movies` returns the *movies* 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 `insertOne` function adds a document to a collection. For example, suppose we want to store a movie. First, we'll create a local variable called `movie` that is a JavaScript object representing our document. It will have the keys "title", "director", and "year" (the year it was release):

```
> movie = {"title" : "Star Wars: Episode IV - A New Hope",
... "director" : "George Lucas",
... "year" : 1977}
{
    "title" : "Star Wars: Episode IV - A New Hope",
    "director" : "George Lucas",
    "year" : 1977
}
```

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

```
> db.blog.insertOne(movie)
{
```

```
        "acknowledged" : true,
        "insertedId" : ObjectId("5721794b349c32b32a012b11")
    }
```

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

```
> db.movies.find()
{
    "_id" : ObjectId("5721794b349c32b32a012b11"),
    "title" : "Star Wars: Episode IV - A New Hope",
    "director" : "George Lucas",
    "year" : 1977
}
```

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 the sudden appearance of the `"_id"` field is explained at the end of this chapter.

## READ

`find` and `findOne` can be used to query a collection. If we just want to see one document from a collection, we can use `findOne`:

```
> db.movies.findOne()
{
    "_id" : ObjectId("5721794b349c32b32a012b11"),
    "title" : "Star Wars: Episode IV - A New Hope",
    "director" : "George Lucas",
    "year" : 1977
}
```

`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 `updateOne`. `updateOne` 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 reviews for our movie we created earlier. We'll need to add an array of reviews as the value for a new key in our document.

To perform the update, we'll need to use an update operator, `$set`.

```
> db.movies.update({title : "Star Wars: Episode IV - A New Hope"},  
...{$set : {reviews: []}})  
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

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

```
> db.movies.find()  
{  
    "_id" : ObjectId("5721794b349c32b32a012b11"),  
    "title" : "Star Wars: Episode IV - A New Hope",  
    "director" : "George Lucas",  
    "year" : 1977,  
    "reviews" : [ ]  
}
```

See ["Updating Documents"](#) for detailed information on updating documents.

## DELETE

`deleteOne` and `deleteMany` permanently delete documents from the database. Both methods take filter document specifying criteria for removal. For example, this would remove the movie we just created:

```
> db.movies.deleteOne({title : "Star Wars: Episode IV - A New Hope"})
```

Use `deleteMany` to delete all documents matching a filter.

## 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 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 (their website 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. The most common types are:

null

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

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

boolean

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

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

number

The shell defaults to using 64-bit floating point numbers. Thus, these numbers look “normal” in the shell:

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

or:

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

For integers, use the NumberInt or NumberLong classes, which represent 4-byte or 8-byte signed integers, respectively.

```
{ "x" : NumberInt("3") }
```

```
{ "x" : NumberLong("3") }
```

## string

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

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

## date

MongoDB stores dates as 64-bit integers representing milliseconds since the Unix epoch (January 1, 1970). The time zone is not stored:

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

## regular expression

Queries can use regular expressions using JavaScript's regular expression syntax:

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

## 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" } }
```

## object id

An object id is a 12-byte ID for documents. See the section “[\\_id and ObjectIds](#)” for details:

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

There are also a few less common types that you may need, including:

## binary data

Binary data is a string of arbitrary bytes. It cannot be manipulated from the shell. Binary data is the only way to save non-UTF-8 strings to the database.

## code

MongoDB also makes it possible to store arbitrary JavaScript in queries and documents.

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

There are a few types that are mostly used internally (or superseded by other types). These will be described in the text as needed.

## Dates

In JavaScript, the `Date` class 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](#).

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

Documents can be used as the *value* for a key. This is called an embedded document. Embedded documents can be used to organize data in a more natural way than just a flat structure of key/value pairs.

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 an embedded document with its own key/value pairs for "street", "city", and "state".

As with arrays, MongoDB “understands” the structure of embedded documents and is able to reach inside 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 representation of information.

The flip side of this is that 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 with an “`_id`” of 123.

## **OBJECTIDS**

`ObjectId` is the default type for “`_id`”. The `ObjectId` class is designed to be lightweight, while still being easy to generate in a globally unique way across different machines.

MongoDB’s distributed nature is the main reason why it uses `ObjectIds` 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 to be a distributed database, it was important to be able to generate unique identifiers in a sharded environment.

`ObjectIds` 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 `ObjectIds` 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 `ObjectIds` are created. The 12 bytes of an `ObjectId` are generated as follows:

0            1            2            3            4    5    6    7    8    9    10    11

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 `ObjectIds` will sort in *roughly* insertion order. This is not a strong guarantee but does have some nice properties, such as making `ObjectIds` 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 `ObjectIds`, some users worry that their servers will need to have synchronized clocks. Although synchronized clocks are a good idea for other reasons, the actual timestamp doesn't matter to `ObjectIds`, 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 `ObjectIds`.

To provide uniqueness among different processes generating `ObjectIds` 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 `ObjectIds` 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.

## Using the MongoDB Shell

This section covers how to use the shell as part of your command line toolkit, customize it, and use some of its more advanced functionality.

Although we connected to a local *mongod* instance above, you can connect your shell to any MongoDB instance that your machine can reach. To connect to a *mongod* on a different machine or port, specify the hostname, port, and database when starting the shell:

```
$ mongo some-host:30000/myDB
MongoDB shell version: 3.3.5
connecting to: some-host:30000/myDB
>
```

db will now refer to *some-host:30000*'s myDB database.

Sometimes it is handy to not connect to a *mongod* at all when starting the *mongo* shell. If you start the shell with --nodb, it will start up without attempting to connect to anything:

```
$ mongo --nodb
MongoDB shell version: 3.3.5
>
```

Once started, you can connect to a *mongod* at your leisure by running new Mongo (*hostname*):

```
> conn = new Mongo("some-host:30000")
connection to some-host:30000
> db = conn.getDB("myDB")
myDB
```

After these two commands, you can use db normally. You can use these commands to connect to a different database or server at any time.

## 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. For MongoDB-specific functionality, the shell includes built-in help that can be accessed by typing **help**:

```
> help
    db.help()                      help on db methods
    db.mycoll.help()                help on collection methods
    sh.help()                       sharding helpers
```

```
...  
show dbs  
show collections  
show users  
...  
show database names  
show collections in current database  
show users in current database
```

Database-level help is provided by `db.help()` and collection-level help by `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.movies.updateOne  
function (filter, update, options) {  
    var opts = Object.extend({}, options || {});  
  
    // Check if first key in update statement contains a $  
    var keys = Object.keys(update);  
    if (keys.length == 0) {  
        throw new Error("the update operation document must contain at  
        least one atomic operator");  
    }  
    ...
```

## Running Scripts with the Shell

In addition to using the shell interactively, you can also pass the shell JavaScript files to execute. Simply pass in your scripts at the command line:

```
$ mongo script1.js script2.js script3.js  
MongoDB shell version: 3.3.5  
connecting to: test  
I am script1.js  
I am script2.js  
I am script3.js  
$
```

The `mongo` shell will execute each script listed and exit.

If you want to run a script using a connection to a non-default host/port `mongod`, specify the address first, then the script(s):

```
$ mongo --quiet server-1:30000/foo script1.js script2.js script3.js
```

This would execute the three scripts with db set to the *foo* database on *server-1:30000*. As shown above, any command line options for running the shell go before the address.

You can print to stdout in scripts (as the scripts above did) using the `print()` function. This allows you to use the shell as part of a pipeline of commands. If you're planning to pipe the output of a shell script to another command use the `--quiet` option to prevent the "MongoDB shell version..." banner from printing.

You can also run scripts from within the interactive shell using the `load()` function:

```
> load("script1.js")
I am script1.js
>
```

Scripts have access to the `db` variable (as well as any other globals). However, shell helpers such as `"use db"` or `"show collections"` do not work from files. There are valid JavaScript equivalents to each of these, as shown in [Table 2-1](#).

*Table 2-1. JavaScript equivalents to shell helpers*

Helper	Equivalent
<code>use video</code>	<code>db.getSiblingDB("video")</code>
<code>show dbs</code>	<code>db.getMongo().getDBs()</code>
<code>show collections</code>	<code>db.getCollectionNames()</code>

You can also use scripts to inject variables into the shell. For example, we could have a script that simply initializes helper functions that you commonly use. The script below, for instance, may be helpful for the replication and sharding sections of the book. It defines a function, `connectTo()`, that connects to the locally-running database on the given port and sets `db` to that connection:

```
// defineConnectTo.js
/**
 * Connect to the database on the given port and return the db object.
 * @param {Number} port - The port to connect to.
 * @return {Object} db - The database object.
 */
```

```

* Connect to a database and set db.
*/
var connectTo = function(port, dbname) {
  if (!port) {
    port = 27017;
  }

  if (!dbname) {
    dbname = "test";
  }

  db = connect("localhost:"+port+"/"+dbname);
  return db;
};

```

If we load this script in the shell, `connectTo` is now defined:

```

> typeof connectTo
undefined
> load('defineConnectTo.js')
> typeof connectTo
function

```

In addition to adding helper functions, you can use scripts to automate common tasks and administrative activities.

By default, the shell will look in the directory that you started the shell in (use `run ("pwd")` to see what directory that is). If the script is not in your current directory, you can give the shell a relative or absolute path to it. For example, if you wanted to put your shell scripts in `~/my-scripts`, you could load `defineConnectTo.js` with `load ("~/myScripts/defineConnectTo.js")`. Note that `load` cannot resolve `~`.

You can use `run ()` to run command-line programs from the shell. Pass arguments to the function as parameters:

```

> run("ls", "-l", "/home/myUser/my-scripts/")
sh70352| -rw-r--r-- 1 myUser myUser 2012-12-13 13:15 defineConnectTo.js
sh70532| -rw-r--r-- 1 myUser myUser 2013-02-22 15:10 script1.js
sh70532| -rw-r--r-- 1 myUser myUser 2013-02-22 15:12 script2.js
sh70532| -rw-r--r-- 1 myUser myUser 2013-02-22 15:13 script3.js

```

This is of limited use, generally, as the output is formatted oddly and it doesn't support pipes.

## Creating a `.mongorc.js`

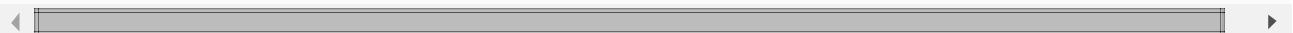
If you have frequently-loaded scripts you might want to put them in your `mongorc.js` file. This file is run whenever you start up the shell.

For example, suppose we would like the shell to greet us when we log in. Create a file called `.mongorc.js` in your home directory, and then add the following lines to it:

```
// .mongorc.js

var compliment = ["attractive", "intelligent", "like Batman"];
var index = Math.floor(Math.random() * 3);

print("Hello, you're looking particularly "+compliment[index]+" today!");
```



Then, when you start the shell, you'll see something like:

```
$ mongo
MongoDB shell version: 3.3.5
connecting to: test
Hello, you're looking particularly like Batman today!
>
```

More practically, you can use this script to set up any global variables you'd like to use, alias long names to shorter ones, and override built-in functions. One of the most common uses for `.mongorc.js` is remove some of the more “dangerous” shell helpers. You can override functions like `dropDatabase` or `deleteIndexes` with no-ops or `undefined` them altogether:

```
var no = function() {
    print("Not on my watch.");
};

// Prevent dropping databases
db.dropDatabase = DB.prototype.dropDatabase = no;

// Prevent dropping collections
DBCollection.prototype.drop = no;

// Prevent dropping indexes
DBCollection.prototype.dropIndex = no;
```

Make sure that, if you change any database functions, you do so on both the `db` variable and the `DB` prototype (as shown in the example above). If you change only one, either the `db` variable won't see the change or all new databases you use (when you run `use anotherDB`) won't see your change.

Now if you try to call any of these functions, it will simply print an error message. Note that this technique does not protect you against malicious users; it can only help with fat-fingering.

You can disable loading your `.mongorc.js` by using the `--norc` option when starting the shell.

## Customizing Your Prompt

The default shell prompt can be overridden by setting the `prompt` variable to either a string or a function. For example, if you are running a query that takes minutes to complete, you may want to have a prompt that displays the current time so you can see when the last operation finished:

```
prompt = function() {
    return (new Date())+> " ";
};
```

Another handy prompt might show the current database you're using:

```
prompt = function() {
    if (typeof db == 'undefined') {
        return '(nodb)> ';
    }

    // Check the last db operation
    try {
        db.runCommand({getLastError:1});
    }
    catch (e) {
        print(e);
    }

    return db+> " ";
};
```

Note that prompt functions should return strings and be very cautious about catching exceptions: it can be very confusing if your prompt turns into an exception!

In general, your prompt function should include a call to `getLastError`. This catches errors on writes and reconnects you “automatically” if the shell gets disconnected (e.g., if you restart `mongod`).

The `.mongorc.js` file is a good place to set your prompt if you want to always use a custom one (or set up a couple of custom prompts that you can switch between in the shell).

## Editing Complex Variables

The multiline support in the shell is somewhat limited: you cannot edit previous lines, which can be annoying when you realize that the first line has a typo and you're currently working on line 15. Thus, for larger blocks of code or objects, you may want to edit them in an editor. To do so, set the EDITOR variable in the shell (or in your environment, but since you're already in the shell):

```
> EDITOR="/usr/bin/emacs"
```

Now, if you want to edit a variable, you can say "edit varname", for example:

```
> var wap = db.books.findOne({title: "War and Peace"})
> edit wap
```

When you're done making changes, save and exit the editor. The variable will be parsed and loaded back into the shell.

Add `EDITOR="/path/to/editor";` to your `.mongorc.js` file and you won't have to worry about setting it again.

## Inconvenient Collection Names

Fetching a collection with the `db.collectionName` syntax almost always works, unless the collection name a reserved word or is an invalid JavaScript property name.

For example, suppose we are trying to access the `version` collection. We cannot say `db.version` because `db.version` is a method on `db` (it returns the version of the running MongoDB server):

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

To actually access the `version` collection, you must use the `getCollection` function:

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

This can also be used for collection names with characters that aren't valid in JavaScript

property names, such as *foo-bar-baz* and *123abc* (JavaScript property names can only contain letters, numbers, "\$" and "\_" and cannot start with a number).

Another way of getting around invalid properties is to use array-access syntax: in JavaScript, `x.y` is identical to `x['y']`. This means that subcollections can be accessed using variables, not just literal names. Thus, 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 (var i in collections) {
    print(db.blog[collections[i]]);
}
```

instead of this:

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

Note that you cannot do `db.blog.i`, which would be interpreted as `test.blog.i`, not `test.blog.posts`. You must use the `db.blog[i]` syntax for `i` to be interpreted as a variable.

You can use this technique to access awkwardly-named collections:

```
> var name = "@#&!"
> db[name].find()
```

Attempting to query `db.@#&!` would be illegal, but `db[name]` would work.

# Chapter 3. Creating, Updating, and Deleting Documents

---

This chapter covers the basics of moving data in 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 Documents

Inserts are the basic method for adding data to MongoDB. To insert a single document, use the collection's `insertOne` method. To insert multiple documents at once, use the `insertMany` method.

```
> db.movies.insertOne({ "title" : "Stand by Me" })
```

`insertOne` will add an `"_id"` key to the document (if we do not supply one) and store the document in MongoDB.

### `insertMany()`

If you need to insert multiple documents into a collection, you can use `insertMany`. This method enables you to pass an array of documents to the database. This is far more efficient because your code will not make a round trip to the database for each document inserted, but will insert them in bulk.

In the shell, you can try this out as follows.

```
> db.movies.insertMany([{"title" : "Ghostbusters"},  
...  
...  
{  
    "acknowledged" : true,  
    "insertedIds" : [  
        ObjectId("572630ba11722fac4b6b4996"),  
        ObjectId("572630ba11722fac4b6b4997"),  
        ObjectId("572630ba11722fac4b6b4998")  
    ]  
}  
> db.movies.find()  
{ "_id" : ObjectId("572630ba11722fac4b6b4996"), "title" : "Ghostbusters" }  
{ "_id" : ObjectId("572630ba11722fac4b6b4997"), "title" : "E.T." }  
{ "_id" : ObjectId("572630ba11722fac4b6b4998"), "title" : "Blade Runner" }
```

Sending dozens, hundreds, or even thousands of documents at a time can make inserts significantly faster.

`insertMany` is useful if you are inserting multiple documents into a single collection. 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 `insertMany` can be used for importing data, as well.

Current versions of MongoDB do not accept messages longer than 48 MB, so there is a limit to how much can be inserted in a single batch insert. If you attempt to insert more than 48 MB, many drivers will split up the batch insert into multiple 48 MB batch inserts. Check your driver documentation for details.

When performing a bulk insert using `insertMany`, if a document halfway through the array produces an error of some type, what happens depends on whether you have opted for ordered or unordered operations. As the second parameter to `insertMany` you may specify an options document. Specify `true` for the key, `"ordered"` in the options document to ensure documents are inserted in the order they are provided. Specify `false` and MongoDB may reorder the inserts to increase performance. Ordered inserts is the default if no ordering is specified. For ordered inserts, the array passed to `insertMany` defines the insertion order. If a document produces an insertion error, no documents beyond that point in the array will be inserted. For unordered inserts, MongoDB will attempt to insert all documents, regardless of whether some insertions produce errors.

```
> db.movies.insertMany([  
    {"_id" : 0, "title" : "Top Gun"},
```

```

        {"_id" : 1, "title" : "Back to the Future"},  

        {"_id" : 1, "title" : "Gremlins"},  

        {"_id" : 2, "title" : "Aliens"}])  

2016-05-01T17:04:06.630-0400 E QUERY      [thread1] BulkWriteError: write  

error at item 1 in bulk operation :  

BulkWriteError({  

    "writeErrors" : [  

        {  

            "index" : 1,  

            "code" : 11000,  

            "errmsg" : "E11000 duplicate key error index: test.movies._id_  

dup key: { : 1.0 }",  

            "op" : {  

                "_id" : 1,  

                "title" : "Back to the Future"  

            }
        }
    ],
    "writeConcernErrors" : [ ],
    "nInserted" : 1,
    "nUpserted" : 0,
    "nMatched" : 0,
    "nModified" : 0,
    "nRemoved" : 0,
    "upserted" : [ ]
})
BulkWriteError@src/mongo/shell/bulk_api.js:371:48
BulkWriteResult/this.toError@src/mongo/shell/bulk_api.js:335:24
Bulk/this.execute@src/mongo/shell/bulk_api.js:1177:1
DBCollection.prototype.insertMany@src/mongo/shell/crud_api.js:281:5
@(shell):1:1

```

---

Because ordered inserts is the default, only the first two documents will be inserted. The third document will produce an error, because you cannot insert two documents with the same `"_id"`.

If, instead, we specify unordered inserts, the first, second, and fourth documents in the array are inserted. The only insert that fails is the third document, again because of a duplicate `"_id"` error.

---

```

> db.movies.insertMany([  

...{_id" : 3, "title" : "Sixteen Candles"},  

...{_id" : 4, "title" : "The Terminator"},  

...{_id" : 4, "title" : "The Princess Bride"},  

...{_id" : 5, "title" : "Scarface"}],  

{"ordered" : false})  

2016-05-01T17:02:25.511-0400 E QUERY      [thread1] BulkWriteError: write  

error at item 2 in bulk operation :  

BulkWriteError({
```

```

    "writeErrors" : [
        {
            "index" : 2,
            "code" : 11000,
            "errmsg" : "E11000 duplicate key error index: test.movies.$_id_
                dup key: { : 4.0 }",
            "op" : {
                "_id" : 4,
                "title" : "The Princess Bride"
            }
        }
    ],
    "writeConcernErrors" : [ ],
    "nInserted" : 3,
    "nUpserted" : 0,
    "nMatched" : 0,
    "nModified" : 0,
    "nRemoved" : 0,
    "upserted" : [ ]
)
BulkWriteError@src/mongo/shell/bulk_api.js:371:48
BulkWriteResult/this.toError@src/mongo/shell/bulk_api.js:335:24
Bulk/this.execute@src/mongo/shell/bulk_api.js:1177:1
DBCollection.prototype.insertMany@src/mongo/shell/crud_api.js:281:5
@(shell):1:1

```

If you have studied these examples closely, you might have noted that the output of these two calls to `insertMany` hint that other operations besides simply inserts might be supported for bulk writes. While `insertMany` does not support operations other than `insert`, MongoDB does support a Bulk Write API that enables you to batch together a number of operations of different types together in one call. While that is beyond the scope of this chapter, you may read about the [Bulk Write API](#) in the MongoDB Documentation.

## Insert Validation

MongoDB does minimal checks on data being inserted: it checks the document's basic structure and adds an `"_id"` field if one does not exist. One of the basic structure checks is size: all documents must be smaller than 16 MB. 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 data 16 MB is, the entire text of *War and Peace* is just 3.14 MB.

These minimal checks also mean that it is fairly easy to insert invalid data (if you are trying to).

Thus, you should only allow trusted sources, such as your application servers, to connect to the database. All of the drivers for major languages (and most of the minor ones, too) do 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.

## insert()

In versions of MongoDB prior to 3.0, `insert()` was the primary method for inserting documents into MongoDB. MongoDB drivers introduced a new CRUD API at the same time as the MongoDB 3.0 server release. As of MongoDB 3.2 the mongo shell also supports this API, which includes `insertOne` and `insertMany` as well as several other methods. The goal of the current CRUD API is to make the semantics of all CRUD operations consistent and clear across the drivers and the shell. While methods such as `insert()` are still supported for backward compatibility, they should not be used in applications going forward. You should instead prefer `insertOne` and `insertMany` for creating documents.

## Removing Documents

Now that there's data in our database, let's delete it. The CRUD API provides `deleteOne` and `deleteMany` for this purpose. Both of these methods take a filter document as their first parameter. The filter specifies a set of criteria to match against in removing documents. To delete the document with the `_id` value of 4, we use `insertOne` in the mongo shell as illustrated below.

```
> db.movies.find()
{ "_id" : 0, "title" : "Top Gun"}
{ "_id" : 1, "title" : "Back to the Future"}
{ "_id" : 3, "title" : "Sixteen Candles"}
{ "_id" : 4, "title" : "The Terminator"}
{ "_id" : 5, "title" : "Scarface"}
> db.movies.deleteOne({"_id" : 4})
{ "acknowledged" : true, "deletedCount" : 1 }
> db.movies.find()
{ "_id" : 0, "title" : "Top Gun"}
{ "_id" : 1, "title" : "Back to the Future"}
{ "_id" : 3, "title" : "Sixteen Candles"}
{ "_id" : 5, "title" : "Scarface"}
```

In the example above, we used a filter that could only match one document since `_id` values are unique in a collection. However, we can also specify a filter that matches multiple documents in a collection. In these cases, `deleteOne` will delete the first document found that matches the filter. Which document is found first depends on several factors including the order in which documents were inserted, what updates were made to documents (for some storage engines),

and what indexes are specified. As with any database operation, be sure you know what effect your use of `deleteOne` will have on your data.

To delete more than one document matching a filter, use `deleteMany`.

```
> db.movies.find()
{ "_id" : 0, "title" : "Top Gun", "year" : 1986 }
{ "_id" : 1, "title" : "Back to the Future", "year" : 1985 }
{ "_id" : 3, "title" : "Sixteen Candles", "year" : 1984 }
{ "_id" : 4, "title" : "The Terminator", "year" : 1984 }
{ "_id" : 5, "title" : "Scarface", "year" : 1983 }
> db.movies.deleteMany({ "year" : 1984 })
{ "acknowledged" : true, "deletedCount" : 2 }
> db.movies.find()
{ "_id" : 0, "title" : "Top Gun", "year" : 1986 }
{ "_id" : 1, "title" : "Back to the Future", "year" : 1985 }
{ "_id" : 5, "title" : "Scarface", "year" : 1983 }
```

As a more realistic use case, suppose we want to remove everyone from the `mailing.list` collection where the value for `"opt-out"` is `true`:

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

In versions of MongoDB prior to 3.0, `remove()` was the primary method for deleted documents in MongoDB. MongoDB drivers introduced the `deleteOne` and `deleteMany` methods at the same time as the MongoDB 3.0 server release. The shell began supporting these methods in MongoDB 3.2. While `remove()` still supported for backward compatibility, you should use `deleteOne` and `deleteMany` in your applications. The current CRUD API provides a cleaner set of semantics and, especially, for multidocument operations helps application developers avoid a couple of common pitfalls with the previous API.

## drop()

It is possible to use `deleteMany` to remove all documents in a collection.

```
> db.movies.find()
{ "_id" : 0, "title" : "Top Gun", "year" : 1986 }
{ "_id" : 1, "title" : "Back to the Future", "year" : 1985 }
{ "_id" : 3, "title" : "Sixteen Candles", "year" : 1984 }
{ "_id" : 4, "title" : "The Terminator", "year" : 1984 }
{ "_id" : 5, "title" : "Scarface", "year" : 1983 }
> db.movies.deleteMany({})
{ "acknowledged" : true, "deletedCount" : 5 }
> db.movies.find()
```

```
>
```

Removing documents is usually a fairly quick operation; however, if you want to clear an entire collection, it is faster to *drop* it

```
> db.movies.drop()  
true
```

and then recreate any indexes on the empty collection.

Once data has been removed, it is gone forever. There is no way to undo a delete or drop operation or recover deleted documents except, of course by restoring a previously backed up version of the data.

## Updating Documents

Once a document is stored in the database, it can be changed using one of several update methods: `updateOne`, `updateMany`, and `replaceOne`. `updateOne` and `updateMany` each take a filter document as their first parameter and a modifier document, which describes changes to make, as the second parameter. `replaceOne` also takes a filter as the first parameter, but as the second parameter `replaceOne` expects a document with which it will replace the document matching the filter.

Updating a document is 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

`replaceOne` 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 move the "friends" and "enemies" fields to a "relationships" subdocument. We can change the structure of the document in the shell and then replace the database's version with an `replaceOne`:

```
> 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.replaceOne({ "name" : "joe" }, joe);
```

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

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

A common mistake is matching more than one document with the criteria and then creating a duplicate `_id` value with the second parameter. The database will throw an error for this, and no documents will be updated.

For example, suppose we create several documents with the same value for "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.replaceOne({ "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`. For the example above, this would be the correct update to use:

```
> db.people.replaceOne({ "_id" : ObjectId("4b2b9f67a1f631733d917a7c") }, joe
```

Using `_id` for the filter will also be efficient since `_id` values form the basis for the primary index of a collection. We'll cover primary and secondary indexes and how indexing affects updates and other operations more in [Chapter 5](#).

## Using Update Operators

Usually only certain portions of a document need to be updated. You can update specific fields in a document using atomic *update operators*. Update operators 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 want to increment a counter each time someone visits a page. We can use update operators 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.updateOne({"url" : "www.example.com"},  
... {"$inc" : {"pageviews" : 1}})  
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

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

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

When using operators, 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 field. If the field 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.updateOne({"_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.updateOne({"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.updateOne({"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.updateOne({"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.updateOne({ "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 make when starting out is to try to set the value of a key to some value by doing an update that resembles this:

```
> db.blog.posts.updateOne({ "author.name" : "joe" }, { "author.name" : "joe s
```

This will result in an error. The update document must contain update operators. Previous versions of the CRUD API did not catch this type of error. Earlier update methods would simply complete a whole document replacement in such situations. It is this type of pitfall that lead to the creation of a new CRUD API.

## INCREMENTING AND DECREMENTING

The `$inc` operator 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.insertOne({ "game" : "pinball", "user" : "joe" })
```

When the ball hits a bumper, the game should increment the player's score. Since 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.updateOne({ "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",
  "user" : "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. We can do this by passing a different value to "\$inc":

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

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

```
> db.games.findOne()
{
  "_id" : ObjectId("4b2d75476cc613d5ee930164"),
  "game" : "pinball",
  "user" : "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.strcounts.insert({"count" : "1"})
WriteResult({ "nInserted" : 1 })
> db.strcounts.update({}, {"$inc" : {"count" : 1}})
WriteResult({
  "nMatched" : 0,
```

```
        "nUpserted" : 0,
        "nModified" : 0,
        "writeError" : {
            "code" : 16837,
            "errmsg" : "Cannot apply $inc to a value of non-numeric type.
{_id: ObjectId('5726c0d36855a935cb57a659')} has the field 'count' of
non-numeric type String"
        }
    })

```

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 following array operations.

## ARRAY OPERATORS

An extensive class of update operators 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.

### ADDING ELEMENTS

"\$push" adds elements to the end of an array if the array 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.updateOne({"title" : "A blog post"},
... {"$push" : {"comments" :
...   {"name" : "joe", "email" : "joe@example.com",
...   "content" : "nice post."}}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> 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 simply use "\$push" again:

```
> db.blog.posts.updateOne({"title" : "A blog post"},
... {"$push" : {"comments" :
...     {"name" : "bob", "email" : "bob@example.com",
...      "content" : "good post."}}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> 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."
    }
  ]
}
```

This is the “simple” form of push, but you can use it for more complex array operations as well. The MongoDB query language provides modifiers for some operators, including \$push. You can push multiple values in one operation using the "\$each" modifier for \$push:

```
> db.stock.ticker.updateOne({"_id" : "GOOG"},
... {"$push" : {"hourly" : {"$each" : [562.776, 562.790, 559.123]}}})
```

This would push three new elements onto the array. Specify a single-element array to get equivalent behavior to the non-\$each form of \$push.

If you only want the array to grow to a certain length, you can use the \$slice modifier with \$push to prevent an array from growing beyond a certain size, effectively making a “top N” list of items:

```
> db.movies.updateOne({"genre" : "horror"},
```

```
...{"$push" : {"top10" : {"$each" : ["Nightmare on Elm Street", "Saw"],  
...  
...  
"$slice" : -10}}})
```

This example limits the array to the last 10 elements pushed.

If the array was smaller than 10 elements (after the push), all elements would be kept. If the array was larger than 10 elements, only the last 10 elements would be kept. Thus, `$slice` can be used to create a queue in a document.

Finally, you can apply the `$sort` modifier to `$push` operations before trimming,

```
> db.movies.updateOne({"genre" : "horror"},  
... {"$push" : {"top10" : {"$each" : [{"name" : "Nightmare on Elm Street",  
...  
...  
"rating" : 6.6},  
...  
{"name" : "Saw", "rating" : 4.3}],  
...  
"$slice" : -10,  
"$sort" : {"rating" : -1}}}})
```

This will sort all of the objects in the array by their `"rating"` field and then keep the first 10. Note that you must include `$each`; you cannot just `$slice` or `$sort` an array with `$push`.

## USING ARRAYS AS SETS

You might want to treat an array as a set, only adding values if they are not 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.updateOne({"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 example, 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.updateOne({ "_id" : ObjectId("4b2d75476cc613d5ee930164") },  
... { "$addToSet" : { "emails" : "joe@gmail.com" } })  
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 0 }  
> db.users.findOne({ "_id" : ObjectId("4b2d75476cc613d5ee930164") })  
{  
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),  
    "username" : "joe",  
    "emails" : [  
        "joe@example.com",  
        "joe@gmail.com",  
        "joe@yahoo.com",  
        []  
    ]  
}  
> db.users.updateOne({ "_id" : ObjectId("4b2d75476cc613d5ee930164") },  
... { "$addToSet" : { "emails" : "joe@hotmail.com" } })  
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }  
> 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 operators if the user wanted to add more than one email address:

```
> db.users.updateOne({ "_id" : ObjectId("4b2d75476cc613d5ee930164") }, { "$adde  
... { "emails" : { "$each" :  
...     [ "joe@php.net", "joe@example.com", "joe@python.org" ] } } })  
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }  
> db.users.findOne({ "_id" : ObjectId("4b2d75476cc613d5ee930164") })  
{  
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),  
    "username" : "joe",  
    "emails" : [  
        "joe@example.com",  
        "joe@gmail.com",  
        "joe@php.net",  
        "joe@python.org"  
    ]  
}
```

```
        "joe@yahoo.com",
        "joe@hotmail.com"
        "joe@php.net"
        "joe@python.org"
    ]
}
```

## REMOVING ELEMENTS

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.insertOne({"todo" : ["dishes", "laundry", "dry cleaning"]})
```

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

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

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

```
> db.lists.findOne()
{
  "_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]`.

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.

## 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.updateOne({ "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.updateOne({ "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.

## 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 updated documents. If a matching document is found, it will be updated normally. Upserts can be handy because they can eliminate the need to “seed” your collection: you can often have the same code create and update documents.

Let’s go back to our example that records 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 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.insertOne({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 to the database. (the third parameter to updateOne and updateMany is an options document that enables us to specify this.

```
db.analytics.updateOne({url : "/blog"}, {"$inc" : {"pageviews" : 1}},
```

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.users.updateOne({ "rep" : 25 }, { "$inc" : { "rep" : 3 } }, { "upsert" : true }
WriteResult({
  "nMatched" : 0,
  "nUpserted" : 1,
  "nModified" : 0,
  "_id" : ObjectId("5727b2a7223502483c7f3acd")
})
> db.users.findOne({ "_id" : ObjectId("5727b2a7223502483c7f3acd") })
{ "_id" : ObjectId("5727b2a7223502483c7f3acd"), "rep" : 28 }
```

The upsert creates a new document with a "rep" of 25 and then increments that by 3, giving us a document where "rep" is 28. If the upsert option were not specified, { "rep" : 25 } would not match any documents, so nothing would happen.

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

Sometimes a field needs to be set when a document is created, but not changed on subsequent updates. This is what "\$setOnInsert" is for. "\$setOnInsert" is an operator that only sets the value of a field when the document is being inserted. Thus, we could do something like this:

```
> db.users.updateOne({}, { "$setOnInsert" : { "createdAt" : new Date() } },
...{ "upsert" : true })
{
  "acknowledged" : true,
  "matchedCount" : 0,
  "modifiedCount" : 0,
  "upsertedId" : ObjectId("5727b4ac223502483c7f3ace")
}
> db.users.findOne()
{
  "_id" : ObjectId("5727b4ac223502483c7f3ace"),
  "createdAt" : ISODate("2016-05-02T20:12:28.640Z")
}
```

If we run this update again, it will match the existing document, nothing will be inserted, and so the "createdAt" field will not be changed:

```
> db.users.updateOne({}, {"$setOnInsert" : {"createdAt" : new Date()}},
...{ "upsert" : true})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 0 }
> db.users.findOne()
{
    "_id" : ObjectId("5727b4ac223502483c7f3ace"),
    "createdAt" : ISODate("2016-05-02T20:12:28.640Z")
}
```

Note that you generally do not need to keep a "createdAt" field, as ObjectIds contain a timestamp of when the document was created. However, "\$setOnInsert" can be useful for creating padding, initializing counters, and for collections that do not use ObjectIds.

## 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. `save` is really just a convenience function so that programmers can quickly modify documents in the shell:

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

Without `save`, the last line would have been a more cumbersome

```
db.testcol.updateOne({ "_id" : x._id }, x).
```

## Updating Multiple Documents

So far in this chapter we have used `updateOne` to illustrate update operations. `updateOne` updates only the first document found that matches the filter criteria. If there are more matching documents, they will remain unchanged. To modify all of the documents matching a filter, use `updateMany`. `updateMany` follows the same semantics as `updateOne` and takes the same parameters. The key difference is in the number of documents that might be changed.

`updateMany` provides a powerful tool for 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 `updateMany` to add a "gift" to their account. For example:

```
> db.users.insertMany([
...{birthday: "10/13/1978"},
```

```

...{birthday: "10/13/1978"},  

...{birthday: "10/13/1978"}])  

{  

    "acknowledged" : true,  

    "insertedIds" : [  

        ObjectId("5727d6fc6855a935cb57a65b"),  

        ObjectId("5727d6fc6855a935cb57a65c"),  

        ObjectId("5727d6fc6855a935cb57a65d")  

    ]  

}  

> db.users.updateMany({birthday : "10/13/1978"},  

... {$set : {"gift" : "Happy Birthday!"}})  

{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }

```

The call to `updateMany` adds a "gift" field to each of the three documents we inserted into the `users` collection immediately before.

## Returning Updated Documents

For some use cases it is important to return the document modified. In earlier versions of MongoDB, `findAndModify` was the method of choice in such situations. It is handy for manipulating queues and performing other operations that need get-and-set style atomicity. However, `findAndModify` is prone to user error due to the fact that it is a complex method combining the functionality of three different types of operations: delete, replace, and update (including upserts).

MongoDB 3.2 introduced three new collection methods to the shell to accommodate the functionality of `findAndModify` but with semantics that are easier to learn and remember. These methods are: `findOneAndDelete`, `findOneAndReplace`, and `findOneAndUpdate`. The primary difference between these methods and, for example, `updateOne` is that they enable you to atomically get the value of a modified document.

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 as "RUNNING". Once we have processed it, we update the status to "DONE". This looks something like the following:

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

    if (result.modifiedCount == 1) {
        do_something(ps);
        db.processes.updateOne({ "_id" : ps._id }, { "$set" : { "status" : "DONE" } });
        break;
    }
    cursor = db.processes.find({ "status" : "READY" });
    cursor.sort({ "priority" : -1 }).limit(1);
}
```

Also, depending on timing, one thread may end up doing all the work while another thread uselessly trails 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 `findOneAndUpdate`. `findOneAndUpdate` can return the item and update it in a single operation. In this case, it looks like the following:

```
> db.processes.findOneAndUpdate({ "status" : "READY" },
... { "$set" : { "status" : "RUNNING" } },
... { "sort" : { "priority" : -1 } })
{
    "_id" : ObjectId("4b3e7a18005cab32be6291f7"),
    "priority" : 1,
```

```
        "status" : "READY"
    }
```

Notice that the status is still "READY" in the returned document because `findOneAndUpdate` defaults to returning the state of the document before it was modified. `findOneAndUpdate` will return the updated document if we set the `returnNewDocument` field in the options document to true. An options document is passed as the third parameter to `findOneAndUpdate`.

```
> db.processes.findOneAndUpdate({ "status" : "READY" },
... { "$set" : { "status" : "RUNNING" } },
... { "sort" : { "priority" : -1 } },
... { "returnNewDocument": true })
{
  "_id" : ObjectId("4b3e7a18005cab32be6291f7"),
  "priority" : 1,
  "status" : "RUNNING"
}
```

Thus, the program becomes the following:

```
ps = db.processes.findOneAndUpdate({ "status" : "READY" },
                                    { "$set" : { "status" : "RUNNING" } },
                                    { "sort" : { "priority" : -1 } },
                                    { "returnNewDocument": true })

do_something(ps)
db.process.update({ "_id" : ps._id }, { "$set" : { "status" : "DONE" } })
```

`findOneAndReplace` takes the same parameters and returns the document matching the filter either before or after the update, depending on the value of the `returnNewDocument`. `findOneAndDelete` is similar except it does not take an update document as a parameter and has a subset of the options of the other two "findOneAnd..." methods. `findOneAndDelete` returns the deleted document.

# 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.
- 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 criteria.

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()
```

matches every document in the collection `c` (and returns these documents in batches).

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: numbers match numbers, 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 returned by default, even if it isn't specifically requested.

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 also 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 couldn't 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](#)”), 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, we can do this:

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

This would find all documents where the "age" field was greater than or equal to 18 AND less than or equal to 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}})
```

Depending on how you create and store dates, an exact match might be less useful, since dates are 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 down your results 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.

While "\$or" will always work, use "\$in" whenever possible as the query optimizer handles it more efficiently.

## \$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, etc., 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](#)").

## Conditional Semantics

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.

There are a few "meta-operators" that go in the outer document: "\$and", "\$or", and "\$nor". They all have a similar form:

```
> db.users.find({"$and" : [{"x" : {"$lt" : 1}}, {"x" : 4}]})
```

This query would match documents with an "x" field both less than 1 and equal to 4. Although these seem like contradictory conditions, it is possible to fulfill if the "x" field is an array: `{"x" : [0, 4]}` would match. Note that the query optimizer does not optimize "\$and" as well as other operators. This query would be more efficient to structure as:

```
> db.users.find({ "x" : { "$lt" : 1, "$in" : [ 4 ] } })
```

## 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 types have special behavior when querying.

### 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" : { "$eq" : null, "$exists" : true } })
```

## Regular Expressions

"\$regex" provides regular expression capabilities for pattern matching strings in queries.

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" : {"$regex" : /joe/i} })
```

Regular expression flags (for example, `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.

### NOTE

MongoDB can leverage an index for queries on prefix regular expressions (e.g., `/^joey/`). Indexes *cannot* be used for case-insensitive searches (`/^joey/i`).

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.insertOne({"bar" : /baz/})
> db.foo.find({"bar" : /baz/})
{
  "_id" : ObjectId("4b23c3ca7525f35f94b60a2d"),
  "bar" : /baz/
}
```

## Querying Arrays

Querying for elements of an array is designed to behave the way querying for scalars does. For example, if the array is a list of fruits, like this:

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

the following query:

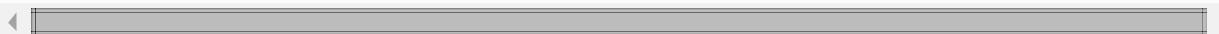
```
> db.food.find({"fruit" : "banana"})
```

will successfully match the document. We can query for it in much the same way as we would if 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.insertOne({_id : 1, fruit : ["apple", "banana", "peach"]})  
> db.food.insertOne({_id : 2, fruit : ["apple", "kumquat", "orange"]})  
> db.food.insertOne({_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 above:

```
> 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(criteria, {"$push" : {"fruit" : "strawberry"}})
```

it can simply be changed to this:

```
> db.food.update(criteria,
... {"$push" : {"fruit" : "strawberry"}, "$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 33th. If there were fewer than 33 elements in the array, it would 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.

## RETURNING A MATCHING ARRAY ELEMENT

"\$slice" is helpful when you know the index of the element, but sometimes you want whichever array element matched your criteria. You can return the matching element with the \$-operator. Given the blog example above, you could get Bob's comment back with:

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

Note that this only returns the first match for each document: if Bob had left multiple comments on this post, only the first one in the "comments" array would be returned.

## ARRAY AND RANGE QUERY INTERACTIONS

Scalars (non-array elements) in documents must match each clause of a query's criteria. For example, if you queried for `{"x" : {"$gt" : 10, "$lt" : 20}}`, "x" would have to be both greater than 10 and less than 20. However, if a document's "x" field is an array, the document matches if there is an element of "x" that matches each part of the criteria *but each query clause can match a different array element*.

The best way to understand this behavior is to see an example. Suppose we have the following documents:

```
{ "x" : 5}  
{ "x" : 15}  
{ "x" : 25}  
{ "x" : [5, 25]}
```

If we wanted to find all documents where "x" is between 10 and 20, one might naively structure a query as `db.test.find({ "x" : { "$gt" : 10, "$lt" : 20 } })` and expect to get back one document: `{ "x" : 15 }`. However, running this, we get two:

```
> db.test.find({ "x" : { "$gt" : 10, "$lt" : 20 } })  
{ "x" : 15}  
{ "x" : [5, 25]}
```

Neither 5 nor 25 is between 10 and 20, but the document is returned because 25 matches the first clause (it is greater than 10) and 5 matches the second clause (it is less than 20).

This makes range queries against arrays essentially useless: a range will match any multi-element array. There are a couple of ways to get the expected behavior.

First, you can use "`$elemMatch`" to force MongoDB to compare both clauses with a single array element. However, the catch is that "`$elemMatch`" won't match non-array elements:

```
> db.test.find({ "x" : { "$elemMatch" : { "$gt" : 10, "$lt" : 20 } }})  
> // no results
```

The document `{ "x" : 15 }` no longer matches the query, because the "x" field is not an array. That said, you should have a good reason for mixing array and scalar values in a field. Many uses cases do not require mixing. For those, "`$elemMatch`" provides a good solution for range queries on array elements.

If you have an index over the field that you're querying on (see [Chapter 5](#)), you can use `min()` and `max()` to limit the index range traversed by the query to your "`$gt`" and "`$lt`" values:

```
> db.test.find({ "x" : { "$gt" : 10, "$lt" : 20 } }).min({ "x" : 10 }).max({ "x" : 15 })
```

Now this will only traverse the index from 10 to 20, missing the 5 and 25 entries. You can only use `min()` and `max()` when you have an index on the field you are querying for, though, and you must pass all fields of the index to `min()` and `max()`.

Using `min()` and `max()` when querying for ranges over documents that may include arrays is generally a good idea: if you look at the index bounds for a `"$gt"/"$lt"` query over an array, you can see that it's horribly inefficient. It basically accepts any value, so it will search every index entry, not just those in the range.

## 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, a query for a full subdocument must exactly match the subdocument. If Joe decides to add a middle name field, 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 5. 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 above: 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. For security, use of "\$where" clauses should be highly restricted or eliminated. End users should never be allowed to execute arbitrary "\$where" clauses.

The most common case for using "\$where" is to compare the values for two keys in a document. For instance, suppose we have documents that look like this:

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

We'd like to return documents where any two of the fields are equal. For example, 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.

"\$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 one of the aggregation tools, which are covered

in Chapter 7.

## Server-Side Scripting

You must be very careful with security when executing JavaScript on the server. If done incorrectly, server-side JavaScript is susceptible to injection attacks similar to those that occur in a relational database. However, by following certain rules around accepting input, you can use JavaScript safely. Alternatively, you can turn off JavaScript execution altogether by running *mongod* with the --noscripting option.

The security issues with JavaScript are all related to executing user-provided programs on the server. You want to avoid doing that, so make sure you aren't accepting user input and passing it directly to *mongod*. For example, suppose you want to print "Hello, *name*!", where *name* is provided by the user. A naive approach might be to write a JavaScript function such as the following:

```
> func = "function() { print('Hello, "+name+"!'); }"
```

If *name* is a user-defined variable, it could be the string "'); db.dropDatabase(); print('", which would turn the code into this:

```
> func = "function() { print('Hello, '); db.dropDatabase(); print('!'); }"
```

Now, if you run this code, your entire database will be dropped!

To prevent this, you should use a scope to pass in the name. In Python, for example, this looks like this:

```
func = pymongo.code.Code("function() { print('Hello, '+username+'!'); }",
                        {"username": name})
```

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 a document that maps variable names to values. This mapping becomes a local scope for the JavaScript function being executed. Thus, in the

example above, the function would have access to a variable called `username`, whose value would be the string that the user gave.

### NOTE

The shell does not have a code type that includes scope; you can only use strings or JavaScript functions with it.

## 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.collection.insertOne({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 JavaScript's 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 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 options 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 4 MB 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 with a `getMore` request. `getMore` requests basically contain an identifier for the query and ask the database if there are any more results, returning the next batch if there are. 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 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 fewer 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 that person clicks 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; there are suggestions for how to avoid them in the next section.

## 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, since it has to find and then discard all the skipped results. Most databases keep more metadata in the index to help with skips, but MongoDB does not yet support this, so large skips should be avoided. Often 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 skips. 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, assuming the date is unique, 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" : {"$lt" : 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.insertOne({ "name" : "joe", "random" : Math.random() })
> db.people.insertOne({ "name" : "john", "random" : Math.random() })
> db.people.insertOne({ "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" : { "$lte" : 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.

```
> db.foo.find(criteria).addSpecial("$maxScan", 20)
```

This can be useful if you want a query to not take too long but are not sure how much of a collection will need to be scanned. This will limit your results to whatever was found in the part of the collection that was scanned (i.e., you may miss other documents that match).

`$min : document`

Start criteria for querying. *document* must exactly match the keys of an index used for the query. This forces the given index to be used for the query.

This is used internally and you should generally use `"$gt"` instead of `"$min"`. You can use `"$min"` to force the lower bound on an index scan, which may be helpful for complex queries.

`$max : document`

End criteria for querying. *document* must exactly match the keys of an index used for the query. This forces the given index to be used for the query.

If this is used internally, you should generally use `"$lt"` instead of `"$max"`. You can use `"$max"` to force bounds on an index scan, which may be helpful for complex queries.

`$showDiskLoc : true`

Adds a `"$diskLoc"` field to the results that shows where on disk that particular result lives. For example:

```
> db.foo.find().__addSpecial('$showDiskLoc', true)
{ "_id" : 0, "$diskLoc" : { "file" : 2, "offset" : 154812592 } }
{ "_id" : 1, "$diskLoc" : { "file" : 2, "offset" : 154812628 } }
```

The file number shows which file the document is in. In this case, if we’re using the *test* database, the document is in *test.2*. The second field gives the byte offset of each document within the file.

## Immortal Cursors

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.

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 way, if a client crashes or is buggy, MongoDB will not be left with thousands of open cursors.

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 kill it to make sure it gets closed. Otherwise, it will sit around in the database hogging resources until the server is restarted.

## Database Commands

There is one very special type of query called a database command. We’ve covered creating, updating, deleting, and finding documents. Database commands do “everything else,” from administrative tasks like shutting down the server and cloning databases to counting documents in a collection and performing aggregations.

Commands are mentioned throughout this text, as they are useful for data manipulation, administration, and monitoring. For example, dropping a collection is done via the "drop" database command:

```
> db.runCommand({ "drop" : "test" });
{
  "nIndexesWas" : 1,
  "msg" : "indexes dropped for collection",
  "ns" : "test.test",
  "ok" : true
}
```

You might be more familiar with the shell helper, which wraps the command and provides a simpler interface:

```
> db.test.drop()
```

Often you can just use the shell helpers, but knowing the underlying commands can be helpful if you're stuck on a box with an old version of the shell and connected to a new version of the database: the shell might not have the wrappers for new database commands, but you can still run them with `runCommand()`.

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. You can see all commands by running the `db.listCommands()` command.

## How Commands Work

A database command always returns a document containing the key "ok". If "ok" is a true

value (1, 1.0, or true), the command was successful; if it is false, then the command failed for some reason.

If "ok" is 0 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 was dropped in the previous section:

```
> db.runCommand({ "drop" : "test" });
{ "errmsg" : "ns not found", "ok" : false }
```

Commands in MongoDB are 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.

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. If you're working on another database and you need to run an admin command, you can use the adminCommand function, instead of runCommand:

```
> use temp
switched to db temp
> db.runCommand({ shutdown:1 })
{ "errmsg" : "access denied; use admin db", "ok" : 0 }
> db.adminCommand({ "shutdown" : 1 })
```

Commands are one of the few places that are field-order-sensitive: the command name must always be the first field in the command. Thus, { "getLastError" : 1, "w" : 2 } will work, but { "w" : 2, "getLastError" : 1 } will not.

## **Part II. Designing Your Application**

---

# Chapter 5. Indexes

---

This chapter introduces MongoDB indexes. Indexes enable you to perform queries efficiently. Indexes are an important part of application development and are even required for certain types of queries. In this chapter we will cover:

- What indexes are and why you'd want to use them
- How to choose which fields to index
- How to enforce and evaluate index usage
- Administrative details on creating and removing indexes

As we'll see in this chapter, choosing the right indexes for your collections is critical to performance.

## Introduction to Indexes

A database index is similar to a book's index. Instead of looking through the whole book, the database takes a shortcut and just looks at an ordered list with references to the content. This allows MongoDB to query orders of magnitude faster.

A query that does not use an index is called a collection scan, which means that the server has to "look through the whole book" to find a query's results. This process 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 collection scans because it is very slow for large collections.

Let's look at an example. To get started, we'll create a collection with 1 million documents in it (or 10 million or 100 million, if you have the patience) and then look at the differences in performance for queries on this collections, first without an index and then with an index.

```
> for (i=0; i<1000000; i++) {
```

```

...
    db.users.insertOne (
...
        {
...
            "i" : i,
...
            "username" : "user"+i,
...
            "age" : Math.floor(Math.random()*120),
...
            "created" : new Date()
...
        }
...
    );
...
}

```

If we do a query on this collection, we can use the `explain()` command to see what MongoDB is doing when it executes the query. The preferred way to use the `explain()` command is through the cursor helper method that wraps this command. The `explain()` cursor method provides information on the execution of a variety of CRUD operations. This method may be run in several verbosity modes. We'll look at `executionStats` mode since this helps us understand the effect of using an index to satisfy queries. Try querying on a specific username to see an example.

```

> db.users.find({ "username": "user101" }).explain("executionStats")
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "test.users",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "username" : {
        "$eq" : "user101"
      }
    },
    "winningPlan" : {
      "stage" : "COLLSCAN",
      "filter" : {
        "username" : {
          "$eq" : "user101"
        }
      }
    },
    "direction" : "forward"
  },
  "rejectedPlans" : [ ]
},
"executionStats" : {
  "executionSuccess" : true,
  "nReturned" : 1,
  "executionTimeMillis" : 479,
  "totalKeysExamined" : 0,
  "totalDocsExamined" : 1000000,
  "executionStages" : {
    "stage" : "COLLSCAN",
    "filter" : {
...

```

```

        "username" : {
            "$eq" : "user101"
        }
    },
    "nReturned" : 1,
    "executionTimeMillisEstimate" : 437,
    "works" : 1000002,
    "advanced" : 1,
    "needTime" : 1000000,
    "needYield" : 0,
    "saveState" : 7827,
    "restoreState" : 7827,
    "isEOF" : 1,
    "invalidates" : 0,
    "direction" : "forward",
    "docsExamined" : 1000000
}
},
"serverInfo" : {
    "host" : "SGB-MBP.local",
    "port" : 27017,
    "version" : "3.4.0-rc2",
    "gitVersion" : "217fb0a4d2b808fa45921e4ca1d73db32330620b"
},
"ok" : 1
}

```

“Using `explain()`” will explain the output fields; for now you can ignore almost all of them. For this example, we want to look at the nested document that is the value of the `“executionStats”` field. Within this document, `“totalDocsExamined”` is the number of documents MongoDB looked at while trying to satisfy the query, which, as you can see, is every document in the collection. That is, MongoDB had to look through every field in every document. This took nearly half a second to accomplish on my laptop: the `“executionTimeMillis”` field shows the number of milliseconds it took to execute the query.

The `“nReturned”` field of the `“executionStats”` field shows the number of results returned: 1, which makes sense because there is only one user with the username `“user101”`. Note that MongoDB had to look through every document in the collection for matches because it did not know that usernames are unique.

To enable MongoDB to respond to queries efficiently, all query patterns in your application should be supported by an index. By query patterns, we simply mean the different types of questions your application asks of the database. In the example above, we queried the users collection by `username`. That is an example of a specific query pattern. In many applications, a

single index will support several query patterns. We will discuss tailoring indexes to query patterns in a later section.

## Creating an Index

Try creating an index on the username field. To create an index, we'll use the `createIndex()` collection method.

```
> db.users.createIndex({ "username" : 1 })
```

Creating the collection should take no longer than a few seconds unless you made your collection especially large. If the `createIndex()` call does not return after a few seconds, run `db.currentOp()` (in a different shell) or check your *mongod*'s log to see the index build's progress.

Once the index build is complete, try repeating the original query:

```
> db.users.find({ "username" : "user101" }).explain("executionStats")
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "test.users",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "username" : {
        "$eq" : "user101"
      }
    },
    "winningPlan" : {
      "stage" : "FETCH",
      "inputStage" : {
        "stage" : "IXSCAN",
        "keyPattern" : {
          "username" : 1
        },
        "indexName" : "username_1",
        "isMultiKey" : false,
        "isUnique" : false,
        "isSparse" : false,
        "isPartial" : false,
        "indexVersion" : 1,
        "direction" : "forward",
        "indexBounds" : {
          "username" : [
            "[\"user101\", \"user101\"]"
          ]
        }
      }
    }
  }
}
```

```
        },
        "rejectedPlans" : [ ]
    },
    "executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 1,
        "executionTimeMillis" : 4,
        "totalKeysExamined" : 1,
        "totalDocsExamined" : 1,
        "executionStages" : {
            "stage" : "FETCH",
            "nReturned" : 1,
            "executionTimeMillisEstimate" : 0,
            "works" : 2,
            "advanced" : 1,
            "needTime" : 0,
            "needYield" : 0,
            "saveState" : 0,
            "restoreState" : 0,
            "isEOF" : 1,
            "invalidates" : 0,
            "docsExamined" : 1,
            "alreadyHasObj" : 0,
            "inputStage" : {
                "stage" : "IXSCAN",
                "nReturned" : 1,
                "executionTimeMillisEstimate" : 0,
                "works" : 2,
                "advanced" : 1,
                "needTime" : 0,
                "needYield" : 0,
                "saveState" : 0,
                "restoreState" : 0,
                "isEOF" : 1,
                "invalidates" : 0,
                "keyPattern" : {
                    "username" : 1
                },
                "indexName" : "username_1",
                "isMultiKey" : false,
                "isUnique" : false,
                "isSparse" : false,
                "isPartial" : false,
                "indexVersion" : 1,
                "direction" : "forward",
                "indexBounds" : {
                    "username" : [
                        "[\"user101\", \"user101\"]"
                    ]
                },
                "keysExamined" : 1,
                "seeks" : 1,
                "dupsTested" : 0,
                "dupsDropped" : 0,
                "isExactMatch" : false
            }
        }
    }
}
```

```
        "seenInvalidated" : 0
    }
},
"serverInfo" : {
    "host" : "SGB-MBP.local",
    "port" : 27017,
    "version" : "3.4.0-rc2",
    "gitVersion" : "217fb0a4d2b808fa45921e4ca1d73db32330620b"
},
"ok" : 1
}
```



This `explain()` output is more complex, but, for now, continue to ignore all the fields other than `"nReturned"`, `"totalDocsExamined"`, and `"executionTimeMillis"` in the `"executionStats"` nested document. As you can see, the query is now almost instantaneous and, even better, has a similar runtime when querying for any username, e.g.:

```
> db.users.find({ "username": "user999999" }).explain("executionStats")
```

An index can make a dramatic difference in query times. However, indexes have their price: write operations (inserts, updated, and deletes) that modify an indexed field will take longer. This is because in addition to updating the document, MongoDB has to update indexes when your data changes. Typically, the tradeoff is worth it. The tricky part becomes figuring out which fields to index.

### TIP

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.

To choose which fields to create indexes for, look through your common queries and queries that need to be fast and try to find a common set of keys from those. For instance, in the example above, we were querying on `"username"`. If that was a particularly common query or was becoming a bottleneck, indexing `"username"` would be a good choice. However, if this was an unusual query or one that was only done by administrators who didn't care how long it took, it would not be a good choice of index.

## Introduction to Compound Indexes

The purpose of an index is to make your queries as efficient as possible. For many query patterns it is necessary to build indexes based on two or more keys. For example, an index keeps all of its values in a sorted order so it makes sorting documents by the indexed key much faster. However, an index can only help with sorting if it is a prefix of the sort. For example, the index on "username" wouldn't help much for this sort:

```
> db.users.find().sort({ "age" : 1, "username" : 1 })
```

This sorts by "age" and then "username", so a strict sorting by "username" isn't terribly helpful. To optimize this sort, you could make an index on "age" *and* "username":

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

This is called a compound index and is useful if your query has multiple sort directions or multiple keys in the criteria. A compound index is an index on more than one field.

Suppose we have a *users* collection that looks something like this, if we run a query with no sorting (called natural order):

```
> db.users.find({}, { "_id" : 0, "i" : 0, "created" : 0 })
{ "username" : "user0", "age" : 69 }
{ "username" : "user1", "age" : 50 }
{ "username" : "user2", "age" : 88 }
{ "username" : "user3", "age" : 52 }
{ "username" : "user4", "age" : 74 }
{ "username" : "user5", "age" : 104 }
{ "username" : "user6", "age" : 59 }
{ "username" : "user7", "age" : 102 }
{ "username" : "user8", "age" : 94 }
{ "username" : "user9", "age" : 7 }
{ "username" : "user10", "age" : 80 }
...
```

If we index this collection by { "age" : 1, "username" : 1 }, the index will have a form we can represent as follows:

```
[0, user100020] -> 8623513776
[0, user1002] -> 8599246768
[0, user100388] -> 8623560880
...
[0, user100414] -> 8623564208
[1, user100113] -> 8623525680
[1, user100280] -> 8623547056
```

```
[1, user100551] -> 8623581744
...
[1, user100626] -> 8623591344
[2, user100191] -> 8623535664
[2, user100195] -> 8623536176
[2, user100197] -> 8623536432
...
```

Each index entry contains an age and a username and points to record identifier. A record identifier is used internally by the storage engine to locate the data for a document. Note that "age" fields are ordered to be strictly ascending and, within each age, "username"s are also in ascending order. In this example dataset, each age has approximately 8,000 usernames associated with it. Here we've included only those necessary to convey the general idea.

The way MongoDB uses this index depends on the type of query you're doing. These are the three most common ways:

```
db.users.find({"age" : 21}).sort({"username" : -1})
```

This is an equality query, which searches for a single value. There may be multiple documents with that value. Due to the second field in the index, the results are already in the correct order for the sort: MongoDB can start with the last match for {"age" : 21} and traverse the index in order:

```
[21, user100154] -> 8623530928
[21, user100266] -> 8623545264
[21, user100270] -> 8623545776
[21, user100285] -> 8623547696
[21, user100349] -> 8623555888
...
```

This type of query is very efficient: MongoDB can jump directly to the correct age and doesn't need to sort the results because traversing the index returns the data in the correct order.

Note that sort direction doesn't matter: MongoDB can traverse the index in either direction.

```
db.users.find({"age" : {"$gte" : 21, "$lte" : 30}})
```

This is a multi-value query, which looks for documents matching multiple values (in this case, all ages between 21 and 30). This particular type of query is sometimes also called a range query query. MongoDB will use the first key in the index, "age", to return the matching documents, like so:

```
[21, user100154] -> 8623530928
[21, user100266] -> 8623545264
[21, user100270] -> 8623545776
...
[21, user999390] -> 8765250224
[21, user999407] -> 8765252400
[21, user999600] -> 8765277104
[22, user100017] -> 8623513392
...
[29, user999861] -> 8765310512
[30, user100098] -> 8623523760
[30, user100155] -> 8623531056
[30, user100168] -> 8623532720
...
```

In general, if MongoDB uses an index for a query it will return the resulting documents in index order.

```
db.users.find({ "age" : { "$gte" : 21, "$lte" :
30 } }).sort({ "username" : 1 })
```

This is a multi-value query, like the previous one, but this time it has a sort. As before, MongoDB will use the index to match the criteria. However, the index doesn't return the usernames in sorted order and the query requested that the results be sorted by username. MongoDB will need to sort the results in memory before returning them rather than simply traversing an index in which the documents are already sorted in the desired order. This type of query is usually less efficient as a consequence.

Of course, the speed depends on how many results match your criteria: if your result set is only a couple of documents, MongoDB won't have much work to do to sort them. If there are more results, it will be slower or may not work at all: if you have more than 32 MB of results MongoDB will just error out, refusing to sort that much data:

```
Error: error: {
    "ok" : 0,
    "errmsg" : "Executor error during find command: OperationFailed
Sort operation used more than the maximum 33554432 bytes of RAM. Add
an index, or specify a smaller limit.",
    "code" : 96,
    "codeName" : "OperationFailed"
}
```

One other index you can use in the last example is the same keys in reverse order: `{"username" : 1, "age" : 1}`. MongoDB will then traverse all the index entries, but

in the order you want them back in. It will pick out the matching documents using the "age" part of the index:

```
[user0, 4]
[user1, 67]
[user10, 11]
[user100, 92]
[user1000, 10]
[user10000, 31]
[user100000, 21] -> 8623511216
[user100001, 52]
[user100002, 69]
[user100003, 27] -> 8623511600
[user100004, 22] -> 8623511728
[user100005, 95]
...
...
```

This is good in that it does not require any giant in-memory sorts. However, it does have to scan the entire index to find all matches. Putting the sort key first is generally a good strategy when designing compound indexes. As we'll see in the next section, this is one of several best practices when considering how to construct compound indexes with consideration for equality queries, multi-value queries, and sorting.

## How MongoDB Selects an Index

Now let's take a look at how MongoDB chooses an index to satisfy a query. Let's imagine we have five indexes. When a query comes in, MongoDB looks at the query's shape. The shape has to do with what fields are being searched on and additional information, such as whether or not there a sort. Based on that information, the system identifies a set of candidate indexes that it might be able to use in satisfying the query.

Let's assume we have a query come in, and three of our five indexes are identified as candidates for this query. MongoDB will then create three query plans, one each for these indexes. In three parallel threads, MongoDB will run the query such that each one will use a different index. The objective here is to see which one is able to return results the fastest.

Visually, we can think of this as a race, something like in the graphic below. The idea here is that the first query plan to reach a goal state is the winner. But more importantly, going forward it will be selected as the index to use for queries that have that same query shape.

TODO: NEED A GRAPHIC HERE.

To win the race, a query thread must be the first to either return all query results or return a threshold number of results (e.g., 100 documents) in sort order. The sort order portion of this is

important given how expensive it is to perform in-memory sorts.

The real value of racing several query plans against one another is that for subsequent queries that have the same query shape, the MongoDB server knows which index to select. The server maintains a cache of query plans. A winning plan is stored in the cache for future use for queries of that shape. Over time, as a collection changes and as the indexes change, eventually a query plan might be evicted from the cache and MongoDB will, again, experiment with possible query plans to find the one that works best for the current collection and set of indexes. Other events that will lead to plans being evicted from the cache are if we rebuild a given index or add or drop an index. Finally, the query plan cache does not survive a restart of a mongod process.

## Using Compound Indexes

In the section above, we've been using compound indexes, which are indexes with more than one key in them. Compound indexes are a little more complicated to think about than single-key indexes, but they are very powerful. This section covers them in more depth.

Now we will walk through an example that gives you an idea of the type of thinking you need to do when you are designing compound indexes. The goal is that our read and write operations are as efficient as possible. But as with so many things, this requires some upfront thinking and some experimentation.

To be sure you get the right indexes in place, it is necessary to test your indexes under some real world workloads and make adjustments from there. However, there are some best practices you will want to apply first as you design your indexes.

First, we need to consider the selectivity of our index. We are interested in the degree to which, for a given query pattern, the index is going to minimize the number of records scanned. We need to consider selectivity in light of all operations necessary to satisfy a query and sometimes make trade offs. We will need to consider, for example, how sorts are handled.

Let's look at an example. For this, we will use a student data set containing approximately one million records. Documents in this dataset resemble the following.

```
{  
    "_id" : ObjectId("585d817db4743f74e2da067c") ,  
    "student_id" : 0,  
    "scores" : [  
        {  
            "type" : "exam",  
            "score" : 38.05000060199827  
        },  
        {  
    ]}
```

```

        "type" : "quiz",
        "score" : 79.45079445008987
    },
    {
        "type" : "homework",
        "score" : 74.50150548699534
    },
    {
        "type" : "homework",
        "score" : 74.68381684615845
    }
],
"class_id" : 127
}

```

We will begin with two indexes already created and look at how MongoDB uses these indexes (or doesn't) in order to satisfy queries. These two indexes are created as follows.

```

db.students.createIndex({"class_id": 1})
db.students.createIndex({student_id: 1, class_id: 1})

```

In working with this data set, we will consider the following query, because it illustrates several of the issues that we have to think about in designing our indexes.

```

db.students.find({student_id:{$gt:500000}, class_id:54})
    .sort({student_id:1})
    .explain("executionStats")

```

Note that in this query we are requesting all records with an id greater than 500,000, so about half of the records. We are also constraining the search to records for the class with id 54. There are about 500 classes represented in this dataset. Finally, we are sorting in ascending order student\_id. Note that this is the same field on which we are doing a multi-value query. Throughout this example we will look at the execution stats that the explain method provides to illustrate how MongoDB will handle this query.

If we run the query above, the output of the explain method tells us how MongoDB used indexes to satisfy this query.

```

{
  "queryPlanner": {
    "plannerVersion": 1,
    "namespace": "school.students",
    "indexFilterSet": false,
    "parsedQuery": {
      "student_id": {
        "$gt": 500000
      },
      "class_id": 54
    }
  }
}

```

```

"$and": [
  {
    "class_id": {
      "$eq": 54
    }
  },
  {
    "student_id": {
      "$gt": 500000
    }
  }
]
},
"winningPlan": {
  "stage": "FETCH",
  "inputStage": {
    "stage": "IXSCAN",
    "keyPattern": {
      "student_id": 1,
      "class_id": 1
    },
    "indexName": "student_id_1_class_id_1",
    "isMultiKey": false,
    "multiKeyPaths": {
      "student_id": [ ],
      "class_id": [ ]
    },
    "isUnique": false,
    "isSparse": false,
    "isPartial": false,
    "indexVersion": 2,
    "direction": "forward",
    "indexBounds": {
      "student_id": [
        "(500000.0, inf.0]"
      ],
      "class_id": [
        "[54.0, 54.0]"
      ]
    }
  }
},
"rejectedPlans": [
  {
    "stage": "SORT",
    "sortPattern": {
      "student_id": 1
    },
    "inputStage": {
      "stage": "SORT_KEY_GENERATOR",
      "inputStage": {
        "stage": "FETCH",
        "filter": {
          "student_id": {

```

```
        "$gt": 500000
    }
},
"inputStage": {
    "stage": "IXSCAN",
    "keyPattern": {
        "class_id": 1
    },
    "indexName": "class_id_1",
    "isMultiKey": false,
    "multiKeyPaths": {
        "class_id": [ ]
    },
    "isUnique": false,
    "isSparse": false,
    "isPartial": false,
    "indexVersion": 2,
    "direction": "forward",
    "indexBounds": {
        "class_id": [
            "[54.0, 54.0]"
        ]
    }
}
}
]
},
"executionStats": {
    "executionSuccess": true,
    "nReturned": 9903,
    "executionTimeMillis": 4325,
    "totalKeysExamined": 850477,
    "totalDocsExamined": 9903,
    "executionStages": {
        "stage": "FETCH",
        "nReturned": 9903,
        "executionTimeMillisEstimate": 3485,
        "works": 850478,
        "advanced": 9903,
        "needTime": 840574,
        "needYield": 0,
        "saveState": 6861,
        "restoreState": 6861,
        "isEOF": 1,
        "invalidates": 0,
        "docsExamined": 9903,
        "alreadyHasObj": 0,
        "inputStage": {
            "stage": "IXSCAN",
            "nReturned": 9903,
            "executionTimeMillisEstimate": 2834,
            "works": 850478,
```

```

    "advanced": 9903,
    "needTime": 840574,
    "needYield": 0,
    "saveState": 6861,
    "restoreState": 6861,
    "isEOF": 1,
    "invalidates": 0,
    "keyPattern": {
        "student_id": 1,
        "class_id": 1
    },
    "indexName": "student_id_1_class_id_1",
    "isMultiKey": false,
    "multiKeyPaths": {
        "student_id": [ ],
        "class_id": [ ]
    },
    "isUnique": false,
    "isSparse": false,
    "isPartial": false,
    "indexVersion": 2,
    "direction": "forward",
    "indexBounds": {
        "student_id": [
            "(500000.0, inf.0]"
        ],
        "class_id": [
            "[54.0, 54.0]"
        ]
    },
    "keysExamined": 850477,
    "seeks": 840575,
    "dupsTested": 0,
    "dupsDropped": 0,
    "seenInvalidated": 0
}
}
},
"serverInfo": {
    "host": "SGB-MBP.local",
    "port": 27017,
    "version": "3.4.1",
    "gitVersion": "5e103c4f5583e2566a45d740225dc250baacfbd7"
},
"ok": 1
}

```

As with most data output from MongoDB, the explain output is JSON. I want to look first at the bottom half of this output, which is almost entirely the executionStats. The executionStats field contains statistics that describe the completed query execution for the winning query plan. We will look at query plans and the query plan output from explain a little later in this example.

Within executionStats, first we will look at totalKeysExamined. This is how many keys within the index MongoDB walked through in order to generate the result set. We can compare totalKeysExamined to nReturned to get a sense for how much of the index MongoDB had to traverse in order to find just the documents matching the query. In this case, there were 850477 index keys examined in order to locate the 9903 matching documents.

This means that the index used in order to satisfy this query was not very selective. This is further emphasized by the fact that this query took more than 4.3 seconds to run. We can that upon review of the executionTimeMillis field. Selectivity is one of our key objectives when we are designing an index, so let's figure out where we went wrong with the existing indexes for this query.

Near the top of the explain output is the winning query plan (see the field `winningPlan`). A query plan describes the steps MongoDB used to satisfy a query. This is, in JSON form, a specific outcome for racing a couple of different query plans against one another. In particular, we are interested in what indexes were used and whether MongoDB had to do an in-memory sort. Below the winning plan are the rejected plans. We'll look at both.

In this case, the winning plan used a compound index based on student\_id and class\_id. This is evident in the following portion of the explain output.

```
"winningPlan": {  
    "stage": "FETCH",  
    "inputStage": {  
        "stage": "IXSCAN",  
        "keyPattern": {  
            "student_id": 1,  
            "class_id": 1  
        },  
    },
```

The explain output presents the query plan as a tree of stages; i.e. a stage can have an inputStage or, if the stage has multiple child stages, inputStages. An input stage provides the documents or index keys to its parent. In this case, there was one input stage, an index scan, and that scan provided the record ids for documents matching the query to its parent, the FETCH stage. The FETCH stage, then, will retrieve the documents themselves and return them in batches as the client requests them.

The losing query plan -- there is only one -- would have used an index based on `class_id`, but then it would have had to do an in memory sort. That is what the below portion of this particular query plan means. When you see a `SORT` stage in a query plan, it means that MongoDB would have been unable to sort the results set in the database using an index. Rather, it would have had to do an in-memory sort.

```
"rejectedPlans": [
  {
    "stage": "SORT",
    "sortPattern": {
      "student_id": 1
    },
  },
```

For this query, the index that won is one that was able to return sorted output. To win it only had to reach the threshold number of sorted result documents (i.e. 101). For the other plan to win, that query thread would have had to return the entire result set (nearly 10,000 documents) first, since those would then need to be sorted in memory.

The issue here is one of selectivity. The multi-value query we are running specifies a broad range of student\_id values, because it's requesting records for which the student\_id is greater than 500,000. That's about half the records in our collection. Here again, for convenience, is the query we are running.

```
db.students.find({student_id:{$gt:500000}, class_id:54})
  .sort({student_id:1})
  .explain("executionStats")
```

Now, I'm sure you can see where we are headed here. This query contains both a multi-value portion and an equality portion. The equality portion is that we are asking for all records in which class\_id is equal to 54. There are only about 500 classes in this data set and while there are a large number students with grades in those classes, class\_id would serve as a much more selective basis on which to execute this query. It is this value that constrains our result set to just under 10,000 records rather than the approximately 850,000 that were identified by the multi-value portion of this query.

It would be better, given the indexes we have if we use the index based on just class\_id. That is to say, the one in the losing query plan. MongoDB does provide a way of forcing the database to use a particular index. However, I cannot stress strongly enough that you should use this with caution. It is not something you should use in a production deployment, but it is a way of overriding what would be the outcome of the query planner.

The cursor hint() method enables us to specify a particular index to use, either by specifying its shape or its name. If we change our query slightly to use hint, as in the following example, the explain output will be quite different.

```
db.students.find({student_id:{$gt:500000}, class_id:54})
  .sort({student_id:1})
```

```
.hint({class_id:1})  
.explain("executionstats")
```

In the resulting explain output, we will see that we are now down from having scanned roughly 850,000 index keys to just about 20,000 in order to get to our results set of just under 10,000. In addition, the execution time is only 272 milliseconds rather than 4.3 seconds as we saw with the query plan using the other index. Now, the fact is, what we would really like to see is that nReturned is very close to totalKeysExamined. In addition, we would like avoid having to use hint() in order to more efficiently execute this query. The way to address both of these concerns is to design a better index.

```
{  
  "queryPlanner": {  
    "plannerVersion": 1,  
    "namespace": "school.students",  
    "indexFilterSet": false,  
    "parsedQuery": {  
      "$and": [  
        {  
          "class_id": {  
            "$eq": 54  
          }  
        },  
        {  
          "student_id": {  
            "$gt": 500000  
          }  
        }  
      ]  
    },  
    "winningPlan": {  
      "stage": "SORT",  
      "sortPattern": {  
        "student_id": 1  
      },  
      "inputStage": {  
        "stage": "SORT_KEY_GENERATOR",  
        "inputStage": {  
          "stage": "FETCH",  
          "filter": {  
            "student_id": {  
              "$gt": 500000  
            }  
          },  
          "inputStage": {  
            "stage": "IXSCAN",  
            "keyPattern": {  
              "class_id": 1  
            },  
            "indexType": "BTREE"  
          }  
        }  
      }  
    }  
  }  
}
```

```
        "indexName": "class_id_1",
        "isMultiKey": false,
        "multiKeyPaths": {
            "class_id": [ ]
        },
        "isUnique": false,
        "isSparse": false,
        "isPartial": false,
        "indexVersion": 2,
        "direction": "forward",
        "indexBounds": {
            "class_id": [
                "[54.0, 54.0]"
            ]
        }
    }
},
"rejectedPlans": [ ]
},
"executionStats": {
    "executionSuccess": true,
    "nReturned": 9903,
    "executionTimeMillis": 272,
    "totalKeysExamined": 20076,
    "totalDocsExamined": 20076,
    "executionStages": {
        "stage": "SORT",
        "nReturned": 9903,
        "executionTimeMillisEstimate": 248,
        "works": 29982,
        "advanced": 9903,
        "needTime": 20078,
        "needYield": 0,
        "saveState": 242,
        "restoreState": 242,
        "isEOF": 1,
        "invalidates": 0,
        "sortPattern": {
            "student_id": 1
        },
        "memUsage": 2386623,
        "memLimit": 33554432,
        "inputStage": {
            "stage": "SORT_KEY_GENERATOR",
            "nReturned": 9903,
            "executionTimeMillisEstimate": 203,
            "works": 20078,
            "advanced": 9903,
            "needTime": 10174,
            "needYield": 0,
            "saveState": 242,
            "restoreState": 242,
            "isEOF": 1
        }
    }
}
```

```
"isEOF": 1,
"invalidates": 0,
"inputStage": {
    "stage": "FETCH",
    "filter": {
        "student_id": {
            "$gt": 500000
        }
    },
    "nReturned": 9903,
    "executionTimeMillisEstimate": 192,
    "works": 20077,
    "advanced": 9903,
    "needTime": 10173,
    "needYield": 0,
    "saveState": 242,
    "restoreState": 242,
    "isEOF": 1,
    "invalidates": 0,
    "docsExamined": 20076,
    "alreadyHasObj": 0,
    "inputStage": {
        "stage": "IXSCAN",
        "nReturned": 20076,
        "executionTimeMillisEstimate": 45,
        "works": 20077,
        "advanced": 20076,
        "needTime": 0,
        "needYield": 0,
        "saveState": 242,
        "restoreState": 242,
        "isEOF": 1,
        "invalidates": 0,
        "keyPattern": {
            "class_id": 1
        },
        "indexName": "class_id_1",
        "isMultiKey": false,
        "multiKeyPaths": {
            "class_id": []
        },
        "isUnique": false,
        "isSparse": false,
        "isPartial": false,
        "indexVersion": 2,
        "direction": "forward",
        "indexBounds": {
            "class_id": [
                "[54.0, 54.0]"
            ]
        },
        "keysExamined": 20076,
        "seeks": 1,
        "dupsTested": 0,
        "isEOF": 1,
        "invalidates": 0
    }
}
```

```

        "dupsDropped": 0,
        "seenInvalidated": 0
    }
}
}
},
"serverInfo": {
    "host": "SGB-MBP.local",
    "port": 27017,
    "version": "3.4.1",
    "gitVersion": "5e103c4f5583e2566a45d740225dc250baacfbd7"
},
"ok": 1
}

```

---

A better index for the query pattern in question is one based on `class_id` and `student_id`, in that order. With `class_id` as the prefix, we are using the equality filter in our query to restrict the keys considered within the index. This is the most selective component of our query and, therefore, effectively constrains the number of keys MongoDB needs to consider in our to satisfy this query. So let's build this index as follows.

```
db.students.createIndex({class_id:1, student_id:1})
```

---

While not true for absolutely every data set, in general, you should design compound indexes such that fields on which you will be using equality filters come before those on which your application will use multi-value filters.

With our new index in place, if we re-run our query, this time no hinting is required and can see from the `executionStats` in the `explain` output that we have a fast query (37 milliseconds) for which the number of results returned (`nReturned`) is equal to the number of keys scanned in the index (`totalKeysExamined`). We can also see that this is due to the fact that the `executionStages`, which reflect the winning query plan, contain an index scan that makes use of the new index we created.

```

...
"executionStats": {
    "executionSuccess": true,
    "nReturned": 9903,
    "executionTimeMillis": 37,
    "totalKeysExamined": 9903,
    "totalDocsExamined": 9903,
    "executionStages": {
        "stage": "FETCH",
        "nReturned": 9903,

```

```
"executionTimeMillisEstimate": 36,
"works": 9904,
"advanced": 9903,
"needTime": 0,
"needYield": 0,
"saveState": 81,
"restoreState": 81,
"isEOF": 1,
"invalidates": 0,
"docsExamined": 9903,
"alreadyHasObj": 0,
"inputStage": {
    "stage": "IXSCAN",
    "nReturned": 9903,
    "executionTimeMillisEstimate": 0,
    "works": 9904,
    "advanced": 9903,
    "needTime": 0,
    "needYield": 0,
    "saveState": 81,
    "restoreState": 81,
    "isEOF": 1,
    "invalidates": 0,
    "keyPattern": {
        "class_id": 1,
        "student_id": 1
    },
    "indexName": "class_id_1_student_id_1",
    "isMultiKey": false,
    "multiKeyPaths": {
        "class_id": [ ],
        "student_id": [ ]
    },
    "isUnique": false,
    "isSparse": false,
    "isPartial": false,
    "indexVersion": 2,
    "direction": "forward",
    "indexBounds": {
        "class_id": [
            "[54.0, 54.0]"
        ],
        "student_id": [
            "(500000.0, inf.0]"
        ]
    },
    "keysExamined": 9903,
    "seeks": 1,
    "dupsTested": 0,
    "dupsDropped": 0,
    "seenInvalidated": 0
}
},
} ,
```

Considering what we know about how indexes are built, you can probably see why this works. The (class\_id, student\_id) index is composed of keys pairs such as the following. Since the student ids are ordered within these key pairs, in order to satisfy our sort, MongoDB simply needs to walk all the key pairs beginning with the first one for class\_id 54.

```
...
[53, 999617]
[53, 999780]
[53, 999916]
[54, 500001]
[54, 500009]
[54, 500048]
...
```

In considering the design of a compound index, we need to know how to address equality filters, multi-value filters, and sort components of common query patterns that will make use of the index. It is necessary to consider these three factors for all compound indexes and if you design your index to balance these concerns correctly, you will get the best performance out of MongoDB for your queries. While we've addressed all three factors for our example query with the class\_id, student\_id index, the query as written represents a special case of the compound index problem because we're sorting on one of the fields we are also filtering on.

To remove the special-case nature of this example, let's sort on final grade instead, changing our query to the following.

```
db.students.find({student_id:{$gt:500000}, class_id:54})
    .sort({final_grade:1})
    .explain("executionStats")
```

If we run this query and look at the explain output, we see that we're now doing an in-memory sort. While the query is still fast at only 136 milliseconds, it is an order of magnitude slower than when sorting on student\_id, because we are now doing an in-memory sort. We can see that we are doing an in-memory sort, because the winning query plan now contains a SORT stage.

```
...
"executionStats": {
    "executionSuccess": true,
    "nReturned": 9903,
    "executionTimeMillis": 136,
    "totalKeysExamined": 9903,
    "totalDocsExamined": 9903,
    "executionStages": {
```

```
"stage": "SORT",
"nReturned": 9903,
"executionTimeMillisEstimate": 36,
"works": 19809,
"advanced": 9903,
"needTime": 9905,
"needYield": 0,
"saveState": 315,
"restoreState": 315,
"isEOF": 1,
"invalidates": 0,
"sortPattern": {
    "final_grade": 1
},
"memUsage": 2386623,
"memLimit": 33554432,
"inputStage": {
    "stage": "SORT_KEY_GENERATOR",
    "nReturned": 9903,
    "executionTimeMillisEstimate": 24,
    "works": 9905,
    "advanced": 9903,
    "needTime": 1,
    "needYield": 0,
    "saveState": 315,
    "restoreState": 315,
    "isEOF": 1,
    "invalidates": 0,
    "inputStage": {
        "stage": "FETCH",
        "nReturned": 9903,
        "executionTimeMillisEstimate": 24,
        "works": 9904,
        "advanced": 9903,
        "needTime": 0,
        "needYield": 0,
        "saveState": 315,
        "restoreState": 315,
        "isEOF": 1,
        "invalidates": 0,
        "docsExamined": 9903,
        "alreadyHasObj": 0,
        "inputStage": {
            "stage": "IXSCAN",
            "nReturned": 9903,
            "executionTimeMillisEstimate": 12,
            "works": 9904,
            "advanced": 9903,
            "needTime": 0,
            "needYield": 0,
            "saveState": 315,
            "restoreState": 315,
            "isEOF": 1,
            "invalidates": 0,
```

```

    "keyPattern": {
        "class_id": 1,
        "student_id": 1
    },
    "indexName": "class_id_1_student_id_1",
    "isMultiKey": false,
    "multiKeyPaths": {
        "class_id": [ ],
        "student_id": [ ]
    },
    "isUnique": false,
    "isSparse": false,
    "isPartial": false,
    "indexVersion": 2,
    "direction": "forward",
    "indexBounds": {
        "class_id": [
            "[54.0, 54.0]"
        ],
        "student_id": [
            "(500000.0, inf.0)"
        ]
    },
    "keysExamined": 9903,
    "seeks": 1,
    "dupsTested": 0,
    "dupsDropped": 0,
    "seenInvalidated": 0
}
}
}
},
...

```

If we can avoid an in-memory sort with a better index design, we should. This will allow us to scale more easily with respect to size of the data set and system load.

In order to do that, we are going to have to make a trade-off. This is commonly the case when designing compound indexes.

As is so often necessary for compound indexes, in order to avoid an in-memory sort we need to examine more keys than the number of documents we return. To use the index to sort, MongoDB needs to be able to walk the index keys in order. This means that we need to include the sort field among the compound index keys.

The keys in our new compound index should be ordered as follows: [class\_id, final\_grade, student\_id]. Note that we include the sort component immediately after the equality filter, but before the multi-value filter. This index will very selectively narrow the set of keys considered

for this query. Then by walking the key triplets matching the equality filter in this index, MongoDB can identify the records that match the multi-value filter and those records will be ordered properly by final grade in ascending order.

This compound index forces MongoDB to examine keys for more documents than will end up being in our results set. However, by using the index to ensure we have sorted documents, we save execution time. We can construct the new index using the following command.

```
db.students.createIndex({class_id:1, final_grade:1, student_id:1})
```

If we, once again issue our query.

```
db.students.find({student_id:{$gt:500000}, class_id:54})
    .sort({final_grade:1})
    .explain("executionStats")
```

We get the following executionStats in the output from explain. This will vary depending on your hardware and what else is going on in the system, but you can see that the willing plan no longer includes an in-memory sort. It is, instead using the index we just created to satisfy the query, including the sort.

```
"executionStats": {
    "executionSuccess": true,
    "nReturned": 9903,
    "executionTimeMillis": 42,
    "totalKeysExamined": 9905,
    "totalDocsExamined": 9903,
    "executionStages": {
        "stage": "FETCH",
        "nReturned": 9903,
        "executionTimeMillisEstimate": 34,
        "works": 9905,
        "advanced": 9903,
        "needTime": 1,
        "needYield": 0,
        "saveState": 82,
        "restoreState": 82,
        "isEOF": 1,
        "invalidates": 0,
        "docsExamined": 9903,
        "alreadyHasObj": 0,
        "inputStage": {
            "stage": "IXSCAN",
            "nReturned": 9903,
            "executionTimeMillisEstimate": 24,
            "works": 9905,
```

```

    "advanced": 9903,
    "needTime": 1,
    "needYield": 0,
    "saveState": 82,
    "restoreState": 82,
    "isEOF": 1,
    "invalidates": 0,
    "keyPattern": {
        "class_id": 1,
        "final_grade": 1,
        "student_id": 1
    },
    "indexName": "class_id_1_final_grade_1_student_id_1",
    "isMultiKey": false,
    "multiKeyPaths": {
        "class_id": [ ],
        "final_grade": [ ],
        "student_id": [ ]
    },
    "isUnique": false,
    "isSparse": false,
    "isPartial": false,
    "indexVersion": 2,
    "direction": "forward",
    "indexBounds": {
        "class_id": [
            "[54.0, 54.0]"
        ],
        "final_grade": [
            "[MinKey, MaxKey]"
        ],
        "student_id": [
            "(500000.0, inf.0)"
        ]
    },
    "keysExamined": 9905,
    "seeks": 2,
    "dupsTested": 0,
    "dupsDropped": 0,
    "seenInvalidated": 0
}
},
},

```

The above provides a concrete example of some best practices for designing compound indexes. While these guidelines do not hold for every situation, they do for most and should be the first ideas you consider when constructing a compound index.

When designing a compound index:

- Keys for equality filters should appear first.

- Keys used for sorting should appear before multi-value fields.
- Keys for multi-value filters should appear last.

Design your compound index using these guidelines and then test it under real-world workloads for the range of query patterns your index is designed to support.

## CHOOSING KEY DIRECTIONS

So far, all of our index entries have been sorted in ascending, or least-to-greatest, order. However, if you need to sort on two (or more) criteria, you may need to have index keys go in different directions. For example, suppose we want to sort the collection above by youngest to oldest and usernames from Z-A. Our previous indexes would not be very efficient for this problem: within each age group users were sorted by "username" ascending A-Z, not Z-A. The compound indexes above do not hold the values in any useful order for getting "age" ascending and "username" descending.

To optimize compound sorts in different directions, use an index with matching directions. In this example, we could use `{"age" : 1, "username" : -1}`, which would organize the data as follows:

```
[21, user999600] -> 8765277104
[21, user999407] -> 8765252400
[21, user999390] -> 8765250224
...
[21, user100270] -> 8623545776
[21, user100266] -> 8623545264
[21, user100154] -> 8623530928
...
[30, user100168] -> 8623532720
[30, user100155] -> 8623531056
[30, user100098] -> 8623523760
```

The ages are arranged from youngest to oldest and, within each age, usernames are sorted from Z to A (or, rather, "9" to "0", given our usernames).

If our application also needed to optimize sorting by `{"age" : 1, "username" : 1}`, we would have to create a second index with those directions. To figure out which directions to use for an index, simply match the directions your sort is using. Note that inverse indexes (multiplying each direction by -1) are equivalent: `{"age" : 1, "username" : -1}` suits the same queries that `{"age" : -1, "username" : 1}` does.

Index direction only really matters when you're sorting based on multiple criteria. If you're only

sorting by a single key, MongoDB can just as easily read the index in the opposite order. For example, if you had a sort on `{"age" : -1}` and an index on `{"age" : 1}`, MongoDB could optimize it just as well as if you had an index on `{"age" : -1}` (so don't create both!). The direction only matters for multikey sorts.

## USING COVERED INDEXES

In the examples above, the query was always used to find the correct document, and then follow a pointer back to fetch the actual document. However, if your query is only looking for the fields that are included in the index, it does not need to fetch the document. When an index contains all the values requested by the user, it is considered to be covering a query. Whenever practical, use covered indexes in preference to going back to documents. You can make your working set much smaller that way.

To make sure a query can use the index only, you should use projections (see [“Specifying Which Keys to Return”](#)) to not return the `_id` field (unless it is part of the index). You may also have to index fields that you aren't querying on, so you should balance your need for faster queries with the overhead this will add on writes.

If you run an `explain` on a covered query, the explain result has an IXSCAN stage that is **not** a descendant of a FETCH stage, and in the `executionStats`, the `totalDocsExamined` is 0.

If you index a field containing arrays, that index can never cover a query (due to the way arrays are stored in indexes, this is covered in more depth in [“Indexing Objects and Arrays”](#)). Even if you exclude the array field from the fields returned, you cannot cover a query using such an index.

## IMPLICIT INDEXES

Compound indexes can do “double duty” and act like different indexes for different queries. If we have an index on `{"age" : 1, "username" : 1}`, the `"age"` field is sorted identically to the way it would be if you had an index on just `{"age" : 1}`. Thus, the compound index can be used the way an index on `{"age" : 1}` by itself would be.

This can be generalized to as many keys as necessary: if an index has N keys, you get a “free” index on any prefix of those keys. For example, if we have an index that looks like `{"a": 1, "b": 1, "c": 1, ..., "z": 1}`, we effectively have indexes on `{"a": 1}`, `{"a": 1, "b": 1}`, `{"a": 1, "b": 1, "c": 1}`, and so on.

Note that this doesn't hold for *any* subset of keys: queries that would use the index `{"b": 1}` or `{"a": 1, "c": 1}` (for example) will not be optimized: only queries that can use a prefix of the index can take advantage of it.

## How \$-Operators Use Indexes

Some queries can use indexes more efficiently than others; some queries cannot use indexes at all. This section covers how various query operators are handled by MongoDB.

### INEFFICIENT OPERATORS

In general, negation is inefficient. "\$ne" queries can use an index, but not very well. They must look at all the index entries other than the one specified by the "\$ne", so it basically has to scan the entire index. For example, here are the index ranges traversed for such a query:

```
> db.example.find({"i" : {"$ne" : 3}}).explain()
{
  "cursor" : "BtreeCursor i_1 multi",
  ...
  "indexBounds" : {
    "i" : [
      [
        {
          "$minElement" : 1
        },
        3
      ],
      [
        3,
        {
          "$maxElement" : 1
        }
      ]
    ],
    ...
  }
}
```

This query looks at all index entries less than 3 and all index entries greater than 3. This can be efficient if a large swath of your collection is 3, but otherwise it must check almost everything.

"\$not" can sometimes use an index but often does not know how. It can reverse basic ranges (`{"key" : {"$lt" : 7}}` becomes `{"key" : {"$gte" : 7}}`) and regular expressions. However, most other queries with "\$not" will fall back to doing a table scan. "\$nin" always uses a table scan.

If you need to perform one of these types of queries quickly, figure out if there's another clause that you could add to the query that could use an index to filter the result set down to a small number of documents before MongoDB attempts to do non-indexed matching.

### RANGES

Compound indexes can help MongoDB efficiently execute queries with multiple clauses. When designing an index with multiple fields, put fields that will be used in exact matches first (e.g., `"x" : 1`) and ranges last (e.g., `"y": {"$gt" : 3, "$lt" : 5}`). This allows the query to find an exact value for the first index key and then search within that for a second index range. For example, suppose we were querying for a specific age and a range of usernames using an `{"age" : 1, "username" : 1}` index. We would get fairly exact index bounds:

```
> db.users.find({"age" : 47,
... "username" : {"$gt" : "user5", "$lt" : "user8"}}).explain()
{
  "cursor" : "BtreeCursor age_1_username_1",
  "n" : 2788,
  "nscanned" : 2788,
  ...,
  "indexBounds" : {
    "age" : [
      [
        47,
        47
      ]
    ],
    "username" : [
      [
        "user5",
        "user8"
      ]
    ]
  },
  ...
}
```

The query goes directly to `"age" : 47` and then searches within that for usernames between `"user5"` and `"user8"`.

Conversely, suppose we use an index on `{"username" : 1, "age" : 1}`. This changes the query plan, as the query must look at all users between `"user5"` and `"user8"` and pick out the ones with `"age" : 47`:

```
> db.users.find({"age" : 47,
... "username" : {"$gt" : "user5", "$lt" : "user8"}}).explain()
{
  "cursor" : "BtreeCursor username_1_age_1",
  "n" : 2788,
  "nscanned" : 319499,
  ...,
```

```

    "indexBounds" : {
        "username" : [
            [
                "user5",
                "user8"
            ]
        ],
        "age" : [
            [
                47,
                47
            ]
        ]
    },
    "server" : "spock:27017"
}

```

This forces MongoDB to scan 10 times the number of index entries as using the previous index would. Using two ranges in a query basically always forces this less-efficient query plan.

## OR QUERIES

As of this writing, MongoDB can only use one index per query. That is, if you create one index on `{"x" : 1}` and another index on `{"y" : 1}` and then do a query on `{"x" : 123, "y" : 456}`, MongoDB will use one of the indexes you created, not use both. The only exception to this rule is `"$or"`. `"$or"` can use one index per `$or` clause, as `$or` performs two queries and then merges the results:

```

> db.foo.find({"$or" : [{"x" : 123}, {"y" : 456}]}).explain()
{
    "clauses" : [
        {
            "cursor" : "BtreeCursor x_1",
            "isMultiKey" : false,
            "n" : 1,
            "nscannedObjects" : 1,
            "nscanned" : 1,
            "nscannedObjectsAllPlans" : 1,
            "nscannedAllPlans" : 1,
            "scanAndOrder" : false,
            "indexOnly" : false,
            "nYields" : 0,
            "nChunkSkips" : 0,
            "millis" : 0,
            "indexBounds" : {
                "x" : [
                    [
                        123,
                        123
                    ]
                ]
            }
        }
    ]
}

```

```

        ]
    }
},
{
    "cursor" : "BtreeCursor y_1",
    "isMultiKey" : false,
    "n" : 1,
    "nscannedObjects" : 1,
    "nscanned" : 1,
    "nscannedObjectsAllPlans" : 1,
    "nscannedAllPlans" : 1,
    "scanAndOrder" : false,
    "indexOnly" : false,
    "nYields" : 0,
    "nChunkSkips" : 0,
    "millis" : 0,
    "indexBounds" : {
        "y" : [
            [
                456,
                456
            ]
        ]
    }
},
],
"n" : 2,
"nscannedObjects" : 2,
"nscanned" : 2,
"nscannedObjectsAllPlans" : 2,
"nscannedAllPlans" : 2,
"millis" : 0,
"server" : "spock:27017"
}

```

As you can see, this explain is the conglomerate of two separate queries. In general, doing two queries and merging the results is much less efficient than doing a single query; thus, whenever possible, prefer "\$in" to "\$or".

If you must use an \$or, keep in mind that MongoDB needs to look through the query results of both queries and remove any duplicates (documents that matched more than one \$or clause).

When running "\$in" queries there is no way, other than sorting, to control the order of documents returned. For example, {"x" : {"\$in" : [1, 2, 3]}} will return documents in the same order as {"x" : {"\$in" : [3, 2, 1]}}.

## Indexing Objects and Arrays

MongoDB allows you to reach into your documents and create indexes on nested fields and

arrays. Embedded object and array fields can be combined with top-level fields in compound indexes and although they are special in some ways, they mostly behave the way “normal” index fields behave.

## INDEXING EMBEDDED DOCS

Indexes can be created on keys in embedded documents in the same way that they are created on normal keys. If we had a collection where each document represented a user, we might have an embedded document that described each user’s location:

```
{  
    "username" : "sid",  
    "loc" : {  
        "ip" : "1.2.3.4",  
        "city" : "Springfield",  
        "state" : "NY"  
    }  
}
```

We could put an index on one of the subfields of "loc", say "loc.city", to speed up queries using that field:

```
> db.users.ensureIndex({"loc.city" : 1})
```

You can go as deep as you’d like with these: you could index "x.y.z.w.a.b.c" (and so on) if you wanted.

Note that indexing the embedded document itself ("loc") has very different behavior than indexing a field of that embedded document ("loc.city"). Indexing the entire subdocument will only help queries that are querying for the entire subdocument. In the example above, the query optimizer could only use an index on "loc" for queries that described the whole subdocument with fields in the correct order (e.g., `db.users.find({"loc" : {"ip" : "123.456.789.000", "city" : "Shelbyville", "state" : "NY"} })`). It could not use the index for queries that looked like `db.users.find({"loc.city" : "Shelbyville"})`.

## INDEXING ARRAYS

You can also index arrays, which allows you to use the index to search for specific array elements efficiently.

Suppose we have a collection of blog posts where each document was a post. Each post has a "comments" field, which is an array of comment subdocuments. If we want to be able to find

the most-recently-commented-on blog posts, we could create an index on the "date" key in the array of embedded "comments" documents of our blog post collection:

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

Indexing an array creates an index entry for each element of the array, so if a post had 20 comments, it would have 20 index entries. This makes array indexes more expensive than single-value ones: for a single insert, update, or remove, every array entry might have to be updated (potentially thousands of index entries).

Unlike the "loc" example in the previous section, you cannot index an entire array as a single entity: indexing an array field indexes each element of the array, not the array itself.

Indexes on array elements do not keep any notion of position: you cannot use an index for a query that is looking for a specific array element, such as "comments.4".

You can, incidentally, index a specific array entry, for example:

```
> db.blog.ensureIndex({ "comments.10.votes": 1 })
```

However, this index would only be useful for queries for exactly the 11th array element (arrays start at index 0).

Only one field in an index entry can be from an array. This is to avoid the explosive number of index entries you'd get from multiple multikey indexes: every possible pair of elements would have to be indexed, causing indexes to be  $n*m$  entries per document. For example, suppose we had an index on {"x" : 1, "y" : 1}:

```
> // x is an array - legal
> db.multi.insert({"x" : [1, 2, 3], "y" : 1})
>
> // y is an array - still legal
> db.multi.insert({"x" : 1, "y" : [4, 5, 6]})
>
> // x and y are arrays - illegal!
> db.multi.insert({"x" : [1, 2, 3], "y" : [4, 5, 6]})
```

cannot index parallel arrays [y] [x]

Were MongoDB to index the final example, it would have to create index entries for {"x" : 1, "y" : 4}, {"x" : 1, "y" : 5}, {"x" : 1, "y" : 6}, {"x" : 2, "y" : 4}, {"x" : 2, "y" : 5}, {"x" : 2, "y" : 6}, {"x" : 3, "y" : 4},

`{"x" : 3, "y" : 5}`, and `{"x" : 3, "y" : 6}` (and these arrays are only three elements long).

## MULTIKEY INDEX IMPLICATIONS

If any document has an array field for the indexed key, the index immediately is flagged as a multikey index. You can see whether an index is multikey from `explain()`'s output: if a multikey index was used, the `"isMultikey"` field will be true. Once an index has been flagged as multikey, it can never be un-multikeyed, even if all of the documents containing arrays in that field are removed. The only way to un-multikey it is to drop and recreate it.

Multikey indexes may be a bit slower than non-multikey indexes. Many index entries can point at a single document so MongoDB may need to do some de-duping before returning results.

## Index Cardinality

Cardinality refers to how many distinct values there are for a field in a collection. Some fields, such as `"gender"` or `"newsletter opt-out"`, might only have two possible values, which is considered a very low cardinality. Others, such as `"username"` or `"email"`, might have a unique value for every document in the collection, which is high cardinality. Still others fall somewhere in between, such as `"age"` or `"zip code"`.

In general, the greater the cardinality of a field, the more helpful an index on that field can be. This is because the index can quickly narrow the search space to a much smaller result set. For a low cardinality field, an index generally cannot eliminate as many possible matches.

For example, suppose we had an index on `"gender"` and were looking for women named Susan. We could only narrow down the result space by approximately 50% before referring to individual documents to look up `"name"`. Conversely, if we indexed by `"name"`, we could immediately narrow down our result set to the tiny fraction of users named Susan and then we could refer to those documents to check the gender.

As a rule of thumb, try to create indexes on high-cardinality keys or at least put high-cardinality keys first in compound indexes (before low-cardinality keys).

## Using `explain()`

As you have seen above, `explain()` gives you lots of information about your queries. It is one of the most important diagnostic tools there is for slow queries. You can find out which indexes are being used and how by looking at a query's explain. For any query, you can add a call to `explain()` at the end (the way you would add a `sort()` or `limit()`, but `explain()` must be the last call).

There are two types of `explain()` output that you'll see most commonly: indexed and non-indexed queries. Special index types may create slightly different query plans, but most fields should be similar. Also, sharding returns a conglomerate of `explain()`s (as covered in Chapter 12), as it runs the query on multiple servers.

The most basic type of `explain()` is on a query that doesn't use an index. You can tell that a query doesn't use an index because it uses a "BasicCursor". Conversely, most queries that use an index use a "BtreeCursor" (some special types of indexes, such as geospatial indexes, use their own type of cursor).

The output to an `explain()` on a query that uses an index varies, but in the simplest case, it looks something like this:

```
> db.users.find({"age" : 42}).explain()
{
  "cursor" : "BtreeCursor age_1_username_1",
  "isMultiKey" : false,
  "n" : 8332,
  "nscannedObjects" : 8332,
  "nscanned" : 8332,
  "nscannedObjectsAllPlans" : 8332,
  "nscannedAllPlans" : 8332,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 91,
  "indexBounds" : {
    "age" : [
      [
        42,
        42
      ]
    ],
    "username" : [
      [
        {
          "$minElement" : 1
        },
        {
          "$maxElement" : 1
        }
      ]
    ]
  },
  "server" : "ubuntu:27017"
}
```

This output first tells you what index was used: `age_1_username_1`. "millis" reports how fast the query was executed, from the server receiving the request to when it sent a response. However, it may not always be the number you are looking for. If MongoDB tried multiple query plans, "millis" will reflect how long it took all of them to run, not the one chosen as the best.

Next is how many documents were actually returned as a result: "n". This doesn't reflect how much work MongoDB did to answer the query: how many index entries and documents did it have to search? Index entries are described by "nscanned". The number of documents scanned is reflected in "nscannedObjects". Finally, if you were using a sort and MongoDB could not use an index for it, "scanAndOrder" would be true. This means that MongoDB had to sort the results in memory, which is generally quite slow and limited to a small number of results.

Now that you know the basics, here is a breakdown of the all of the fields in more detail:

`"cursor" : "BtreeCursor age_1_username_1"`

`BtreeCursor` means that an index was used, specifically, the index on `age` and `username`: `{"age" : 1, "username" : 1}`. You may also see `reverse` (if the query is traversing the index in reverse direction—say for a sort) or `multi`, if it is using a multikey index.

`"isMultiKey" : false`

If this query used a multikey index (see "[Indexing Objects and Arrays](#)").

`"n" : 8332`

This is the number of documents returned by the query.

`"nscannedObjects" : 8332`

This is the number of times MongoDB had to follow an index pointer to the actual document on disk. If the query contains criteria that is not part of the index or requests fields back that aren't contained in the index, MongoDB must look up the document each index entry points to.

`"nscanned" : 8332`

The number of index entries looked at if an index was used. If this was a table scan, it is the

number of documents examined.

"scanAndOrder" : false

If MongoDB had to sort results in memory.

"indexOnly" : false

If MongoDB was able to fulfill this query using only the index entries (as discussed in ["Using covered indexes"](#)).

In this example, MongoDB found all matching documents using the index, which we know because "nscanned" is the same as "n". However, the query was told to return every field in the matching documents and the index only contained the "age" and "username" fields. If we changed the query to have a second argument, {"\_id" : 0, "age" : 1, "username" : 1}, then it would be covered by the index and "indexOnly" would be true.

"nYields" : 0

The number of times this query yielded (paused) to allow a write request to proceed. If there are writes waiting to go, queries will periodically release their lock and allow them to do so. However, on this system, there were no writes waiting so the query never yielded.

"millis" : 91

The number of milliseconds it took the database to execute the query. The lower this number is, the better.

"indexBounds" : { . . . }

This field describes how the index was used, giving ranges of the index traversed. As the first clause in the query was an exact match, the index only needed to look at that value: 42. The second index key was a free variable, as the query didn't specify any restrictions to it. Thus, the database looked for values between negative infinity ("\$minElement" : 1) and infinity("\$maxElement" : 1) for usernames within "age" : 42.

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
}

```

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" : {
          "$exists" : true
        }
      }
    ],
    [
      {
        "age" : 14,
        "username" : /.*/
      },
      {
        "age" : 14,
        "username" : /.*/
      }
    ]
  ],
  "nscanned" : 13,
  "nscannedObjects" : 13,
  "n" : 13,
  "millis" : 5
}

```

```
        }
    ],
    "nscanned" : 2,
    "nscannedObjects" : 2,
    "n" : 2,
    "millis" : 2
}
```

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})
```

If a query is not using the index that you want it to and you use a hint to change it, run an `explain()` on the hinted query before deploying. If you force MongoDB to use an index on a query that it does not know how to use an index for, you could end up making the query less efficient than it was without the index.

## The Query Optimizer

MongoDB's query optimizer works a bit differently than any other database's. Basically, if an index exactly matches a query (you are querying for "x" and have an index on "x"), the query optimizer will use that index. Otherwise, there might be a few possible indexes that could work well for your query. MongoDB will select a subset of likely indexes and run the query once with each plan, in parallel. The first plan to return 100 results is the "winner" and the other plans' executions are halted.

This plan is cached and used subsequently for that query until the collection has seen a certain amount of churn. Once the collection has changed a certain amount since the initial plan evaluation, the query optimizer will re-race the possible plans. Plans will also be reevaluated after index creation or every 1,000 queries.

The "allPlans" field in `explain()`'s output shows each plan the query tried running.

## When Not to Index

Indexes are most effective at retrieving small subsets of data and some types of queries are faster without indexes. Indexes become less and less efficient as you need to get larger

percentages of a collection because using an index requires two lookups: one to look at the index entry and one following the index's pointer to the document. A table scan only requires one: looking at the document. In the worst case (returning all of the documents in a collection) using an index would take twice as many lookups and would generally be significantly slower than a collection scan.

Unfortunately, there isn't a hard-and-fast rule about when an index helps and when it hinders as it really depends on the size of your data, size of your indexes, size of your documents, and the average result set size ([Table 5-1](#)). As a rule of thumb: if a query is returning 30% or more of the collection, start looking at whether indexes or table scans are faster. However, this number can vary from 2% to 60%.

*Table 5-1. Properties that affect the effectiveness of indexes*

<b>Indexes often work well for</b>	<b>Collection scans often work well for</b>
Large collections	Small collections
Large documents	Small documents
Selective queries	Non-selective queries

Let's say we have an analytics system that collects statistics. Your application queries the system for all documents for a given account to generate a nice graph of all data from an hour ago to the beginning of time:

```
> db.entries.find({ "created_at" : { "$lt" : hourAgo } })
```

We index "created\_at" to speed up this query.

When we first launch, this is a tiny result set and returns instantly. But after a couple weeks, it starts being a lot of data, and after a month this query is already taking too long to run.

For most applications, this is probably the “wrong” query: do you really want a query that’s returning most of your data set? Most applications, particularly those with large data sets, do not. However, there are some legitimate cases where you may want most or all of your data: you might be exporting this data to a reporting system or using it for a batch job. In these cases, you would like to return this large proportion of the data set as fast as possible.

You can force it to do a collection scan by hinting `{"$natural" : 1}`. As described in “[Capped Collections](#)”, `$natural` specifies insertion order when used in a sort when using the WiredTiger storage engine, and order on disk when using MMAPv1.

```
> db.entries.find({"created_at" : {"$lt" : hourAgo}}).hint({"$natural" : 1})
```

Take care when using `"$natural"` with MMAPv1. On-disk order is generally meaningless for an active collection, because as documents grow and shrink they'll be moved around on disk and new documents will be written in the “holes” they left. However, for insert-only workloads, `$natural` can be useful for giving you the latest (or earliest) documents, especially when using the WiredTiger storage engine in which the record identifier does not change as documents change size.

## Types of Indexes

There are a few index options you can specify when building an index that change the way the index behaves. The most common variations are described in the following sections, and more advanced or special-case options are described in the next chapter.

### Unique Indexes

Unique indexes guarantee that each value will appear at most once in the index. 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.users.ensureIndex({"username" : 1}, {"unique" : true})
```

For example, suppose that we try to insert the following documents on the collection above:

```
> db.users.insert({username: "bob"})
> db.users.insert({username: "bob"})
E11000 duplicate key error index: test.users.$username_1  dup key: { : "bo
```

If you check the collection, you'll see that only the first `"bob"` was stored. Throwing duplicate key exceptions is not very efficient, so use the unique constraint for the occasional duplicate, not to filter out zillions of duplicates a second.

A unique index that you are probably already familiar with is the index on `"_id"`, which is

automatically created whenever you create a collection. This is a normal unique index (aside from the fact that it cannot be dropped as other unique indexes can be).

## WARNING

If a key does not exist, the index stores its value as `null` for that document. This means that if you create a unique index and try to insert more than one document that is missing the indexed field, the inserts will fail because you already have a document with a value of `null`. See “[Sparse Indexes](#)” for advice on handling this.

In some cases a value won’t be indexed. Index buckets are of limited size and if an index entry exceeds it, it just won’t be included in the index. This can cause confusion as it makes a document “invisible” to queries that use the index. All fields must be smaller than 1024 bytes to be included in an index. MongoDB does not return any sort of error or warning if a document’s fields cannot be indexed due to size. This means that keys longer than 8 KB will not be subject to the unique index constraints: you can insert identical 8 KB strings, for example.

## 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 across all keys in an index entry can appear in the index at most once.

For example, if we had a unique index on `{"username" : 1, "age" : 1}`, the following inserts would be legal:

```
> db.users.insert({"username" : "bob"})
> db.users.insert({"username" : "bob", "age" : 23})
> db.users.insert({"username" : "fred", "age" : 23})
```

However, attempting to insert a second copy of any of these documents would cause a duplicate key exception.

GridFS, the standard method for storing large files in MongoDB (see “[Storing Files with GridFS](#)”), 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.

## DROPPING DUPLICATES

If you attempt to build a unique index on an existing collection, it will fail to build if there are any duplicate values:

```
> db.users.ensureIndex({ "age" : 1 }, { "unique" : true })
E11000 duplicate key error index: test.users.$age_1  dup key: { : 12 }
```

Generally, you'll need to process your data (the aggregation framework can help) and figure out where the duplicates are and what to do with them.

In a few rare cases, you may just want to delete 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 }
```

"dropDups" forces the unique index build, but it's a very drastic option; you have no control over which documents are dropped and which are kept (and MongoDB gives you no indication of which documents were dropped, if any). If your data is of any importance, do not use "dropDups".

## Sparse Indexes

As mentioned in an earlier section, unique indexes count null as a value, so you cannot have a unique index with more than one document missing the key. However, there are lots of cases where you may want the unique index to be enforced only if the key exists. If you have a field that may or may not exist but must be unique when it does, you can combine the unique option with the sparse option.

## NOTE

If you are familiar with sparse indexes on relational databases, MongoDB's sparse indexes are a completely different concept. MongoDB sparse indexes are basically indexes that need not include every document as an entry.

To create a sparse index, include the `sparse` option. For example, if providing an email address was optional but, if provided, should be unique, we could do:

```
> db.users.ensureIndex({ "email" : 1}, { "unique" : true, "sparse" : true})
```

Sparse indexes do not necessarily have to be unique. To make a non-unique sparse index, simply do not include the `unique` option.

One thing to be aware of is that the same query can return different results depending on whether or not it uses the sparse index. For example, suppose we had a collection where most of the documents had "`x`" fields, but one does not:

```
> db.foo.find()
{ "_id" : 0 }
{ "_id" : 1, "x" : 1 }
{ "_id" : 2, "x" : 2 }
{ "_id" : 3, "x" : 3 }
```

When we do a query on "`x`", it will return all matching documents:

```
> db.foo.find({ "x" : { "$ne" : 2 } })
{ "_id" : 0 }
{ "_id" : 1, "x" : 1 }
{ "_id" : 3, "x" : 3 }
```

If we create a sparse index on "`x`", the `"_id" : 0` document won't be included in the index. So now if we query on "`x`", MongoDB will use the index and not return the `["_id" : 0]` document:

```
> db.foo.find({ "x" : { "$ne" : 2 } })
{ "_id" : 1, "x" : 1 }
{ "_id" : 3, "x" : 3 }
```

You can use `hint()` to force it to do a table scan if you need documents with missing fields.

## Index Administration

As shown in the previous section, you can create new indexes using the `createIndex` function. An index only needs to be created once per collection. If you try to create the same index again, nothing will happen.

All of the information about a database's indexes is stored in the `system.indexes` collection. This is a reserved collection, so you cannot modify its documents or remove documents from it. You can manipulate it only through `createIndex` and the `dropIndexes` database command.

When you create an index, you can see its meta information in `system.indexes`. You can also run `db.collectionName.getIndexes()` to see all index information about a given collection:

```
> db.students.getIndexes()
[  
  {  
    "v" : 2,  
    "key" : {  
      "_id" : 1  
    },  
    "name" : "_id_",  
    "ns" : "school.students"  
  },  
  {  
    "v" : 2,  
    "key" : {  
      "class_id" : 1  
    },  
    "name" : "class_id_1",  
    "ns" : "school.students"  
  },  
  {  
    "v" : 2,  
    "key" : {  
      "student_id" : 1,  
      "class_id" : 1  
    },  
    "name" : "student_id_1_class_id_1",  
    "ns" : "school.students"  
  }  
]
```

The important fields are the `"key"` and `"name"`. The key can be used for hinting, max, min, and other places where an index must be specified. This is a place where field order matters: an

index on {"class\_id" : 1, "student\_id" : 1} is not the same as an index on {"student\_id" : 1, "class\_id" : 1}. The index name is used as identifier for a lot of administrative index operations, such as `dropIndex`. Whether or not the index is multikey is not specified in its spec.

The "v" field is used internally for index versioning. If you have any indexes that do not have a "v" : 1 field, they are being stored in an older, less efficient format. You can upgrade them by ensuring that you're running at least MongoDB version 2.0 and dropping and rebuilding the index.

## Identifying Indexes

Each index in a collection has a 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 `createIndex`:

```
> db.soup.createIndex({ "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 if the index creation succeeded or why it didn't.

## Changing Indexes

As your application grows and changes, you may find that your data or queries have changed and that indexes that used to work well no longer do. You can remove unneeded indexes using the `dropIndex` command:

```
> db.people.dropIndex("x_1_y_1")
{ "nIndexesWas" : 3, "ok" : 1 }
```

Use the "name" field from the index description to specify which index to drop.

Building new indexes is time-consuming and resource-intensive. By default, MongoDB will build an index as fast as possible, blocking all reads and writes on a database until the index build has finished. If you would like your database to remain somewhat responsive to reads and

writes, use the `background` option when building an index. This forces the index build to occasionally yield to other operations, but may still have a severe impact on your application (see “[Building Indexes](#)” for more information). Background indexing is also much slower than foreground indexing.

If you have the choice, creating indexes on existing documents is slightly faster than creating the index first and then inserting all documents.

There is more on the operational aspects of building indexes in [Chapter 16](#).

# Chapter 6. Special Index and Collection Types

---

This chapter covers the special collections and index types MongoDB has available, including:

- Capped collections for queue-like data
- TTL indexes for caches
- Full-text indexes for simple string searching
- Geospatial indexes for 2D and spherical geometries
- GridFS for storing large files

## Geospatial Indexes

MongoDB has two types of geospatial indexes: `2dsphere` and `2d`. `2dsphere` indexes work with spherical geometries that model the surface of the earth based on the WGS84 datum. This datum models the surface of the earth as an oblate spheroid. Meaning that there is some flattening at the poles. Distance calculations using `2sphere` indexes, therefore, take the shape of the earth into account and provide a more accurate treatment of distance between, for example, two cities, than do `2d` indexes. Use `2d` indexes for points stored on a two-dimensional plane.

`2dsphere` allows you to specify geometries for points, lines, and polygons in the [GeoJSON](#) format. A point is given by a two-element array, representing `[longitude, latitude]`:

```
{  
    "name" : "New York City",  
    "loc" : {  
        "type" : "Point",  
        "coordinates" : [50, 2]  
    }  
}
```

---

A line is given by an array of points:

```
{  
  "name" : "Hudson River",  
  "loc" : {  
    "type" : "Line",  
    "coordinates" : [[0,1], [0,2], [1,2]]  
  }  
}
```

A polygon is specified the same way a line is (an array of points), but with a different "type":

```
{  
  "name" : "New England",  
  "loc" : {  
    "type" : "Polygon",  
    "coordinates" : [[0,1], [0,2], [1,2]]  
  }  
}
```

The field that we are naming, "loc", in this example can be called anything, but the field names in the embedded object that is the value of this field are specified by GeoJSON and cannot be changed.

You can create a geospatial index using the "2dsphere" type with `createIndex`:

```
> db.openStreetMap.createIndex({"loc" : "2dsphere"})
```

To create a 2dsphere index, pass a document to `createIndex` that contains a field for which the name specifies the field containing geometries you want to index for the collection in question and specify "2dsphere" as the value.

## Types of Geospatial Queries

There are several types of geospatial query that you can perform: intersection, within, and nearness. To query, specify what you're looking for as a GeoJSON object that looks like `{"$geometry" : geoJsonDesc}`.

For example, you can find documents that intersect the query's location using the `"$geoIntersects"` operator:

```
> var eastVillage = {  
... "type" : "Polygon",  
... "coordinates" : [  
... [-73.9917900, 40.7264100],  
... [-73.9917900, 40.7321400],  
... [-73.9829300, 40.7321400],  
... [-73.9829300, 40.7264100]  
... ]}  
> db.openStreetMap.find(  
... {"loc" : {"$geoIntersects" : {"$geometry" : eastVillage}}})
```

This would find all point-, line-, and polygon-containing documents that had a point in the East Village in New York City.

You can use "\$geoWithin" to query for things that are completely contained in an area, for instance: "What restaurants are in the East Village?"

```
> db.openStreetMap.find({"loc" : {"$geoWithin" : {"$geometry" : eastVillage}}})
```

Unlike our first query, this would not return things that merely pass through the East Village (such as streets) or partially overlap it (such as a polygon describing Manhattan).

Finally, you can query for nearby locations with "\$near":

```
> db.openStreetMap.find({"loc" : {"$near" : {"$geometry" : eastVillage}}})
```

Note that \$near is the only geospatial operator that implies a sort: results from "\$near" are always returned in distance from closest to farthest.

## Using Geospatial Indexes

MongoDB's geospatial indexing allows you to efficiently execute spatial queries on a collection that contains geospatial shapes and points. To showcase the capabilities of geospatial features and compare different approaches, we will go through the process of writing queries for a simple geospatial application. We will go a little deeper on a few concepts central to geospatial indexes and then demonstrate their use with \$geoWithin, \$geoIntersects, and \$geoNear.

Suppose you are designing a mobile application to help users find restaurants in New York City. The application must:

- Determine the neighborhood the user is currently in
- Show the number of restaurants in that neighborhood
- Find restaurants within a specified distance

We will use a 2dsphere index to query for this data on spherical geometry.

## 2D VS SPHERICAL GEOMETRY IN QUERIES

Geospatial queries can use either spherical or 2d (flat) geometries, depending on both the query and the type of index in use. The following table shows what kind of geometry each geospatial operator uses.

Query Type	Geometry Type	Notes
<code>\$near</code> (GeoJSON point, 2dsphere index)	Spherical	
<code>\$near</code> (legacy coordinates, 2d index)	Flat	
<code>\$geoNear</code> (GeoJSON point, 2dsphere index)	Spherical	
<code>\$geoNear</code> (legacy coordinates, 2d index)	Flat	
<code>\$nearSphere</code> (GeoJSON point, 2dsphere index)	Spherical	
<code>\$nearSphere</code> (legacy coordinates, 2d index)	Spherical	Use GeoJSON points instead.
<code>\$geoWithin : { \$geometry: ... }</code>	Spherical	
<code>\$geoWithin : { \$box: ... }</code>	Flat	
<code>\$geoWithin : { \$polygon: ... }</code>	Flat	
<code>\$geoWithin : { \$center: ... }</code>	Flat	
<code>\$geoWithin : { \$centerSphere: ... }</code>	Spherical	

\$geoIntersects

Spherical

Note also that 2d indexes support both flat geometries and distance-only calculations on spheres (i.e. using \$nearSphere). However, queries using spherical geometries will be more performant and accurate with a 2dsphere index.

Note also that the \$geoNear operator listed above is an aggregation operator. The aggregation framework is discussed in chapter TODO. In addition to the \$near query operation, the \$geoNear aggregation operator or the special command, geoNear, enable us to query for nearby locations. Keep in mind that the \$near query operator will not work on collections that are distributed using sharding, MongoDB's scaling solution (see chapter TODO).

The geoNear command and the \$geoNear aggregation operator require that a collection have at most one 2dsphere index and at most one 2d index, whereas geospatial query operators (e.g. \$near and \$geoWithin) permit collections to have multiple geospatial indexes.

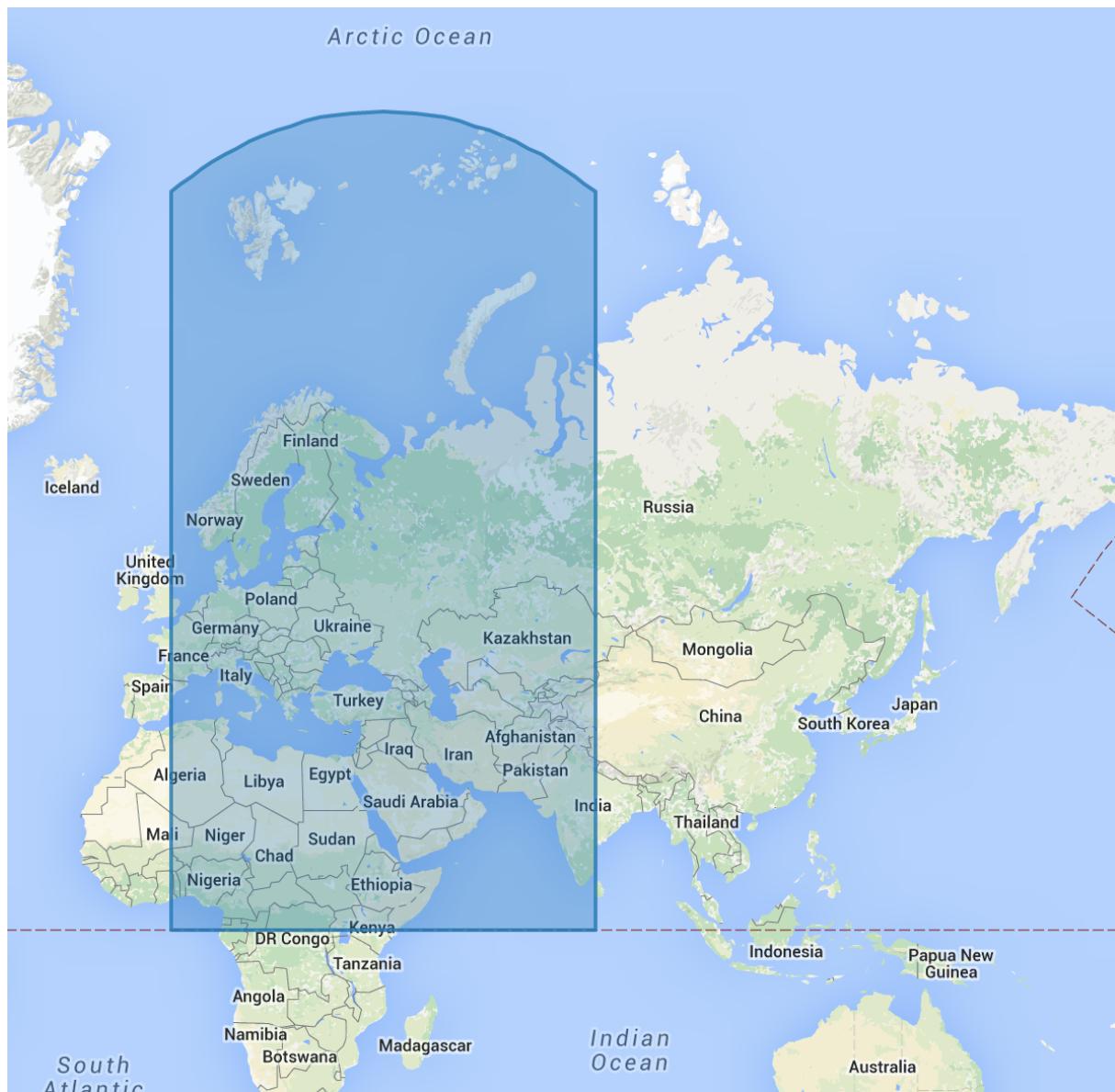
The geospatial index restriction for the geoNear command and the \$geoNear aggregation operator exists because neither the geoNear command nor the \$geoNear syntax includes the location field. As such, index selection among multiple 2d indexes or 2dsphere indexes is ambiguous.

No such restriction applies for geospatial query operators since these operators take a location field, eliminating the ambiguity.

## DISTORTION

Spherical geometry will appear distorted when visualized on a map due to the nature of projecting a three dimensional sphere, such as the earth, onto a flat plane.

For example, take the specification of the spherical square defined by the longitude latitude points (0,0), (80,0), (80,80), and (0,80). The following figure depicts the area covered by this region:



## SEARCHING FOR RESTAURANTS

In this example, we will work with neighborhood and restaurant datasets based in New York City. You may download the example datasets from

<https://raw.githubusercontent.com/mongodb/docs-assets/geospatial/neighborhoods.json> and  
<https://raw.githubusercontent.com/mongodb/docs-assets/geospatial/restaurants.json>.

We can import the datasets into our database using `mongoimport` as follows.

```
mongoimport <path to neighborhoods.json> -c neighborhoods
mongoimport <path to restaurants.json> -c restaurants
```

We can create a `2dsphere` index on each collection using the `createIndex` command in the `mongo` shell:

```
db.neighborhoods.createIndex({location:"2dsphere"})
```

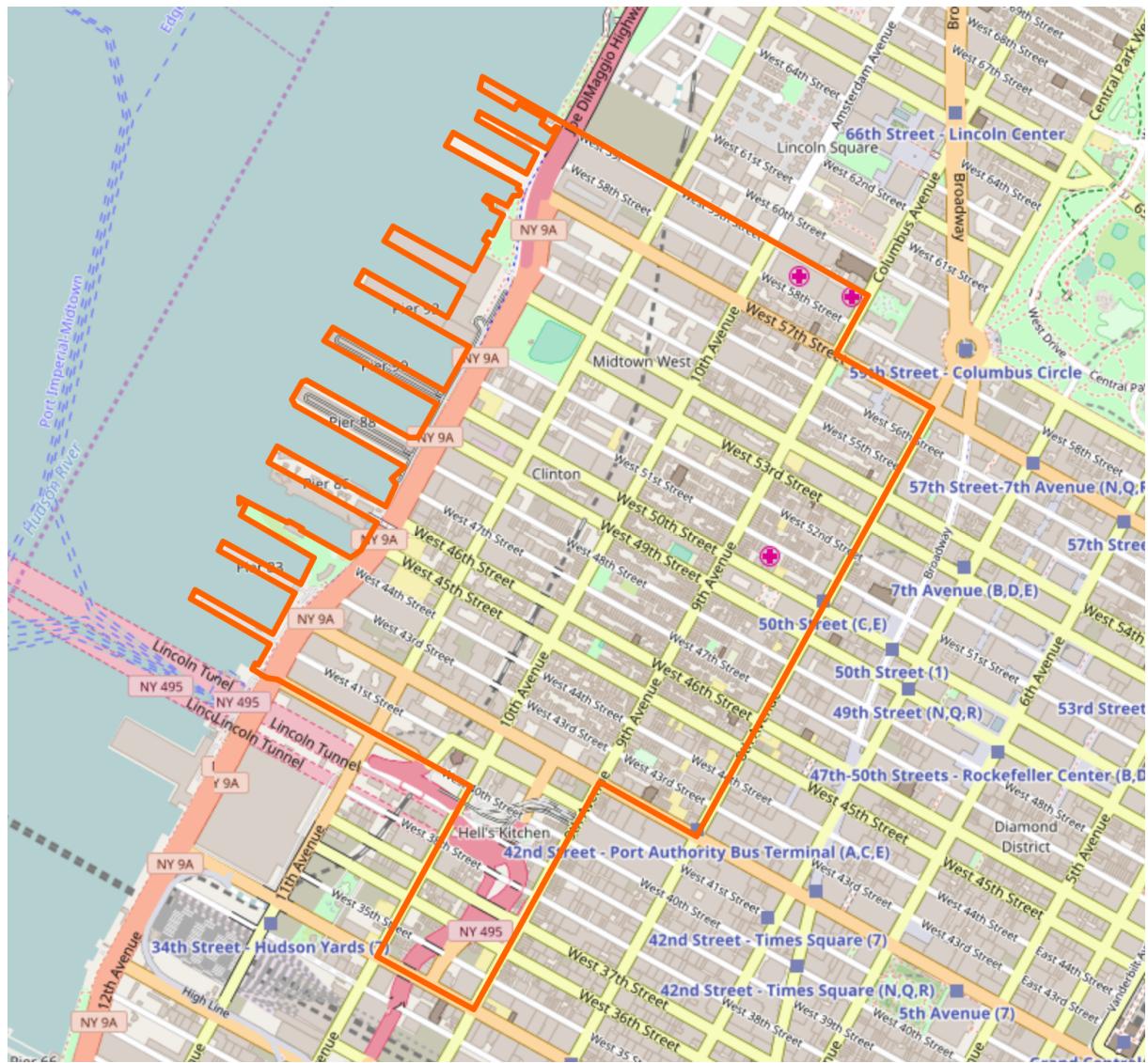
```
db.restaurants.createIndex({location:"2dsphere"})
```

## EXPLORING THE DATA

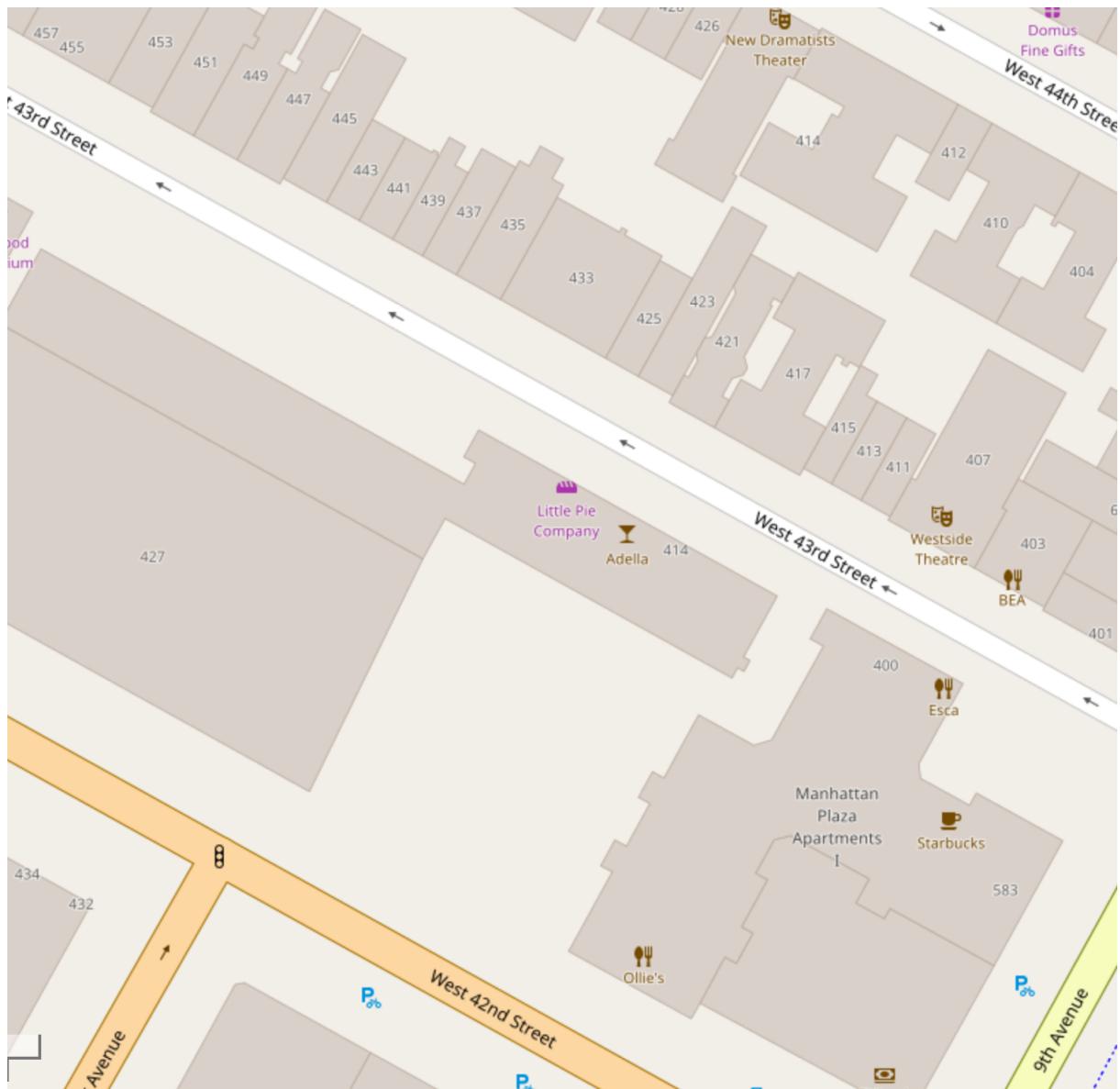
We can get a sense for the schema used for documents in these collections with a couple of quick queries in the `mongo` shell:

```
> db.neighborhoods.find({name: "East Village"})
{
  "_id": ObjectId("55cb9c666c522cafdb053a4b"),
  "geometry": {
    "coordinates": [
      [
        [
          [-73.99, 40.75],
          .
          .
          .
          [-73.98, 40.76],
          [-73.99, 40.75]
        ]
      ],
      "type": "Polygon"
    ],
    "name": "East Village"
  }
}

> db.restaurants.find({name: "Little Pie Company"})
{
  "_id": ObjectId("55cba2476c522cafdb053dea"),
  "location": {
    "coordinates": [
      -73.99331699999999,
      40.7594404
    ],
    "type": "Point"
  },
  "name": "Little Pie Company"
}
```



The bakery corresponds to the location shown in the following figure.



## FIND THE CURRENT NEIGHBORHOOD

Assuming the user's mobile device can give a reasonably accurate location for the user, it is simple to find the user's current neighborhood with `$geoIntersects`.

Suppose the user is located at -73.93414657 longitude and 40.82302903 latitude. To find the current neighborhood, we can specify a point using the special `$geometry` field in GeoJSON format:

```
db.neighborhoods.findOne({geometry:{$geoIntersects:{$geometry:{type:"Point"}}
```

This query will return the following result:

```
{
  "_id":ObjectId("55cb9c666c522cafdb053a68"),
  "geometry":{
```

```
        "type": "Polygon",
        "coordinates": [[[[-73.93383000695911, 40.81949109558767], ...]]],
        "name": "Central Harlem North-Polo Grounds"
    }
```

---

## FIND ALL RESTAURANTS IN THE NEIGHBORHOOD

We can also query to find all restaurants contained in a given neighborhood. To do so, we can execute the following in the mongo shell to find the neighborhood containing the user, and then count the restaurants within that neighborhood. Suppose the user is somewhere in the Hell's Kitchen neighborhood.

```
> var neighborhood = db.neighborhoods.findOne({
  geometry: {
    $geoIntersects: {
      $geometry: {
        type: "Point",
        coordinates: [-73.93414657, 40.82302903]
      }
    }
  }
}) ;

> db.restaurants.find({
  location: {
    $geoWithin: {
      // Use the geometry from neighborhood object we retrieved above
      $geometry: neighborhood.geometry
    }
  }
},
// Project just the name of each matching restaurant
{name: 1, _id: 0});
```

This query will tell you that there are 127 restaurants in the requested neighborhood that have the following names.

```
{
  "name": "White Castle"
}
{
  "name": "Touch Of Dee'S"
}
{
  "name": "McDonald'S"
}
{
  "name": "Popeyes Chicken & Biscuits"
}
```

```
{
  "name": "Make My Cake"
}
{
  "name": "Manna Restaurant Ii"
}
...
{
  "name": "Harlem Coral Llc"
}
```

## FIND RESTAURANTS WITHIN A DISTANCE

To find restaurants within a specified distance of a point, we can use either `$geoWithin` with `$centerSphere` to return results in unsorted order, or `$nearSphere` with `$maxDistance` if you need results sorted by distance.

To find restaurants within a circular region, use `$geoWithin` with `$centerSphere`. `$centerSphere` is a MongoDB-specific syntax to denote a circular region by specifying the center and the radius in radians. `$geoWithin` does not return the documents in any specific order, so it might return the furthest documents first.

The following will find all restaurants within five miles of the user:

```
db.restaurants.find({
  location: {
    $geoWithin: {
      $centerSphere: [
        [-73.93414657, 40.82302903],
        5/3963.2
      ]
    }
  }
})
```

`$centerSphere`'s second argument accepts the radius in radians. The query converts the distance to radians by dividing by the approximate equatorial radius of the earth, 3963.2 miles.

Applications can use `$centerSphere` without having a geospatial index. However, geospatial indexes support much faster queries than the unindexed equivalents. Both `2dsphere` and `2d` geospatial indexes support `$centerSphere`.

You may also use `$nearSphere` and specify a `$maxDistance` term in meters. This will return all restaurants within five miles of the user in sorted order from nearest to farthest:

```
var METERS_PER_MILE = 1609.34;

db.restaurants.find({
  location: {
    $nearSphere: {
      $geometry: {
        type: "Point",
        coordinates: [-73.93414657, 40.82302903]
      },
      $maxDistance: 5*METERS_PER_MILE
    }
  }
}) ;
```

## Compound Geospatial Indexes

As with other types of indexes, you can combine geospatial indexes with other fields to optimize more complex queries. A possible query mentioned above was: “What restaurants are in Hell’s Kitchen?” Using only a geospatial index, we could narrow the field to everything in Hell’s Kitchen, but narrowing it down to only “restaurants” or “pizza” would require another field in the index:

```
db.openStreetMap.createIndex({"tags" : 1, "location" : "2dsphere"})
```

Then we can quickly find a pizza place in Hell’s Kitchen:

```
db.openStreetMap.find({"loc" : {"$geoWithin" : {"$geometry" : hellsKitchen
... "tags" : "pizza"}}
```

We can have the “vanilla” index field either before or after the “2dsphere” field, depending on whether we’d like to filter by the vanilla field or the location first. Choose whichever is more selective -- will filter out more results as the first index term.

## 2D Indexes

For non-spherical maps (video game maps, time series data, etc.) you can use a “2d” index, instead of “2dsphere”:

```
db.hyrule.createIndex({"tile" : "2d"})
```

“2d” indexes assume a perfectly flat surface, instead of a sphere. Thus, “2d” indexes should

not be used with spheres unless you don't mind massive distortion around the poles.

Documents should use a two-element array for their `2d` indexed field. The elements in this array should reflect the longitude and latitude coordinates respectively. A sample document might look like this:

```
{  
  "name" : "Water Temple",  
  "tile" : [ 32, 22 ]  
}
```

Do not use a `2d` index if you plan to store GeoJSON data. "`2d`" indexes can only index points. You can store an array of points, but it will be stored as exactly that: an array of points, not a line. This is an important distinction for "`$geoWithin`" queries, in particular. If you store a street as an array of points, the document will match `$geoWithin` if one of those points is within the given shape. However, the line created by those points might not be wholly contained in the shape.

By default, `2d` indexes assume that your values are going to range from -180 to 180. If you are expecting larger or smaller bounds, you can specify what the minimum and maximum values will be as options to `createIndex`:

```
db.starTrek.createIndex({"light-years" : "2d"}, {"min" : -1000, "max" : 1000})
```

This will create a spatial index calibrated for a  $2,000 \times 2,000$  square.

"`2d`" indexes support the "`$geoWithin`", `$nearSphere`, and "`$near`" query selectors. Use `$geoWithin` to query for points within a shape defined on a flat surface. "`$geoWithin`" can query for all points within a rectangle, polygon, circle, or sphere. The `$geoWithin` operator uses the `$geometry` operator to specify the GeoJSON object. Returning to our Legend of Zelda grid indexed as follows.

```
db.hyrule.createIndex({"tile" : "2d"})
```

The following queries for documents within a rectangle defined by [10, 10] at the bottom left corner and by [100, 100] at the top right corner.

```
db.places.find({  
  tile: {
```

```
$geoWithin: {  
    $box: [[10, 10], [100, 100]]  
}  
}  
})
```

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

To query for documents that are within the circle centered on [-17 , 20.5] and with a radius of 25 we can issue the following command.

```
db.hyrule.find({  
    tile: {  
        $geoWithin: {  
            $center: [-17, 20.5], 25  
        }  
    }  
})
```

The following query returns all documents with coordinates that exist within the polygon defined by [0, 0], [3, 6], and [6 , 0].

```
db.hyrule.find({  
    tile: {  
        $geoWithin: {  
            $polygon: [[0, 0], [3, 6], [6, 0]]  
        }  
    }  
})
```

As this example illustrates, you specify a polygon as an array of points. This example would locate all documents containing points within the given triangle. The final point in the list will be “connected to” the first point to form the polygon.

MongoDB also supports rudimentary spherical queries on flat 2d indexes for legacy reasons. In general, spherical calculations should use a 2dsphere index, as described in 2dsphere Indexes. However, to query for legacy coordinate pairs within a sphere, use \$geoWithin with the \$centerSphere operator. Specify an array that contains:

- The grid coordinates of the circle’s center point
- The circle’s radius measured in radians.

```
db.hyrule.find({  
    loc: {  
        $geoWithin: {  
            $centerSphere: [[88, 30], 10/3963.2]  
        }  
    }  
})
```

To query for nearby points, use `$near`. Proximity queries return the documents with coordinate pairs closest to the defined point and sort the results by distance.

```
db.hyrule.find({"tile" : {"$near" : [20, 21]}})
```

This finds all of the documents in the `hyrule` collection, in order by distance from the point (20, 21). 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 documents nearest to (20, 21):

```
db.hyrule.find({"tile" : {"$near" : [20, 21]} }).limit(10)
```

## Indexes for Full Text Search

Text indexes in MongoDB support full-text search requirements. Use this type of index if your application needs to enable users to submit keyword queries that should match titles, descriptions, and text in other fields within a collection. In previous chapters, we've queried for strings using exact matches and regular expressions, but these techniques have some limitations. Searching a large block of text for a regular expression is slow and it's tough to take morphology (e.g., that "entry" should match "entries") and other challenges presented by human language into account. Text indexes give you the ability to search text quickly and provide support for common search engine requirements such as language-appropriate tokenization, stop words, and stemming.

Text indexes require a number of keys proportional to the words in the fields being indexed. As a consequence, creating a text can consume a large amount system resources. You should create a text index at a time when it will not negatively impact the performance of your application for users or build the index in the background, if possible. To ensure good performance, as with all indexes, you should take care that any text index you create, together with all other indexes fit in RAM. See [Chapter 16](#) for more information on creating indexes with minimal impact on your application.

Writes to a collection require that all indexes are updated. If you are using text search, strings will be tokenized and stemmed and the index updated in, potentially, many places. For this reason, writes involving text indexes are usually more expensive than writes to single-field, compound, or even multi-key indexes. Thus, you will tend to see poorer write performance on text-indexed collections than on others. It will also slow down data movement if you are sharding: all text must be reindexed when it is migrated to a new shard.

## Create a Text Index

Suppose we have a collection of Wikipedia articles that we want to index. To run a search over the text, we first need to create a "text" index. The following call to `createIndex()` will create a text index based on the terms in both the "title" and "body" fields.

```
> db.articles.createIndex({ "title": "text",
                           "body" : "text"})
```

This is not like “nomal” multikey indexes where there is an ordering on the keys. By default, each field is given equal consideration in text indexes. You can control the relative importance MongoDB attaches to each field by specifying a weight:

```
> db.articles.createIndex({ "title": "text",
                           "body": "text" },
                           { "weights" : {
                               "title" : 3,
                               "body" : 2 }})
```

The weights above would weight "title" fields at a ratio of 3:2 in comparison to "body" fields.

You cannot change field weights after index creation (without dropping the index and recreating it), so you may want to play with weights on a sample data set before creating the index on your production data.

For some collections, you may not know which fields a document will contain. You can create a full-text index on all string fields in a document by creating an index on "\$\*\*": this not only indexes all top-level string fields, but also searches embedded documents and arrays for string fields:

```
> db.articles.createIndex({ "$**" : "text" })
```

## Text Search

Use the `$text` query operator to perform text searches on a collection with a text index.

`$text` will tokenize the search string using whitespace and most punctuation as delimiters, and perform a logical OR of all such tokens in the search string. For example, you could use the following query to find all articles containing any of the terms from the list “crater”, “meteor”, “moon”. Note that because our index is based on terms in both the title and body of an article, this query will match documents in which those terms are found in either field. For purposes of this example, we will project the title so that we can fit more results on this page of text.

```
> db.articles.find({$text: {$search: "impact crater lunar"}},  
                     {title: 1}  
                     ).limit(10)  
{ "_id" : "170375", "title" : "Chengdu" }  
{ "_id" : "34331213", "title" : "Avengers vs. X-Men" }  
{ "_id" : "498834", "title" : "Culture of Tunisia" }  
{ "_id" : "602564", "title" : "ABC Warriors" }  
{ "_id" : "40255", "title" : "Jupiter (mythology)" }  
{ "_id" : "80356", "title" : "History of Vietnam" }  
{ "_id" : "22483", "title" : "Optics" }  
{ "_id" : "8919057", "title" : "Characters in The Legend of Zelda series" }  
{ "_id" : "20767983", "title" : "First inauguration of Barack Obama" }  
{ "_id" : "17845285", "title" : "Kushiel's Mercy" }
```



You can see that the results with our initial query are not terribly relevant. As with all technologies, it’s important to have a good grasp of how text indexes work in MongoDB in order to use them effectively. In this case, there are two problems with the way we’ve issued the query. The first is that our query is pretty broad, given that MongoDB issues the query using a logical OR of “impact”, “crater”, and “lunar”. The second problem is that, by default, a text search does not sort the results by relevance.

We can begin to address the problem of the query itself by using a phrase in our query. You can search for exact phrases by wrapping them in double-quotes. For example, the following will find all documents containing the phrase “impact crater”. Possibly surprising is that for this query, is that MongoDB will issue this query as “impact crater” AND “lunar”.

```
> db.articles.find({$text: {$search: "\"impact crater\" lunar"}},  
                     {title: 1}  
                     ).limit(10)  
{ "_id" : "2621724", "title" : "Schjellerup (crater)" }  
{ "_id" : "2622075", "title" : "Steno (lunar crater)" }  
{ "_id" : "168118", "title" : "South Pole-Aitken basin" }  
{ "_id" : "1509118", "title" : "Jackson (crater)" }  
{ "_id" : "10096822", "title" : "Victoria Island structure" }
```

```
{ "_id" : "968071", "title" : "Buldhana district" }
{ "_id" : "780422", "title" : "Puchezh-Katunki crater" }
{ "_id" : "28088964", "title" : "Svedberg (crater)" }
{ "_id" : "780628", "title" : "Zeleny Gai crater" }
{ "_id" : "926711", "title" : "Fracastorius (crater)" }
```

To make sure the semantics of this are clear, let's look at an expanded example. For the following query, MongoDB will issue the query as “impact crater” AND (“lunar” OR “meteor”). MongoDB performs a logical AND of the phrase with the individual terms in the search string and a logical OR of the individual terms with one another.

```
db.articles.find({$text: {$search: "\"impact crater\" lunar meteor"}},
                  {title: 1}
                 ).limit(10)
```

If you want to issue a logical AND between individual terms in a query, treat each term as a phrase by wrapping it in quotes. The following query will return documents containing “impact crater” AND “lunar” AND “meteor”.

```
> db.articles.find({$text: {$search: "\"impact crater\" \"lunar\" \"meteor\"",
                           {title: 1}
                          ).limit(10)
{ "_id" : "168118", "title" : "South Pole-Aitken basin" }
{ "_id" : "330593", "title" : "Giordano Bruno (crater)" }
{ "_id" : "421051", "title" : "Opportunity (rover)" }
{ "_id" : "2693649", "title" : "Pascal Lee" }
{ "_id" : "275128", "title" : "Tektite" }
{ "_id" : "14594455", "title" : "Beethoven quadrangle" }
{ "_id" : "266344", "title" : "Space debris" }
{ "_id" : "2137763", "title" : "Wegener (lunar crater)" }
{ "_id" : "929164", "title" : "Dawes (lunar crater)" }
{ "_id" : "24944", "title" : "Plate tectonics" }
```

Now that we have a better understanding of using phrases and logical ANDs in our queries, let's return to the problem of the results not being sorted by relevance. While the results above are certainly relevant, this is mostly due to the fairly strict query we've issued. We can do better by sorting for relevance.

Text queries cause some metadata to be associated with each query result. The metadata is not displayed in the query results unless we explicitly project it using the \$meta operator. So, in addition to the title, we will project the relevance score calculated for each document. The relevance score is stored in the metadata field named "textScore". For this example, I will return to our query of “impact crater” AND lunar.

```
> db.articles.find({$text: {$search: "\"impact crater\" lunar"}},  
                    {title: 1, score: {$meta: "textScore"}}  
                  ).limit(10)  
{"_id": "2621724", "title": "Schjellerup (crater)", "score": 2.85298713235}  
{"_id": "2622075", "title": "Steno (lunar crater)", "score": 2.47666396103}  
{"_id": "168118", "title": "South Pole-Aitken basin", "score": 2.980198136}  
{"_id": "1509118", "title": "Jackson (crater)", "score": 2.341913728632478}  
{"_id": "10096822", "title": "Victoria Island structure", "score": 1.78205}  
{"_id": "968071", "title": "Buldhana district", "score": 1.627978339350180}  
{"_id": "780422", "title": "Puchezh-Katunki crater", "score": 1.9295977011}  
{"_id": "28088964", "title": "Svedberg (crater)", "score": 2.4977678571428}  
{"_id": "780628", "title": "Zeleny Gai crater", "score": 1.486607142857142}  
{"_id": "926711", "title": "Fracastorius (crater)", "score": 2.75118771114}
```

In these results you can see the relevance score projected with the title for each result. Note that they are not sorted. To sort the results in order of relevance score, we must add a call to `sort()`, again using `$meta` to specify the `"textScore"` field value. Note that we must use the same field name in our sort as we used in our projection. In this case, we used the field name `"score"` for the relevance score value displayed in our search results. As you can see, the results are now sorted in decreasing order of relevance.

```
> db.articles.find({$text: {$search: "\"impact crater\" lunar"}},  
                    {title: 1, score: {$meta: "textScore"}}  
                  ).sort({score: {$meta: "textScore"}}).limit(10)  
{"_id": "1621514", "title": "Lunar craters", "score": 3.1655242042922014}  
{"_id": "14580008", "title": "Kuiper quadrangle", "score": 3.0847527829208}  
{"_id": "1019830", "title": "Shackleton (crater)", "score": 3.0764711199321}  
{"_id": "2096232", "title": "Geology of the Moon", "score": 3.064981949458}  
{"_id": "927269", "title": "Messier (crater)", "score": 3.0638183133686008}  
{"_id": "206589", "title": "Lunar geologic timescale", "score": 3.062029541}  
{"_id": "14536060", "title": "Borealis quadrangle", "score": 3.05730107196}  
{"_id": "14609586", "title": "Michelangelo quadrangle", "score": 3.05722401}  
{"_id": "14568465", "title": "Shakespeare quadrangle", "score": 3.049525641}  
{"_id": "275128", "title": "Tektite", "score": 3.0378807169646915}
```

Text search is also available in the aggregation pipeline. We discuss the aggregation pipeline in chapter TODO.

## Optimize Full Text Search

There are a couple ways to optimize full text searches. If you can first narrow your search results by other criteria, you can create a compound index with a prefix of the other criteria and then the full-text fields:

```
> db.blog.createIndex({ "date" : 1, "post" : "text" })
```

This is referred to as partitioning the full-text index, as it breaks it into several smaller trees based on "date" (in the example above). This makes full-text searches for a specific date or date range much faster.

You can also use a postfix of other criteria to cover queries with the index. For example, if we were only returning the "author" and "post" fields, we could create a compound index on both:

```
> db.blog.createIndex({ "post" : "text", "author" : 1 })
```

These prefix and postfix forms can be combined:

```
> db.blog.createIndex({ "date" : 1, "post" : "text", "author" : 1 })
```

## Searching in Other Languages

When a document is inserted (or the index is first created), MongoDB looks at the indexes fields and stems each word, reducing it to an essential unit. However, different languages stem words in different ways, so you must specify what language the index or document is. Thus, text-type indexes allow a "default\_language" option to be specified, which defaults to "english" but can be set to a number of other languages (see the online documentation for an up-to-date list).

For example, to create a French-language index, we could say:

```
> db.users.createIndex({ "profil" : "text",
                           "intérêts" : "text" },
                           { "default_language" : "french" })
```

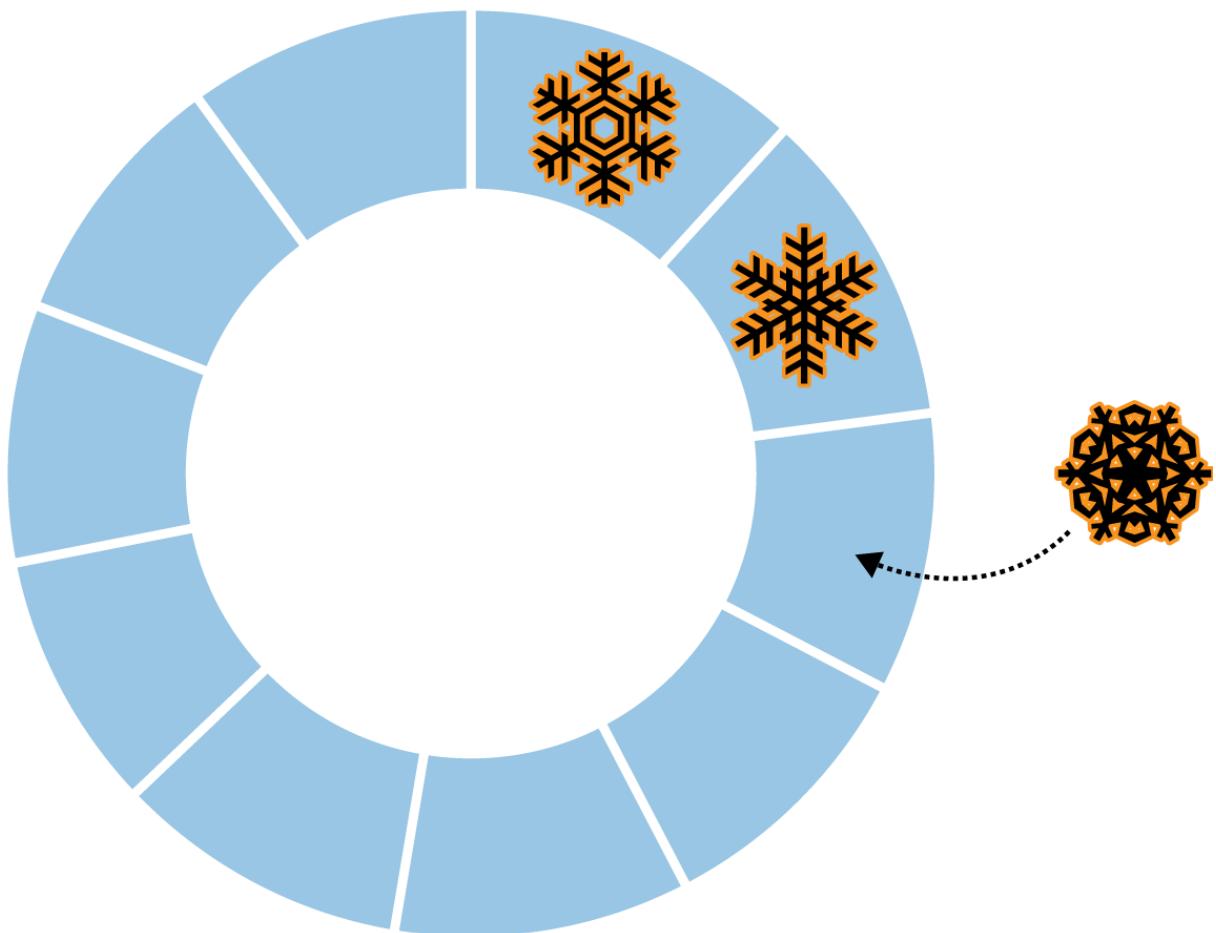
Then French would be used for stemming, unless otherwise specified. You can, on a per-document basis, specify another stemming language by having a "language" field that describes the document's language:

```
> db.users.insert({ "username" : "swedishChef",
                     ... "profile" : "Bork de bork", "language" : "swedish" })
```

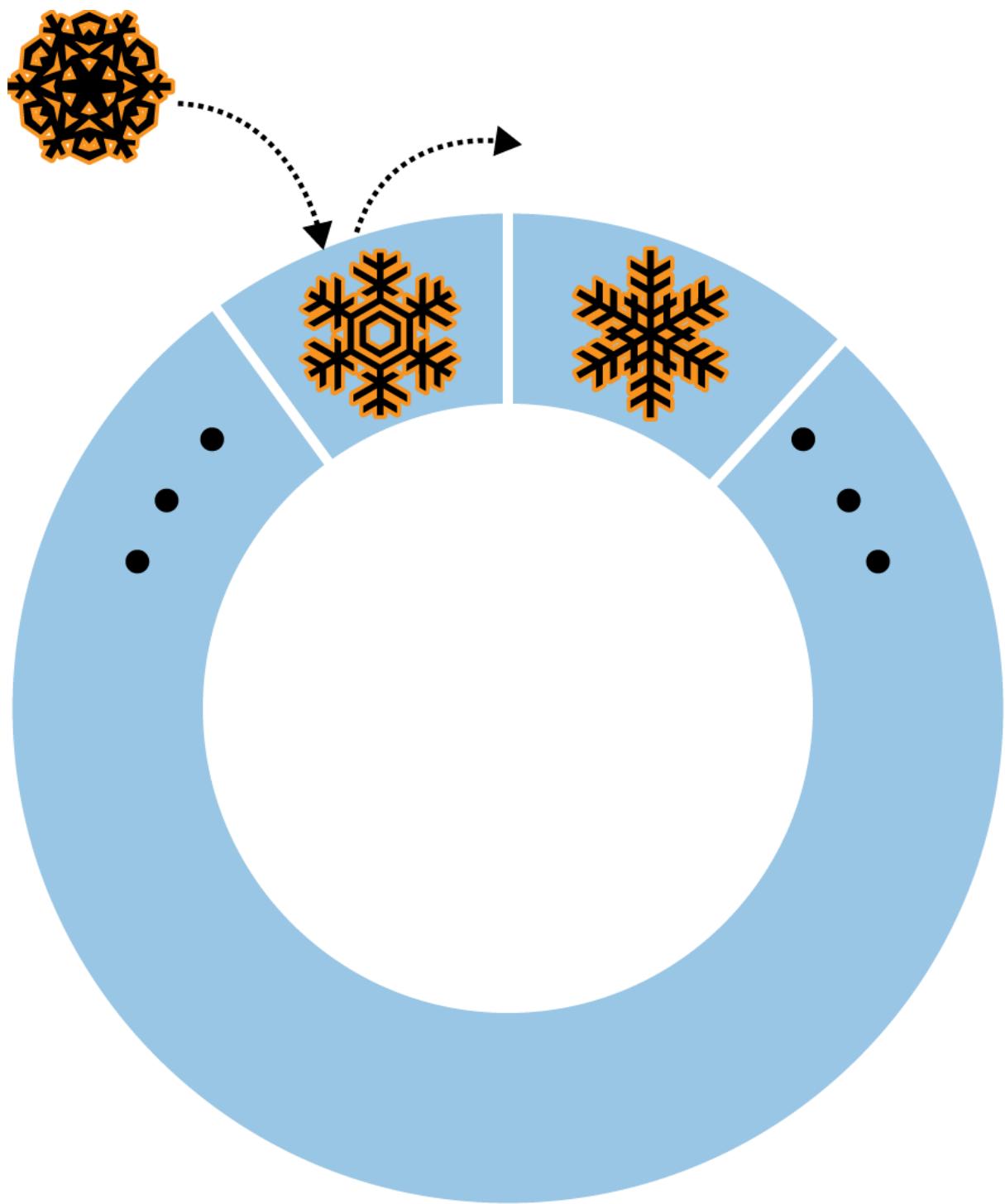
## Capped Collections

“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 6-1](#)). 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 will be deleted, and the new one will take its place (see [Figure 6-2](#)). 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 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 6-1. New documents are inserted at the end of the queue*



*Figure 6-2. When the queue is full, the oldest element will be replaced by the newest*

Capped collections have a different access pattern than most MongoDB collections: data is written sequentially over a fixed section of disk. This makes them tend to perform writes quickly on spinning disk, especially if they can be given their own disk (so as not to be “interrupted” by other collections’ random writes).

## NOTE

Capped collections cannot be sharded.

Capped collections tend to be useful for logging, although they lack flexibility: you cannot control when data ages out, other than setting a size when you create the collection.

## 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})
```

The previous command creates a capped collection, *my\_collection*, that is a fixed size of 100,000 bytes.

`createCollection` can also specify a limit on the number of documents in a capped collection in addition to the limit size:

```
> db.createCollection("my_collection2",
                      {"capped" : true, "size" : 100000, "max" : 100});
```

You could use this to keep, say, the latest 10 news articles or limit a user to 1,000 documents.

Once a capped collection has been created, it cannot be changed (it must be dropped and recreated if you wish to change its properties). Thus, you should think carefully about the size of a large collection before creating it.

## NOTE

When limiting the number of documents in a capped collection, you must specify a size limit as well. Age-out will be based on whichever limit is reached first: it can neither hold more than "max" documents nor take up more than "size" space.

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 }
```

There is no way to “uncap” a capped collection (other than dropping it).

## Tailable Cursors

Tailable cursors are a special type of 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 run out of results, they can continue to fetch new results as documents are added to the collection. Tailable cursors can be used only on capped collections, since insert order is not tracked for normal collections.

Tailable cursors are often used for processing documents as they are inserted onto a “work queue” (the capped collection). Because tailable cursors will time out after 10 minutes of no results, it is important to include logic to re-query the collection if they die. 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());

        }
    }
}
```

The cursor will process results or wait for more results to arrive until the cursor dies (it will time out if there are no inserts for 10 minutes or someone kills the query operation).

## Time-To-Live (TTL) Indexes

As mentioned in the previous section, capped collections give you limited control over when their contents are overwritten. If you need a more flexible age-out system, time-to-live (TTL) indexes allow you to set a timeout for each document. When a document reaches a preconfigured age, it will be deleted. This type of index is useful for caching use cases such as session storage.

You can create a TTL index by specifying the `expireAfterSeconds` option in the second argument to `createIndex`:

```
> // 24-hour timeout
> db.sessions.createIndex({ "lastUpdated" : 1 }, { "expireAfterSeconds" : 60 * 60 * 24 })
```

This creates a TTL index on the `"lastUpdated"` field. If a document's `"lastUpdated"` field exists and is a date, the document will be removed once the server time is `expireAfterSeconds` seconds ahead of the document's time.

To prevent an active session from being removed, you can update the `"lastUpdated"` field to the current time whenever there is activity. Once `"lastUpdated"` is 24 hours old, the document will be removed.

MongoDB sweeps the TTL index once per minute, so you should not depend on to-the-second granularity. You can change the `expireAfterSeconds` using the `collMod` command:

```
> db.runCommand( { "collMod" : "someapp.cache" , "index" : { "keyPattern" :
{ "lastUpdated" : 1 } , "expireAfterSeconds" : 3600 } } );
```

You can have multiple TTL indexes on a given collection. They cannot be compound indexes but can be used like “normal” indexes for the purposes of sorting and query optimization.

## Storing Files with GridFS

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, you might be able to use GridFS instead of a separate tool for file storage.
- 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 easier.

- 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 2 GB chunks.

There are some downsides, too:

- Slower performance: accessing files from MongoDB will not be as fast as going directly through the filesystem.
- You can only modify documents by deleting them and resaving the whole thing. MongoDB stores files as multiple documents so it cannot lock all of the chunks in a file at the same time.

GridFS is generally best when you have large files you'll be accessing in a sequential fashion that won't be changing much.

## Getting Started with GridFS: mongofiles

The easiest way to try out 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

All the client libraries have GridFS APIs. For example, with PyMongo (the Python driver for MongoDB) you can 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 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 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. 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"`

The `"_id"` of the file document that contains the metadata for the file this chunk is from.

`"n"`

The chunk's position in the file, relative to the other chunks.

`"data"`

The bytes in 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.

As mentioned above, you are not limited to the required fields in `fs.files`: feel free to keep any other file metadata in this collection as well. You might want to keep information such as download count, MIME type, or user rating with a file's metadata.

Once you understand the underlying GridFS specification, it becomes trivial to implement features that the driver you're using might not provide helpers for. For example, you can use the `distinct` command to get a list of unique filenames stored in GridFS:

```
> db.fs.files.distinct("filename")
[ "foo.txt" , "bar.txt" , "baz.txt" ]
```

This allows your application a great deal of flexibility in loading and collecting information about files.

# Chapter 7. Introduction to the Aggregation Framework

---

Many applications require data analysis of one form or another. MongoDB provides powerful support for running analytics natively using the aggregation framework. In this chapter, we introduce the aggregation framework and some of the fundamental tools this framework provides:. In the next chapter, we'll dive deeper, looking at more advanced aggregation features, including the ability to perform joins across collections.

- The aggregation framework
- Aggregation stages
- Aggregation expressions
- Aggregation accumulators

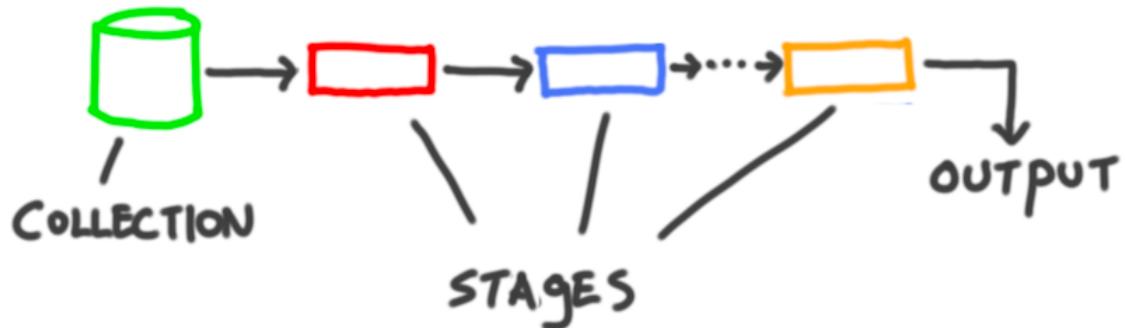
## Pipelines, Stages, and Tunables

The aggregation framework is a set of analytics tools within MongoDB that allow you to do analytics on documents in one or more collections.

The aggregation framework is based on the concept of a pipeline. The idea with an aggregation pipeline is that we take input from a MongoDB collection and pass the documents from that collection through one or more stages, each of which performs a different

operation on its inputs. Each stage takes as input whatever the stage before it produced as output. The inputs and outputs for all stages are documents -- a stream of documents, if you will.

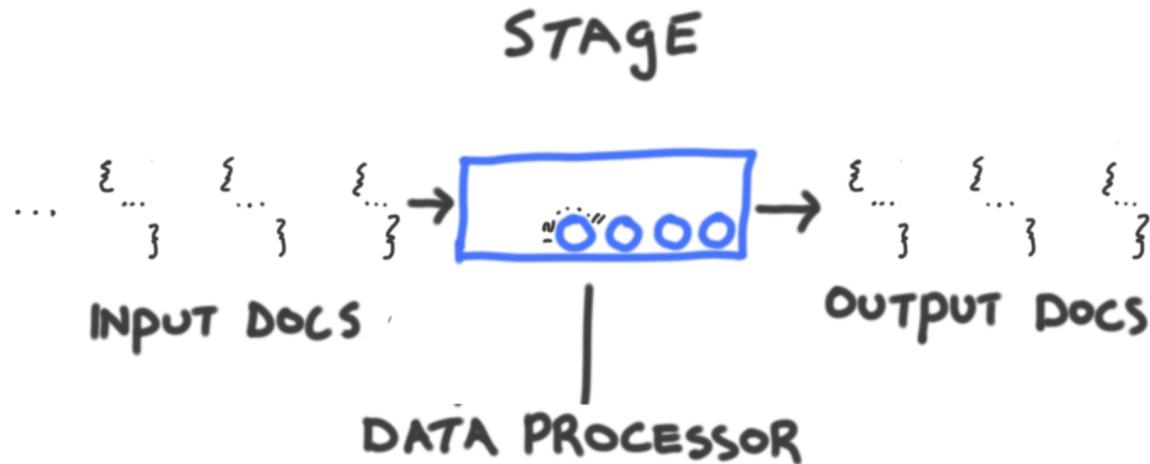
# PIPELINE



If you're familiar with pipelines in a Linux shell, such as bash, this is a very similar idea. Each stage has a specific job that it does. It's expecting a specific form of document and produces a specific output, which is itself a stream of documents. At the end of the pipeline, we get access to the output, much in the same way that we would by executing a find query. By that I simply mean that we get a stream of documents back that we can then do additional work, with whether it's creating a report of some kind, generating a website, or some other type of task.

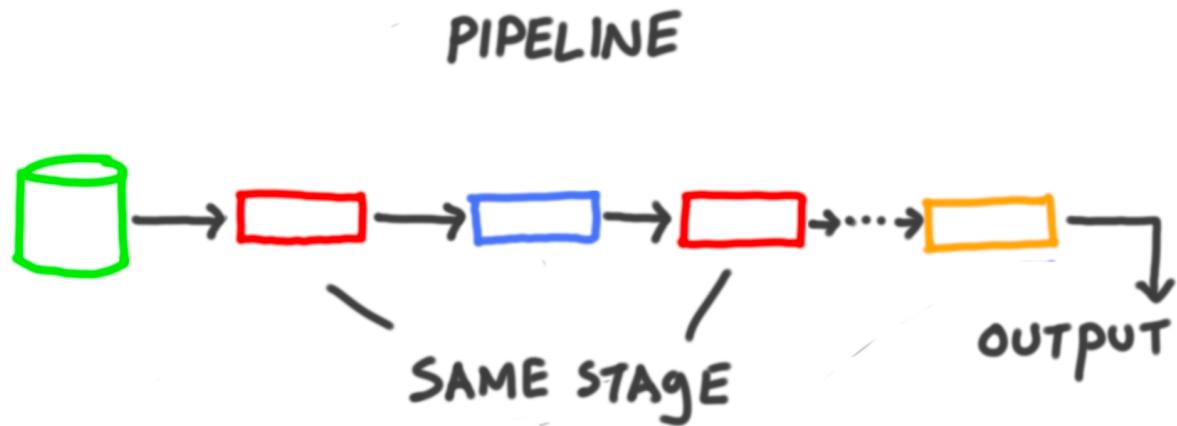
Now, let's dive in a little deeper and consider individual stages. An individual stage of an aggregation pipeline is a data processing unit. As I mentioned, that stage takes a stream of input documents one at a time, processes each document one at a time, and produces an output stream of documents one at a time.

Each stage provides a set of knobs or tunables, that we can control to parameterize the stage to perform whatever task we're interested in doing. A stage performs a generic task -- a general purpose task of some kind and we parameterize the stage for the particular collection that we're working with and exactly what we would like that stage to do with those documents.



These tunables typically take the form of operators that we can supply that will modify fields, perform arithmetic operations reshape documents, or do some sort of accumulation task as well as a variety of other things.

Before we dive in to look at some concrete examples, I want to introduce one more aspect of pipelines that is especially important to keep in mind as you begin to work with them. Frequently, we want to include the same type of stage multiple times within a single pipeline. For example, we may want to perform an initial filter so that we don't have to pass the entire collection into our pipeline. But then later on, following some additional processing, we might want to want to filter further, applying a different set of criteria.



To recap, pipelines work with a MongoDB collection. They're composed of stages, each of which does a different data processing task on its input and produces documents as output to be passed to the next stage. Finally, at the end, a pipeline produces output that we can then do something with in our application. In many cases, in order to perform the analysis we need to do, we will include the same type of stage multiple times within an individual pipeline.

## Getting Started with Stages: Familiar Operations

As our first steps in developing aggregation pipelines, we will look at building some pipelines that involve operations that are already familiar to you. For this we will look at the match, project, sort, skip, and limit stages.

To work through these aggregation examples, we will use a collection of company data. As an overview, let's look at an example based on a document containing data on Facebook, Inc. In this collection we have a have a number of fields that specify details on companies, such as number of employees, when the company was founded.

There are also fields for the rounds of funding a company has gone through, important milestones for the company, whether or not a company has been through an initial public offering (IPO), and if so, the details of the IPO.

```
{
    "_id" : "52cdef7c4bab8bd675297d8e",
    "name" : "Facebook",
    "category_code" : "social",
    "founded_year" : 2004,
    "description" : "Social network",
    "funding_rounds" : [
        {
            "id" : 4,
            "round_code" : "b",
            "raised_amount" : 27500000,
            "raised_currency_code" : "USD",
            "funded_year" : 2006,
            "investments" : [
                {
                    "company" : null,
                    "financial_org" : {
                        "name" : "Greylock Partners",
                        "permalink" : "greylock"
                    },
                    "person" : null
                },
                {
                    "company" : null,
                    "financial_org" : {
                        "name" : "Meritech Capital Partners",
                        "permalink" : "meritech-capital-partners"
                    },
                    "person" : null
                },
                {
                    "company" : null,
                    "financial_org" : {
                        "name" : "Founders Fund",
                        "permalink" : "founders-fund"
                    }
                }
            ]
        }
    ]
}
```

```

        },
        "person" : null
    },
    {
        "company" : null,
        "financial_org" : {
            "name" : "SV Angel",
            "permalink" : "sv-angel"
        },
        "person" : null
    }
]
},
{
    "id" : 2197,
    "round_code" : "c",
    "raised_amount" : 15000000,
    "raised_currency_code" : "USD",
    "funded_year" : 2008,
    "investments" : [
        {
            "company" : null,
            "financial_org" : {
                "name" : "European Founders Fund",
                "permalink" : "european-founders-fund"
            },
            "person" : null
        }
    ]
},
"ipo" : {
    "valuation_amount" : NumberLong("104000000000"),
    "valuation_currency_code" : "USD",
    "pub_year" : 2012,
    "pub_month" : 5,
    "pub_day" : 18,
    "stock_symbol" : "NASDAQ:FB"
}
}

```

---

As our first aggregation example, let's do a simple filter looking for all companies that were founded in 2004.

---

```

db.companies.aggregate([
    {$match: {founded_year: 2004}},
])

```

---

This is equivalent to the following operation using the find query.

```
db.companies.find({founded_year: 2004})
```

Now, let's add a project stage to our pipeline to reduce the output to just a few fields per document. Let's exclude the `_id`, but include the name and `founded_year`. Our pipeline will be as follows.

```
db.companies.aggregate([
  {$match: {founded_year: 2004}},
  {$project: {
    _id: 0,
    name: 1,
    founded_year: 1
  }}
])
```

If we run this, we get output that looks like the following.

```
{"name": "Digg", "founded_year": 2004 }
{"name": "Facebook", "founded_year": 2004 }
{"name": "AddThis", "founded_year": 2004 }
{"name": "Veoh", "founded_year": 2004 }
{"name": "Pando Networks", "founded_year": 2004 }
{"name": "Jobster", "founded_year": 2004 }
{"name": "AllPeers", "founded_year": 2004 }
{"name": "blinkx", "founded_year": 2004 }
{"name": "Yelp", "founded_year": 2004 }
{"name": "KickApps", "founded_year": 2004 }
{"name": "Flickr", "founded_year": 2004 }
{"name": "FeedBurner", "founded_year": 2004 }
{"name": "Dogster", "founded_year": 2004 }
{"name": "Sway", "founded_year": 2004 }
{"name": "Loomia", "founded_year": 2004 }
{"name": "Redfin", "founded_year": 2004 }
{"name": "Wink", "founded_year": 2004 }
{"name": "Techmeme", "founded_year": 2004 }
{"name": "Eventful", "founded_year": 2004 }
{"name": "Oodle", "founded_year": 2004 }
...
...
```

Let's unpack this aggregation pipeline in a little more detail. The first thing you will notice is that we're using the `aggregate` method. This is the method we call when we want to run an aggregation query. To aggregate, we pass in an aggregation pipeline. A pipeline is an array with documents as elements. Each of the documents must stipulate a particular stage operator. In the aggregation above, we have a pipeline that has two stages: a match stage for filtering and a project stage with which we're limiting the output to just two fields per document.

The match stage filters against the collection and passes the resulting documents to the project stage one at a time. Project performs its operation, reshaping the documents and passes the output out of the pipeline and back to us.

So now let's extend our pipeline a bit further to include limit. In this case, we are still going to match using the same query, but limit our result set to five, and then project out the fields we want. For simplicity, let's limit our output to just the name of each company.

```
db.companies.aggregate([
  {$match: {founded_year: 2004}},
  {$limit: 5},
  {$project: {
    _id: 0,
    name: 1}}
])
```

The result is as follows.

```
{"name": "Digg"}
{"name": "Facebook"}
{"name": "AddThis"}
{"name": "Veoh"}
{"name": "Pando Networks"}
```

Note that I have constructed this pipeline so that we limit before the project stage. If we ran the project stage first and then the limit as in the following query, we would get exactly the same results. However, the difference is that we would pass hundreds of documents through the project stage before finally limiting to five.

```
db.companies.aggregate([
  {$match: {founded_year: 2004}},
  {$project: {
    _id: 0,
    name: 1}},
  {$limit: 5}
])
```

Regardless of what type of optimizations the MongoDB query planner might be capable of in a given release, I pause here to encourage you to always consider the efficiency of your aggregation pipeline. Ensure that you are limiting the number of documents that need to be passed on from one stage to another as you build your pipeline.

This requires careful consideration of the entire flow of documents through a pipeline. In the

case of the query above, we really are only interested in the first five documents that match our query, regardless of how they are sorted, so it's perfectly fine to limit, as our second stage.

However, if the order matters, then we'll need to sort before the limit stage. Sort works in a manner similar to what we have seen already, except that in the aggregation framework, we specify sort as a stage within a pipeline as follows. In this case, we will sort by name in ascending order.

```
db.companies.aggregate([
  { $match: { founded_year: 2004 } },
  { $sort: { name: 1 } },
  { $limit: 5 },
  { $project: {
    _id: 0,
    name: 1 } }
])
```

with the following result from our companies collection.

```
{"name": "1915 Studios"}
{"name": "1Scan"}
{"name": "2GeeksinaLab"}
{"name": "2GeeksinaLab"}
{"name": "2threads"}
```

Note that we're looking at a different set of five companies now, getting instead the first five documents in alphanumeric order by name.

Finally, to close this subsection, let's take a look at including a skip stage. Here we will sort first. We will then skip the first 10 documents and, again, limit our result set to five documents.

```
db.companies.aggregate([
  {$match: {founded_year: 2004}},
  {$sort: {name: 1}},
  {$skip: 10},
  {$limit: 5},
  {$project: {
    _id: 0,
    name: 1}},
])
```

Let's review our pipeline one more time. We have five stages. First we're filtering the companies collection looking only for documents where the founded year is 2004. Then we're sorting based on the name in ascending order, skipping the first 10, and then limiting our end

results to five. Finally, we pass those five documents on to project, where we're going to reshape the documents such that our output documents contain just the name.

Here, we've looked at constructing pipelines using stages that perform operations that should already be familiar to you. These operations are provided in the aggregation framework because they are necessary for the types of analytics that we want to accomplish using stages discussed in later sections. As we move through the rest of this chapter, we will take a deep dive into the other operations that the aggregation framework provides.

## Expressions

As we move deeper into our discussion of the aggregation framework, it is important to have a sense for the different types of expressions available for use as you construct aggregation pipelines. The aggregation framework supports many different classes of expressions:

- **Boolean** expressions allow us to and, or, and not other expressions.
- **Set** expressions that allow us to work with arrays as sets. In particular, we can get the intersection or union of two or more sets. We can also take the difference of two sets and perform a number of other set operations.
- **Comparison** expressions enable us to express many different types of range filters.
- **Arithmetic** expressions enable us to calculate the ceiling, the floor, natural log and log, as well as simple arithmetic operations like multiplication, division, addition, and subtraction. We can even perform more complex operations such as calculating the square root of a value.
- **String** expressions allow us to concatenate, find substrings, perform operations having to do with case, and text search operations.
- **Array** expressions provide a lot of power for manipulating arrays including the ability to filter array elements, slicing an array, or just taking a range of values from a specific array.
- **Variable** expressions, which we won't dive into too deeply, allow us to work with literals, expressions for parsing date values, and conditional expressions.
- **Accumulators** provide the ability to calculate sums, descriptive statistics, and many other types of values.

## Project Stage and Reshaping Documents

Now I'd like to take a deeper dive into the project stage and reshaping documents. Here we will explore the types of reshaping operations that should be most common in the applications that you develop. We have seen some simple projections in aggregation pipelines. Now let's look at some that are a little more complex.

First, let's look at promoting nested fields. In the following pipeline, we are doing a match.

```
db.companies.aggregate([
  {$match: {"funding_rounds.investments.financial_org.permalink": "greylock"}},
  {$project: {
    _id: 0,
    name: 1,
    ipo: "$ipo.pub_year",
    valuation: "$ipo.valuation_amount",
    funders: "$funding_rounds.investments.financial_org.permalink"
  }
])
.pretty()
```

As an example of the relevant fields for documents in our companies collection, we can look at a portion of the Facebook document again.

```
{
  "_id" : "52cdef7c4bab8bd675297d8e",
  "name" : "Facebook",
  "category_code" : "social",
  "founded_year" : 2004,
  "description" : "Social network",
  "funding_rounds" : [
    {
      "id" : 4,
      "round_code" : "b",
      "raised_amount" : 27500000,
      "raised_currency_code" : "USD",
      "funded_year" : 2006,
      "investments" : [
        {
          "company" : null,
          "financial_org" : {
            "name" : "Greylock Partners",
            "permalink" : "greylock"
          },
          "person" : null
        },
        {
          "company" : null,
          "financial_org" : {
            "name" : "Meritech Capital Partners",
            "permalink" : "meritech-capital-partners"
          },
          "person" : null
        }
      ]
    }
  ]
}
```

```

        "person" : null
    } ,
    {
        "company" : null,
        "financial_org" : {
            "name" : "Founders Fund",
            "permalink" : "founders-fund"
        },
        "person" : null
    } ,
    {
        "company" : null,
        "financial_org" : {
            "name" : "SV Angel",
            "permalink" : "sv-angel"
        },
        "person" : null
    }
]
},
{
    "id" : 2197,
    "round_code" : "c",
    "raised_amount" : 15000000,
    "raised_currency_code" : "USD",
    "funded_year" : 2008,
    "investments" : [
        {
            "company" : null,
            "financial_org" : {
                "name" : "European Founders Fund",
                "permalink" : "european-founders-fund"
            },
            "person" : null
        }
    ]
},
"ipo" : {
    "valuation_amount" : NumberLong("104000000000"),
    "valuation_currency_code" : "USD",
    "pub_year" : 2012,
    "pub_month" : 5,
    "pub_day" : 18,
    "stock_symbol" : "NASDAQ:FB"
}
}

```

Going back to our match:

---

```

db.companies.aggregate([
    {$match: {"funding_rounds.investments.financial_org_permalink": "greyloc}},
    {$project: {

```

```

        _id: 0,
        name: 1,
        ipo: "$ipo.pub_year",
        valuation: "$ipo.valuation_amount",
        funders: "$funding_rounds.investments.financial_org.permalink"
    } }
]) .pretty()

```

we are filtering for all companies that had a funding round in which Greylock Partners participated. The permalink value, “greylock”, is the unique identifier for such documents. Here is another view on the Facebook document with just the relevant fields displayed.

```

{
  ...
  "name" : "Facebook",
  ...
  "funding_rounds" : [
    ...
    "investments" : [
      ...
      "financial_org" : {
        "name" : "Greylock Partners",
        "permalink" : "greylock"
      },
      ...
    ],
    {
      ...
      "financial_org" : {
        "name" : "Meritech Capital Partners",
        "permalink" : "meritech-capital-partners"
      },
      ...
    },
    {
      ...
      "financial_org" : {
        "name" : "Founders Fund",
        "permalink" : "founders-fnd"
      },
      ...
    },
    {
      ...
      "company" : null,
      "financial_org" : {
        "name" : "SV Angel",
        "permalink" : "sv-angel"
      },
      ...
    },
    ...
  ],
}

```

```

    ...
  ] },
{
  ...
  "investments" : [ {
    ...
    "financial_org" : {
      "name" : "European Founders Fund",
      "permalink" : "european-founders-fund"
    },
    ...
  } ]
},
"ipo" : {
  "valuation_amount" : NumberLong("104000000000"),
  "valuation_currency_code" : "USD",
  "pub_year" : 2012,
  "pub_month" : 5,
  "pub_day" : 18,
  "stock_symbol" : "NASDAQ:FB"
}
}

```

---

The project stage we have defined in this aggregation pipeline will suppress the `_id` and include the name. It will also promote some nested fields. This project uses dot notation to express fields paths that reach into the `ipo` field and the `funding_rounds` field to select values from those nested documents and arrays. This project stage will make those the values of top-level fields in the documents it produces as output.

```

{
  "name" : "Digg",
  "funders" : [
    [
      "greylock",
      "omidyar-network"
    ],
    [
      "greylock",
      "omidyar-network",
      "floodgate",
      "sv-angel"
    ],
    [
      "highland-capital-partners",
      "greylock",
      "omidyar-network",
      "svb-financial-group"
    ]
  ]
}
```

```

{
  "name" : "Facebook",
  "ipo" : 2012,
  "valuation" : NumberLong("104000000000"),
  "funders" : [
    [
      "accel-partners"
    ],
    [
      "greylock",
      "meritech-capital-partners",
      "founders-fund",
      "sv-angel"
    ],
    ...
    [
      "goldman-sachs",
      "digital-sky-technologies-fo"
    ]
  ]
}

{
  "name" : "Revision3",
  "funders" : [
    [
      "greylock",
      "sv-angel"
    ],
    [
      "greylock"
    ]
  ]
}
...

```

In the output, each document has a field for name. Each also has a field for funders. For those companies that have gone through an IPO, the field “ipo” contains the year the company went public and the “valuation” field contains the value of the company at IPO. Note that in all of these documents, these are top level fields and that the values for those fields were promoted from nested documents and arrays.

The \$ character used to specify the value for ipo, valuation, and funders in our project stage, indicates that the value should be interpreted as a field path and used to select the value that should be projected for each field respectively.

One thing you might have noticed is that we’re seeing multiple values printed out for funders. In fact, we’re seeing an array of arrays. The reason for that is because for every funding round, we’re going to get potentially many investors because we know that funding rounds are

represented by an array.

Based on our review of the Facebook example document, we know that all of the funders are listed within an array called investments. What's happening here is that our stage specifies that we want to project the financial\_org.permalink value, for each entry in the investments array, for every funding round. So, an array of arrays of funders' names is built up.

In later lessons we will look at how to perform arithmetic operations and other operations on strings, dates, and a number of other value types to project documents of all shapes and sizes. Just about the only thing we can't do from a project stage, relatively speaking, is change the data type for a value. We have lots of power in what type of documents we construct using a project stage. Here is where we begin to see some hints of that.

## \$unwind

When working with array fields in an aggregation pipeline, it is often necessary to include one or more unwind stages. Unwind allows us to produce output such that there is one output document for each element in a specified array field.

```
{ key1: "value1",
  key2: "value2",
  key3: [ "elem1",
            "elem2",
            "elem3" ] }    ⇒ $unwind
                                         ↓
```

```
{ key1: "value1",
  key2: "value2",
  key3: "elem1" }      { key1: "value1",
                        key2: "value2",
                        key3: "elem2" }      { key1: "value1",
                                         key2: "value2",
                                         key3: "elem3" }
```

In this example in figure TODO, we have an input document that has three keys and values. The third key has an array as its value, with elements one, two, and three. Unwind, if run on this type of input document, and configured to unwind the “key3” field, will produce documents that look like this. The thing that might not be intuitive to you about this is that in each of these output documents, there will be a key3 field. That key3 field will have a single value rather than an array value, one for each one of the elements that were in this array.

If there were 10 elements in this array, unwind would produce 10 output documents. Let's go

back to our companies examples, and take a look at the use of unwind stages. We'll start with the following aggregation pipeline. Note that in this pipeline, as in an earlier section, we are simply matching on a specific funder and promoting values from embedded funding\_rounds documents using a project stage.

```
db.companies.aggregate([
  {$match: {"funding_rounds.investments.financial_org.permalink": "greylock"}},
  {$project: [
    _id: 0,
    name: 1,
    amount: "$funding_rounds.raised_amount",
    year: "$funding_rounds.funded_year"
  }]
])
```

Once again, here is our example of the data model for documents in this collection.

```
{
  "_id" : "52cdef7c4bab8bd675297d8e",
  "name" : "Facebook",
  "category_code" : "social",
  "founded_year" : 2004,
  "description" : "Social network",
  "funding_rounds" : [
    {
      "id" : 4,
      "round_code" : "b",
      "raised_amount" : 27500000,
      "raised_currency_code" : "USD",
      "funded_year" : 2006,
      "investments" : [
        {
          "company" : null,
          "financial_org" : {
            "name" : "Greylock Partners",
            "permalink" : "greylock"
          },
          "person" : null
        },
        {
          "company" : null,
          "financial_org" : {
            "name" : "Meritech Capital Partners",
            "permalink" : "meritech-capital-partners"
          },
          "person" : null
        },
        {
          "company" : null,
          "financial_org" : {
```

```

        "name" : "Founders Fund",
        "permalink" : "founders-fund"
    },
    "person" : null
},
{
    "company" : null,
    "financial_org" : {
        "name" : "SV Angel",
        "permalink" : "sv-angel"
    },
    "person" : null
}
],
},
{
    "id" : 2197,
    "round_code" : "c",
    "raised_amount" : 15000000,
    "raised_currency_code" : "USD",
    "funded_year" : 2008,
    "investments" : [
        {
            "company" : null,
            "financial_org" : {
                "name" : "European Founders Fund",
                "permalink" : "european-founders-fund"
            },
            "person" : null
        }
    ]
},
{
    "ipo" : {
        "valuation_amount" : NumberLong("104000000000"),
        "valuation_currency_code" : "USD",
        "pub_year" : 2012,
        "pub_month" : 5,
        "pub_day" : 18,
        "stock_symbol" : "NASDAQ:FB"
    }
}

```

The aggregation query above will produce results such as the following.

```
{
    "name" : "Digg",
    "amount" : [
        8500000,
        2800000,
        28700000,
        5000000
    ],
}
```

```

    "year" : [
        2006,
        2005,
        2008,
        2011
    ]
}
{
    "name" : "Facebook",
    "amount" : [
        500000,
        12700000,
        27500000,
        ...

```

This aggregation query produces documents that have arrays for both amount and year, because we're accessing the raised amount and the funded year for every element in the funding rounds array.

To fix this, we can include an unwind stage before our project stage in this aggregation pipeline, and parameterize this by specifying that it is the funding rounds array that should be unwound.

Returning again to our Facebook example, and looking at funding rounds, we can see that for each funding round, there is a `raised_amount` field, and a `funded_year` field.



The unwind stage will produce an output document for each element of the funding rounds array. In this example, our values are strings, but regardless of the type of value, unwind will produce an output document for each one.

```
db.companies.aggregate([
```

```

        { $match: {"funding_rounds.investments.financial_org.permalink": "greylo",
        { $unwind: "$funding_rounds" },
        { $project: {
            _id: 0,
            name: 1,
            amount: "$funding_rounds.raised_amount",
            year: "$funding_rounds.funded_year"
        } }
    })

```

The unwind stage produces an exact copy of every one of the documents that it receives as input. All fields will have the same key and value, with one exception, and that is that the funding rounds field. Rather than being an array of funding rounds documents, instead it will have a value that is a single document, which is an individual funding round.

---

```

{"name": "Digg", "amount": 8500000, "year": 2006 }
{"name": "Digg", "amount": 2800000, "year": 2005 }
{"name": "Digg", "amount": 28700000, "year": 2008 }
{"name": "Digg", "amount": 5000000, "year": 2011 }
{"name": "Facebook", "amount": 500000, "year": 2004 }
{"name": "Facebook", "amount": 12700000, "year": 2005 }
{"name": "Facebook", "amount": 27500000, "year": 2006 }
{"name": "Facebook", "amount": 240000000, "year": 2007 }
{"name": "Facebook", "amount": 60000000, "year": 2007 }
{"name": "Facebook", "amount": 15000000, "year": 2008 }
{"name": "Facebook", "amount": 100000000, "year": 2008 }
{"name": "Facebook", "amount": 60000000, "year": 2008 }
{"name": "Facebook", "amount": 200000000, "year": 2009 }
{"name": "Facebook", "amount": 210000000, "year": 2010 }
{"name": "Facebook", "amount": 1500000000, "year": 2011 }
{"name": "Revision3", "amount": 1000000, "year": 2006 }
{"name": "Revision3", "amount": 8000000, "year": 2007 }
...

```

---

Now let's add an additional field to our output documents. In doing so, we'll actually identify a little bit of a problem with this aggregation pipeline as currently written.

---

```

db.companies.aggregate([
    { $match: {"funding_rounds.investments.financial_org.permalink": "greylo",
    { $unwind: "$funding_rounds" },
    { $project: {
        _id: 0,
        name: 1,
        funder: "$funding_rounds.investments.financial_org.permalink",
        amount: "$funding_rounds.raised_amount",
        year: "$funding_rounds.funded_year"
    } }
])

```

---

```
])
```

In adding the “funder” field we know have a field path value that will access the investments field of the funding rounds embedded document that it gets from unwind and, for the financial organization, selects the permalink value. Note that this is very similar to what we’re doing here, in terms of our match filter. Let’s have a look at our output.

```
{
  "name" : "Digg",
  "funder" : [
    "greylock",
    "omidyar-network"
  ],
  "amount" : 8500000,
  "year" : 2006
}
{
  "name" : "Digg",
  "funder" : [
    "greylock",
    "omidyar-network",
    "floodgate",
    "sv-angel"
  ],
  "amount" : 2800000,
  "year" : 2005
}
{
  "name" : "Digg",
  "funder" : [
    "highland-capital-partners",
    "greylock",
    "omidyar-network",
    "svb-financial-group"
  ],
  "amount" : 28700000,
  "year" : 2008
}
...
{
  "name" : "Farecast",
  "funder" : [
    "madrona-venture-group",
    "wrf-capital"
  ],
  "amount" : 1500000,
  "year" : 2004
}
{
  "name" : "Farecast",
```

```

    "funder" : [
        "greylock",
        "madrona-venture-group",
        "wrf-capital"
    ],
    "amount" : 7000000,
    "year" : 2005
}
{
    "name" : "Farecast",
    "funder" : [
        "greylock",
        "madrona-venture-group",
        "par-capital-management",
        "pinnacle-ventures",
        "sutter-hill-ventures",
        "wrf-capital"
    ],
    "amount" : 12100000,
    "year" : 2007
}

```

---

In order to understand what we're seeing here, we need to go back to our document and look at the investments field.

The funding\_rounds.investments field is itself an array. Multiple funders can participate in each funding round, so investments will list every one of those funders. Looking at the results as we originally saw with the raised\_amount and funded\_year field, we're now seeing an array for funder because investments is an array-valued field.

Another problem is that given the way we've written our pipeline, many documents are passed to the project stage that represent funding rounds that greylock did not participate in. We can see this by looking at the funding rounds for Farecast. This problem stems from the fact that our match stage selects companies with which greylock participated in at least one funding round. If we are interested in considering only funding rounds in which greylock participated, we need figure out a way to filter differently.

One possibility is reverse the order in which we do our unwind and match stages. That is to say, do the unwind first and then do the match. This guarantees that we will only match documents coming out of unwind. But in thinking through this approach, we should quickly realize that, with unwind as the first stage, we would be doing a scan through the entire collection.

For efficiency, we want to match as early as possible in our pipeline. This enables the aggregation framework to make use of indexes, for example. In order to select only those funding rounds in which greylock participated, we can include a second match stage.

---

```
db.companies.aggregate([
  { $match: { "funding_rounds.investments.financial_org.permalink": "greylock" },
    $unwind: "$funding_rounds" },
  { $match: { "funding_rounds.investments.financial_org.permalink": "greylock" } },
  { $project: {
      _id: 0,
      name: 1,
      individualFunder: "$funding_rounds.investments.person.permalink",
      fundingOrganization: "$funding_rounds.investments.financial_org.permalink",
      amount: "$funding_rounds.raised_amount",
      year: "$funding_rounds.funded_year"
    }
  }
])
```

This pipeline will filter for companies where greylock participated in at least one funding round. It will then unwind the funding rounds and then filter again so that any documents that represent funding rounds, that greylock did not participate in will be removed from what's passed on to project.

As I mentioned when introducing the aggregation framework, it is often the case that we need to include multiple stages of the same type. This is a good example in that we're filtering to reduce the number of documents that we're looking at initially by narrowing down our set of documents for consideration to those for which Greylock participated in at least one funding round. Then, through our unwind stage, we end up with a number of documents that represent funding rounds from companies that greylock did, in fact, fund, but individual funding rounds that greylock did not participate in. We can get rid of all the funding rounds we're not interested in by simply including another filter, using a second match stage.

## Array Expressions

Now let's turn our attention to array expressions. As part of our deep dive, we'll take a look at using array expressions in project stages.

The first expression I'd like to look at is a filter expression. A filter expression selects a subset of the elements in an array based on filter criteria.

Here again working with our companies data set, we'll match using the same criteria for funding rounds in which Greylock participated. Take a look at the rounds field in this pipeline.

```
db.companies.aggregate([
  { $match: { "funding_rounds.investments.financial_org.permalink": "greylock" },
    $project: [
      _id: 0,
```

```
name: 1,
founded_year: 1,
rounds: { $filter: {
    input: "$funding_rounds",
    as: "round",
    cond: { $gte: ["$$round.raised_amount", 100000000] } } }
} },
{ $match: {"rounds.investments.financial_org.permalink": "greylock" } },
].pretty()
```

The rounds field uses a filter expression. The \$filter operator is designed to work with array fields and requires options we must supply. The first option to \$filter is “input”. For “input” we simply specify an array. In this case, we use a field path specifier to identify the funding\_rounds array found in documents in our companies collection. Next, we specify the name we’d like to use for this funding\_rounds array throughout the rest of our filter expression. Then, as the third option, we need to specify a condition. The condition should provide criteria used to filter whatever array we’ve provided as input selecting a subset. In this case, we’re filtering such that we only select elements where the raised\_amount for a funding\_round is greater than or equal to 100 million.

In specifying the condition, we’ve made use of \$\$|. We use \$\$ to reference a variable defined within the expression we’re working in. The as clause defines a variable within our filter expression. This variable has the name “round” because that’s what we labeled it in the as clause. This is to disambiguate a reference to a variable from a field path. In this case, our comparison expression takes an array of two values and will return true if the first value provided is greater than or equal to the second value.

Now let’s consider what documents the project stage of this pipeline will produce, given this filter. The output documents will have fields for name, founded\_year, and rounds. The values for rounds will be arrays composed of the elements that match our filter condition, that being that the raised amount is greater than \$100,000,000.

In the match stage that follows, as we did previously, we will simply filter the input documents for those that were funded in some way by greylock. Documents output by this pipeline will resemble the following.

```
{
  "name" : "Dropbox",
  "founded_year" : 2007,
  "rounds" : [
    {
      "id" : 25090,
      "round_code" : "b",
```

```
"source_description" : "Dropbox Raises $250M In Funding, Boasts 45 M",
"raised_amount" : 250000000,
"raised_currency_code" : "USD",
"funded_year" : 2011,
"investments" : [
  {
    "financial_org" : {
      "name" : "Index Ventures",
      "permalink" : "index-ventures"
    }
  },
  {
    "financial_org" : {
      "name" : "RIT Capital Partners",
      "permalink" : "rit-capital-partners"
    }
  },
  {
    "financial_org" : {
      "name" : "Valiant Capital Partners",
      "permalink" : "valiant-capital-partners"
    }
  },
  {
    "financial_org" : {
      "name" : "Benchmark",
      "permalink" : "benchmark-2"
    }
  },
  {
    "company" : null,
    "financial_org" : {
      "name" : "Goldman Sachs",
      "permalink" : "goldman-sachs"
    },
    "person" : null
  },
  {
    "financial_org" : {
      "name" : "Greylock Partners",
      "permalink" : "greylock"
    }
  },
  {
    "financial_org" : {
      "name" : "Institutional Venture Partners",
      "permalink" : "institutional-venture-partners"
    }
  },
  {
    "financial_org" : {
      "name" : "Sequoia Capital",
      "permalink" : "sequoia-capital"
    }
  }
]
```

```

        },
        {
            "financial_org" : {
                "name" : "Accel Partners",
                "permalink" : "accel-partners"
            }
        },
        {
            "financial_org" : {
                "name" : "Glynn Capital Management",
                "permalink" : "glynn-capital-management"
            }
        },
        {
            "financial_org" : {
                "name" : "SV Angel",
                "permalink" : "sv-angel"
            }
        }
    ]
}
]
}

```

Only the rounds array items for which the raised amount exceeds \$100,000,000 will pass through the filter. In the case of Dropbox, there is just one round that meets that criteria. You have a lot of flexibility in how you set up filter expressions, but this is the basic form and provides a concrete example of a use case for this particular array expression.

Next, let's look at the array element operator. For our example we'll continue working with funding rounds, but in this case I simply want to pull out the first round and the last round. I might be interested, for example, in seeing when these rounds occurred or in comparing their amounts. These are things I can do with date and arithmetic expressions as we'll see in another section.

The `$arrayElemAt` operator enables us to select an element at a particular slot within an array. The following pipeline provides an example of using `$arrayElemAt`.

---

```

db.companies.aggregate([
    { $match: { "founded_year": 2010 } },
    { $project: {
        _id: 0,
        name: 1,
        founded_year: 1,
        first_round: { $arrayElemAt: [ "$funding_rounds", 0 ] },
        last_round: { $arrayElemAt: [ "$funding_rounds", -1 ] }
    } }

```

```
]).pretty()
```

Note the syntax for using \$arrayElemAt within a project stage. We define a field that we want projected out and as the value of the value specify a document with \$arrayElemAt as the field name and an two element array as the value. The first element should be a field path that specifies the array field we want to select from. The second element identifies the slot within that array that we want. Remember that arrays are 0 indexed.

In many cases, the length of an array is not readily available. To select array slots starting from the end of the array, use negative integers. The last element in an array is identified with -1.

A simple output document for this aggregation pipeline would resemble the following.

```
{
  "name" : "vufind",
  "founded_year" : 2010,
  "first_round" : {
    "id" : 19876,
    "round_code" : "angel",
    "source_url" : "",
    "source_description" : "",
    "raised_amount" : 250000,
    "raised_currency_code" : "USD",
    "funded_year" : 2010,
    "funded_month" : 9,
    "funded_day" : 1,
    "investments" : [ ]
  },
  "last_round" : {
    "id" : 57219,
    "round_code" : "seed",
    "source_url" : "",
    "source_description" : "",
    "raised_amount" : 500000,
    "raised_currency_code" : "USD",
    "funded_year" : 2012,
    "funded_month" : 7,
    "funded_day" : 1,
    "investments" : [ ]
  }
}
```

Related to \$arrayElemAt is the slice expression. This allows us to return not just one, but multiple items from an array in sequence beginning with a particular index.

```
db.companies.aggregate([
  { $match: { "founded_year": 2010 } },
```

```
{ $project: {  
    _id: 0,  
    name: 1,  
    founded_year: 1,  
    early_rounds: { $slice: [ "$funding_rounds", 1, 3 ] }  
} }  
]).pretty()
```

Here, again with the funding rounds array, we begin at index 1 and take the next three elements from the array. Perhaps we know that in this data set the first funding round isn't all that interesting and I simply want some early ones but not the very first one.

Filtering and selecting individual elements or slices of arrays are among the more common operations we need to perform on arrays. Probably the most common, however, is determining an array's size or length. To do this we can use the `$size` operator.

```
db.companies.aggregate([  
  { $match: { "founded_year": 2004 } },  
  { $project: {  
    _id: 0,  
    name: 1,  
    founded_year: 1,  
    total_rounds: { $size: "$funding_rounds" }  
  } }  
]).pretty()
```

When used in project, a size expression will simply provide a value that is the number of elements in the array.

In this section, we have explored some of the most common array expressions. There are many more and the list grows with each release. Please review the aggregation framework quick reference in the MongoDB documentation for a summary of all expressions that are available.

## Accumulators

At this point, we have covered a few different types of expressions. Next, let's look at what accumulators the aggregation framework has to offer. Accumulators are essentially another type of expression but one we think about in their own class, because they calculate values from field values found in multiple documents.

Accumulators the aggregation framework provides enable us to perform operations such as summing all values in a particular field (`$sum`), calculating an average (`$avg`), etc. We also consider `$first` and `$last` to be accumulators because these consider values in all documents that pass through the stage in which they are used. `$max` and `$min` are two more examples of

accumulators that consider a stream of documents and saves just one of the values it sees.

We also have accumulators for arrays. We can \$push values onto an array as documents pass through a pipeline stage. \$addToSet is very similar to \$push except that no duplicate values will be included in the resulting array.

Then there are some expressions for calculating descriptive statistics for calculating the standard deviation of a sample and of a population. Both work with a stream of documents that pass through a pipeline stage.

Prior to MongoDB 3.2, accumulators were available only in the group stage. MongoDB 3.2 introduced the ability to access a subset of accumulators within the project stage. The primary difference between the accumulators in the group stage and the project stage is that in the project stage accumulators, such as \$sum and \$average, must operate on arrays within a single document. Whereas, accumulators in the group stage, as we'll see in a later section, provide you with the ability to perform calculations on values across multiple documents.

That's a quick overview of accumulators to provide some context and set the stage for our deep dive into examples.

## Using Accumulators in Project Stages

The following provides a pretty good example of using an accumulator in a project stage. Note that our match stage filters for documents that contain a funding\_rounds field and for which the funding rounds array is not empty.

```
db.companies.aggregate([
  { $match: { "funding_rounds": { $exists: true, $ne: [] } } },
  { $project: {
    _id: 0,
    name: 1,
    largest_round: { $max: "$funding_rounds.raised_amount" }
  } }
])
```

Because the value for \$funding\_rounds is an array within each company document, we can use an accumulator. Remember that in project stages accumulators must work on an array-valued field. In this case, we're able to do something pretty cool here. We are easily identifying the largest value in an array by reaching into an embedded document within that array and projecting the max value in output documents.

```
{ "name" : "Wetpaint", "largest_round" : 25000000 }
```

```

{ "name" : "Digg", "largest_round" : 28700000 }
{ "name" : "Facebook", "largest_round" : 15000000000 }
{ "name" : "Omnidrive", "largest_round" : 800000 }
{ "name" : "Geni", "largest_round" : 10000000 }
{ "name" : "Twitter", "largest_round" : 400000000 }
{ "name" : "StumbleUpon", "largest_round" : 17000000 }
{ "name" : "Gizmoz", "largest_round" : 6500000 }
{ "name" : "Scribd", "largest_round" : 13000000 }
{ "name" : "Slacker", "largest_round" : 40000000 }
{ "name" : "Lala", "largest_round" : 20000000 }
{ "name" : "eBay", "largest_round" : 6700000 }
{ "name" : "MeetMoi", "largest_round" : 2575000 }
{ "name" : "Joost", "largest_round" : 45000000 }
{ "name" : "Babelgum", "largest_round" : 13200000 }
{ "name" : "Plaxo", "largest_round" : 9000000 }
{ "name" : "Cisco", "largest_round" : 2500000 }
{ "name" : "Yahoo!", "largest_round" : 4800000 }
{ "name" : "Powerset", "largest_round" : 12500000 }
{ "name" : "Technorati", "largest_round" : 10520000 }
...

```

Again, in project stages, we can use a certain subset of accumulators to calculate values based on array-valued fields. As another example, let's use the \$sum accumulator to calculate the total funding for each company in our collection.

```

db.companies.aggregate([
  { $match: { "funding_rounds": { $exists: true, $ne: [ ] } } },
  { $project: {
    _id: 0,
    name: 1,
    total_funding: { $sum: "$funding_rounds.raised_amount" }
  } }
])

```

We've looked a couple ways of using accumulators in project stages. Again, I encourage you to review the Aggregation Quick Reference in the MongoDB docs for a complete overview of the accumulator expressions available.

## Introduction to Grouping

Accumulators are historically the province of the group stage in the MongoDB aggregation framework. The group stage performs a function that is similar to the SQL group by command. In a group stage, we can aggregate together values from multiple documents and perform some type of aggregate operation on them, such as calculating an average. Let's take a look at an example.

```

db.companies.aggregate([
  { $group: {
    _id: { founded_year: "$founded_year" },
    average_number_of_employees: { $avg: "$number_of_employees" }
  }},
  { $sort: { average_number_of_employees: -1 } }
])

```

Here we're going to use a group stage to aggregate together all companies based on the year they were founded. We're then going to calculate the average number of employees for each company. The output for this pipeline resembles the following.

```

{ "_id" : { "founded_year" : 1847 }, "average_number_of_employees" : 40500
{ "_id" : { "founded_year" : 1896 }, "average_number_of_employees" : 38800
{ "_id" : { "founded_year" : 1933 }, "average_number_of_employees" : 32000
{ "_id" : { "founded_year" : 1915 }, "average_number_of_employees" : 18600
{ "_id" : { "founded_year" : 1903 }, "average_number_of_employees" : 17100
{ "_id" : { "founded_year" : 1865 }, "average_number_of_employees" : 12500
{ "_id" : { "founded_year" : 1921 }, "average_number_of_employees" : 10700
{ "_id" : { "founded_year" : 1835 }, "average_number_of_employees" : 10000
{ "_id" : { "founded_year" : 1952 }, "average_number_of_employees" : 92900
{ "_id" : { "founded_year" : 1946 }, "average_number_of_employees" : 91500
{ "_id" : { "founded_year" : 1947 }, "average_number_of_employees" : 88510
{ "_id" : { "founded_year" : 1898 }, "average_number_of_employees" : 80000
{ "_id" : { "founded_year" : 1968 }, "average_number_of_employees" : 73550
{ "_id" : { "founded_year" : 1957 }, "average_number_of_employees" : 70055
{ "_id" : { "founded_year" : 1969 }, "average_number_of_employees" : 67635
{ "_id" : { "founded_year" : 1928 }, "average_number_of_employees" : 51000
{ "_id" : { "founded_year" : 1963 }, "average_number_of_employees" : 50503
{ "_id" : { "founded_year" : 1959 }, "average_number_of_employees" : 47432
{ "_id" : { "founded_year" : 1902 }, "average_number_of_employees" : 41171
{ "_id" : { "founded_year" : 1887 }, "average_number_of_employees" : 35000
...

```

The output includes documents that have a document as their `_id` value and then a report on the average number of employees. This is the type of analysis we might do as a first step in assessing the correlation between the year in which a company was founded and its growth, possibly normalizing for how old the company is.

As you can see, the pipeline we built has two stages: a group stage and a sort stage. Fundamental to the group stage is the `_id` field that we specify as part of the document. This is the value of the `$group` operator itself, using a very strict interpretation.

We use `_id` to define what the group stage uses to organize the documents that it sees. Since the

group stage is first, the aggregate command will pass all documents in the companies collection through this stage. The group stage will take every document that has the same value for founded year and treat them as

a single group. In constructing the value for this field, this stage will use the average accumulator to calculate an average number of employees for all companies with the same founded year.

You can think of it this way. Each time the group stage encounters a document with a specific founded year, it adds the value for `number_of_employees` to a running sum of number of employees and adds one to a count for number of documents seen so far. Once all documents have passed through the group stage, it can then calculate the average using that running sum and count for every grouping of documents it identified based on the founded year.

At the end of this pipeline, we sort the documents into descending order by `average_number_of_employees`.

Let's look at another example. One field we've not yet considered in the companies data set is the relationships. The relationships field appears in documents in the following form.

```
{
  "_id" : "52cdef7c4bab8bd675297d8e",
  "name" : "Facebook",
  "permalink" : "facebook",
  "category_code" : "social",
  "founded_year" : 2004,
  ...
  "relationships" : [
    {
      "is_past" : false,
      "title" : "Founder and CEO, Board Of Directors",
      "person" : {
        "first_name" : "Mark",
        "last_name" : "Zuckerberg",
        "permalink" : "mark-zuckerberg"
      }
    },
    {
      "is_past" : true,
      "title" : "CFO",
      "person" : {
        "first_name" : "David",
        "last_name" : "Ebersman",
        "permalink" : "david-ebersman"
      }
    },
    ...
  ]
}
```

```

] ,
"funding_rounds" : [
    ...
{
    "id" : 4,
    "round_code" : "b",
    "source_url" : "http://www.facebook.com/press/info.php?factsheet",
    "source_description" : "Facebook Funding",
    "raised_amount" : 27500000,
    "raised_currency_code" : "USD",
    "funded_year" : 2006,
    "funded_month" : 4,
    "funded_day" : 1,
    "investments" : [
        {
            "company" : null,
            "financial_org" : {
                "name" : "Greylock Partners",
                "permalink" : "greylock"
            },
            "person" : null
        },
        {
            "company" : null,
            "financial_org" : {
                "name" : "Meritech Capital Partners",
                "permalink" : "meritech-capital-partners"
            },
            "person" : null
        },
        {
            "company" : null,
            "financial_org" : {
                "name" : "Founders Fund",
                "permalink" : "founders-fund"
            },
            "person" : null
        },
        {
            "company" : null,
            "financial_org" : {
                "name" : "SV Angel",
                "permalink" : "sv-angel"
            },
            "person" : null
        }
    ]
},
...
"ipo" : {
    "valuation_amount" : NumberLong("104000000000"),
    "valuation_currency_code" : "USD",
    "pub_year" : 2012,
    "pub_month" : 5,
}
]
}

```

```
    "pub_day" : 18,
    "stock_symbol" : "NASDAQ:FB"
  },
  ...
}
```

Let's look for people who have been associated with a large number of companies. The relationships field gives us the ability to dive in and look at people who have, in one way or another been associated with a relatively large number of companies. Now let's take a look at this aggregation.

```
db.companies.aggregate( [
  { $match: { "relationships.person": { $ne: null } } },
  { $project: { relationships: 1, _id: 0 } },
  { $unwind: "$relationships" },
  { $group: {
    _id: "$relationships.person",
    count: { $sum: 1 }
  }},
  { $sort: { count: -1 } }
]).pretty()
```

We're matching on relationships.person. If we look at our Facebook example document, we can see how relationships are structured and get a sense for what it means to do this. We are filtering for all relationships for which person is not null. Then we project out all relationships for documents that match. We will pass only relationships to the next stage in the pipeline, which is unwind. We unwind relationships so that every relationship in the array comes through to the group stage that follows. In the group stage, we use a field path to identify the person within relationship documents. All documents with the same person will be grouped together. As we saw previously, it's perfectly fine for a document to be the value around which we group. So every match to a document for a first name, last name, and permalink for a person will be aggregated together. We use the sum accumulator to count the number of relationships in which each person has participated. Finally, we sort into descending order. The output for this pipeline resembles the following.

```
{
  "_id" : {
    "first_name" : "Tim",
    "last_name" : "Hanlon",
    "permalink" : "tim-hanlon"
  },
  "count" : 28
}
```

```

    "_id" : {
        "first_name" : "Pejman",
        "last_name" : "Nozad",
        "permalink" : "pejman-nozad"
    },
    "count" : 24
}
{
    "_id" : {
        "first_name" : "David S.",
        "last_name" : "Rose",
        "permalink" : "david-s-rose"
    },
    "count" : 24
}
{
    "_id" : {
        "first_name" : "Saul",
        "last_name" : "Klein",
        "permalink" : "saul-klein"
    },
    "count" : 24
}
...

```

Tim Hanlon is the individual who has participated in the most relationships with companies in this collection. It could be that Mr. Hanlon has actually had a relationship with 28 companies, but we can't know that for sure, because it's possible that he has had multiple relationships with one or more companies, each with a different title. I use this example to illustrate a very important point about aggregation pipelines. Make sure you fully understand what it is you're working with as you do calculations, particularly when you're calculating aggregate values using accumulator expressions of some kind.

In this case, we can say that Tim Hanlon appears 28 times in relationship documents throughout the companies in our collection. We would have to dig a little deeper to see exactly how many unique companies he was associated with. I will leave the construction of that pipeline as an exercise to you.

## The `_id` Field in Group Stages

Before we go any further discussing group, I'd like to talk a little more about the `_id` field within the group stage and look at some best practices for constructing `_ids` in group aggregation stages. To do this, I'd like to walk through a few examples that illustrate several different ways in which we commonly group documents. As our first example, let's look at this pipeline.

```
db.companies.aggregate([

```

```

{ $match: { founded_year: { $gte: 2013 } } },
{ $group: {
  _id: { founded_year: "$founded_year" },
  companies: { $push: "$name" }
} },
{ $sort: { "_id.founded_year": 1 } }
]).pretty()

```

The output for this pipeline resembles the following.

```

{
  "_id" : {
    "founded_year" : 2013
  },
  "companies" : [
    "Fixya",
    "Wamba",
    "Advaliant",
    "Fluc",
    "iBazar",
    "Gimigo",
    "SEOGroup",
    "Clowdy",
    "WhosCall",
    "Pikk",
    "Tongxue",
    "Shopseen",
    "VistaGen Therapeutics"
  ]
}
...

```

In our output we have documents with two fields. One is `_id` and the other is `companies`. Each of these documents contains a list of the companies founded in whatever the `founded_year` is, `companies` being an array of company names.

What I'd like to call your attention to is how we've constructed `_id` field in the group stage. Why not just provide the founded year rather than putting it inside a document with a field labeled founded year. The reason we don't do it that way is that if we don't label the group value, it's not explicit that we are grouping on the year in which the company was founded. In order to avoid confusion, it is a best practice to explicitly label values on which we group.

In some circumstances it might be necessary to use another approach in which our `_id` value is a document composed of multiple fields. In this case, we're actually grouping documents on the basis of founded year and their category code.

```
db.companies.aggregate([
  { $match: { founded_year: { $gte: 2010 } } },
  { $group: {
    _id: { founded_year: "$founded_year", category_code: "$category_code" },
    companies: { $push: "$name" }
  } },
  { $sort: { "_id.founded_year": 1 } }
]).pretty()
```

It is perfectly fine to use documents with multiple fields as our `_id` value in group stages. In other cases, it might also be necessary to do something like this.

```
db.companies.aggregate([
  { $group: {
    _id: { ipo_year: "$ipo.pub_year" },
    companies: { $push: "$name" }
  } },
  { $sort: { "_id.ipo_year": 1 } }
]).pretty()
```

In this case, we're grouping documents based on the year in which they IPO-ed, and that year is actually a field of an embedded document. It is common practice to use field paths that reach into embedded documents as the value on which to group in a group stage. In this case, the output will resemble the following.

```
{
  "_id" : {
    "ipo_year" : 1999
  },
  "companies" : [
    "Akamai Technologies",
    "TiVo",
    "XO Group",
    "Nvidia",
    "Blackberry",
    "Blue Coat Systems",
    "Red Hat",
    "Brocade Communications Systems",
    "Juniper Networks",
    "F5 Networks",
    "Informatica",
    "Iron Mountain",
    "Perficient",
    "Sitestar",
    "Oxford Instruments"
  ]
}
```

At this point I should call your attention to the fact that in the examples in this section we have been using an accumulator we haven't seen before. One called \$push. As the group stage processes documents in its input stream, a \$push expression will add this value to an array that it builds through its run. In the case of the pipeline immediately above, group is building an array composed of company names. The \$push operator is yet another accumulator.

Finally, the last example I'd like to cover is one we've already seen, but which I include here for the sake of completeness.

```
db.companies.aggregate( [
  { $match: { "relationships.person": { $ne: null } } },
  { $project: { relationships: 1, _id: 0 } },
  { $unwind: "$relationships" },
  { $group: {
    _id: "$relationships.person",
    count: { $sum: 1 }
  } },
  { $sort: { count: -1 } }
] )
```

In the example above in which we were grouping on IPO year, we used a field path that resolved to a scalar value -- the ipo year. In this case, our field path resolves to a document containing three fields: first name, last name, and permalink. The group stage supports grouping on document values.

Above are several ways in which we can construct \_id values in group stages. In general, bear in mind that what we want to do here is make sure that in our output, the semantics of our \_id value are clear.

## Group vs Project

To round out our discussion of the group aggregation stage, I'd like to look at a couple of additional accumulators, and call attention to the fact that they are not available in the project stage. This is a means of thinking a little bit deeper about what we can do in project with respect to accumulators and what we can do in group. As an example let's take a look at this aggregation query.

```
db.companies.aggregate([
  { $match: { funding_rounds: { $ne: [ ] } } },
  { $unwind: "$funding_rounds" },
  { $sort: { "funding_rounds.funded_year": 1,
    "funding_rounds.funded_month": 1,
```

```

        "funding_rounds.funded_day": 1 } } ,
} $group: {
  _id: { company: "$name" },
  funding: {
    $push: {
      amount: "$funding_rounds.raised_amount",
      year: "$funding_rounds.funded_year"
    } }
} },
] ).pretty()

```

Here, we begin by filtering for document for which the array funding\_rounds is not empty. Then we unwind funding rounds. Therefore, sort and the group stage will see one document for each element of the funding\_rounds array for every company.

Our sort stage in this pipeline sorts on first year, then month, then day, all in ascending order. This means that this stage will output the oldest funding rounds first. And as you are aware from the indexes chapter, we can support this type of sort with a compound index.

In the group stage that follows sort, we will group by company name and use the \$push accumulator to construct a sorted array of funding rounds. The sorting rounds array will be sorted for each company because we sorted all funding rounds, globally, in the sort stage.

Documents output from this pipeline will resemble the following.

```

{
  "_id" : {
    "company" : "Green Apple Media"
  },
  "funding" : [
    {
      "amount" : 30000000,
      "year" : 2013
    },
    {
      "amount" : 100000000,
      "year" : 2013
    },
    {
      "amount" : 2000000,
      "year" : 2013
    }
  ]
}

```

In this pipeline, with \$push, we are accumulating an array. In this case, we have specified our push expression so that it adds documents to the end of the accumulation array. Since the

funding rounds are in chronological order, pushing onto the end of the array guarantees that the the funding amounts for each company are sorted in chronological order.

Push expressions only work in group stages. The reason is because group stages are designed to take an input stream of documents and accumulate values by processing each document in turn. Project stages, on the other hand, work with each document in their input stream individually.

Let's take a look at one other example. This is a little longer, but it builds on the previous one.

```
db.companies.aggregate([
  { $match: { funding_rounds: { $exists: true, $ne: [ ] } } },
  { $unwind: "$funding_rounds" },
  { $sort: { "funding_rounds.funded_year": 1,
    "funding_rounds.funded_month": 1,
    "funding_rounds.funded_day": 1 } },
  { $group: {
    _id: { company: "$name" },
    first_round: { $first: "$funding_rounds" },
    last_round: { $last: "$funding_rounds" },
    num_rounds: { $sum: 1 },
    total_raised: { $sum: "$funding_rounds.raised_amount" }
  } },
  { $project: {
    _id: 0,
    company: "$_id.company",
    first_round: {
      amount: "$first_round.raised_amount",
      article: "$first_round.source_url",
      year: "$first_round.funded_year"
    },
    last_round: {
      amount: "$last_round.raised_amount",
      article: "$last_round.source_url",
      year: "$last_round.funded_year"
    },
    num_rounds: 1,
    total_raised: 1,
  } },
  { $sort: { total_raised: -1 } }
]).pretty()
```

Again, we are unwinding `funding_rounds` and sorting chronologically. However, in this case, instead of accumulating an array of summary `funding_rounds`, we are using two accumulators we've not yet looked at: `$first` and `$last`. We did briefly touch on these two operators when we introduced accumulators. A `first` expression simply saves the first value that passes through the input stream for the stage. A `last` expression simply tracks the values that pass through the group stage and just hangs on to the last one.

So you can see that as with \$push, we can't use \$first and \$last in project stages because, again, project stages are not designed to accumulate values based on multiple documents streaming through them. Rather they are designed to reshape documents individually.

In addition to \$first and \$last, we also use \$sum to calculate the total number of funding\_rounds using a sum expression. For this expression we can just specify the value, 1. A sum expression like this simply serves to count the number of documents that it sees in each grouping.

Finally, this pipeline includes a fairly complex project stage. However, all it is really doing is making the output prettier. Rather than have a first round value, or entire documents for the first and last funding rounds, this project stage creates a summary. Note that this maintains good semantics, because I'm clearly labeling each value. For first\_round we'll produce a simple embedded document that contains just the essential details of amount, article, and year, pulling those values from the original funding round document that will be the value of \$first\_round. The project stage does something similar for \$last\_round. Finally, this project stage just passing through to output documents the num\_rounds and total\_raised values for documents it receives in its input stream.

Documents output from this pipeline resemble the following.

```
{
  "first_round" : {
    "amount" : 7500000,
    "article" : "http://www.teslamotors.com/display_data/pressguild.swf",
    "year" : 2004
  },
  "last_round" : {
    "amount" : 10000000,
    "article" : "http://www.bizjournals.com/sanfrancisco/news/2012/10/10/te
    "year" : 2012
  },
  "num_rounds" : 11,
  "total_raised" : 823000000,
  "company" : "Tesla Motors"
}
```

And with that, we've concluded an overview of the group stage. In this chapter, we have covered a number of different accumulators, some that are available in project, and we've also covered how to think about when to use group versus project when considering various accumulators.

## **Part III. Replication**

---

# Chapter 8. Setting Up a Replica Set

---

This chapter introduces MongoDB’s high availability system: replica sets. It covers:

- What replica sets are
- How to set up a replica set
- What configuration options are available for replica set members

## Introduction to Replication

Since the first chapter, we’ve been using a standalone server, a single *mongod* server. It’s an easy way to get started but a dangerous way to run in production. What if your server crashes or becomes unavailable? Your database will be unavailable for at least a little while. If there are problems with the hardware, you might have to move your data to another machine. In the worst case, disk or network issues could leave you with corrupt or inaccessible data.

Replication is a way of keeping identical copies of your data on multiple servers and is recommended for all production deployments. Replication keeps your application running and your data safe, even if something happens to one or more of your servers.

With MongoDB, you set up replication by creating a replica set. A replica set is a group of servers with one primary, the server taking client requests, and multiple secondaries, servers that keep copies of the primary’s data. If the primary crashes, the secondaries can elect a new primary from amongst themselves.

If you are using replication and a server goes down, you can still access your data from the other servers in the set. If the data on a server is damaged or inaccessible, you can make a new copy of the data from one of the other members of the set.

This chapter introduces replica sets and covers how to set up replication on your system. If you are less interested in replication mechanics and simply want to create a replica set for testing/development or production, use MongoDB’s cloud solution, [MongoDB Atlas](#). It’s easy

to use and provides a free-tier option for experimentation. Alternatively, to manage MongoDB clusters in your own infrastructure, you can use [Ops Manager](#).

## Setting up a Replica Set, Part 1

In this example we'll setup a three-node replica set on a single machine. For experimenting with replica set mechanics this is the type of setup that you might script just to get a replica set up and running and then poke at it with administrative commands in the mongo shell or simulate network partitions or server failures to better understand how MongoDB handles high availability and disaster recovery. Given the variety of virtualization and cloud options available, it is nearly as easy for you to bring up a test replica set with each member on a dedicated host. We've provided a vagrant script [TODO] to allow you to experiment with this option. In production, you should always use a replica set and allocate a dedicated host to each member to avoid resource contention and provide isolation against server failure.

To get started with our test replica set, let's first create separate data directories for each node. On Linux or macOS run the following in the terminal to create the three directories.

```
mkdir -p ~/data/rs{1,2,3}
```

This will create the directories `~/data/rs1`, `~/data/rs2`, and `~/data/rs3` (`~` identifies your home directory).

On Windows, to create these directories, run the following in the Command Prompt (cmd) or PowerShell.

```
md c:\data\rs1 c:\data\rs2 c:\data\rs3
```

On Linux or macOS run each of the following commands in a separate terminal.

```
mongod --replSet mdbDefGuide --dbpath ~/data/rs1 --port 27017 --smallfiles  
mongod --replSet mdbDefGuide --dbpath ~/data/rs2 --port 27018 --smallfiles  
mongod --replSet mdbDefGuide --dbpath ~/data/rs3 --port 27019 --smallfiles
```

On Windows, run each of the following commands in its own Command Prompt or PowerShell window.

```
mongod --replSet mdbDefGuide --dbpath c:\data\rs1 --port 27017 --smallfile:  
mongod --replSet mdbDefGuide --dbpath c:\data\rs2 --port 27018 --smallfile:
```

```
mongod --replSet mdbDefGuide --dbpath c:\data\rs3 --port 27019 --smallfile:
```

Once you've started the *mongods*, you should have three separate *mongod* processes running.

In general, the principles we will walk through in the rest of this chapter apply to replica sets used in production deployments where each *mongod* has a dedicated host. However, there are additional details pertaining to securing replica sets that we address in chapter [TODO]. I'll touch on those briefly below as a preview.

## Networking Considerations

Every member of a set must be able to make connections to every other member of the set (including itself). If you get errors about members not being able to reach other members that you know are running, you may have to change your network configuration to allow connections between them.

The processes we launched above can just as easily be running on separate servers. However, with the release of MongoDB 3.6, *mongod* binds to localhost (127.0.0.1) only by default. In order for each member of replica set to communicate with the others, you must also bind to an ip address that is reachable by other members. If the *mongod* we started above were running on a server with a network interface having an ip address of 198.51.100.1 and we wanted to run this *mongod* as a member of replica set with each member on different servers, we could specify the command-line parameter *bind\_ip* or use *bind\_ip* in the configuration file for this *mongod*.

```
mongod --bind_ip localhost,192.51.100.1 --replSet mdbDefGuide --dbpath ~/d:  
--port 27017 --smallfiles --oplogSize 200
```

We would make a similar modification to launch the other *mongods* as well in this case, regardless of whether running on Linux, macOS, or Windows.

## Security Considerations

Before you bind to ip addresses other than *localhost*, when configuring a replica set, you should enable authorization controls and specify an authentication mechanism. In addition it is a good idea to encrypt data on disk and communication among replica set members and between the set and clients. I go into detail on securing replica sets in chapter [TODO].

## Setting up a Replica Set, Part 2

Returning to our example, with the work we've done so far, each *mongod* does not yet know that the others exist. To tell them about one another, we need to create a configuration that lists each of the members and send this configuration to one of our *mongod* processes. It will take care of propagating the configuration to the other members.

In a fourth terminal, Windows Command Prompt, or PowerShell window, launch a mongo shell that connects to one of the running mongod instances. You can do this by typing the following command. With this command, we'll connect to the mongod running on port 27017.

```
mongo --port 27017
```

Then, in the mongo shell, create a configuration document and pass this to the `rs.initiate()` helper to initiate a replica set. This will initiate a replica set containing three members and propagate the configuration to the rest of the *mongods* so that a replica set is formed.

```
> rsconf = {
    _id: "mdbDefGuide",
    members: [
        {_id: 0, host: "localhost:27017"},
        {_id: 1, host: "localhost:27018"},
        {_id: 2, host: "localhost:27019"}
    ]
}
> rs.initiate(rsconf)
{ "ok" : 1, "operationTime" : Timestamp(1501186502, 1) }
```

There are several important parts of a replica set configuration document. The config's "`_id`" is the name of the replica set that you passed in on the command line (in this example, "`mdbDefGuide`"). Make sure that this name matches exactly.

The next part of the document is an array of members of the set. Each of these needs two fields: an "`_id`" that is an integer and unique among the replica set members and a hostname. This config document is your replica set configuration.

Note that we are using *localhost* as a hostname for the members in this set. This is for example purposes only. In later chapters where we discuss securing replica sets, we'll look at configurations that are more appropriate for production deployments. MongoDB allows all-*localhost* replica sets for testing locally but will protest if you try to mix *localhost* and non-*localhost* servers in a config.

This `config` object is your replica set configuration. The member running on `localhost:27017` will parse the configuration and send messages to the other members, alerting them of the new configuration. Once they have all loaded the configuration, they will elect a primary and start handling reads and writes.

### NOTE

Unfortunately, you cannot convert a standalone server to a replica set without some downtime for restarting it and initializing the set. Thus, even if you only have one server to start out with, you may want to configure it as a one-member replica set. That way, if you want to add more members later, you can do so without downtime.

If you are starting a brand-new set, you can send the configuration to any member in the set. If you are starting with data on one of the members, you must send the configuration to the member with data. You cannot initiate a replica set with data on more than one member.

Once initiated, you should have a fully functional replica set. The replica set should elect a primary. You can view the status of replica set using `rs.status()`. The output from `rs.status()` tells you quite a bit about the replica set including a number of things we've not yet covered, but don't worry, we'll get there! At this point, I'd like you to simply pay attention to the `members` array. Note that all three of our `mongod` instances are listed in this array and that one of them, in this case the `mongod` running on port 27017, has been elected primary. The other two are secondaries. If you try this for yourself you will certainly have different values for date and the several `Timestamp` values in this output, but you might also find that a different `mongod` was elected primary. That is fine. Any one of them could be elected primary.

```
rs.status()
{
    "set" : "mdbDefGuide",
    "date" : ISODate("2017-07-27T20:23:31.457Z"),
    "myState" : 1,
    "term" : NumberLong(1),
    "heartbeatIntervalMillis" : NumberLong(2000),
    "optimes" : {
        "lastCommittedOpTime" : {
            "ts" : Timestamp(1501187006, 1),
            "t" : NumberLong(1)
        },
        "appliedOpTime" : {
            "ts" : Timestamp(1501187006, 1),
            "t" : NumberLong(1)
        }
    }
}
```

```

        "t" : NumberLong(1)
    },
    "durableOpTime" : {
        "ts" : Timestamp(1501187006, 1),
        "t" : NumberLong(1)
    }
},
"members" : [
{
    "_id" : 0,
    "name" : "localhost:27017",
    "health" : 1,
    "state" : 1,
    "stateStr" : "PRIMARY",
    "uptime" : 688,
    "optime" : {
        "ts" : Timestamp(1501187006, 1),
        "t" : NumberLong(1)
    },
    "optimeDate" : ISODate("2017-07-27T20:23:26Z"),
    "electionTime" : Timestamp(1501186514, 1),
    "electionDate" : ISODate("2017-07-27T20:15:14Z"),
    "configVersion" : 1,
    "self" : true
},
{
    "_id" : 1,
    "name" : "localhost:27018",
    "health" : 1,
    "state" : 2,
    "stateStr" : "SECONDARY",
    "uptime" : 508,
    "optime" : {
        "ts" : Timestamp(1501187006, 1),
        "t" : NumberLong(1)
    },
    "optimeDurable" : {
        "ts" : Timestamp(1501187006, 1),
        "t" : NumberLong(1)
    },
    "optimeDate" : ISODate("2017-07-27T20:23:26Z"),
    "optimeDurableDate" : ISODate("2017-07-27T20:23:26Z"),
    "lastHeartbeat" : ISODate("2017-07-27T20:23:30.818Z"),
    "lastHeartbeatRecv" : ISODate("2017-07-27T20:23:30Z"),
    "pingMs" : NumberLong(0),
    "syncingTo" : "localhost:27017",
    "configVersion" : 1
},
{
    "_id" : 2,
    "name" : "localhost:27019",
    "health" : 1,
    "state" : 2,
    "stateStr" : "SECONDARY",

```

```
        "uptime" : 508,
        "optime" : {
            "ts" : Timestamp(1501187006, 1),
            "t" : NumberLong(1)
        },
        "optimeDurable" : {
            "ts" : Timestamp(1501187006, 1),
            "t" : NumberLong(1)
        },
        "optimeDate" : ISODate("2017-07-27T20:23:26Z"),
        "optimeDurableDate" : ISODate("2017-07-27T20:23:26Z"),
        "lastHeartbeat" : ISODate("2017-07-27T20:23:30.818Z"),
        "lastHeartbeatRecv" : ISODate("2017-07-27T20:23:30.818Z"),
        "pingMs" : NumberLong(0),
        "syncingTo" : "localhost:27017",
        "configVersion" : 1
    }
],
"ok" : 1,
"operationTime" : Timestamp(1501187006, 1)
}
```

## rs Helper Functions

Note the `rs` in the `rs.initiate()` command above. `rs` is a global variable that contains replication helper functions (run `rs.help()` to see the helpers it exposes). These functions are almost always just wrappers around database commands. For example, the following database command is equivalent to `rs.initiate(config)`:

```
> db.adminCommand({"replSetInitiate" : config})
```

It is good to have a familiarity with both the helpers and the underlying commands, because it might be easier to use the command form instead of the helper.

## Observing Replication

If your replica set elected the mongod on port 27017 as primary, then the mongo shell we used to initiate the replica set is currently connected to the primary. You should see the prompt change to something like the following, which indicates that we are connected to the primary of the replica set having the `_id` "mdbDefGuide". To simplify and for the sake of clarity, we'll abbreviate the mongo shell prompt to just `>` throughout replication examples.

```
mdbDefGuide:PRIMARY>
```

If your replica set elected a different node primary, quit the shell and connect to the primary by specifying the correct port number in the command line as we did when launching the *mongo* shell above. For example, if your set's primary is on port 27018, connect using the following command.

```
mongo --port 27018
```

Now that you're connected to the primary, let's try doing some writes and see what happens. First, insert 1,000 documents:

```
> use test
> for (i=0; i<1000; i++) {db.coll.insert({count: i})}
>
> // make sure the docs are there
> db.coll.count()
1000
```

Now check one of the secondaries and verify that they have a copy of all of these documents. Connect to either of the secondaries:. We could do this by quitting the shell and connecting using the port number of one of the secondaries, but it's easy to simply acquire a connection to one of the secondaries by instantiating a connection object using the `Mongo()` constructor within the shell we're already running.

First, use your connection to the test db on the primary to run the *isMaster* command. This will show you the status of the replica set: in a much more concise form than `rs.status()`. It is also a convenient means of determining which member is primary when writing application code or scripting.

```
> db.isMaster()
{
    "hosts" : [
        "localhost:27017",
        "localhost:27018",
        "localhost:27019"
    ],
    "setName" : "mdbDefGuide",
    "setVersion" : 1,
    "ismaster" : true,
    "secondary" : false,
    "primary" : "localhost:27017",
    "me" : "localhost:27017",
    "electionId" : ObjectId("7fffffff0000000000000004"),
    "lastWrite" : {
        "opTime" : {
```

```

        "ts" : Timestamp(1501198208, 1),
        "t" : NumberLong(4)
    },
    "lastWriteDate" : ISODate("2017-07-27T23:30:08Z"),
    "maxBsonObjectSize" : 16777216,
    "maxMessageSizeBytes" : 48000000,
    "maxWriteBatchSize" : 1000,
    "localTime" : ISODate("2017-07-27T23:30:08.722Z"),
    "maxWireVersion" : 6,
    "minWireVersion" : 0,
    "readOnly" : false,
    "compression" : [
        "snappy"
    ],
    "ok" : 1,
    "operationTime" : Timestamp(1501198208, 1)
}

```

---

If at any point an election is called and the mongod we're connected to becomes a secondary, we can use the *isMaster* command to determine which member has become primary. Given that both *localhost:27018* and *localhost:27019* are both secondaries, we can use either for our purposes. Let's instantiate a connection to *localhost:27019*.

```

> secondaryConn = new Mongo("localhost:27019")
connection to localhost:27019
>
> secondaryDB = secondaryConn.getDB("test")
test

```

---

Now, if we attempt to do a read on the collection that has been replicated to the secondary, we'll get an error. Let's attempt to do a `find()` on this collection and then review the error and why we get it.

```

> secondaryDB.coll.find()
Error: error: {
    "operationTime" : Timestamp(1501200089, 1),
    "ok" : 0,
    "errmsg" : "not master and slaveOk=false",
    "code" : 13435,
    "codeName" : "NotMasterNoSlaveOk"
}

```

---

Secondaries may fall behind the primary (or lag) and not have the most current writes, so secondaries will refuse read requests by default to prevent applications from accidentally reading stale data. Thus, if you attempt to query a secondary, you'll get an error that it's not

primary. This is to protect your application from accidentally connecting to a secondary and reading stale data. To allow queries on the secondary, we set an “I’m okay with reading from secondaries” flag, like so:

```
> secondaryConn.setSlaveOk()
```

Note that `slaveOk` is set on the *connection* (`secondaryConn`), not the database (`secondaryDB`).

Now you’re all set to read from this member. Query it normally:

```
> secondaryDB.coll.find()
{ "_id" : ObjectId("597a750696fd35621b4b85db"), "count" : 0 }
{ "_id" : ObjectId("597a750696fd35621b4b85dc"), "count" : 1 }
{ "_id" : ObjectId("597a750696fd35621b4b85dd"), "count" : 2 }
{ "_id" : ObjectId("597a750696fd35621b4b85de"), "count" : 3 }
{ "_id" : ObjectId("597a750696fd35621b4b85df"), "count" : 4 }
{ "_id" : ObjectId("597a750696fd35621b4b85e0"), "count" : 5 }
{ "_id" : ObjectId("597a750696fd35621b4b85e1"), "count" : 6 }
{ "_id" : ObjectId("597a750696fd35621b4b85e2"), "count" : 7 }
{ "_id" : ObjectId("597a750696fd35621b4b85e3"), "count" : 8 }
{ "_id" : ObjectId("597a750696fd35621b4b85e4"), "count" : 9 }
{ "_id" : ObjectId("597a750696fd35621b4b85e5"), "count" : 10 }
{ "_id" : ObjectId("597a750696fd35621b4b85e6"), "count" : 11 }
{ "_id" : ObjectId("597a750696fd35621b4b85e7"), "count" : 12 }
{ "_id" : ObjectId("597a750696fd35621b4b85e8"), "count" : 13 }
{ "_id" : ObjectId("597a750696fd35621b4b85e9"), "count" : 14 }
{ "_id" : ObjectId("597a750696fd35621b4b85ea"), "count" : 15 }
{ "_id" : ObjectId("597a750696fd35621b4b85eb"), "count" : 16 }
{ "_id" : ObjectId("597a750696fd35621b4b85ec"), "count" : 17 }
{ "_id" : ObjectId("597a750696fd35621b4b85ed"), "count" : 18 }
{ "_id" : ObjectId("597a750696fd35621b4b85ee"), "count" : 19 }
Type "it" for more
```

You can see that all of our documents are there.

Now, try to write to a secondary:

```
> secondaryDB.coll.insert({"count" : 1001})
WriteResult({ "writeError" : { "code" : 10107, "errmsg" : "not master" } })
> secondaryDB.coll.count()
1000
```

You can see that the secondary does not accept the write. The secondary will only perform

writes that it gets through replication, not from clients.

There is one other interesting feature that you should try out: automatic failover. If the primary goes down, one of the secondaries will automatically be elected primary. To try this out, stop the primary:

```
> db.adminCommand({ "shutdown" : 1 })
```

You'll see some error messages generated when you run this command because the *mongod* running on port 27017 (the member we're connected to) will terminate and the shell we're using will lose its connection.

```
2017-07-27T20:10:50.612-0400 E QUERY      [thread1] Error: error doing query
while attempting to run command 'shutdown' on host '127.0.0.1:27017'  :
DB.prototype.runCommand@src/mongo/shell/db.js:163:1
DB.prototype.adminCommand@src/mongo/shell/db.js:179:16
@(shell):1:1
2017-07-27T20:10:50.614-0400 I NETWORK  [thread1] trying reconnect to 127.0.0.1:27017
2017-07-27T20:10:50.615-0400 I NETWORK  [thread1] reconnect 127.0.0.1:27017
MongoDB Enterprise mdbDefGuide:SECONDARY>
2017-07-27T20:10:56.051-0400 I NETWORK  [thread1] trying reconnect to 127.0.0.1:27017
2017-07-27T20:10:56.051-0400 W NETWORK  [thread1] Failed to connect to 127.0.0.1:27017
in(checking socket for error after poll), reason: Connection refused
2017-07-27T20:10:56.051-0400 I NETWORK  [thread1] reconnect 127.0.0.1:27017
MongoDB Enterprise >
MongoDB Enterprise > secondaryConn.isMaster()
2017-07-27T20:11:15.422-0400 E QUERY      [thread1] TypeError: secondaryConn
@(shell):1:1
```

This isn't a problem. It won't cause the shell to crash. Go ahead and run `isMaster` on the secondary to see who has become the new primary:

```
> secondaryDB.isMaster()
```

The output from `isMaster` should look something like this:

```
{
  "hosts" : [
    "localhost:27017",
    "localhost:27018",
    "localhost:27019"
  ],
  "setName" : "mdbDefGuide",
  "setVersion" : 1,
```

```

    "ismaster" : true,
    "secondary" : false,
    "primary" : "localhost:27018",
    "me" : "localhost:27018",
    "electionId" : ObjectId("7fffffff0000000000000005"),
    "lastWrite" : {
        "opTime" : {
            "ts" : Timestamp(1501200681, 1),
            "t" : NumberLong(5)
        },
        "lastWriteDate" : ISODate("2017-07-28T00:11:21Z")
    },
    "maxBsonObjectSize" : 16777216,
    "maxMessageSizeBytes" : 48000000,
    "maxWriteBatchSize" : 1000,
    "localTime" : ISODate("2017-07-28T00:11:28.115Z"),
    "maxWireVersion" : 6,
    "minWireVersion" : 0,
    "readOnly" : false,
    "compression" : [
        "snappy"
    ],
    "ok" : 1,
    "operationTime" : Timestamp(1501200681, 1)
}

```

Note that the primary has switched to 27018. Your primary may be the other server; whichever secondary noticed that the primary was down first will be elected. Now you can send writes to the new primary.

*isMaster* is a very old command, predating replica sets to when MongoDB only supported master-slave replication. Thus, it does not use the replica set terminology consistently: it still calls the primary a “master.” You can generally think of “master” as equivalent to “primary” and “slave” as equivalent to “secondary.”

Go ahead and bring back up the server we had running at *localhost:27017*. You simply need to find the command-line interface from which you launched it. You’ll see some messages indicating that it terminated. Just run it again using the same command you used to launch it originally.

Congratulations! You just set up, used, and even poked a little at a replica set to force a shutdown and an election for a new primary.

There are a few key concepts to remember:

- Clients can send a primary all the same operations they could send a standalone server (reads, writes, commands, index builds, etc.).

- Clients cannot write to secondaries.
- Clients, by default, cannot read from secondaries. By explicitly setting an “I know I’m reading from a secondary” setting, clients can read from secondaries.

## Changing Your Replica Set Configuration

Replica set configurations can be changed at any time: members can be added, removed, or modified. There are shell helpers for some common operations; for example, to add a new member to the set, you can use `rs.add`:

```
> rs.add("localhost:27020")
```

Similarly, you can remove members;

```
> rs.remove("localhost:27017")
{ "ok" : 1, "operationTime" : Timestamp(1501202441, 2) }
```

You can check that a reconfiguration succeeded by run `rs.config()` in the shell. It will print the current configuration:

```
> rs.config()
{
  "_id" : "mdbDefGuide",
  "version" : 3,
  "protocolVersion" : NumberLong(1),
  "members" : [
    {
      "_id" : 1,
      "host" : "localhost:27018",
      "arbiterOnly" : false,
      "buildIndexes" : true,
      "hidden" : false,
      "priority" : 1,
      "tags" : {

      },
      "slaveDelay" : NumberLong(0),
      "votes" : 1
    },
    {
      "_id" : 2,
      "host" : "localhost:27019",
      "arbiterOnly" : false,
      "buildIndexes" : true,
      "hidden" : false,
```

```

        "priority" : 1,
        "tags" : {

    },
    "slaveDelay" : NumberLong(0),
    "votes" : 1
},
{
    "_id" : 3,
    "host" : "localhost:27020",
    "arbiterOnly" : false,
    "buildIndexes" : true,
    "hidden" : false,
    "priority" : 1,
    "tags" : {

    },
    "slaveDelay" : NumberLong(0),
    "votes" : 1
}
],
"settings" : {
    "chainingAllowed" : true,
    "heartbeatIntervalMillis" : 2000,
    "heartbeatTimeoutSecs" : 10,
    "electionTimeoutMillis" : 10000,
    "catchUpTimeoutMillis" : -1,
    "getLastErrorModes" : {

    },
    "getLastErrorDefaults" : {
        "w" : 1,
        "wtimeout" : 0
    },
    "replicaSetId" : ObjectId("597a49c67e297327b1e5b116")
}
}
}

```

Each time you change the configuration, the "version" field will increase. It starts at version 1.

You can also modify existing members, not just add and remove them. To make modifications, create the configuration document that you want in the shell and call `rs.reconfig`. For example, suppose we have a configuration such as the one shown here:

---

```

> rs.config()
{
    "_id" : "testReplSet",
    "version" : 2,
    "members" : [

```

```

    {
        "_id" : 0,
        "host" : "198.51.100.1:27017"
    },
    {
        "_id" : 1,
        "host" : "localhost:27018"
    },
    {
        "_id" : 2,
        "host" : "localhost:27019"
    }
]
}

```

Someone accidentally added member 0 by IP, instead of its hostname. To change that, first we load the current configuration in the shell and then we change the relevant fields:

```

> var config = rs.config()
> config.members[0].host = "localhost:27017"

```

Now that the config document is correct, we need to send it to the database using the `rs.reconfig` helper:

```
> rs.reconfig(config)
```

`rs.reconfig` is often more useful than `rs.add` and `rs.remove` for complex operations, such as modifying members' configuration or adding/removing multiple members at once. You can use it to make any legal configuration change you need: simply create the config document that represents your desired configuration and pass it to `rs.reconfig`.

## How to Design a Set

To plan out your set, there are certain replica set concepts that you must be familiar with. The next chapter goes into more detail about these, but the most important is that replica sets are all about majorities: you need a majority of members to elect a primary, a primary can only stay primary as long as it can reach a majority, and a write is safe when it's been replicated to a majority. This majority is defined to be “more than half of all members in the set,” as shown in [Table 8-1](#).

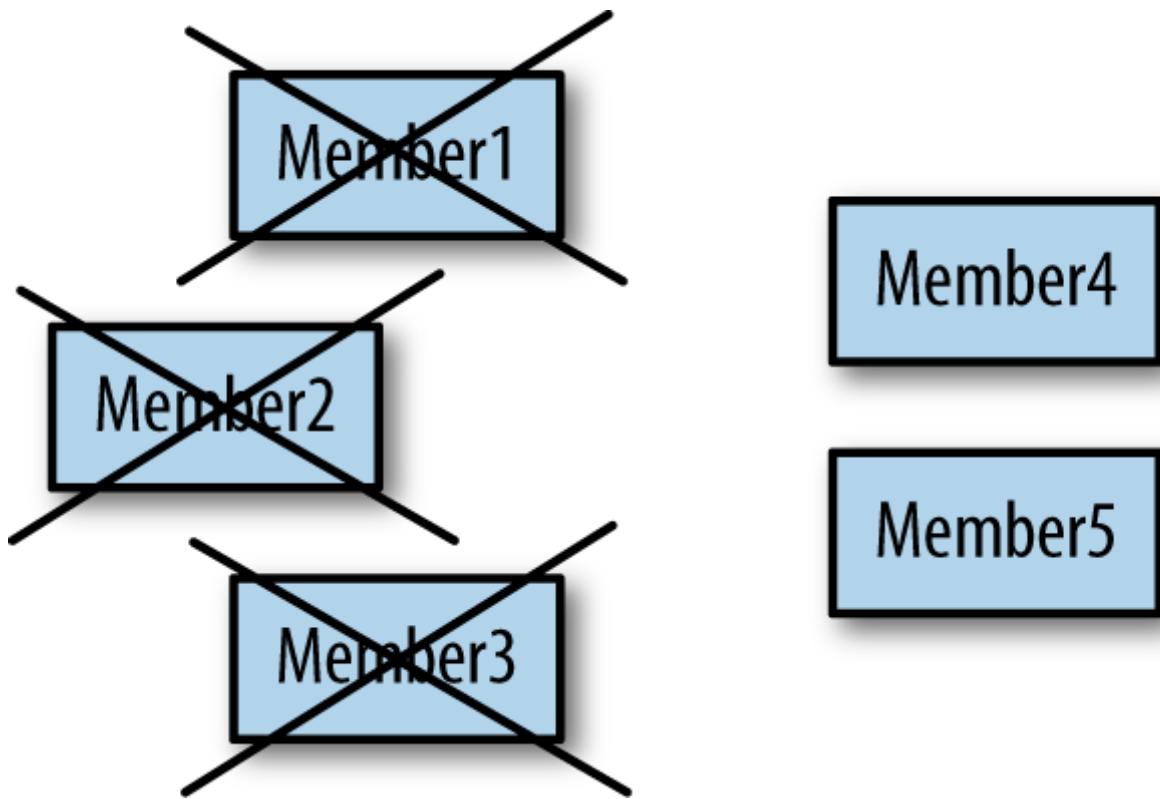
*Table 8-1. What is a majority?*

Number of members in the set	Majority of the set
------------------------------	---------------------

1	1
2	2
3	2
4	3
5	3
6	4
7	4

Note that it doesn't matter how many members are down or unavailable; majority is based on the set's configuration.

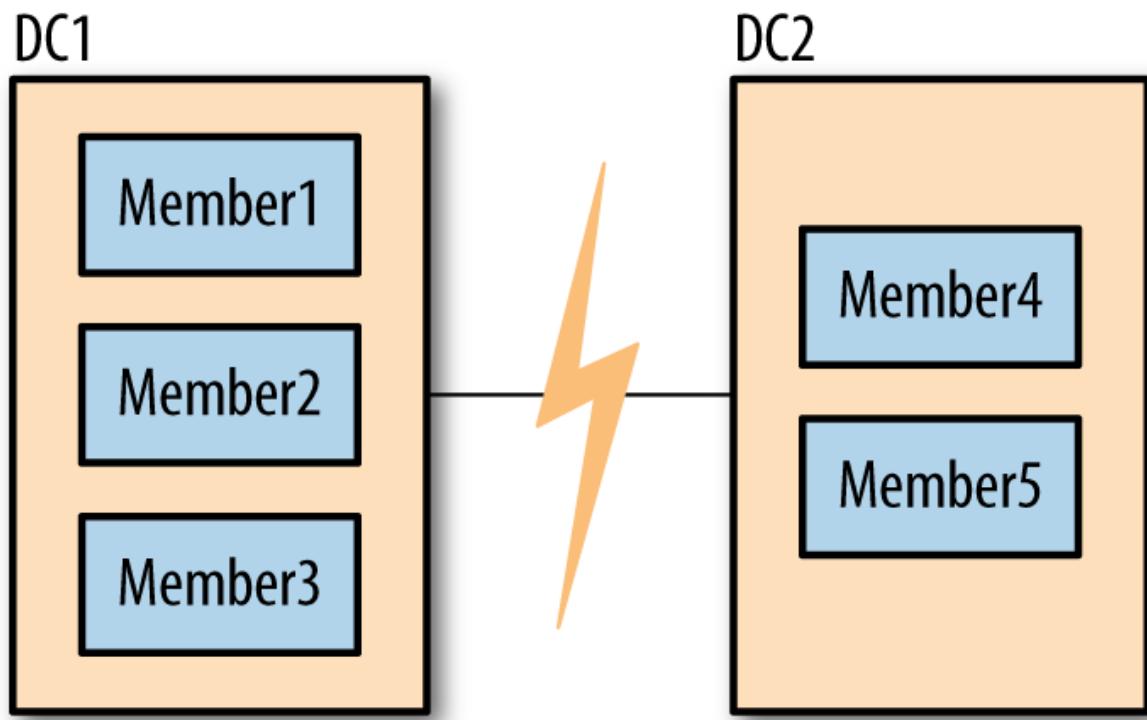
For example, suppose that we have a five-member set and three members go down, as shown in Figure 8-1. There are still two members up. These two members cannot reach a majority of the set (at least three members), so they cannot elect a primary. If one of them were primary, it would step down as soon as it noticed that it could not reach a majority. After a few seconds, your set would consist of two secondaries and three unreachable members.



*Figure 8-1. With a minority of the set available, all members will be secondaries*

Many users find this frustrating: why can't the two remaining members elect a primary? The problem is that it's possible that the other three members didn't go down, and that it was the network that went down, as shown in Figure 8-2. In this case, the three members on the left will elect a primary, since they can reach a majority of the set (three members out of five).

In the case of a network partition, we do not want both sides of the partition to elect a primary: otherwise the set would have two primaries. Then both primaries would be writing to the data and the data sets would diverge. Requiring a majority to elect or stay primary is a neat way of avoiding ending up with more than one primary.



*Figure 8-2. For the members, a network partition looks identical to servers on the other side of the partition going down*

It is important to configure your set in such a way that you'll usually be able to have one primary. For example, in the five-member set described above, if members 1, 2, and 3 are in one data center and members 4 and 5 are in another, there should almost always be a majority available in the first data center (it's more likely to have a network break between data centers than within them).

There are a couple of common configurations that are recommended:

- A majority of the set in one data center, as in Figure 8-2. This is a good design if you have a primary data center where you always want your replica set's primary to be located. So long as your primary data center is healthy, you will have a primary. However, if that data center becomes unavailable, your secondary data center will not be able to elect a new primary.
- An equal number of servers in each data center, plus a tie-breaking server in a third location. This is a good design if your data centers are “equal” in preference, since generally servers from either data center will be able to see a majority of the set. However, it involves having three separate locations for servers.

More complex requirements might require different configurations, but you should keep in mind how your set will acquire a majority under adverse conditions.

All of these complexities would disappear if MongoDB supported having more than one

primary. However, multimaster would bring its own host of complexities. With two primaries, you would have to handle conflicting writes (for example, someone updates a document on one primary and someone deletes it on another primary). There are two popular ways of handling conflicts in systems that support multiple writers: manual reconciliation or having the system arbitrarily pick a “winner.” Neither of these options is a very easy model for developers to code against, seeing that you can’t be sure that the data you’ve written won’t change out from under you. Thus, MongoDB chose to only support having a single primary. This makes development easier but can result in periods when the replica set is read-only.

## How Elections Work

When a secondary cannot reach a primary, it will contact all the other members and request that it be elected primary. These other members do several sanity checks: Can they reach a primary that the member seeking election cannot? Is the member seeking election up to date with replication? Is there anyone with a higher priority available who should be elected instead?

In version 3.2, MongoDB introduced version 1 of the replication protocol `{protocolVersion: 1}`. Protocol version 1 is based on the RAFT consensus protocol developed by Diego Ongaro and John Ousterhout at Stanford University.

Replica set members send heartbeats (pings) to each other every two seconds. If a heartbeat does not return within 10 seconds, the other members mark the delinquent member as inaccessible. The election algorithm will make a “best-effort” attempt to have the secondary with the highest priority available call an election. Member priority affects both the timing and the outcome of elections; secondaries with higher priority call elections relatively sooner than secondaries with lower priority, and are also more likely to win. However, a lower priority instance can be elected as primary for brief periods, even if a higher priority secondary is available. Replica set members continue to call elections until the highest priority member available becomes primary.

To be elected primary, a member must be up to date with replication, as far as the members it can reach know. All replicated operations are strictly ordered by an ascending identifier, so the candidate must have operations later than or equal to any member it can reach.

## Member Configuration Options

The replica sets we have set up so far have been fairly uniform in that every member has the same configuration as every other member. However, there are many situations when you don’t want members to be identical: you might want one member to preferentially be primary or make a member invisible to clients so that no read requests can be routed to it. These and many other configuration options can be specified in the member subdocuments of the replica set

configuration. This section outlines the member options that you can set.

## Priority

Priority is how strongly this member “wants” to become primary. Priority can range from 0 to 100 and the default is 1. Setting priority to 0 has a special meaning: members with 0 priority can never become primary. These members are called passive members.

The highest-priority member will always be elected primary (so long as they can reach a majority of the set and have the most up-to-date data). For example, suppose you add a member with priority of 1.5 to the set, like so:

```
> rs.add({"host" : "server-4:27017", "priority" : 1.5})
```

Assuming the other members of the set have priority 1, once *server-4* caught up with the rest of the set, the current primary would automatically step down and *server-4* would elect itself. If *server-4* was, for some reason, unable to catch up, the current primary would stay primary. Setting priorities will never cause your set to go primary-less. It will also never cause a member who is behind to become primary (until it has caught up).

The absolute value of a priority only matters in relation to whether it is greater or less than the other priorities in the set: members with priorities of 500, 1, and 1 will behave the same way as another set with priorities 2, 1, and 1.

## Hidden

Clients do not route requests to hidden members and hidden members are not preferred as replication sources (although they will be used if more desirable sources are not available). Thus, many people will hide less powerful or backup servers.

For example, suppose you had a set that looked like this:

```
> rs.isMaster()
{
    ...
    "hosts" : [
        "server-1:27017",
        "server-2:27017",
        "server-3:27017"
    ],
    ...
}
```

To hide *server-3*, add the `hidden: true` field to its configuration. A member must have a priority of 0 to be hidden (you can't have a hidden primary):

```
> var config = rs.config()
> config.members[2].hidden = 0
0
> config.members[2].priority = 0
0
> rs.reconfig(config)
```

Now running `isMaster()` will show:

```
> rs.isMaster()
{
  ...
  "hosts" : [
    "server-1:27107",
    "server-2:27017"
  ,
  ...
}
```

`rs.status()` and `rs.config()` will still show the member; it only disappears from `isMaster()`. When clients connect to a replica set, they call `isMaster()` to determine the members of the set. Thus, hidden members will never be used for read requests.

To unhide a member, change the `hidden` option to `false` or remove the option entirely.

## Creating Election Arbiters

The example above shows the disadvantages two-member sets have for majority requirements. However, many people with small deployments do not want to keep three copies of their data, feeling that two is enough and keeping a third copy is not worth the administrative, operational, and financial costs.

For these deployments, MongoDB supports a special type of member called an arbiter, whose only purpose is to participate in elections. Arbiters hold no data and aren't used by clients: they just provide a majority for two-member sets..

As arbiters don't have any of the traditional responsibilities of a *mongod* server, you can run an arbiter as a lightweight process on a wimpier server than you'd generally use for MongoDB. It's often a good idea, if possible, to run an arbiter in a separate failure domain from the other members, so that it has an "outside perspective" on the set, as described in the recommended

deployments listed in “How to Design a Set”.

You start up an arbiter in the same way that you start a normal *mongod*, using the `--replSet` name option and an empty data directory. You can add it to the set using the `rs.addArb()` helper:

```
> rs.addArb("server-5:27017")
```

Equivalently, you can specify the `arbiterOnly` option in the member configuration:

```
> rs.add({_id : 4, "host" : "server-5:27017", "arbiterOnly" : true})
```

An arbiter, once added to the set, is an arbiter forever: you cannot reconfigure an arbiter to become a nonarbiter, or vice versa.

One other thing that arbiters are good for is breaking ties in larger clusters. If you have an even number of nodes, you may have half the nodes vote for one member and half for another. Adding an arbiter can add a deciding vote.

## USE AT MOST ONE ARBITER

Note that, in both of the use cases above, you need *at most* one arbiter. You do not need an arbiter if you have an odd number of nodes. A common misconception seems to be that you should add extra arbiters “just in case.” However, it doesn’t help elections go any faster or provide any data safety to add extra arbiters.

Suppose you have a three members set. Two members are required to elect a primary. If you add an arbiter, you’ll have a four member set, so three members will be required to choose a primary. Thus, your set is potentially less stable: instead of requiring 67% of your set to be up, you’re now requiring 75%.

Having extra members can also make elections take longer. If you have an even number of nodes because you added an arbiter, your arbiters can cause ties, not prevent them.

## THE DOWNSIDE TO USING AN ARBITER

If you have a choice between a data node and an arbiter, choose a data node. Using an arbiter instead of a data node in a small set can make some operational tasks more difficult. For example, suppose you are running a replica set with two “normal” members and one arbiter, and one of the data-holding members goes down. If that member is well and truly dead (the data is unrecoverable), you will have to get a copy of the data from the current primary to the new

server you'll be using as a secondary. Copying data can put a lot of stress on a server and, thus, slow down your application. (Generally, copying a few gigabytes to a new server is trivial but more than a hundred starts becoming impractical.)

Conversely, if you have three data-holding members, there's more "breathing room" if a server completely dies. You can use the remaining secondary to bootstrap a new server instead of depending on your primary.

In the two-member-plus-arbiter scenario, the primary is the last remaining good copy of your data *and* the one trying to handle load from your application while you're trying to get another copy of your data online.

Thus, if possible, use an odd number of "normal" members instead of an arbiter.

## **Building Indexes**

Sometimes a secondary does not need to have the same (or any) indexes that exist on the primary. If you are using a secondary only for backup data or offline batch jobs, you might want to specify "buildIndexes" : false in the member's configuration. This option prevents the secondary from building any indexes.

This is a permanent setting: members that have "buildIndexes" : false specified can never be reconfigured to be "normal" index-building members again. If you want to change a non-index-building member to an index-building one, you must remove it from the set, delete all of its data, re-add it to the set, and allow it to resync from scratch.

Again, this option requires the member's priority to be 0.

# Chapter 9. Components of a Replica Set

---

This chapter covers how the pieces of a replica set fit together, including:

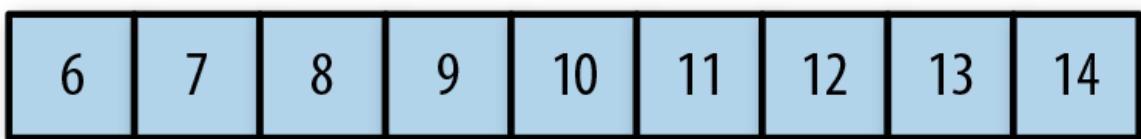
- How replica set members replicate new data
- How bringing up new members works
- How elections work
- Possible server and network failure scenarios

## Syncing

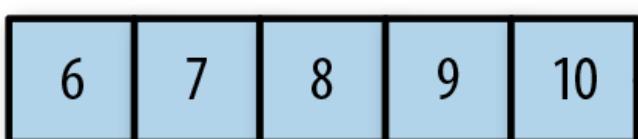
Replication is concerned with keeping an identical copy of data on multiple servers. The way MongoDB accomplishes this is by keeping a log of operations, or oplog, containing every write that a primary performs. This is a capped collection that lives in the *local* database on the primary. The secondaries query this collection for operations to replicate.

Each secondary maintains its own oplog, recording each operation it replicates from the primary. This allows any member to be used as a sync source for any other member, as shown in [Figure 9-1](#). Secondaries fetch operations from the member they are syncing from, apply the operations to their data set, and then write the operations to the oplog. If applying an operation fails (which should only happen if the underlying data has been corrupted or in some way differs from the primary), the secondary will exit.

## Primary

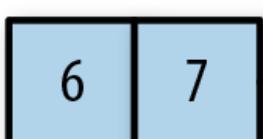


## Secondary #1



Query for ops {"\$gt":10}

## Secondary #2



Query for ops {"\$gt":7}

Figure 9-1. Oplog keep an ordered list of write operations that have occurred. Each member has its own copy of the oplog, which should be identical to the primary's (modulo some lag).

If a secondary goes down for any reason, when it restarts it will start syncing from the last operation in its oplog. As operations are applied to data and then written to the oplog, the secondary may replay operations that it has already applied to its data. MongoDB is designed to handle this correctly: replaying oplog ops multiple times yields the same result as replaying them once. Each operation in the oplog is idempotent. That is, oplog operations produce the same results whether applied once or multiple times to the target dataset.

Because the oplog is a fixed size, it can only hold a certain number of operations. In general, the oplog will use space at approximately the same rate as writes come into the system: if you're writing 1 KB/minute on the primary, your oplog is probably going to fill up at about 1 KB/minute. However, there are a few exceptions: operations that effect multiple documents, such as removes or a multi-updates, that will be exploded into many oplog entries. The single operation on the primary will be split into one oplog op per document affected. Thus, if you remove 1,000,000 documents from a collection with `db.coll.remove()`, it will become 1,000,000 oplog entries removing one document at a time. If you are doing lots of bulk operations, this can fill up your oplog more quickly than you might expect.

In most cases, the default oplog size is sufficient. If you can predict your replica set's workload to resemble one of the following patterns, then you might want to create an oplog that is larger than the default. Conversely, if your application predominantly performs reads with a minimal

amount of write operations, a smaller oplog may be sufficient. The following workloads might require a larger oplog size.

- **Updates to Multiple Documents at Once:** The oplog must translate multi-updates into individual operations in order to maintain idempotency. This can use a great deal of oplog space without a corresponding increase in data size or disk use.
- **Deletions Equal the Same Amount of Data as Inserts** If you delete roughly the same amount of data as you insert, the database will not grow significantly in disk use, but the size of the operation log can be quite large.
- **Significant Number of In-Place Updates** If a significant portion of the workload is updates that do not increase the size of the documents, the database records a large number of operations but does not change the quantity of data on disk.

Before mongod creates an oplog, you can specify its size with the `oplogSizeMB` option. However, after you have started a replica set member for the first time, you can only change the size of the oplog using the `Change the Size of the Oplog` procedure.

MongoDB uses two forms of data synchronization: initial sync to populate new members with the full data set, and replication to apply ongoing changes to the entire data set.

## Initial Sync

Initial sync copies all the data from one member of the replica set to another member. When a member of the set starts up, it will check if it is in a valid state to begin syncing from someone. If not, it will attempt to make a full copy of data from another member of the set. This is called initial syncing and there are several steps to the process, which you can follow in the *mongod*'s log:

First, MongoDB clones all databases except the local database. To clone, the `mongod` scans every collection in each source database and inserts all data into its own copies of these collections on the target member. Prior to beginning the clone operations, any existing data on the target member will be dropped at this point. Only do an initial sync for a member if you do not want the data in your data directory or have moved it elsewhere, as *mongod*'s first action is to delete it all.

Initial sync builds all collection indexes as the documents are copied for each collection. In earlier versions of MongoDB, only the `_id` indexes are built during this stage.

Initial sync pulls newly added oplog records during the data copy. Ensure that the target member has enough disk space in the local database to temporarily store these oplog records for

the duration of this data copy stage.

Once all databases are cloned, mongod applies all changes to the data set that occurred while the copy was in progress. Using the oplog from the source, the mongod updates its data set to reflect the current state of the replica set. Note that changes might include any type of write including inserts, updates, and deletes. This process might mean that mongod has to reclone certain documents that were moved and, therefore, missed by the cloner:

```
Mon Jan 30 15:38:36 [rsSync] oplog sync 1 of 3
Mon Jan 30 15:38:36 [rsBackgroundSync] replSet syncing to: server-1:27017
Mon Jan 30 15:38:37 [rsSyncNotifier] replset setting oplog notifier to
server-1:27017
Mon Jan 30 15:38:37 [repl writer worker 2] replication update of non-mod
failed:
{ ts: Timestamp 1352215827000|17, h: -5618036261007523082, v: 2, op: "i
ns: "db1.someColl", o2: { _id: ObjectId('50992a2a7852201e750012b7') }
o: { $set: { count.0: 2, count.1: 0 } } }
Mon Jan 30 15:38:37 [repl writer worker 2] replication info
adding missing object
Mon Jan 30 15:38:37 [repl writer worker 2] replication missing object
not found on source. presumably deleted later in oplog
```

This is roughly what the logs will look like if some documents had to be recloned. Depending on the level of traffic and the types of operations that were happening on the sync source, you may or may not have missing objects.

Then the second oplog application occurs, which applies operations that happened during the first oplog application. This one generally passes without much fanfare. It is only distinct from the first application in that there should no longer be anything to reclone.

At this point, the data should exactly match the data set as it existed at some point on the primary. The member finishes the initial sync process and transitions to normal syncing, which allows it to become a secondary:

Doing an initial sync is very easy from an operator perspective: start up a *mongod* with a clean data directory. However, it is often preferable to restore from backup instead. Restoring from backup is often faster than copying all of your data through *mongod*.

Also, cloning can ruin the sync source's working set. Many deployments end up with a subset of their data that's frequently accessed and always in memory (because the OS is accessing it frequently). Performing an initial sync forces the member to page all of its data into memory, evicting the frequently-used data. This can slow down a member dramatically as requests that

were being handled by data in RAM are suddenly forced to go to disk. However, for small data sets and servers with some breathing room, initial syncing is a good, easy option.

One of the most common issues people run into with initial sync is when it takes too long. In these cases, the new member can “fall off” the end of sync source’s oplog: the new member gets so far behind the sync source that it can no longer catch up because the sync source’s oplog has overwritten the data the member would need to use to continue replicating.

There is no way to fix this other than attempting the initial sync at a less-busy time or restoring from a backup. The initial sync cannot proceed if the member has fallen off of the sync source’s oplog. The next section covers this in more depth.

## Replication

Secondary members replicate data continuously after the initial sync. Secondary members copy the oplog from their sync source and apply these operations in an asynchronous process.

Secondaries may automatically change their sync from source as needed based on changes in the ping time and state of other members’ replication. There are several rules that govern which members a given node can sync from. For example, replica set members with 1 vote cannot sync from members with 0 votes. Secondaries avoid syncing from delayed members and hidden members. Elections and different classes of replica set members are discussed in later sections.

## Handling Staleness

If a secondary falls too far behind the actual operations being performed on the sync source, the secondary will go *stale*. A stale secondary is unable to continue catching up because every operation in the sync source’s oplog is too far ahead: it would be skipping operations if it continued to sync. This could happen if the secondary has had downtime, has more writes than it can handle, or is too busy handling reads.

When a secondary goes stale, it will attempt to replicate from each member of the set in turn to see if there’s anyone with a longer oplog that it can bootstrap from. If there is no one with a long-enough oplog, replication on that member will halt and it will need to be fully resynced (or restored from a more recent backup).

To avoid out-of-sync secondaries, it’s important to have a large oplog so that the primary can store a long history of operations. A larger oplog will obviously use more disk space. But in general this is a good trade-off to make because the disk space tends to be cheap and little of the oplog is usually in use, and therefore it doesn’t take up much RAM. For more information on sizing the oplog, see “[Resizing the Oplog](#)”.

## Heartbeats

Members need to know about the other members' states: who's primary, who they can sync from, and who's down. To keep an up-to-date view of the set, a member sends out a heartbeat request to every other member of the set every two seconds. A heartbeat request is a short message that checks everyone's state.

One of the most important functions of heartbeats is to let the primary know if it can reach a majority of the set. If a primary can no longer reach a majority of the servers, it will demote itself and become a secondary.

### Member States

Members also communicate what state they are in via heartbeats. We've already discussed two states: primary and secondary. There are several other normal states that you'll often see members be in:

#### STARTUP

This is the state MongoDB goes into when you first start a member. It's the state when MongoDB is attempting to load a member's replica set configuration. Once the configuration has been loaded, it transitions to STARTUP2.

#### STARTUP2

This state will last throughout the initial sync process but on a normal member, it should only ever last a few seconds. It just forks off a couple of threads to handle replication and elections and then transitions into the next state: RECOVERING.

#### RECOVERING

This state means that the member is operating correctly but is not available for reads. This state is a bit overloaded: you may see it in a variety of situations.

On startup, a member has to make a couple checks to make sure it's in a valid state before accepting reads; therefore, all members will go through recovering state briefly on startup before becoming secondaries. A member can also go into RECOVERING state during long-running operations such as compact or in response to the `replSetMaintenance` command.

A member will also go into RECOVERING state if it has fallen too far behind the other members to catch up. This is, generally, a failure state that requires resyncing the member. The member does not go into an error state at this point because it lives in hope that

someone will come online with a long-enough oplog that it can bootstrap itself back to non-staleness.

## ARBITER

Arbiters have a special state and should always be in state ARBITER during normal operation.

There are also a few states that indicate a problem with the system. These include:

## DOWN

If a member was up but then becomes unreachable. Note that a member reported as “down” might, in fact, still be up, just unreachable due to network issues.

## UNKNOWN

If a member has never been able to reach another member, it will not know what state it’s in, so it will report it as unknown. This generally indicates that the unknown member is down or that there are network problems between the two members.

## REMOVED

This is the state of a member that has been removed from the set. If a removed member is added back into the set, it will transition back into its “normal” state.

## ROLLBACK

This state is used when a member is rolling back data, as described in [“Rollbacks”](#). At the end of the rollback process, a server will transition back into the recovering state and then become a secondary.

## Elections

A member will seek election if it cannot reach a primary (and is itself eligible to become primary). A member seeking election will send out a notice to all of the members it can reach. These members may know of reasons that this member is an unsuitable primary: it may be behind in replication or there may already be a primary that the member seeking election cannot reach. In these cases, the other members will not allow the election to proceed.

Assuming that there is no reason to object, the other members will vote for the member seeking election. If the member seeking election receives votes from a majority of the set, the election

was successful and will transition into primary state. If it did not receive a majority of votes, it will remain a secondary and may try to become a primary again later. A primary will remain primary until it cannot reach a majority of members, goes down, is stepped down, or the set is reconfigured.

Assuming that the network is healthy and a majority of the servers are up, elections should be fast. It will take a member up to two seconds to notice that a primary has gone down (due to the heartbeats mentioned earlier) and it will immediately start an election, which should only take a few milliseconds. However, the situation is often non-optimal: an election may be triggered due to networking issues or overloaded servers responding too slowly. In these cases, an election might take more time, even up to a few minutes. TODO: should probably add a brief discussion of RAFT here.

## Rollbacks

The election process described in the previous section means that if a primary does a write and goes down before the secondaries have a chance to replicate it, the next primary elected may not have the write. For example, suppose we have two data centers, one with the primary and a secondary, and the other with three secondaries, as shown in Figure 9-2.

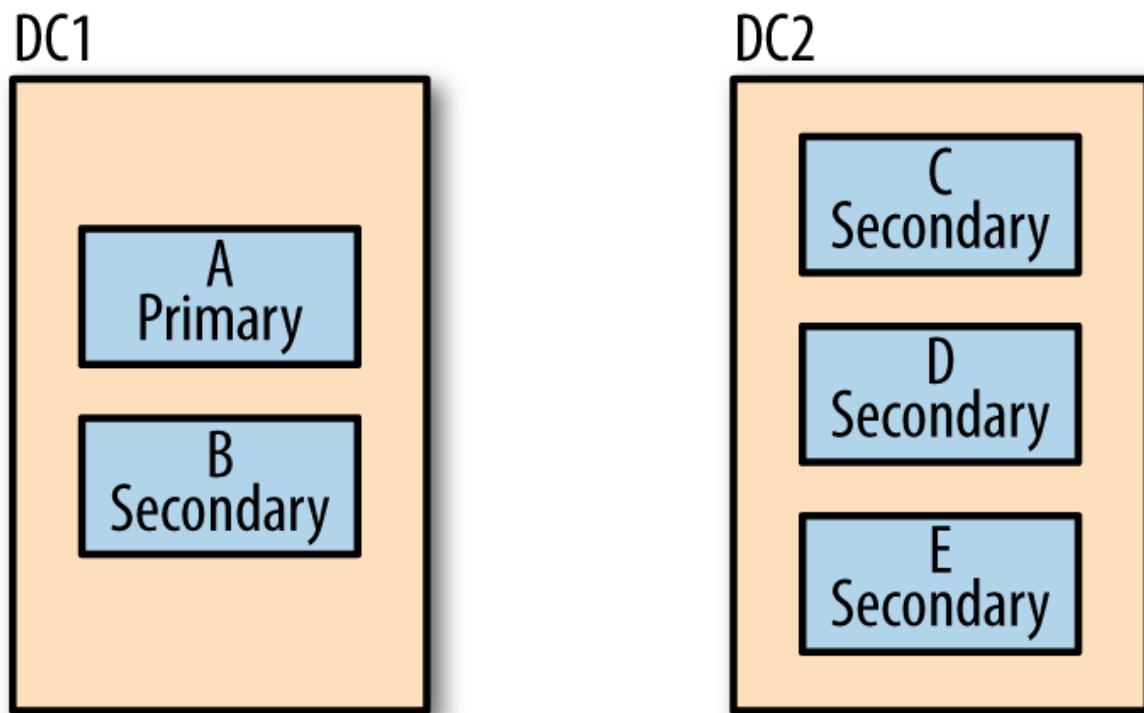
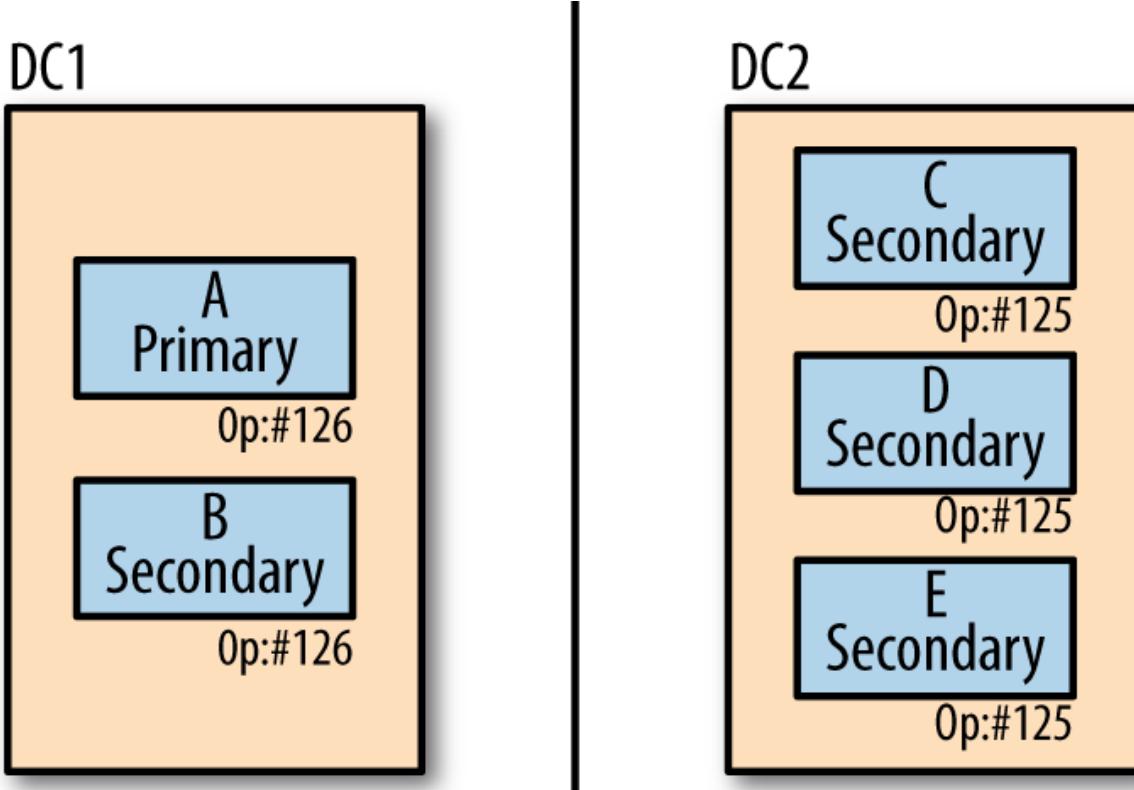


Figure 9-2. A possible two-data-center configuration

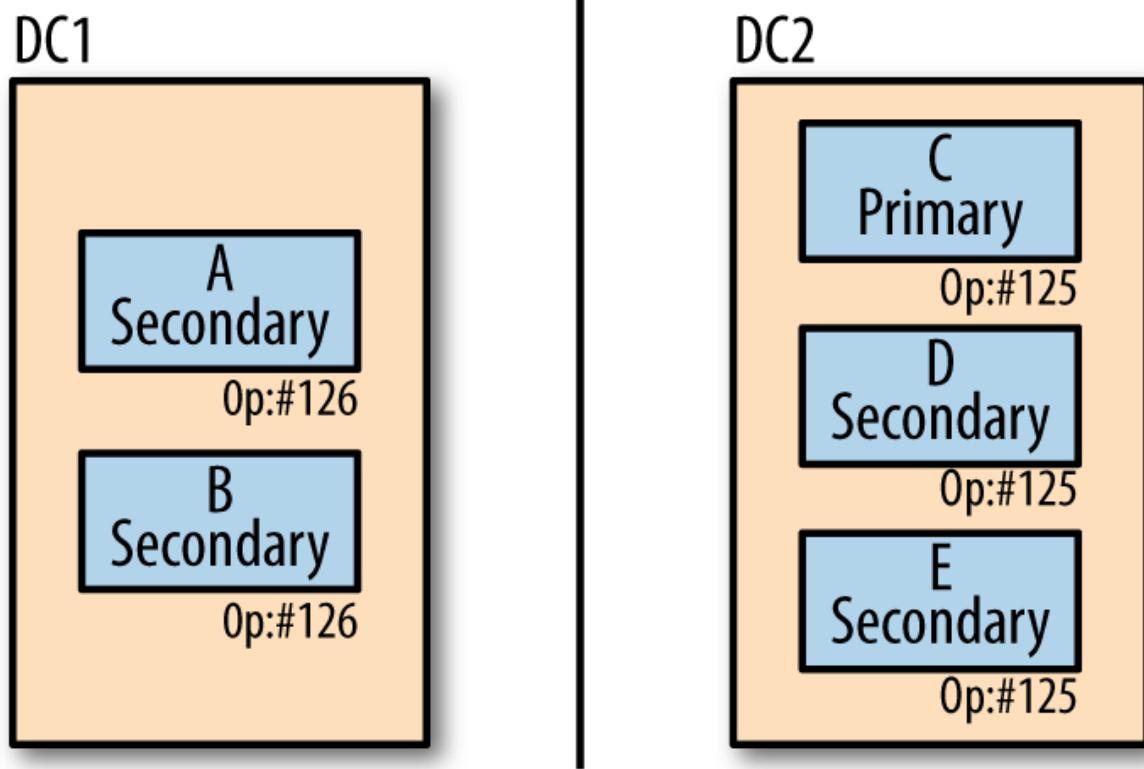
Suppose that there is a network partition between the two data centers, as shown in Figure 9-3. The servers in the first data center are up to operation 126, but that data center hasn't yet

replicated to the servers in the other data center.



*Figure 9-3. Replication across data centers can be slower than within a single data center*

The servers in the other data center can still reach a majority of the set (three out of five servers). Thus, one of them may be elected primary. This new primary begins taking its own writes, as shown in [Figure 9-4](#).



*Figure 9-4. Unreplicated writes won't match writes on the other side of a network partition*

When the network is repaired, the servers in the first data center will look for operation 126 to start syncing from the other servers but will not be able to find it. When this happens, *A* and *B* will begin a process called rollback. Rollback is used to undo ops that were not replicated before failover. The servers with 126 in their oplogs will look back through the oplogs of the servers in the other data center for a common point. They'll find that operation 125 is the latest operation that matches. Figure 9-5 shows what the oplogs would look like.

A

123	124	125	126	127	128
-----	-----	-----	-----	-----	-----

B

123	124	125	126'	127'	128'	129'	130'	131'
-----	-----	-----	------	------	------	------	------	------

Figure 9-5. Two members with conflicting oplogs: A apparently crashed before replicating ops 126–128, so these operations are not present on B, which has more recent operations. A will have to rollback these three operations before resuming syncing.

At this point, the server will go through the ops it has and write its version of each document affected by those ops to a `.bson` file in a `rollback` directory of your data directory. Thus, if (for example) operation 126 was an update, it will write the document updated by 126 to `collectionName.bson`. Then it will copy the version of that document from the current primary.

The following is a paste of the log entries generated from a typical rollback:

```
Fri Oct  7 06:30:35 [rsSync] replSet syncing to: server-1
Fri Oct  7 06:30:35 [rsSync] replSet our last op time written: Oct  7
          06:30:05:3
Fri Oct  7 06:30:35 [rsSync] replset source's GTE: Oct  7 06:30:31:1
Fri Oct  7 06:30:35 [rsSync] replSet rollback 0
Fri Oct  7 06:30:35 [rsSync] replSet ROLLBACK
Fri Oct  7 06:30:35 [rsSync] replSet rollback 1
Fri Oct  7 06:30:35 [rsSync] replSet rollback 2 FindCommonPoint
Fri Oct  7 06:30:35 [rsSync] replSet info rollback our last optime: Oct
          06:30:05:3
Fri Oct  7 06:30:35 [rsSync] replSet info rollback their last optime: Oct
          06:30:31:2
Fri Oct  7 06:30:35 [rsSync] replSet info rollback diff in end of log time:
          -26 seconds
Fri Oct  7 06:30:35 [rsSync] replSet rollback found matching events at Oct
          06:30:03:4118
Fri Oct  7 06:30:35 [rsSync] replSet rollback findcommonpoint scanned : 6
Fri Oct  7 06:30:35 [rsSync] replSet replSet rollback 3 fixup
Fri Oct  7 06:30:35 [rsSync] replSet rollback 3.5
Fri Oct  7 06:30:35 [rsSync] replSet rollback 4 n:3
Fri Oct  7 06:30:35 [rsSync] replSet mininvalid=Oct  7 06:30:31 4e8ed4c7:2
Fri Oct  7 06:30:35 [rsSync] replSet rollback 4.6
Fri Oct  7 06:30:35 [rsSync] replSet rollback 4.7
Fri Oct  7 06:30:35 [rsSync] replSet rollback 5 d:6 u:0
```

```
Fri Oct  7 06:30:35 [rsSync] replSet rollback 6
Fri Oct  7 06:30:35 [rsSync] replSet rollback 7
Fri Oct  7 06:30:35 [rsSync] replSet rollback done
Fri Oct  7 06:30:35 [rsSync] replSet RECOVERING
Fri Oct  7 06:30:36 [rsSync] replSet syncing to: server-1
Fri Oct  7 06:30:36 [rsSync] replSet SECONDARY
```

The server begins syncing from another member (*server-1*, in this case) and realizes that it cannot find its latest operation on the sync source. At that point, it starts the rollback process by going into rollback state ("replSet ROLLBACK").

At step 2, it finds the common point between the two oplogs, which was 26 seconds ago. It then begins undoing the operations from the last 26 seconds from its oplog. Once the rollback is complete, it transitions into recovering state and begins syncing normally again.

To apply operations that have been rolled back to the current primary, first use *mongorestore* to load them into a temporary collection:

```
$ mongorestore --db stage --collection stuff \
> /data/db/rollback/important.stuff.2012-12-19T18-27-14.0.bson
```

Now you should examine the documents (using the shell) and compare them to the current contents of the collection from whence they came. For example, if someone had created a “normal” index on the rollback member and a unique index on current primary, you’d want to make sure that there weren’t any duplicates in the rolled-back data and resolve them if there were.

Once you have a version of the documents that you like in your staging collection, load it into your main collection:

```
> staging.staff.find().forEach(function(doc) {
...     prod.staff.insert(doc);
... })
```

If you have any insert-only collections, you can directly load the rollback documents into the collection. However, if you are doing updates on the collection you will need to be more careful about how you merge rollback data.

One often-misused member configuration option is the number of votes each member has. Manipulating the number of votes is almost always not what you want and causes a lot of rollbacks (which is why it was not included in the list of member properties in the last chapter).

Do not change the number of votes unless you are prepared to deal with regular rollbacks.

For more information on preventing rollbacks, see [Chapter 10](#).

## When Rollbacks Fail

In some cases, MongoDB decides that the rollback is too large to undertake. Rollback can fail if there are more than 300 MB of data or about 30 minutes of operations to roll back. In these cases, you must resync the node that is stuck in rollback.

The most common cause of this is when secondaries are lagging and the primary goes down. If one of the secondaries becomes primary, it will be missing a lot of operations from the old primary. The best way to make sure you don't get a member stuck in rollback is to keep your secondaries as up to date as possible.

# Chapter 10. Connecting to a Replica Set from Your Application

---

This chapter covers how applications interact with replica sets, including:

- How connections and failovers work
- Waiting for replication on writes
- Routing reads to the correct member

## Client-to-Replica-Set Connection Behavior

MongoDB client libraries (“drivers” in MongoDB parlance) are designed to manage communication with MongoDB servers, regardless of whether the server is a standalone MongoDB instance or a replica set. For replica sets, by default, drivers will connect to the primary and route all traffic to it. Your application can perform reads and writes as though it were talking to a standalone server while your replica set quietly keeps hot standbys ready in the background.

Connections to a replica set are similar to connections to a single server. Use the `MongoClient` class (or equivalent) in your driver and provide a seed list for the driver to connect to. A seed list is simply a list of server addresses. Seeds are members of the replica set your application will read from and write data to. You do not need to list all members in the seed list (although you can). When the driver connects to the seeds, it will discover the other members from them. A connection string usually looks something like this:

```
"mongodb://server-1:27017,server-2:27017,server-3:27017"
```

See your driver’s documentation for details.

All MongoDB drivers adhere to the server discovery and monitoring ([SDAM](#)) spec. They persistently monitor the topology of your replica set to detect any changes in your application’s

ability to reach all members of the set. In addition, the drivers monitor the set to maintain information on which member is primary.

The purpose of replica sets is to make your data highly available in the face of network partitions or servers going down. In ordinary circumstances, replica sets respond gracefully to such problems by electing a new primary so that applications can continue to read and write data. If a primary goes down, the driver will automatically find the new primary (once one is elected) and will route requests to it as soon as possible. However, while there is no reachable primary, your application will be unable to perform writes.

There may be no primary available for a brief time (during an election) or for an extended period of time (if no reachable member can become primary). By default, the driver will not service any requests—read or write—during this period. If necessary to your application, you can configure the driver to use secondaries for read requests.

A common desire is to have the driver hide the entire election process (the primary going away and a new primary being elected) from the user. However, this is not possible or desirable in many cases, so no driver handles failover this way. First, a driver can only hide a lack of primary for so long. Second, a driver often finds out that the primary went down because an operation failed, which means that the driver doesn't know whether or not the primary processed the operation before going down. This is a fundamental distributed systems problem that is impossible to avoid. So we need a strategy for dealing with the problem when it emerges. Should we retry the operation on the new primary, if one is elected quickly? Assume it got through on the old primary? Check and see if the new primary has the operation?

The correct strategy, it turns out, is to retry at most one time. Huh? To explain, let's consider our options. These boil down to: don't retry; give up after retrying some fixed number of times; retry at most once. We also need to consider the type of error that could be the source of our problem. There are three types of errors we might see in attempting to write to a replica set: a transient network error; a persistent outage (either network or server); or an error caused by a command the server rejects as incorrect (e.g., not authorized). For each type of error, let's consider our retry options.

For the sake of this discussion, let's look at the example of a write to simply increment a counter. If our application attempts to increment our counter, but gets no response from the server, we don't know whether the server received the message and performed the update. So, if we follow a strategy of not retrying this write, for a transient network error, we might undercount. For a persistent outage or a command error. Not retrying is the correct strategy, because no amount of retrying the write operation will have the desired effect.

If we follow a strategy of retrying some fixed number of times, for transient network errors, we might overcount (in the case where our first attempt succeeded). For a persistent outage or command error, retrying multiple times will simply waste cycles.

Let's look now at the strategy of retrying just once. For a transient network error, we might overcount. For a persistent outage or command error, this is the correct strategy. However, what if we could ensure that our operations are idempotent. Idempotent operations are those that have the same outcome whether do them once or multiple times. With idempotent operations, retrying network errors once has the best chance of correctly dealing with all three types of errors.

As of MongoDB 3.6, the server and all MongoDB drivers support a retriable writes option. See your driver's document for details on how to use this option. With retriable rights, the drivers will automatically follow the retry-at-most-once strategy. Command errors will be returned to the application for client-side handling. Network errors will be retried once after an appropriate delay that shoud accommodate a primary election under ordinary circumstances. With retryable writes turned on, the server maintains a unique identifier for each writer operation and can therefore determine when the driver is attempting to retry a command that already succeeded. Rather than apply the write again, it will simply return a message indicating the write succeeded and thereby overcome the problem caused by the transient network issue.

## **Waiting for Replication on Writes**

Depending on the requirements of your application, you might want to require that all writes are replicated to a majority of the replica set before they are acknowledged by the server. In the rare circumstance where the primary of a set goes down and the newly elected primary (formerly a secondary) did not replicate the very last writes to the former primary, those writes will be rolled back when the former primary comes back up. They can be recovered, but it requires manual intervention. For many applications having a small number of writes rolled back is not a problem. In a blog application, for example, there is little real danger in rolling back a single comment from one reader.

However, for other applications, rollback of any writes should be avoided. Suppose your application sends a write to the primary. It receives confirmation that the write was written, but the primary crashes before any secondaries have had a chance to replicate that write. Now your application thinks that it'll be able to access that write and the current members of the replica set don't have a copy of it.

At some point, a secondary may be elected primary and start taking new writes. When the former primary comes back up, it will discover that it has writes that the current primary does

not. To correct this, it will undo any writes that do not match the sequence of operations on the current primary. These operations are not lost, but they are written to special rollback files that have to be manually applied to the current primary. MongoDB cannot automatically apply these writes, since they may conflict with other writes that have happened since the crash. Thus, the write essentially disappears until an admin gets a chance to apply the rollback files to the current primary. See [Chapter 9](#) for more details on rollbacks.

Writing to a majority prevents this situation: if the application gets a confirmation that the write succeeded, then the new primary would have to have a copy of the write to be elected (a member must be up to date to be elected primary). If the application does not receive acknowledgement from the server or receives an error, then the application would know to try again, given that the write had not been propagated to a majority of the set before the primary crashed.

To ensure that writes will be persisted no matter what happens to the set, you must ensure that the write propagates to a majority of the members of the set. We can achieve this using write concern.

As of MongoDB 2.6, write concern is integrated with write operations. For example, in JavaScript, we can use write concern as follows.

```
try {
    db.products.insertOne(
        { "_id": 10, "item": "envelopes", "qty": 100, type: "Self-Sealing"
         { writeConcern: { "w" : "majority", "wtimeout" : 100 } }
    );
} catch (e) {
    print (e);
}
```

The specific syntax in your driver will vary depending on the programming language, but semantics remain the same. In the example above, we specify a write concern of “majority”. Upon success, the server will respond with a message such as the following.

```
{ "acknowledged" : true, "insertedId" : 10 }
```

But the server will not respond, until this write operation has replicated to a majority of the members of the replica set. Only then, will our application receive acknowledgement that this write succeeded. If the write does not succeed within the timeout we’ve specified, the server will respond with an error message.

```
WriteConcernError({  
    "code" : 64,  
    "errInfo" : {  
        "wtimeout" : true  
    },  
    "errmsg" : "waiting for replication timed out"  
})
```

Write concern majority and the replica set election protocol ensure that in the event of a primary election, only secondaries that are up to date with acknowledged writes will be elected primary. In this way, we ensure that rollback will not happen. With the timeout option, we also have a tunable that enables us to detect and flag any long-running writes at the application layer.

## Other Options for “w”

“majority” is not the only write concern option. MongoDB also lets you specify an arbitrary number of servers to replicate to by passing “w” a number, as below:

```
db.products.insertOne(  
    { "_id": 10, "item": "envelopes", "qty": 100, type: "Self-Sealing" },  
    { writeConcern: { "w" : 2, "wtimeout" : 100 } }  
)
```

This would wait until two members (the primary and one secondary) had the write.

Note that the “w” value includes the primary. If you want the write propagated to n secondaries, you should set “w” to n+1 (to include the primary). Setting “w” : 1 is the same as not passing the “w” option at all because it just checks that the write was successful on the primary.

The downside to using a literal number is that you have to change your application if your replica set configuration changes.

## Custom Replication Guarantees

Writing to a majority of a set is considered “safe.” However, some sets may have more complex requirements: you may want to make sure that a write makes it to at least one server in each data center or a majority of the nonhidden nodes. Replica sets allows you to create custom rules that you can pass to `getLastError` to guarantee replication to whatever combination of servers you need.

### Guaranteeing One Server per Data Center

Network issues between data centers are much more common than within data centers and it is more likely for an entire data center to go dark than an equivalent smattering of servers across multiple data centers. Thus, you might want some data-center-specific logic for writes.

Guaranteeing a write to every data center before confirming success means that, in the case of a write followed by the data center going offline, every other data center will have at least one local copy.

To set this up, first classify the members by data center. You do this by adding a "tags" field to their replica set configuration:

```
> var config = rs.config()
> config.members[0].tags = { "dc" : "us-east" }
> config.members[1].tags = { "dc" : "us-east" }
> config.members[2].tags = { "dc" : "us-east" }
> config.members[3].tags = { "dc" : "us-east" }
> config.members[4].tags = { "dc" : "us-west" }
> config.members[5].tags = { "dc" : "us-west" }
> config.members[6].tags = { "dc" : "us-west" }
```

The "tags" field is an object, as each member can have multiple tags. It might be a "high quality" server in the "us-east" data center, for example, in which case we'd want a tags field such as { "dc": "us-east", "quality" : "high" }.

The second step is to add a rule by creating a "getLastErrorMode" field in our replica set config. The name "getLastErrorMode" is vestigial in the sense that prior to MongoDB 2.6, applications used a method called "getLastError" to specify write concern. In replica configs, for "getLastErrorMode" each rule is of the form "*name* : { "key" : *number* }". "*name*" is the name for the rule, which should describe what the rule does in a way that clients can understand, as they'll be using this name when they call *getLastError*. In this example, we might call this rule "eachDC" or something more abstract such as "user-level safe".

The "key" field is the key field from the tags, so in this example it will be "dc". The *number* is the number groups that are needed to fulfil this rule. In this case, *number* is 2 (because we want at least one server from "us-east" and one from "us-west"). *number* always means "at least one server from each of *number* groups."

We add "getLastErrorModes" to the replica set config and reconfigure to create the rule:

```
> config.settings = {}
> config.settings.getLastErrorModes = [{ "eachDC" : { "dc" : 2 } }]
```

```
> rs.reconfig(config)
```

"getLastErrorModes" lives in the "settings" subobject of a replica set config, which contains a few set-level optional settings.

Now we can use this rule for writes:

```
db.products.insertOne(  
  { "_id": 10, "item": "envelopes", "qty": 100, type: "Self-Sealing" },  
  { writeConcern: { "w" : "eachDC", wtimeout : 1000 } }  
) ;
```

Note that rules are somewhat abstracted away from the application developer: they don't have to know which servers are in "eachDC" to use the rule, and the rule can change without their application having to change. We could add a datacenter or change set members and the application would not have to know.

## Guaranteeing a Majority of Nonhidden Members

Often, hidden members are somewhat second-class citizens: you're never going to fail over to them and they certainly aren't taking any reads. Thus, you may only care that nonhidden members received a write and let the hidden members sort it out for themselves.

Suppose we have five members, *host0* through *host4*, *host4* being a hidden member. We want to make sure that a majority of the nonhidden members have a write, that is, at least three of *host0*, *host1*, *host2*, and *host3*. To create a rule for this, first we tag each of the nonhidden members with its own tag:

```
> var config = rs.config()  
> config.members[0].tags = [ { "normal" : "A" } ]  
> config.members[1].tags = [ { "normal" : "B" } ]  
> config.members[2].tags = [ { "normal" : "C" } ]  
> config.members[3].tags = [ { "normal" : "D" } ]
```

The hidden member, *host4*, is not given a tag.

Now we add a rule for the majority of these servers:

```
> config.settings.getLastErrorModes = [ { "visibleMajority" : { "normal" : 3 } } ]  
> rs.reconfig(config)
```

Finally, you can use this rule in your application:

```
db.products.insertOne(  
  { "_id": 10, "item": "envelopes", "qty": 100, type: "Self-Sealing" },  
  { writeConcern: { "w" : "visibleMajority", wtimeout : 1000 } }  
) ;
```

This will wait until at least three of the nonhidden member have the write.

## Creating Other Guarantees

The rules you can create are limitless. Remember that there are two steps to creating a custom replication rule:

1. Tag members by assigning them key-value pairs. The keys describe classifications; for example, you might have keys such as "data\_center" or "region" or "serverQuality". Values determine which group a server belongs to within a classification. For example, for the key "data\_center", you might have some servers tagged "us-east", some "us-west", and others "aust".
2. Create a rule based on the classifications you create. Rules are always of the form `{"name" : {"key" : number}}`, where at least one server from `number` groups must have a write before it has succeeded. For example, you could create a rule `{"twoDCs" : {"data_center" : 2}}`, which would mean that at least one server in two of the data centers tagged must confirm a write before it is successful.

Then you can use this rule in `getLastError`.

Rules are immensely powerful ways to configure replication, although they are complex to understand and set up. Unless you have fairly involved replication requirements, you should be perfectly safe sticking with `"w" : "majority"`.

## Sending Reads to Secondaries

By default, drivers will route all requests to the primary. This is generally what you want, but you can configure other options by setting read preferences in your driver. Read preferences let you specify the types of servers queries should be sent to.

Sending read requests to secondaries is generally a bad idea. There are some specific situations in which it makes sense, but you should generally send all traffic to the primary. If you are considering sending reads to secondaries, make sure to weigh the pros and cons very carefully

before allowing it. This section covers why it's a bad idea and the specific conditions when it makes sense to do so.

## Consistency Considerations

Applications that require strongly consistent reads should not read from secondaries.

Secondaries should usually be within a few milliseconds of the primary. However, there is no guarantee of this. Sometimes secondaries can fall behind by minutes, hours, or even days due to load, misconfiguration, network errors, or other issues. Client libraries cannot tell how up to date a secondary is, so clients will cheerfully send queries to secondaries that are far behind. Hiding a secondary from client reads can be done but is a manual process. Thus, if your application needs data that is predictably up to date, it should not read from secondaries.

If your application needs to read its own writes (e.g., insert a document and then query for it and find it) you should not send the read to a secondary (unless the write waits for replication to all secondaries using "w" as shown earlier). Otherwise, an application may perform a successful write, attempt to read the value, and not be able to find it (because it sent the read to a secondary, which hadn't replicated yet). Clients can issue requests faster than replication can copy operations.

To always send read requests to the primary, set your read preference to Primary (or leave it alone, since Primary is the default). If there is no primary, queries will error out. This means that your application cannot perform queries if the primary goes down. However, it is certainly an acceptable option if your application can deal with downtime during failovers or network partitions or if getting stale data is unacceptable.

## Load Considerations

Many users send reads to secondaries to distribute load. For example, if your servers can only handle 10,000 queries a second and you need to handle 30,000, you might set up a couple of secondaries and have them take some of the load. However, this is a dangerous way to scale because it's easy to accidentally overload your system and difficult to recover from once you do.

For example, suppose that you have the situation above: 30,000 reads per second. You decide to create a replica set with four members to handle this: each secondary is well below its maximum load and the system works perfectly.

Until one of the secondaries crashes.

Now each of the remaining members are handling 100% of their possible load. If you need to

rebuild the member that crashed, it may need to copy data from one of the other servers, overwhelming the remaining servers. Overloading a server often makes it perform slower, lowering the set's capacity even further and forcing other members to take more load, causing them to slow down in a death spiral.

Overloading can also cause replication to slow down, making the remaining secondaries fall behind. Suddenly you have a member down, a member lagging, and everything is too overloaded to have any wiggle room.

If you have a good idea of how much load a server can take, you might feel like you can plan this out better: use five servers instead of four and the set won't be overloaded if one goes down. However, even if you plan it out perfectly (and only lose the number of servers you expected), you still have to fix the situation with the other servers under more stress than they would be otherwise.

A better choice is to use sharding to distribute load. We'll cover how to set sharding up in [Chapter 12](#).

## Reasons to Read from Secondaries

There are a few cases in which it's reasonable to send application reads to secondaries. For instance, you may want your application to still be able to perform reads if the primary goes down (and you do not care if those reads are somewhat stale). This is the most common case for distributing reads to secondaries: you'd like a temporary read-only mode when your set loses a primary. This read preference is called `Primary preferred`.

One common argument for reading from secondaries is to get low-latency reads. You can specify `Nearest` as your read preference to route requests to the lowest-latency member based on average ping time from the driver to the replica set member. If your application needs to access the same document with low latency in multiple data centers, this is the only way to do it. If, however, your documents are more location-based (application servers in this data center need low-latency access to some of your data, or application servers in another data center need low-latency access to other data), this should be done with sharding. Note that you must use sharding if your application requires low-latency reads *and* low-latency writes: replica sets only allow writes to one location (wherever the primary is).

You must be willing to sacrifice consistency if you are reading from members that may not have replicated all the writes yet. Alternatively, you could sacrifice write speed if you wanted to wait until writes had been replicated to all members.

If your application can truly function acceptably with arbitrarily stale data, you can use

`Secondary` or `Secondary preferred` read preferences. `Secondary` will always send read requests to a secondary. If there are no secondaries available, this will error out rather than send reads to the primary. It can be used for applications that do not care about stale data and want to use the primary for writes only. If you have any concerns about staleness of data, this is not recommended.

`Secondary preferred` will send read requests to a secondary, if one is available. If no secondaries are available, requests will be sent to the primary.

Sometimes, read load is drastically different than write load: you're reading entirely different data than you're writing. You might want dozens of indexes for offline processing that you don't want to have on the primary. In this case, you might want to set up a secondary with different indexes than the primary. If you'd like to use a secondary for this purpose, you'd probably create a connection directly to it from the driver, instead of using a replica set connection.

Consider which of the options makes sense for your application. You can also combine options: if some read requests must be from the primary, use `Primary` for those. If you are OK with other reads not having the most up-to-date data, use `Primary preferred` for those. And if certain requests require low latency over consistency, use `Nearest` for those.

# Chapter 11. Administration

---

This chapter covers replica set administration, including:

- Techniques for performing maintenance on individual members
- Configuring sets under a variety of circumstances
- Getting information about and resizing your oplog
- Doing some more exotic set configurations
- Converting from master-slave to a replica set

## Starting Members in Standalone Mode

A lot of maintenance tasks cannot be performed on secondaries (because they involve writes) and shouldn't be performed on primaries because of the impact this could have on application performance. Thus, the following sections frequently mention starting up a server in standalone mode. This means restarting the member so that it is a standalone server, not a member of a replica set (temporarily).

To start up a member in standalone mode, first look at the command line argument. Suppose it looks something like this:

---

```
> db.serverCmdLineOpts()
{
    "argv" : [ "mongod", "-f", "/var/lib/mongod.conf" ],
    "parsed" : {
        "replicaSet": "mySet",
        "port": "27017",
        "dbpath": "/var/lib/db"
    },
    "ok" : 1
}
```

---

To perform maintenance on this server we can restart it without the `replSet` option. This will allow us to read and write to it as a normal standalone `mongod`. We don't want the other servers in the set to be able to contact it, so we'll make it listen on a different port (so that the other members won't be able to find it). Finally, we want to keep the `dbpath` the same, as we are presumably starting it up this way to manipulate the server's data somehow.

First, we shutdown the server from the mongo shell.

```
> db.shutdownServer()
```

Then, in an operating system shell (e.g., bash), restart mongod on another port and without the `replSet` parameter.

```
$ mongod --port 30000 --dbpath /var/lib/db
```

It will now be running as a standalone server, listening on port 30000 for connections. The other members of the set will attempt to connect to it on port 27017 and assume that it is down.

When we have finished performing maintenance on the server, we can shut it down and restart it with its original options. It will automatically sync up with the rest of the set, replicating any operations that it missed while it was “away.”

## Replica Set Configuration

Replica set configuration is always kept in a document in the `local.system.replset` collection. This document is the same on all members of the set. Never update this document using `update`. Always use an `rs` helper or the `replSetReconfig` command.

### Creating a Replica Set

You create a replica set by starting up the `mongods` that you want to be members and then passing one of them a configuration through `rs.initiate`:

```
> var config = {
... "_id" : setName,
... "members" : [
...     {"_id" : 0, "host" : host1},
...     {"_id" : 1, "host" : host2},
...     {"_id" : 2, "host" : host3}
... ]
> rs.initiate(config)
```

You should always pass a config object to `rs.initiate`. If you do not, MongoDB will attempt to automatically generate a config for a one-member replica set. It might not use the hostname that you want or correctly configure the set.

You only call `rs.initiate` on one member of the set. The member that receives the initiate will pass the configuration on to the other members.

## Changing Set Members

When you add a new set member, it should either have nothing in its data directory (in which case it will initial sync) or have a copy of the data from another member.

Connect to the primary and add the new member:

```
> rs.add("spock:27017")
```

Alternatively, you can specify a more complex member config as a document:

```
> rs.add({_id : 5, "host" : "spock:27017", "priority" : 0, "hidden" : true})
```

You can also remove members by their "host" field:

```
> rs.remove("spock:27017")
```

You can change a member's settings by reconfiguring. There are a few restrictions in changing a member's settings:

- You cannot change a member's `"_id"`.
- You cannot make the member you're sending the reconfig to (generally the primary) priority 0.
- You cannot turn an arbiter into a nonarbiter and visa versa.
- You cannot change a member with `"buildIndexes" : false` to `"buildIndexes" : true`.

Notably, you *can* change a member's "host" field. Thus, if you incorrectly specify a host (say, use a public IP instead of a private one) you can later go back and simply change the

config to use the correct IP.

To change a hostname, you could do something like this:

```
> var config = rs.config()
> config.members[0].host = "spock:27017"
spock:27017
> rs.reconfig(config)
```

This same strategy applies to change any other option: fetch the config with `rs.config()`, modify any parts of it that you wish, and reconfigure the set by passing `rs.reconfig()` the new configuration.

## Creating Larger Sets

Replica sets are limited to 50 members and only 7 voting members. This is to reduce the amount of network traffic required for everyone to heartbeat everyone else and to limit the amount of time elections take.

If you are creating a replica set that has more than 7 members, every additional member must be given 0 votes. You can do this by specifying it in the member's config:

```
> rs.add({_id : 7, host : "server-7:27017", votes : 0})
```

This prevents these members from casting positive votes in elections.

## Forcing Reconfiguration

When you permanently lose a majority of the set, you may want to reconfigure the set while it doesn't have a primary. This is a little tricky: usually you'd send the reconfig to the primary. In this case, you can force reconfigure the set by sending a reconfig command to a secondary.

Connect to a secondary in the shell and pass it a reconfig with the "force" option:

```
> rs.reconfig(config, {force : true})
```

Forced reconfigurations follow the same rules as a normal reconfiguration: you must send a valid, well-formed configuration with the correct options. The "force" option doesn't allow invalid configs; it just allows a secondary to accept a reconfig.

Forced reconfigurations bump the replica set "version" number by a large amount. You may see it jump by tens or hundreds of thousands. This is normal: it is to prevent version number

collisions (just in case there's a reconfig on either side of a network partition).

When the secondary receives the reconfig, it will update its configuration and pass the new config along to the other members. The other members of the set will only pick up on a change of config if they recognize the sending server as a member of their current config. Thus, if some of your members have changed hostnames, you should force reconfig from a member that kept its old hostname. If everyone has a new hostname, you should shut down each member of the set, start it up in standalone mode, change its *local.system.replset* document manually, and then restart the member.

## Manipulating Member State

There are several ways to manually change member state for maintenance or in response to load. Note that there is no way to force a member to become primary other than configuring the set appropriately, in this case, by giving the replica set member a priority higher than any other member of the set.

### Turning Primaries into Secondaries

You can demote a primary to a secondary using the `stepDown` function:

```
> rs.stepDown()
```

This makes the primary step down into secondary state for 60 seconds. If no other primary has been elected in that time period, it will be able to attempt a reelection. If you would like it to remain a secondary for a longer or shorter amount of time, you can specify your own number of seconds for it to stay in SECONDARY state:

```
> rs.stepDown(600) // 10 minutes
```

### Preventing Elections

If you need to do some maintenance on the primary but don't want any of the other eligible members to become primary in the interim, you can force them to stay secondaries by running `freeze` on each of them:

```
> rs.freeze(10000)
```

Again, this takes a number of seconds to remain secondary.

When you have finished whatever maintenance you are doing and want to “unfreeze” the other members you can run the command again, giving a timeout of 0 seconds:

```
> rs.freeze(0)
```

This will allow the member to hold an election, if it chooses.

You can also unfreeze primaries that have been stepped down by running `rs.freeze(0)`.

## Monitoring Replication

It is important to be able to monitor the status of a set: not only that everyone is up, but what states they are in and how up-to-date the replication is. There are several commands you can use to see replica set information. MongoDB hosting services and management tools including: Atlas, Cloud Manager, and Ops Manager also keeps some useful stats on replication.

Often issues with replication are transient: a server could not reach another server but now it can. The easiest way to see issues like this is to look at the logs. Make sure you know where the logs are being stored (and that they *are* being stored) and that you can access them.

### Getting the Status

One of the most useful commands you can run is `replSetGetStatus`, which gets the current information about every member of the set (from the view of the member you’re running it on). There is a helper for this command in the shell:

```
> rs.status()

{
  "set" : "replica",
  "date" : ISODate("2016-11-02T20:02:16.543Z"),
  "myState" : 1,
  "term" : NumberLong(1),
  "heartbeatIntervalMillis" : NumberLong(2000),
  "optimes" : {
    "lastCommittedOpTime" : {
      "ts" : Timestamp(1478116934, 1),
      "t" : NumberLong(1)
    },
    "readConcernMajorityOpTime" : {
      "ts" : Timestamp(1478116934, 1),
      "t" : NumberLong(1)
    },
    "appliedOpTime" : {
      "ts" : Timestamp(1478116934, 1),
      "t" : NumberLong(1)
    }
}
```

```

} ,
"durabilityOpTime" : {
    "ts" : Timestamp(1478116934, 1),
    "t" : NumberLong(1)
}
} ,

"members" : [
{
    "_id" : 0,
    "name" : "m1.example.net:27017",
    "health" : 1,
    "state" : 1,
    "stateStr" : "PRIMARY",
    "uptime" : 269,
    "optime" : {
        "ts" : Timestamp(1478116934, 1),
        "t" : NumberLong(1)
    },
    "optimeDate" : ISODate("2016-11-02T20:02:14Z"),
    "infoMessage" : "could not find member to sync from",
    "electionTime" : Timestamp(1478116933, 1),
    "electionDate" : ISODate("2016-11-02T20:02:13Z"),
    "configVersion" : 1,
    "self" : true
},
{
    "_id" : 1,
    "name" : "m2.example.net:27017",
    "health" : 1,
    "state" : 2,
    "stateStr" : "SECONDARY",
    "uptime" : 14,
    "optime" : {
        "ts" : Timestamp(1478116934, 1),
        "t" : NumberLong(1)
    },
    "optimeDurable" : {
        "ts" : Timestamp(1478116934, 1),
        "t" : NumberLong(1)
    },
    "optimeDate" : ISODate("2016-11-02T20:02:14Z"),
    "optimeDurableDate" : ISODate("2016-11-02T20:02:14Z"),
    "lastHeartbeat" : ISODate("2016-11-02T20:02:15.618Z"),
    "lastHeartbeatRecv" : ISODate("2016-11-02T20:02:14.866Z"),
    "pingMs" : NumberLong(0),
    "syncingTo" : "m3.example.net:27017",
    "configVersion" : 1
},
{
    "_id" : 2,
    "name" : "m3.example.net:27017",
    "health" : 1,
    "state" : 2,

```

```

        "stateStr" : "SECONDARY",
        "uptime" : 14,
        "optime" : {
            "ts" : Timestamp(1478116934, 1),
            "t" : NumberLong(1)
        },
        "optimeDurable" : {
            "ts" : Timestamp(1478116934, 1),
            "t" : NumberLong(1)
        },
        "optimeDate" : ISODate("2016-11-02T20:02:14Z"),
        "optimeDurableDate" : ISODate("2016-11-02T20:02:14Z"),
        "lastHeartbeat" : ISODate("2016-11-02T20:02:15.619Z"),
        "lastHeartbeatRecv" : ISODate("2016-11-02T20:02:14.787Z"),
        "pingMs" : NumberLong(0),
        "syncingTo" : "m1.example.net:27018",
        "configVersion" : 1
    }
],
"ok" : 1
}

```

---

These are some of the most useful fields:

`self`

This field is only present in the member `rs.status()` was run on, in this case, *server-2*.

`stateStr`

A string describing the state of the server. See “Member States” to see descriptions of the various states.

`uptime`

The number of seconds a member has been reachable (or the time since this server was started for the “`self`” member). Thus, *server-2* has been up for 161989 seconds, or about 45 hours. *server-1* has been available for the last 21 hours and *server-3* has been available for the last 7 hours.

`optimeDate`

The last optime in each member’s oplog (where that member is synced to). Note that this is the state of each member as reported by the heartbeat, so the optime reported here may be off by a couple of seconds.

lastHeartbeat

The time this server last received a heartbeat from the member. If there have been network issues or the server has been busy, this may be longer than two seconds ago.

pingMs

The running average of how long heartbeats to this server have taken. This is used in determining which member to sync from.

errmsg

Any status message that the member chose to return in the heartbeat request. These are often merely informational, not error messages. For example, the "errmsg" field in *server-3* indicates that this server is in the process of initial syncing. The hexadecimal number 507e9a30:851 is the timestamp of the operation this member needs to get to to complete the initial sync.

There are several fields that give overlapping information: "state" is the same as "stateStr", it's simply the internal id for the state. "health" merely reflects whether a given server is reachable (1) or unreachable (0), which is also shown by "state" and "stateStr" (they'll be UNKNOWN or DOWN if the server is unreachable). Similarly, "optime" and "optimeDate" are the same value represented in two ways: one represents milliseconds since the epoch ("t" : 135...) and the other is a more human-readable date.

Note that this report is from the point of view of whichever member of the set you run it on: it may be incorrect or out of date due to network issues.

## Visualizing the Replication Graph

If you run `rs.status()` on a secondary, there will be a top-level field called "syncingTo". This gives the host that this member is replicating from. By running the `replSetGetStatus` command on each member of the set, you can figure out the replication graph. For example, assuming `server1` was a connection to *server1*, `server2` was a connection to *server2*, and so on, you might have something like:

```
> server1.adminCommand({replSetGetStatus: 1})['syncingTo']
server0:27017
> server2.adminCommand({replSetGetStatus: 1})['syncingTo']
server1:27017
> server3.adminCommand({replSetGetStatus: 1})['syncingTo']
server1:27017
```

```
> server4.adminCommand({replSetGetStatus: 1})['syncingTo']
server2:27017
```

Thus, *server0* is the source for *server1*, *server1* is the replication source for *server2* and *server3*, and *server2* is the replication source for *server4*.

MongoDB determines who to sync to based on ping time. When one member heartbeats another, it times how long that request took. MongoDB keeps a running average of these times. When it has to choose a member to sync from, it looks for the member that is closest to it and ahead of it in replication (thus, you cannot end up with a replication cycle: members will only replicate from the primary or secondaries that are strictly further ahead).

Thus, if you bring up a new member in a secondary data center, it is more likely to sync from the other members in that data center than a member in your primary data center (thus minimizing WAN traffic), as shown in Figure 11-1.

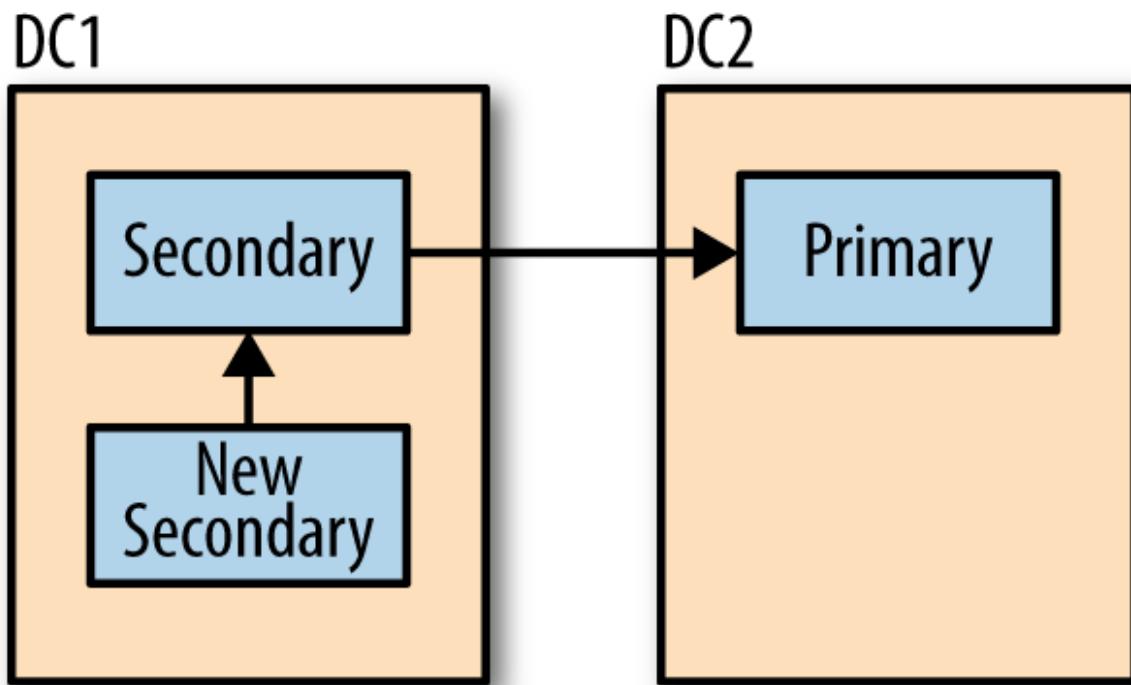
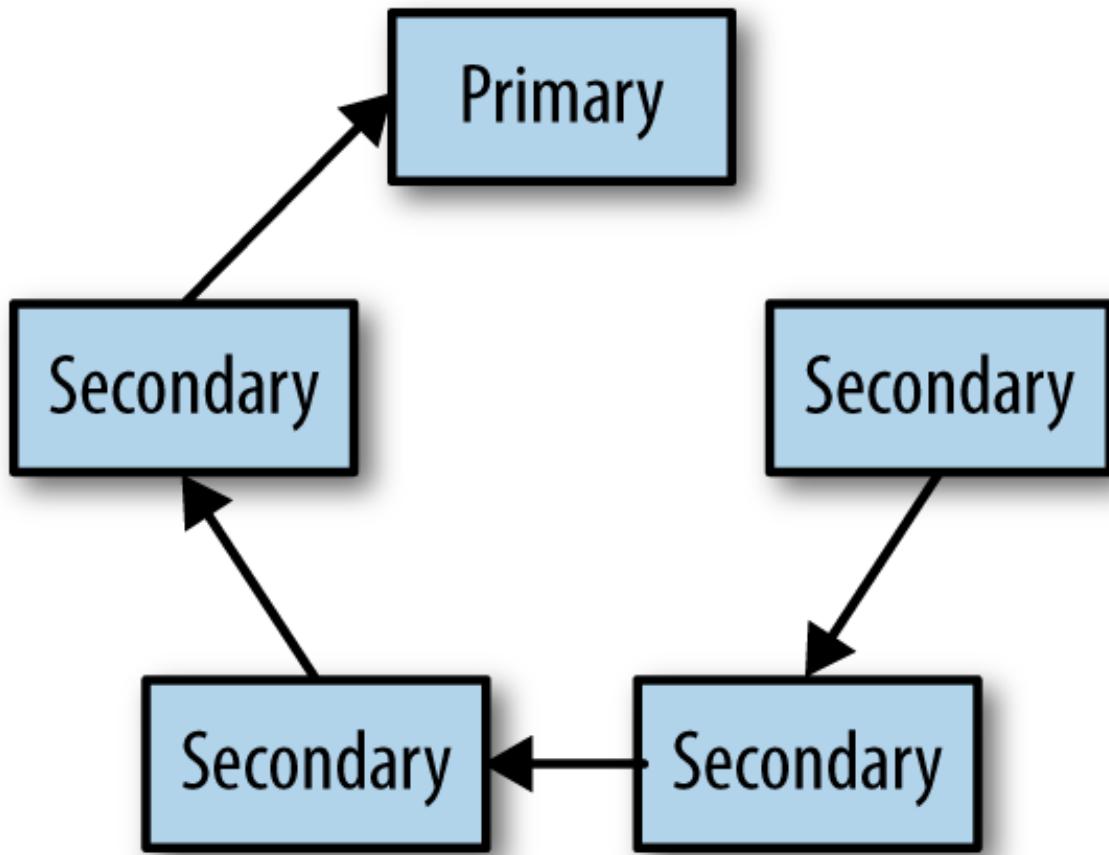


Figure 11-1. New secondaries will generally choose to sync from a member in the same data center

However, there are some downsides to automatic replication chaining: more replication hops mean that it takes a bit longer to replicate writes to all servers. For example, let's say that everything is in one data center but, due to the vagaries of network speeds when you added members, MongoDB ends up replicating in a line, as shown in Figure 11-2.



*Figure 11-2. As replication chains get longer, it takes longer for all members to get a copy of the data*

This is highly unlikely, but not impossible. It is, however, probably undesirable: each secondary in the chain will have to be a bit further behind than the secondary “in front” of it. You can fix this by modifying the replication source for a member using the `replSetSyncFrom` command (or the `rs.syncFrom()` helper).

Connect to the secondary whose replication source you want to change and run this command, passing it the server you’d prefer this member to sync from:

```
> secondary.adminCommand({ "replSetSyncFrom" : "server0:27017" })
```

It may take a few seconds to switch sync sources, but if you run `rs.status()` on that member again, you should see that the “`syncingTo`” field now says “`server0:27017`”.

At this point, `server4` will continue replicating from `server0` until `server0` becomes unavailable or, if it happened to be a secondary, falls significantly behind the other members.

## Replication Loops

A replication loop is when members end up replicating from one another, for example, *A* is

syncing from *B* who is syncing from *C* who is syncing from *A*. As none of the members in a replication loop can be a primary, the members will not receive any new operations to replicate and fall behind. On the plus side, replication loops should be impossible when members choose who to sync from automatically.

However, you can force replication loops using the *replSetSyncFrom* command. Inspect the `rs.status()` output carefully before manually changing sync targets and be careful not to create loops. The *replSetSyncFrom* command will warn you if you do not choose to sync from a member who is strictly ahead, but it will allow it.

## Disabling Chaining

Chaining is when a secondary syncs from another secondary (instead of the primary). As mentioned earlier, members may decide to sync from other members automatically. You can disable chaining, forcing everyone to sync from the primary, by changing the "chainingAllowed" setting to `false` (if not specified, it defaults to `true`):

```
> var config = rs.config()
> // create the settings subobject, if it does not already exist
> config.settings = config.settings || {}
> config.settings.chainingAllowed = false
> rs.reconfig(config)
```

With `chainingAllowed` set to `false`, all members will sync from the primary. If the primary becomes unavailable, they will fall back to syncing from secondaries.

## Calculating Lag

One of the most important metrics to track for replication is how well the secondaries are keeping up with the primary. Lag is how far behind a secondary is, which means the difference in timestamp between the last operation the primary has performed and the timestamp of the last operation the secondary has applied.

You can use `rs.status()` to see a member's replication state, but you can also get a quick summary (along with oplog size) by running `rs.printReplicationInfo()` or `rs.printSlaveReplicationInfo()`.

`rs.printReplicationInfo` gives a summary of the primary's oplog:

```
> rs.printReplicationInfo();
  configured oplog size: 10.48576MB
  log length start to end: 34secs (0.01hrs)
```

```
oplog first event time: Tue Apr 10 2018 09:27:57 GMT-0400 (EDT)
oplog last event time: Tue Apr 10 2018 10:27:47 GMT-0400 (EDT)
now:                  Tue Apr 10 2018 10:27:47 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 10 MB and is only able to fit about an hour of operations.

If this were a real deployment, the oplog should probably be larger (see “[Resizing the Oplog](#)” for instructions on changing oplog size). 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 secondary falls off the end of the oplog before finishing its initial sync.

#### NOTE

The log length is computed by taking the time difference between the first and last operation in the oplog once the oplog has filled up. If the server has just started with nothing in the oplog, then the earliest 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 length is a more useful metric for servers that have been operating long enough to write through their entire oplog at least once.

You can also use the `rs.printSlaveReplicationInfo()` function to get the `syncedTo` value for each member and the time when the last oplog entry was written to each secondary, as shown in the following example:

```
> rs.printSlaveReplicationInfo();
source: m1.example.net:27017
    syncedTo: Tue Apr 10 2018 10:27:47 GMT-0400 (EDT)
    0 secs (0 hrs) behind the primary
source: m2.example.net:27017
    syncedTo: Tue Apr 10 2018 10:27:43 GMT-0400 (EDT)
    0 secs (0 hrs) behind the primary
source: m3.example.net:27017
    syncedTo: Tue Apr 10 2018 10:27:39 GMT-0400 (EDT)
    0 secs (0 hrs) behind the primary
```

Remember that a replica set member’s lag is calculated relative to the primary, not against “wall time.” This usually is irrelevant, but on very low-write systems, this can cause phantom replication lag “spikes.” For example, suppose you do a write once an hour. Right after that

write, before it's replicated, the secondary will look like it's an hour behind the primary. However, it'll be able to catch up with that "hour" of operations in a few milliseconds. This can sometimes cause confusion when monitoring a low-throughput system.

## Resizing the Olog

Your primary's oplog should be thought of as your maintenance window. If your primary has an oplog that is an hour long, then you only have one hour to fix anything that goes wrong before your secondaries fall too far behind and must be resynced from scratch. Thus, you generally want to have an oplog that can hold a couple days to a week's worth of data, to give yourself some breathing room if something goes wrong.

Unfortunately, there's no easy way to tell how long your oplog is going to be before it fills up and there's no way to resize it while your server is running. However, it is possible to cycle through your servers, taking each one offline, making its oplog larger, and then adding it back into the set. Remember that each server that could become a primary should have a large enough oplog to give you a sane maintenance window.

To increase the size of your oplog, perform the following steps:

1. If this is currently the primary, step it down and wait for the other servers to catch up.
2. Shut down the server.
3. Start it up as a standalone server.
4. Temporarily store the last insert in the oplog in another collection:

```
> use local
> // op: "i" finds the last insert
> var cursor = db.oplog.rs.find({"op" : "i"})
> var lastInsert = cursor.sort({"$natural" : -1}).limit(1).next()
> db.tempLastOp.save(lastInsert)
>
> // make sure it was saved! It's very important that you don't lose this
> db.tempLastOp.findOne()
```

We could use the last update or delete, but \$-operators cannot be inserted into a collection.

5. Drop the current oplog:

```
> db.oplog.rs.drop()
```

6. Create a new oplog:

```
> db.createCollection("oplog.rs", {"capped" : true, "size" : 10000})
```

7. Put the last op back in the oplog:

```
> var temp = db.tempLastOp.findOne()
> db.oplog.rs.insert(temp)
>
> // make sure that this was actually inserted
> db.oplog.rs.findOne()
```

Make sure that the last op was inserted into the oplog. If it was not, the server will drop all of its data and resync when you add it back into the set.

8. Finally, restart the server as a member of the replica set. Remember that it only has one op in the oplog to start out with, so you won't be able to see its true oplog length (how long it is in time) for a while. Also, it won't be a very good sync source if other members are behind.

You generally should not decrease the size of your oplog: although it may be months long, there is usually ample disk space for it and it does not use up any valuable resources like RAM or CPU.

## Building Indexes

If you send an index build to the primary, the primary will build the index normally and then the secondaries will build the index when they replicate the “build index” operation. Although this is the easiest way to build an index, index builds are resource-intensive operations that can make members unavailable. If all of your secondaries start building an index at the same time, almost every member of your set will be offline until the index build completes.

Therefore, you may want to build an index on one member at a time to minimize impact on your application. To accomplish this, do the following:

1. Shut down a secondary.
2. Restart it as a standalone server.
3. Build the index on the standalone server.
4. When the index build is complete, restart the server as a member of the replica set.

5. Repeat steps 1 through 4 for each secondary in the replica set.

You should now have a set where every member other than the primary has the index built. Now there are two options, and you should choose the one that will impact your production system the least:

1. Build the index on the primary. If you have an “off” time when you have less traffic, that would probably be a good time to build it. You also might want to modify read preferences to temporarily shunt more load onto secondaries while the build is in progress.

The primary will replicate the index build to the secondaries, but they will already have the index so it will be a no-op for them.

2. Step down the primary, then follow steps 1 through 4 as outlined earlier. This requires a failover, but you will have a normally-functioning primary while the old primary is building its index. After its index build is complete, you can reintroduce it to the set.

Note that you could also use this technique to build different indexes on a secondary than you have on the rest of the set. This could be useful for offline processing, but make sure a member with different indexes can never become primary: its priority should always be 0.

If you are building a unique index, make sure that the primary is not inserting duplicates or that you build the index on the primary first. Otherwise, the primary could be inserting duplicates that would then cause replication errors on secondaries. If this occurs, the secondary will shut itself down. You will have to restart it as a stand alone, remove the unique index, and restart it.

## Replication on a Budget

If it is difficult get more than one high-quality server, consider getting a secondary server that is strictly for disaster recovery, with less RAM, CPU, slower disk IO, etc. The good server will always be your primary and the cheaper server will never handle any client traffic (configure your clients to send all reads to the primary). Here are all the options to set for the cheaper box:

```
"priority" : 0
```

You do not want this server to ever become primary.

```
"hidden" : true
```

You do not want clients ever sending reads to this secondary.

```
"buildIndexes" : false
```

This is optional, but it can decrease the load this server has to handle considerably. If you ever need to restore from this server, you'll need to rebuild indexes.

```
"votes" : 0
```

If you only have two machines, set the votes on this secondary to 0 so that the primary can stay primary if this machine goes down. If you have a third server (even just your application server), run an arbiter on that instead of setting votes to 0.

This will give you the safety and security of having a secondary without having to invest in two high-performance servers.

## **Part IV. Sharding**

---

# Chapter 12. Introduction to Sharding

---

This chapter covers how to scale with MongoDB:

- What sharding is and the components of a cluster
- How to configure sharding
- The basics of how sharding interacts with your application

## Introduction to Sharding

Sharding refers to the process of splitting data up across machines; the term partitioning is also sometimes used to describe this concept. By putting a subset of data on each machine, it becomes possible to store more data and handle more load without requiring larger or more powerful machines, just a larger quantity of less-powerful machines. Sharding may be used for other purposes as well, including splitting a dataset based on geography to locate a subset of documents in a collection (e.g., users based in a particular locale) close to the application servers from which they are most commonly accessed. Other uses of sharding include partitioning data for other purposes such as placing more frequently accessed data on more performant hardware.

Manual sharding can be done with almost any database software. Manual sharding is when an application maintains connections to several different database servers, each of which are completely independent. The application 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 tries to both abstract the architecture away from the application and simplify the administration of such a system. MongoDB allows your application to ignore the fact that it isn't talking to a standalone MongoDB server, to some extent. On the operations side, MongoDB automates balancing data across shards and makes it easier to add and remove capacity.

Sharding is the most complex way of configuring MongoDB, both from a development and operational point of view. There are many components to configure and monitor and data moves around the cluster automatically. You should be comfortable with standalone servers and replica sets before attempting to deploy or use a sharded cluster. Also, as with replica sets, the recommended means of configuring and deploying sharded clusters is using Ops Manager, if you need maintain control of your computing infrastructure or MongoDB Atlas, if you can leave the infrastructure management to MongoDB (you have the option of running in Amazon AWS, Microsoft Azure, or Google Compute Cloud).

## Understanding the Components of a Cluster

MongoDB's sharding allows you to create a cluster of many machines (shards) and break up a collection across them, putting a subset of data on each shard. This allows your application to grow beyond the resource limits of a standalone server or replica set.

### NOTE

Many people are confused about the difference between replication and sharding. Remember that replication creates an exact copy of your data on multiple servers, so every server is a mirror-image of every other server. Conversely, every shard contains a different subset of data.

One of the goals of sharding is to make a cluster of 2, 3, 10, or even hundreds of shards look like a single machine to your application. To hide these details from the application, we run one or more routing processes called a `mongos` in front of the shards. A `mongos` keeps a “table of contents” that tells it which shard contains which data. Applications can connect to this router and issue requests normally, as shown in Figure 12-1. 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, if necessary, merges them, and sends them back to the application. As far as the application knows, it’s connected to a stand-alone `mongod`, as in Figure 12-2.

## A One-Minute Test Setup

As in the replication section, we will start by setting up a quick cluster on a single machine. First, start a mongo shell with the `--nodb` option:

```
mongo --nodb
```

To create a cluster, use the `ShardingTest` class: Run the following in the mongo shell you just launched.

```
st = ShardingTest({
  name:"one-min-shards",
  chunkSize:1,
  shards:2,
  rs:{
    nodes:3,
    oplogSize:10
  },
  other:{
    enableBalancer:true
  }
});
```

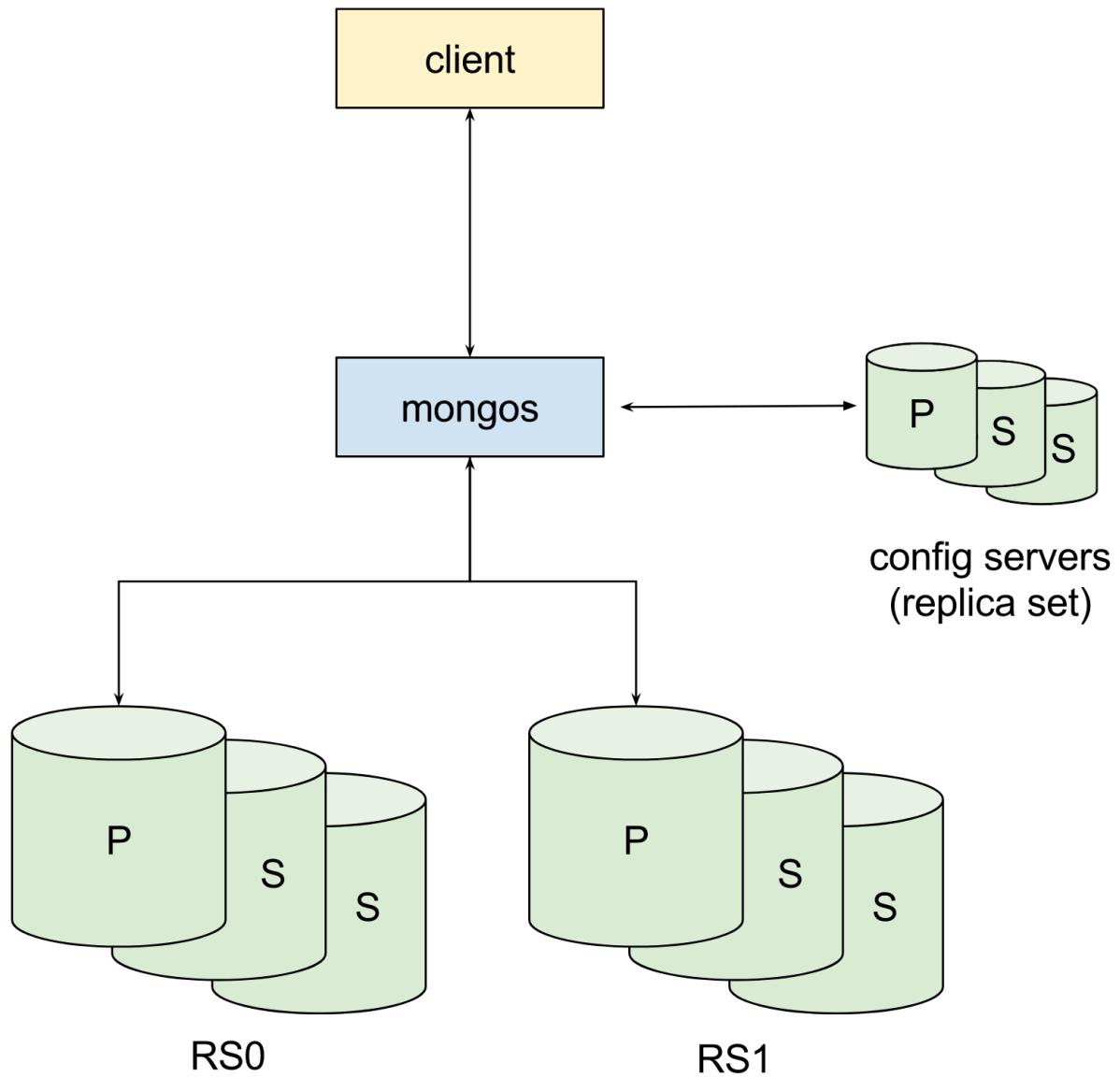
The `chunksize` option is covered in [Chapter 15](#). For now, simply set it to 1. As for the other options we just passed to `ShardingTest`: `name` simply provides a label for our sharded cluster; `shards` specifies that our cluster will be composed of two shards (we do this to keep the resource requirements low for this example); `rs` defines each shard as a three-node replica set with an `oplogSize` of 10 MiB (again, to keep resource utilization low). Though it is possible to run a one standalone mongod for each shard, it paints a clearer picture of the typical architecture of a sharded cluster if we create each shard as a replica set. In the last option specified, we are instructing `ShardingTest` to enable the balancer once the cluster is spun up. This will ensure that data is evenly distributed across both shards.

`ShardingTest` is a class designed for internal use by MongoDB Engineering and is therefore undocumented externally. However, because it ships with MongoDB server, it provides the most straightforward means of experimenting with a sharded cluster. `ShardingTest` was originally designed to support server test suites and is still used for this purpose. By default it provides a number of conveniences that help in keeping resource utilization as small as possible and in setting up the relatively complex architecture of a sharded cluster.

In running the command specified above, `ShardingTest` will do a lot for us automatically. It will create a new cluster with two shards, each of which is a replica set. It will configure the replica sets and launch each node with the necessary options to establish replication protocols. It will launch a `mongos` to manage requests across the shards so that clients can interact with the cluster as if communicating with a standalone mongod, to some extent. Finally, it will launch an additional replica set for the config servers that maintain routing table information necessary to ensure queries are directed to the correct shard. Remember that the use cases for sharding are to split a dataset to address hardware and cost constraints or provide better performance to applications (e.g. geographical partitioning). MongoDB sharding provides these capabilities in a

way that is seamless to the application in many respects.

Once ShardingTest has finished setting up our cluster, we will have 10 processes up and running to which we can connect: two replica sets of three nodes each, one config server replica set of three nodes, and one mongos. By default, these processes should begin at port 20000. The mongos should be running at port 20009. Other processes you have running on your local machine and previous calls to ShardingTest can have an effect on which ports ShardingTest uses, but you should not have too much difficulty determining the ports on which your cluster processes are running.



*Figure 12-1. Sharded client connection*

**client**



**mongod**

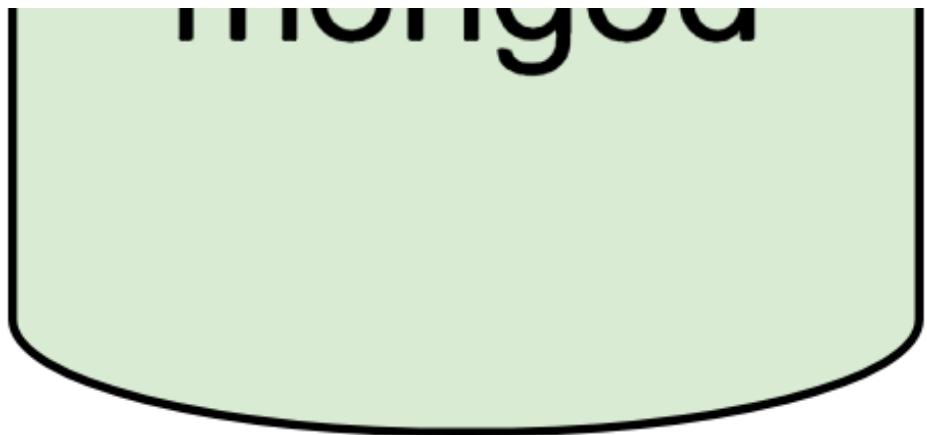


Figure 12-2. Nonsharded client connection

We will connect to the *mongos* to play around with the cluster. Your entire cluster will be dumping its logs to your current shell, so open up a second terminal window and launch another mongo shell.

```
mongo --nodb
```

Use this shell to connect to your cluster's *mongos*: Again, your *mongos* should be running on port 2009.

```
> db = (new Mongo("localhost:20009")).getDB("accounts")
```

Note that the prompt in your mongo shell should change to reflect that you are connected to a *mongos*. Now you are in the situation show in [Figure 12-1](#): the shell is the client and is connected to a *mongos*. You can start passing requests to the *mongos* and it'll route them to the shards. You don't really have to know anything about the shards, like how many there are or what their addresses are. So long as there are some shards out there, you can pass the requests to the *mongos* and allow it to forward them appropriately.

Start by inserting some data:

```
> for (var i=0; i<100000; i++) {  
...     db.users.insert({ "username" : "user"+i, "created_at" : new Date() })  
... }  
> db.users.count()  
100000
```

As you can see, interacting with *mongos* works the same way as interacting with a stand-alone

server does.

You can get an overall view of your cluster by running `sh.status()`. It will give you a summary of your shards, databases, and collections:

```
> sh.status()
--- Sharding Status ---
sharding version: {
  "_id": 1,
  "minCompatibleVersion": 5,
  "currentVersion": 6,
  "clusterId": ObjectId("5a4f93d6bcde690005986071")
}
shards:
{
  "_id" : "one-min-shards-rs0",
  "host" : "one-min-shards-rs0/Shannons-MBP:20000,Shannons-MBP:20001,Shannons-MBP:20002",
  "state" : 1 }
{
  "_id" : "one-min-shards-rs1",
  "host" : "one-min-shards-rs1/Shannons-MBP:20003,Shannons-MBP:20004,Shannons-MBP:20005",
  "state" : 1 }
active mongoses:
"3.6.1" : 1
autosplit:
  Currently enabled: no
balancer:
  Currently enabled: no
  Currently running: no
  Failed balancer rounds in last 5 attempts: 0
  Migration Results for the last 24 hours:
    No recent migrations
databases:
  { "_id" : "accounts", "primary" : "one-min-shards-rs1", "partitioned" : false }
  { "_id" : "config", "primary" : "config", "partitioned" : true }
  config.system.sessions
shard key: { "_id" : 1 }
unique: false
balancing: true
chunks:
  one-min-shards-rs0      1
  { "_id" : { "$minKey" : 1 } } -->> { "_id" : { "$maxKey" : 1 } }
on : one-min-shards-rs0 Timestamp(1, 0)
```

`sh` is similar to `rs`, but for sharding: it is a global variable that defines a number of sharding helper functions. Run `sh.help()` to see what it defines. As you can see from the `sh.status()` output, you have two shards and two databases (`config` is created automatically).

Your *accounts* database may have a different primary shard than shown above. A primary shard is a “home base” shard that is randomly chosen for each database. All of your data will be on this primary shard. MongoDB cannot automatically distribute your data yet because it doesn’t know how (or if) you want it to be distributed. You have to tell it, per-collection, how you want it to distribute data.

### NOTE

A primary shard is different than a replica set primary. A primary shard refers to the entire replica set composing a shard. A primary in a replica set is the single server in the set that can take writes.

To shard a particular collection, first enable sharding on the collection’s database. To do so, run the `enableSharding` command:

```
> sh.enableSharding("accounts")
```

Now sharding is enabled on the *accounts* database, which allows you to shard collections within the database.

When you shard a collection, you choose a shard key. This is a field or two that MongoDB uses to break up data. For example, if you choose to shard on "username", MongoDB would break up the data into ranges of usernames: "a1-steak-sauce" through "defcon", "defcon1" through "howie1998", and so on. Choosing a shard key can be thought of as choosing an ordering for the data in the collection. This is a similar concept to indexing, and for good reason: the shard key becomes the most important index on your collection as it gets bigger. To even create a shard key, the field(s) must be indexed.

Before enabling sharding, we have to create an index on the key we want to shard by:

```
> db.users.createIndex({"username" : 1})
```

Now we’ll shard the collection by "username":

```
> sh.shardCollection("accounts.users", {"username" : 1})
```

Although we are choosing a shard key without much thought here, it is an important decision

that should be carefully considered in a real system. See Chapter 14 for more advice on choosing a shard key.

If you wait a few minutes and run `sh.status()` again, you'll see that there's a lot more information displayed than there was before:

```
MongoDB Enterprise mongos> sh.status()
--- Sharding Status ---
sharding version: {
  "_id" : 1,
  "minCompatibleVersion" : 5,
  "currentVersion" : 6,
  "clusterId" : ObjectId("5a4f93d6bcde690005986071")
}
shards:
{  "_id" : "one-min-shards-rs0",
  "host" : "one-min-shards-rs0/Shannons-MBP:20000,Shannons-MBP:20001,Shan
  "state" : 1 }
{  "_id" : "one-min-shards-rs1",
  "host" : "one-min-shards-rs1/Shannons-MBP:20003,Shannons-MBP:20004,Shan
  "state" : 1 }
active mongoses:
  "3.6.1" : 1
autosplit:
  Currently enabled: no
balancer:
  Currently enabled: yes
  Currently running: no
  Failed balancer rounds in last 5 attempts: 0
Migration Results for the last 24 hours:
  6 : Success
databases:
{  "_id" : "accounts",  "primary" : "one-min-shards-rs1",  "partitioned"
accounts.users
  shard key: { "username" : 1 }
  unique: false
  balancing: true
chunks:
  one-min-shards-rs0  6
  one-min-shards-rs1  7
  { "username" : { "$minKey" : 1 } } -->>
    { "username" : "user17256" } on : one-min-shards-rs0 Timestamp(2, 0)
  { "username" : "user17256" } -->>
    { "username" : "user24515" } on : one-min-shards-rs0 Timestamp(3, 0)
  { "username" : "user24515" } -->>
    { "username" : "user31775" } on : one-min-shards-rs0 Timestamp(4, 0)
  { "username" : "user31775" } -->>
    { "username" : "user39034" } on : one-min-shards-rs0 Timestamp(5, 0)
  { "username" : "user39034" } -->>
    { "username" : "user46294" } on : one-min-shards-rs0 Timestamp(6, 0)
  { "username" : "user46294" } -->>
```

```

        { "username" : "user53553" } on : one-min-shards-rs0 Timestamp(7, 0)
{ "username" : "user53553" } -->>
        { "username" : "user60812" } on : one-min-shards-rs1 Timestamp(7, 1)
{ "username" : "user60812" } -->>
        { "username" : "user68072" } on : one-min-shards-rs1 Timestamp(1, 7)
{ "username" : "user68072" } -->>
        { "username" : "user75331" } on : one-min-shards-rs1 Timestamp(1, 8)
{ "username" : "user75331" } -->>
        { "username" : "user82591" } on : one-min-shards-rs1 Timestamp(1, 9)
{ "username" : "user82591" } -->>
        { "username" : "user89851" } on : one-min-shards-rs1 Timestamp(1, 10)
{ "username" : "user89851" } -->>
        { "username" : "user9711" } on : one-min-shards-rs1 Timestamp(1, 11)
{ "username" : "user9711" } -->>
        { "username" : { "$maxKey" : 1 } } on : one-min-shards-rs1 Timestamp
        { "_id" : "config", "primary" : "config", "partitioned" : true }

config.system.sessions
    shard key: { "_id" : 1 }
    unique: false
    balancing: true
    chunks:
        one-min-shards-rs0 1
        { "_id" : { "$minKey" : 1 } } -->>
        { "_id" : { "$maxKey" : 1 } } on : one-min-shards-rs0 Timestamp(1, 0)

```

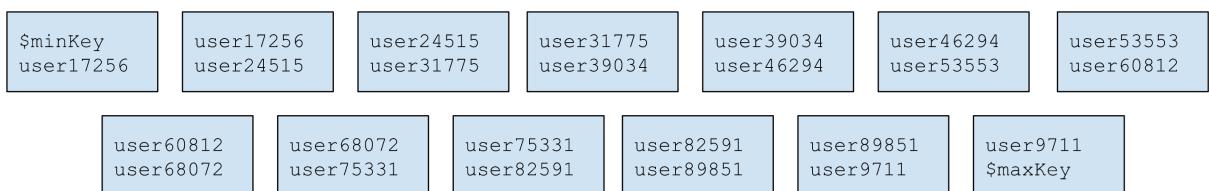
The collection has been split up into 13 chunks, where each chunk is a subset of your data. These are listed by shard key range (the `{ "username" : minValue } -->> { "username" : maxValue }` denotes the range of each chunk). Looking at the `"on" : shard` part of the output, you can see that these chunks have been evenly distributed between the shards.

This process of a collection being split into chunks is shown in Figure 12-3 through Figure 12-5. Before sharding, the collection is essentially a single chunk. Sharding splits this into smaller chunks based on the shard key, as shown in Figure 12-4. These chunks can then be distributed across the cluster, as Figure 12-5 shows.

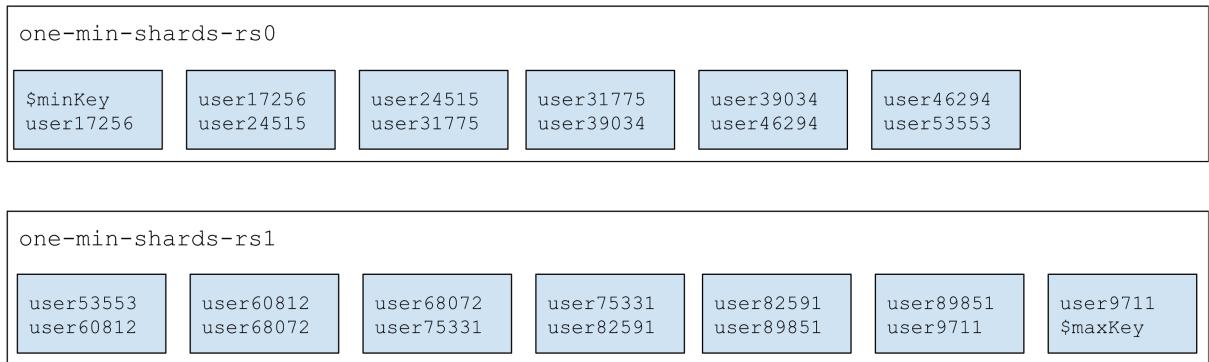
TODO: Update graphics below to reflect the new shard-key ranges and shards defined above.



*Figure 12-3. Before a collection is sharded, it can be thought of as a single chunk from the smallest value of the shard key to the largest*



*Figure 12-4. Sharding splits the collection into many chunks based on shard key ranges*



*Figure 12-5. Chunks are evenly distributed across the available shards*

Notice the keys at the beginning and end of the chunk list: `$minKey` and `$maxKey`. `$minKey` can be thought of as “negative infinity.” It is smaller than any other value in MongoDB. Similarly, `$maxKey` is like “positive infinity.” It is greater than any other value. Thus, you’ll always see these as the “caps” on your chunk ranges. The values for your shard key will always be between `$minKey` and `$maxKey`. These values are actually BSON types and should not be used in your application; they are mainly for internal use. If you wish to refer to them in the shell, use the `MinKey` and `MaxKey` constants.

Now that the data is distributed across multiple shards, let’s try doing some queries. First, try a query on a specific username:

```
> db.users.find({username: "user12345"})
{
  "_id" : ObjectId("5a4fb11dbb9ce6070f377880"),
  "username" : "user12345",
  "created_at" : ISODate("2018-01-05T17:08:45.657Z")
}
```

As you can see, querying works normally. However, let’s run an `explain` to see what MongoDB is doing under the covers:

```
> db.users.find({username: "user12345"}).explain()
```

```
{  
    "queryPlanner" : {  
        "mongosPlannerVersion" : 1,  
        "winningPlan" : {  
            "stage" : "SINGLE_SHARD",  
            "shards" : [{  
                "shardName" : "one-min-shards-rs0",  
                "connectionString" :  
                    "one-min-shards-rs0/Shannons-MBP:20000,Shannons-MBP:20001,Shanno:  
                "serverInfo" : {  
                    "host" : "Shannons-MBP",  
                    "port" : 20000,  
                    "version" : "3.6.1",  
                    "gitVersion" : "025d4f4fe61efd1fb6f0005be20cb45a004093d1"  
                },  
                "plannerVersion" : 1,  
                "namespace" : "accounts.users",  
                "indexFilterSet" : false,  
                "parsedQuery" : {  
                    "username" : {  
                        "$eq" : "user12345"  
                    }  
                },  
                "winningPlan" : {  
                    "stage" : "FETCH",  
                    "inputStage" : {  
                        "stage" : "SHARDING_FILTER",  
                        "inputStage" : {  
                            "stage" : "IXSCAN",  
                            "keyPattern" : {  
                                "username" : 1  
                            },  
                            "indexName" : "username_1",  
                            "isMultiKey" : false,  
                            "multiKeyPaths" : {  
                                "username" : [ ]  
                            },  
                            "isUnique" : false,  
                            "isSparse" : false,  
                            "isPartial" : false,  
                            "indexVersion" : 2,  
                            "direction" : "forward",  
                            "indexBounds" : {  
                                "username" : [  
                                    "[\"user12345\", \"user12345\"]"  
                                ]  
                            }  
                        }  
                    }  
                },  
                "rejectedPlans" : [ ]  
            }]  
        }  
    },  
},
```

```

"ok" : 1,
"$clusterTime" : {
  "clusterTime" : Timestamp(1515174248, 1),
  "signature" : {
    "hash" : BinData(0, "AAAAAAAAAAAAAAAAAAAAA="),
    "keyId" : NumberLong(0)
  }
},
"operationTime" : Timestamp(1515173700, 201)
}

```

From the `winningPlan` field in the explain output, we can see that our cluster satisfied this query using a single shard – *one-min-shards-rs0*. Based on the output of `sh.status()` above, we can see that `user12345` does fall within the key range for the first chunk listed for that shard in our cluster.

Because "username" is the shard key, *mongos* was able to route the query directly to the correct shard. Contrast that with the results for querying for all of the users:

```

> db.users.find().explain()
{
  "queryPlanner": {
    "mongosPlannerVersion": 1,
    "winningPlan": {
      "stage": "SHARD_MERGE",
      "shards": [
        {
          "shardName": "one-min-shards-rs0",
          "connectionString": "one-min-shards-rs0/Shannons-MBP:20000,Shannons-MBP:20001,Shan",
          "serverInfo": {
            "host": "Shannons-MBP.fios-router.home",
            "port": 20000,
            "version": "3.6.1",
            "gitVersion": "025d4f4fe61ef1fb6f0005be20cb45a004093d1"
          },
          "plannerVersion": 1,
          "namespace": "accounts.users",
          "indexFilterSet": false,
          "parsedQuery": {
            ...
          },
          "winningPlan": {
            "stage": "SHARDING_FILTER",
            "inputStage": {
              "stage": "COLLSCAN",
              "direction": "forward"
            }
          }
        }
      ]
    }
  }
}

```

```

    "rejectedPlans": [
        ]
    },
    {
        "shardName": "one-min-shards-rs1",
        "connectionString": "one-min-shards-rs1/Shannons-MBP:20003,Shannons-MBP:20004,Shannons-MBP:20005",
        "serverInfo": {
            "host": "Shannons-MBP.fios-router.home",
            "port": 20003,
            "version": "3.6.1",
            "gitVersion": "025d4f4fe61efd1fb6f0005be20cb45a004093d1"
        },
        "plannerVersion": 1,
        "namespace": "accounts.users",
        "indexFilterSet": false,
        "parsedQuery": {

        },
        "winningPlan": {
            "stage": "SHARDING_FILTER",
            "inputStage": {
                "stage": "COLLSCAN",
                "direction": "forward"
            }
        },
        "rejectedPlans": [
            ]
        }
    }
},
"ok": 1,
"$clusterTime": {
    "clusterTime": Timestamp(1515174893, 1),
    "signature": {
        "hash": BinData(0, "AAAAAAAAAAAAAAAAAAAAA="),
        "keyId": NumberLong(0)
    }
},
"operationTime": Timestamp(1515173709, 514)
}

```

As you can see from this `explain`, this query has to visit both shards to find all the data. In general, if we are not using the shard key in the query, *mongos* will have to send the query to every shard.

Queries that contain the shard key and can be sent to a single shard or subset of shards are called

targeted queries. Queries that must be sent to all shards are called scatter-gather queries: *mongos* scatters the query to all the shards and then gathers up the results.

Once you are finished experimenting, shut down the set. Switch back to your original shell and hit Enter a few times to get back to the command line. Then run `st.stop()` to cleanly shut down all of the servers:

```
> st.stop()
```

If you are ever unsure of what an operation will do, it can be helpful to use `ShardingTest` to spin up a quick local cluster and try it out.

# Chapter 13. Configuring Sharding

---

In the previous chapter, you set up a “cluster” on one machine. This chapter covers how to set up a more realistic cluster and how each piece fits, in particular:

- How to set up config servers, shards, and *mongos* processes
- How to add capacity to a cluster
- How data is stored and distributed

## When to Shard

Deciding when to shard is a balancing act. You generally do not want to shard too early because it adds operational complexity to your deployment and forces you to make design decisions that are difficult to change later. On the other hand, you do not want to wait too long to shard because it is difficult to shard an overloaded system without downtime.

In general, sharding is used to:

- Increase available RAM
- Increase available disk space
- Reduce load on a server
- Read or write data with greater throughput than a single `mongod` can handle

Thus, good monitoring is important to decide when sharding will be necessary. Carefully measure each of these metrics. Generally people speed toward one of these bottlenecks much faster than the others, so figure out which one your deployment will need to provision for first and make plans well in advance about when and how you plan to convert your replica set.

## Starting the Servers

The first step in creating a cluster is to start up all of the processes required. As mentioned in the previous chapter, we need to set up the *mongos* and the shards. There’s also a third component, the config servers, which are an important piece. Config servers are normal `mongod` servers that store the cluster configuration: who the shards are, what collections are sharded by, and the chunks. MongoDB

3.2 introduced the use of replica sets as config servers. Replica sets replace the original syncing mechanism used by config servers. The ability to use the original mechanism was removed in MongoDB 3.4.

## Config Servers

Config servers are the brains of your cluster: they hold all of the metadata about which servers hold what data. Thus, they must be set up first and the data they hold is *extremely* important: make sure that they are running with journaling enabled and that their data is stored on non-ephemeral drives. In production deployments, your config server replica set should consist of at least three members. Each config server should be on a separate physical machine, preferable geographically distributed.

The config servers must be started before any of the *mongos* processes, as *mongos* pulls its configuration from them. Start each config server first on a separate machine. :

```
$ # machine for config server 1
$ mongod --configsvr --replSet configRS --bind_ip localhost,198.51.100.51 mongo
$ 
$ # machine for config server 2
$ mongod --configsvr --replSet configRS --bind_ip localhost,198.51.100.52 mongo
$ 
$ # machine for config server 3
$ mongod --configsvr --replSet configRS --bind_ip localhost,198.51.100.53 mongo
```

Then initiate the config servers as a replica set. To do this, connect a mongo shell to one of the replica set members.

```
mongo --host <hostname> --port <port>
> rs.initiate(
  {
    _id: "configRS",
    configsvr: true,
    members: [
      { _id : 0, host : "cfg1.example.net:27019" },
      { _id : 1, host : "cfg2.example.net:27019" },
      { _id : 2, host : "cfg3.example.net:27019" }
    ]
  }
)
```

Here I'm using *configRS* as the replica set name. Note that this name appears both on the command line when instantiating each config server and in the call to `rs.initiate()`.

The `--configsvr` option indicates to the *mongod* that you are planning to use it as a config server. When running with this option, clients (i.e. other cluster components) cannot write data to any database other than config and admin.

The admin database contains the collections related to the authentication and authorization as well as

the other system.\* collections for internal use. The config database contains the collections that contain the sharded cluster metadata. MongoDB writes data to the config database when the metadata changes, such as after a chunk migration or a chunk split.

When writing to config servers, MongoDB uses a write concern of "majority". When reading from config servers, MongoDB uses a read concern level of "majority". This ensures that sharded cluster meta data will not be committed to the config server replica set until it can't be rolled back. It also ensures that only meta data that will survive a failure of the config servers will be read. This is necessary to ensure all mongos routers have a consistent view of which how data is organized in a sharded cluster.

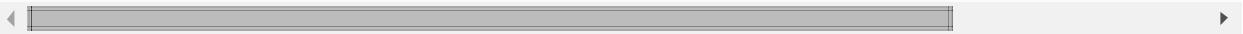
In terms of provisioning, config servers do not need much space or many resources. A generous estimate is 1 KB of config server space per 200 MB of actual data: they really are just tables of contents. As they don't use many resources, you can deploy config servers on machines running other things, like app servers, shard *mongods*, or *mongos* processes.

If all of your config servers are lost, you must dig through the data on your shards to figure out which data is where. This is possible, but slow and unpleasant. Take frequent backups of config server data. Always take a backup of your config servers before performing any cluster maintenance.

## The mongos Processes

Once you have three config servers running, start a *mongos* process for your application to connect to. *mongos* processes need to know where the config servers are, so you must always start *mongos* with the `--configdb` option:

```
$ # machine for mongos
$ mongos --configdb configRS/cfg1.example.net:27019,cfg2.example.net:27019,cfg3.
--bind_ip localhost,198.51.100.100 --logpath /var/log/mongos.log
```



By default, *mongos* runs on port 27017. Note that it does not need a data directory (*mongos* holds no data itself, it loads the cluster configuration from the config servers on startup). Make sure that you set `logpath` to save the *mongos* log somewhere safe.

You can start as many *mongos* processes as you'd like. A common setup is one *mongos* process per application server (running on the same machine as the application server).

## Adding a Shard from a Replica Set

Finally, you're ready to add a shard. There are two possibilities: you may have an existing replica set or you may be starting from scratch. We will cover starting from an existing set below. If you are starting from scratch, initialize an empty set and follow the steps below.

If you already have a replica set serving your application, that will become your first shard. To convert it into a shard, we need to make some small configuration modifications to the members and then tell

the *mongos* how to find the replica set that will comprise the shard.

For example, if you have a replica set named `rs0` on `svr1.example.net`, `svr2.example.net`, `svr3.example.net`, you would first connect to one of the members using the mongo shell and use `rs.status()` to determine which members is primary and which are secondaries.:

```
$ mongo srv1.example.net
> rs.status()
{
  "set" : "rs0",
  "date" : ISODate("2018-11-02T20:02:16.543Z"),
  "myState" : 1,
  "term" : NumberLong(1),
  "heartbeatIntervalMillis" : NumberLong(2000),
  "optimes" : {
    "lastCommittedOpTime" : {
      "ts" : Timestamp(1478116934, 1),
      "t" : NumberLong(1)
    },
    "readConcernMajorityOpTime" : {
      "ts" : Timestamp(1478116934, 1),
      "t" : NumberLong(1)
    },
    "appliedOpTime" : {
      "ts" : Timestamp(1478116934, 1),
      "t" : NumberLong(1)
    },
    "durableOpTime" : {
      "ts" : Timestamp(1478116934, 1),
      "t" : NumberLong(1)
    }
  },
  "members" : [
    {
      "_id" : 0,
      "name" : "svr1.example.net:27017",
      "health" : 1,
      "state" : 1,
      "stateStr" : "PRIMARY",
      "uptime" : 269,
      "optime" : {
        "ts" : Timestamp(1478116934, 1),
        "t" : NumberLong(1)
      },
      "optimeDate" : ISODate("2018-11-02T20:02:14Z"),
      "infoMessage" : "could not find member to sync from",
      "electionTime" : Timestamp(1478116933, 1),
      "electionDate" : ISODate("2018-11-02T20:02:13Z"),
      "configVersion" : 1,
      "self" : true
    },
    {
      "_id" : 1,
      "name" : "svr2.example.net:27017",
      "health" : 1,
      "state" : 2,
      "stateStr" : "SECONDARY",
      "uptime" : 269,
      "optime" : {
        "ts" : Timestamp(1478116934, 1),
        "t" : NumberLong(1)
      },
      "optimeDate" : ISODate("2018-11-02T20:02:14Z"),
      "infoMessage" : "could not find member to sync from",
      "electionTime" : null,
      "electionDate" : null,
      "configVersion" : 1,
      "self" : false
    }
  ]
}
```

```

        "uptime" : 14,
        "optime" : {
            "ts" : Timestamp(1478116934, 1),
            "t" : NumberLong(1)
        },
        "optimeDurable" : {
            "ts" : Timestamp(1478116934, 1),
            "t" : NumberLong(1)
        },
        "optimeDate" : ISODate("2018-11-02T20:02:14Z"),
        "optimeDurableDate" : ISODate("2018-11-02T20:02:14Z"),
        "lastHeartbeat" : ISODate("2018-11-02T20:02:15.618Z"),
        "lastHeartbeatRecv" : ISODate("2018-11-02T20:02:14.866Z"),
        "pingMs" : NumberLong(0),
        "syncingTo" : "m1.example.net:27017",
        "configVersion" : 1
    },
    {
        "_id" : 2,
        "name" : "svr3.example.net:27017",
        "health" : 1,
        "state" : 2,
        "stateStr" : "SECONDARY",
        "uptime" : 14,
        "optime" : {
            "ts" : Timestamp(1478116934, 1),
            "t" : NumberLong(1)
        },
        "optimeDurable" : {
            "ts" : Timestamp(1478116934, 1),
            "t" : NumberLong(1)
        },
        "optimeDate" : ISODate("2018-11-02T20:02:14Z"),
        "optimeDurableDate" : ISODate("2018-11-02T20:02:14Z"),
        "lastHeartbeat" : ISODate("2018-11-02T20:02:15.619Z"),
        "lastHeartbeatRecv" : ISODate("2018-11-02T20:02:14.787Z"),
        "pingMs" : NumberLong(0),
        "syncingTo" : "m1.example.net:27017",
        "configVersion" : 1
    }
],
"ok" : 1
}

```

Beginning with MongoDB 3.4, for sharded clusters, mongod instances for shards **must** be configured with the shardsvr option, either via the configuration file setting sharding.clusterRole or via the command line option --shardsvr.

We will need to do this for each of the members of the replica set we are in the process of converting to a shard. We'll do this by first restarting each secondary in turn with the --shardsvr option. Then we'll step down the primary and restart it with the --shardsvr option.

After shutting down a secondary, restart it as follows.

```
$ mongod --replSet "rs0" --shardsvr --port 27017 --bind_ip localhost,<ip address>
```

Note that you'll need to use the correct ip address for each secondary for the --bind\_ip parameter.

Now connect a mongo shell to the primary.

```
$ mongo m1.example.net
```

Step down the primary.

```
rs.stepDown()
```

Restart the former primary with the --shardsvr option.

```
mongod --replSet "rs0" --shardsvr --port 27017 --bind_ip localhost,<ip address>
```

Now we're ready to add our replica set as shard. Connect a mongo shell to the admin database of the mongos.

```
mongo mongos1.example.net:27017/admin
```

Add a shard to the cluster using the sh.addShard() method.

```
sh.addShard("rs0/srv1.example.net:27017,srv2.example.net:27017,srv3.example.net:27017")
```

You can specify all the members of the set, but you do not have to. *mongos* will automatically detect any members that were not included in the seed list. If you run sh.status(), you'll see that MongoDB soon lists the shard as

```
"rs0/srv1.example.net:27017,srv2.example.net:27017,srv3.example.net:27017"
```

The set name, “rs0”, is taken on as an identifier for this shard. If we ever want to remove this shard or migrate data to it, we'll use “rs0” to describe it. This works better than using a specific server (e.g., *srv1.example.net*), as replica set membership and status can change over time.

Once you've added the replica set as a shard you can convert your application from connecting to the replica set to connecting to the *mongos*. When you add the shard, *mongos* registers that all the databases in the replica set are “owned” by that shard, so it will pass through all queries to your new shard. *mongos* will also automatically handle failover for your application as your client library would: it will pass the errors through to you.

Test failing over a shard's primary in a development environment to ensure that your application

handles the errors received from *mongos* correctly (they should be identical to the errors that you receive from talking to the primary directly).

### NOTE

Once you have added a shard, you *must* set up all clients to send requests to the *mongos* instead of contacting the replica set. Sharding will not function correctly if some clients are still making requests to the replica set directly (not through the *mongos*). Switch all clients to contacting the *mongos* immediately after adding the shard and set up a firewall rule to ensure that they are unable to connect directly to the shard.

Prior to MongoDB 3.6 it was possible to create a standalone mongod as a shard. This is no longer an option in versions of MongoDB 3.6. All shards must be replica sets.

## Adding Capacity

When you want to add more capacity, you'll need to add more shards. To add a new, empty shard, create a replica set. Make sure it has a distinct name from any of your other shards. Once it is initialized and has a primary, add it to your cluster by running the `addShard` command through *mongos*, specifying the new replica set's name and its hosts as seeds.

If you have several existing replica sets that are not shards, you can add all of them as new shards in your cluster so long as they do not have any database names in common. For example, if you had one replica set with a “blog” database, one with a “calendar” database, and one with the “mail”, “tel”, and “music” databases, you could add each replica set as a shard and end up with a cluster with three shards and five databases. However, if you had a fourth replica set that also had a database named “tel”, *mongos* would refuse to add it to the cluster.

## Sharding Data

MongoDB won't distribute your data automatically until you tell it how to do so. You must explicitly tell both the database and collection that you want them to be distributed. For example, suppose we want to shard the *artists* collection in the *music* database on the "name" key. First, we enable sharding for the database, *music*:

```
> db.enableSharding("music")
```

Sharding a database is always prerequisite to sharding one of its collections.

Once you've enabled sharding on the database level, you can shard a collection by running `sh.shardCollection`:

```
> sh.shardCollection("music.artists", {"name" : 1})
```

---

Now the collection will be sharded by the "name" key. If you are sharding an existing collection there must be an index on the "name" field; otherwise the `shardCollection` call will return an error. If you get an error, create the index (*mongos* will return the index it suggests as part of the error message) and retry the `shardCollection` command.

If the collection you are sharding does not yet exist, *mongos* will automatically create the shard key index for you.

The `shardCollection` command splits the collection into chunks, which are the unit MongoDB uses to move data around. Once the command returns successfully, MongoDB will begin balancing the collection across the shards in your cluster. This process is not instantaneous. For large collections it may take hours to finish this initial balancing.

## How MongoDB Tracks Cluster Data

Each *mongos* must always know where to find a document, given its shard key. Theoretically, MongoDB could track where each and every document lived, but this becomes unwieldy for collections with millions or billions of documents. Thus, MongoDB groups documents into chunks, which are documents in a given range of the shard key. A chunk always lives on a single shard, so MongoDB can keep a small table of chunks mapped to shards.

For example, if a user collection's shard key is `{"age" : 1}`, one chunk might be all documents with an "age" field between 3 and 17. If *mongos* gets a query for `{"age" : 5}`, it can route the query to the shard where the 3–17 chunk lives.

As writes occur, the number and size of the documents in a chunk might change. Inserts can make a chunk contain more documents, removes fewer. If we were making a game for children and preteens, our chunk for ages 3–17 might get larger and larger (one would hope). Almost all of our users would be in that chunk, and so on a single shard, somewhat defeating the point of distributing our data. Thus, once a chunk grows to a certain size, MongoDB automatically splits it into two smaller chunks. In this example, the chunk might be split into one chunk containing documents with ages 3 through 11 and the another containing 12 through 17. Note that these two chunks still cover the entire age range that the original chunk covered: 3–17. As these new chunks grow, they can be split into still smaller chunks until there is a chunk for each age.

You cannot have chunks with overlapping ranges, like 3–15 and 12–17. If you could, MongoDB would need to check both chunks when attempting to find an age in the overlap, like 14. It is more efficient to only have to look in one place, particularly once chunks begin moving around the cluster.

A document always belongs to one and only one chunk. One consequence to this rule is that you cannot use an array field as your shard key, since MongoDB creates multiple index entries for arrays. For example, if a document had [5, 26, 83] in its "age" field, it would belong in up to three chunks.

## NOTE

A common misconception is that the data in a chunk is physically grouped on disk. This is incorrect: chunks have no effect on how *mongod* stores collection data.

## Chunk Ranges

Each chunk is described by the range it contains. A newly sharded collection starts off with a single chunk and every document lives in this chunk. This chunk's bounds are negative infinity to infinity, shown as `$minKey` and `$maxKey` in the shell.

As this chunk grows, MongoDB will automatically split it into two chunks, with the range negative infinity to `<some value>` and `<some value>` to infinity. `<some value>` is the same for both chunks: the lower chunk contains everything up to (but not including) `<some value>` and the upper chunk actually contains `<some value>`.

This may be more intuitive with an example: suppose we were sharding by "age" as described earlier. All documents with "age" between 3 and 17 are contained on one chunk:  $3 \leq \text{age} < 17$ . When this is split, we end up with two ranges:  $3 \leq \text{age} < 12$  on one chunk and  $12 \leq \text{age} < 17$  on the other. 12 is called the split point.

Chunk information is stored in the `config.chunks` collection. If you looked at the contents of that collection, you'd see documents that looked something like this (some fields have been omitted for clarity):

```
> db.chunks.find(criteria, {"min" : 1, "max" : 1})
{
  "_id" : "test.users-age_-100.0",
  "min" : {"age" : -100},
  "max" : {"age" : 23}
}
{
  "_id" : "test.users-age_23.0",
  "min" : {"age" : 23},
  "max" : {"age" : 100}
}
{
  "_id" : "test.users-age_100.0",
  "min" : {"age" : 100},
  "max" : {"age" : 1000}
}
```

Based on the `config.chunks` documents shown, here are a few examples of where various documents would live:

```
{"_id" : 123, "age" : 50}
```

This document would live in the second chunk, as that chunk contains all documents with "age" between 23 and 100.

```
{"_id" : 456, "age" : 100}
```

This document would live on the third chunk, as lower bounds are inclusive. The second chunk contains all documents up to "age" : 100, but not any documents where "age" equals 100.

```
{"_id" : 789, "age" : -101}
```

This document would not be in any of these chunks. It would be in some chunk with a range lower than the first chunk's.

With a compound shard key, shard ranges work the same way that sorting by the two keys would work. For example, suppose that we had a shard key on { "username" : 1, "age" : 1 }. Then we might have chunk ranges such as:

```
{
  "_id" : "test.users-username_MinKeyage_MinKey",
  "min" : {
    "username" : { "$minKey" : 1 },
    "age" : { "$minKey" : 1 }
  },
  "max" : {
    "username" : "user107487",
    "age" : 73
  }
}
{
  "_id" : "test.users-username_\\"user107487\\\"age_73.0",
  "min" : {
    "username" : "user107487",
    "age" : 73
  },
  "max" : {
    "username" : "user114978",
    "age" : 119
  }
}
{
  "_id" : "test.users-username_\\"user114978\\\"age_119.0",
  "min" : {
    "username" : "user114978",
    "age" : 119
  },
  "max" : {
    "username" : "user122468",
    "age" : 68
  }
}
```

Thus, *mongos* can easily find on which chunk someone with a given username (or a given username and age) lives. However, given just an age, *mongos* would have to check all, or almost all, chunks. If

we wanted to be able to target queries on age to the right chunk, we'd have to use the "opposite" shard key: `{"age" : 1, "username" : 1}`. This is often a point of confusion: a range over the second half of a shard key will cut across multiple chunks.

## Splitting Chunks

*mongos* tracks how much data it inserts per chunk and, once that reaches a certain threshold, checks if the chunk needs to be split, as shown in [Figure 13-1](#) and [Figure 13-2](#). If the chunk does need to be split, *mongos* will update the chunk's metadata on the config servers. Chunk splits are just a metadata change (no data is moved). New chunk documents are created on the config servers and the old chunk's range ("max") is modified. Once that process is complete, the *mongos* resets its tracking for the original chunk and creates new trackers for the new chunks.

When *mongos* asks a shard if a chunk needs to be split, the shard makes a rough calculation of the chunk size. If it finds that the chunk is getting large, it finds split points and sends those to the *mongos* (as shown in [Figure 13-3](#)).

A shard may not be able to find any split points though, even for a large chunk, as there are a limited number of ways to legally split a chunk. Any two documents with the same shard key must live in the same chunk so chunks can only be split between documents where the shard key's value changes. For example, if the shard key was "age", the following chunk could be split at the points where the shard key changed, as indicated:

```
{ "age" : 13, "username" : "ian" }
{ "age" : 13, "username" : "randolph" }
----- // split point
{ "age" : 14, "username" : "randolph" }
{ "age" : 14, "username" : "eric" }
{ "age" : 14, "username" : "hari" }
{ "age" : 14, "username" : "mathias" }
----- // split point
{ "age" : 15, "username" : "greg" }
{ "age" : 15, "username" : "andrew" }
```

*mongos* will not necessarily split a chunk at every split point available, but those are the possibilities it has to choose from.

For example, if the chunk contained the following documents, it could not be split (unless the application started inserting fractional ages):

```
{ "age" : 12, "username" : "kevin" }
{ "age" : 12, "username" : "spencer" }
{ "age" : 12, "username" : "alberto" }
{ "age" : 12, "username" : "tad" }
```

Thus, having a variety of values for your shard key is important. Other important properties will be covered in the next chapter.

If one of the config servers is down when a *mongos* tries to do a split, the *mongos* won't be able to update the metadata (as shown in Figure 13-4). All config servers must be up and reachable for splits to happen. If the *mongos* continues to receive write requests for the chunk, it will keep trying to split the chunk and fail. As long as the config servers are not healthy, splits will continue not to work and all the split attempts can slow down the *mongos* and shard involved (which repeats the process shown in Figure 13-1 through Figure 13-4 for each incoming write). This process of *mongos* repeatedly attempting to split a chunk and being unable to is called a split storm. The only way to prevent split storms is to ensure that your config servers are up and healthy as much of the time as possible. You can also restart a *mongos* to reset its write counter (so that it is no longer at the split threshold).

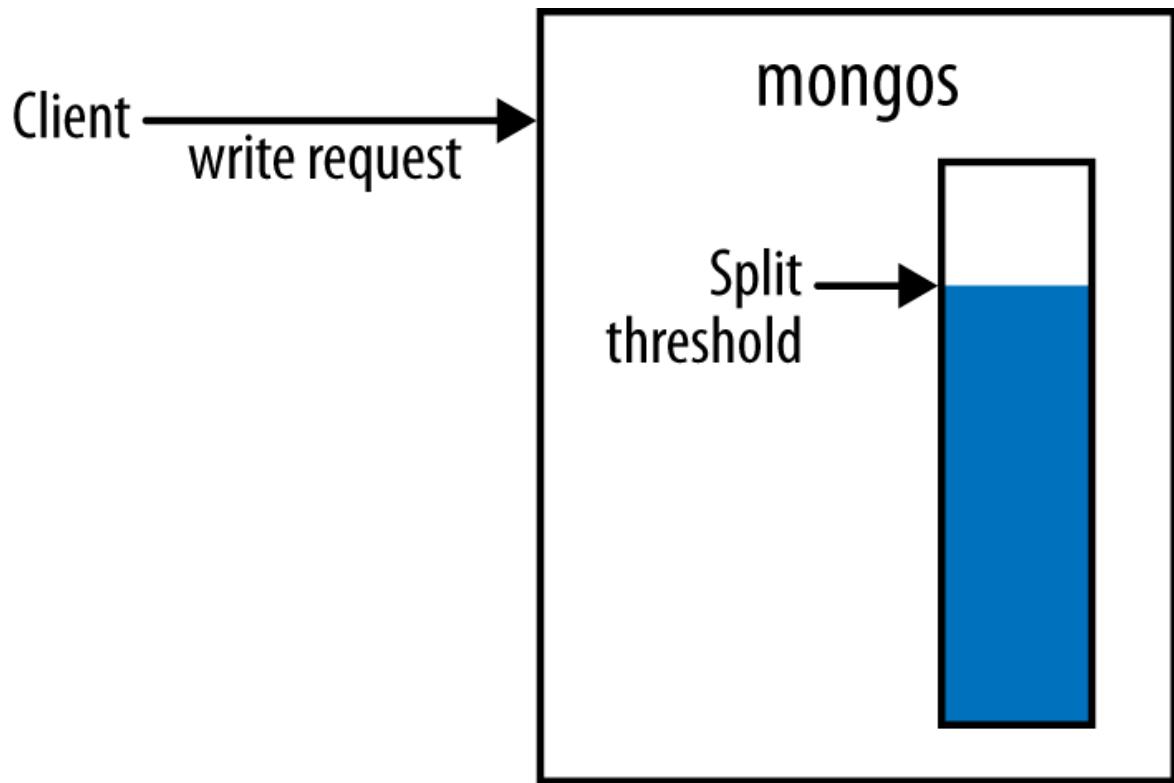


Figure 13-1. When a client writes to a chunk, *mongos* will check its split threshold for the chunk

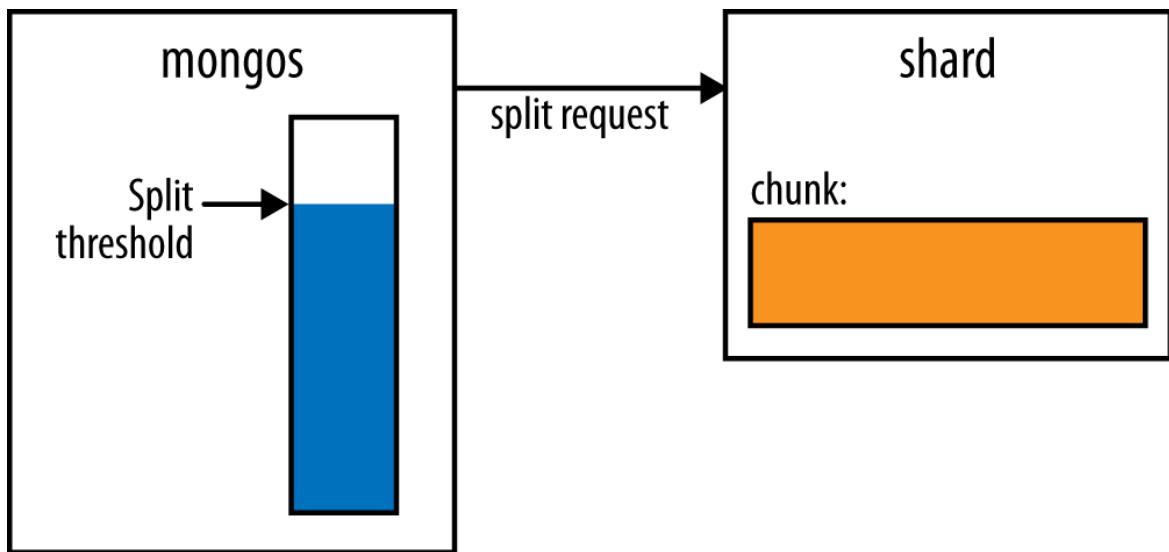


Figure 13-2. If the split threshold has been reached, mongos will send a request for split points to the shard

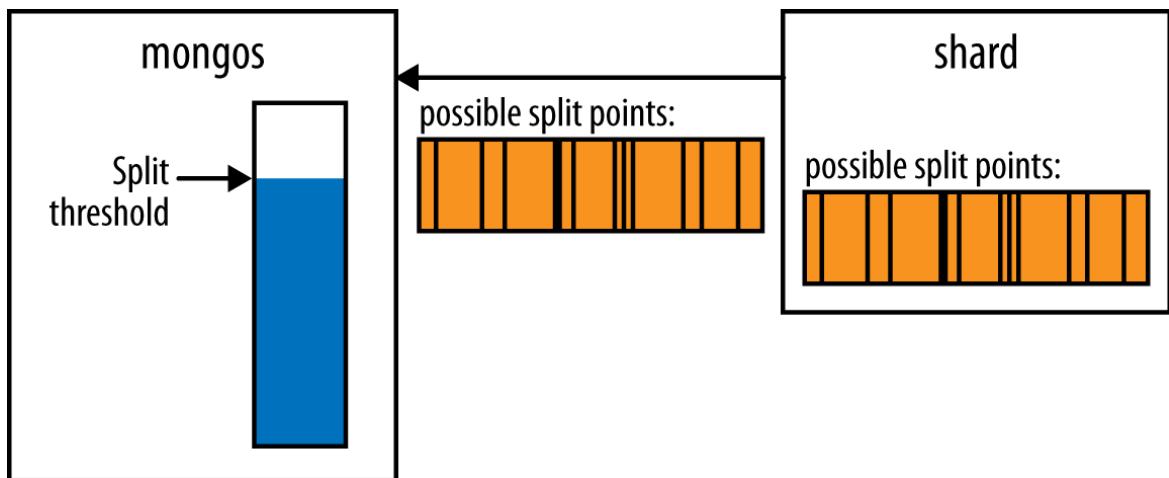
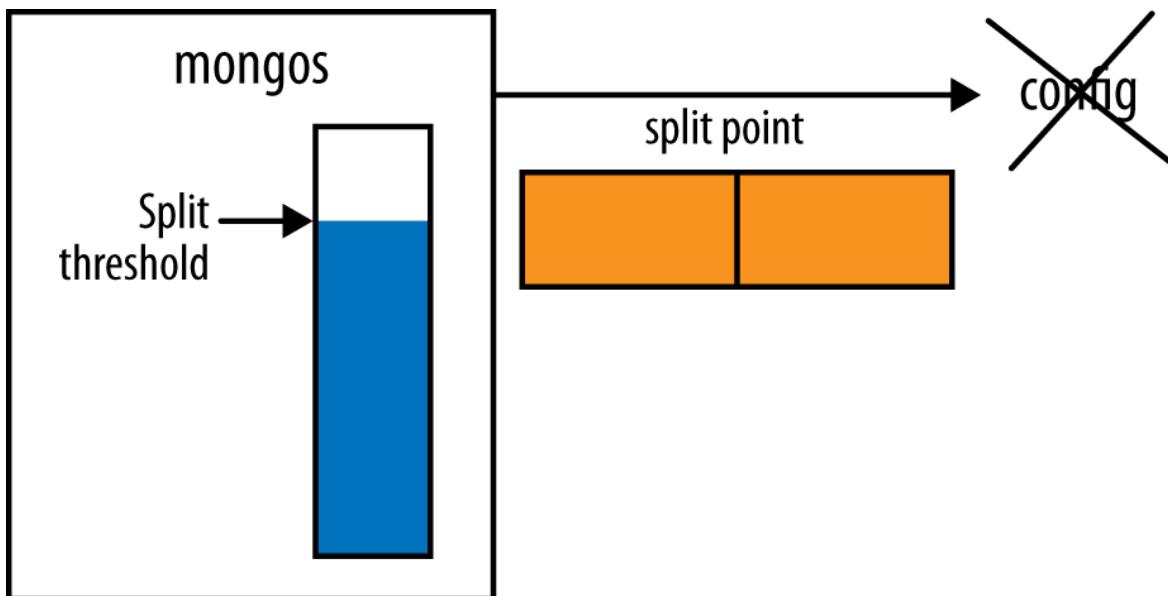


Figure 13-3. The shard calculates split points for the chunk and sends them to the mongos



*Figure 13-4. The mongos chooses a split point and attempts to inform the config server but cannot reach it. Thus, it is still over its split threshold for the chunk and any subsequent writes will trigger this process again.*

Another issue is that *mongos* might never realize that it needs to split a large chunk. There is no global counter of how big each chunk is. Each *mongos* simply calculates whether the writes it has received have reached a certain threshold (as shown in Figure 13-5). This means that if your *mongos* processes go up and down frequently a *mongos* might never receive enough writes to hit the split threshold before it is shut down again and your chunks will get larger and larger (as shown in Figure 13-6).

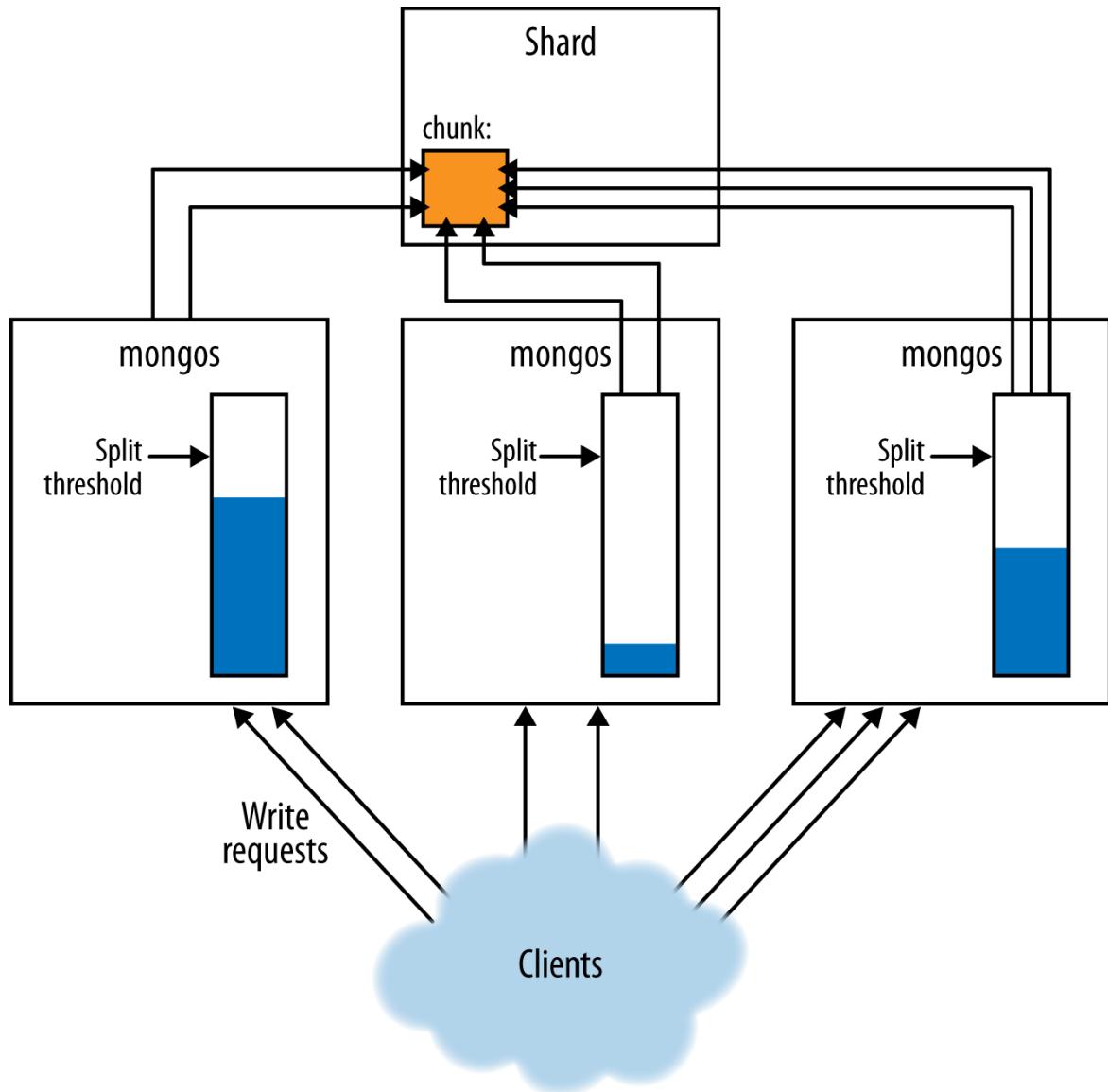
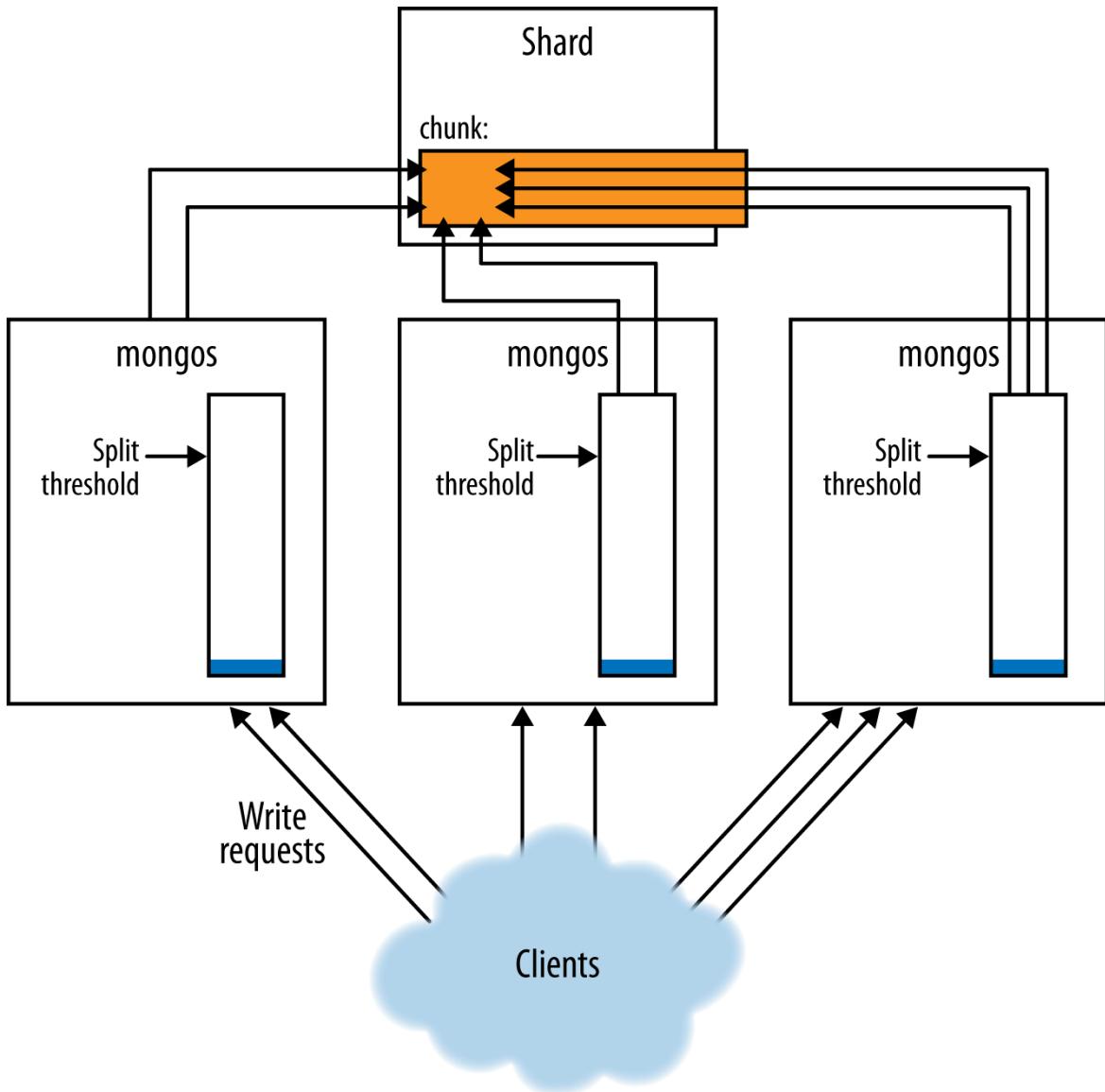


Figure 13-5. As mongos processes perform writes, their counters increase toward the split threshold



*Figure 13-6. If mongos processes are regularly restarted their counters may never hit the threshold, making chunks grow without bound*

The first way to prevent this is to have fewer *mongos* churn. Leave *mongos* processes up, when possible, instead of spinning them up when they are needed and then turning them off when they are not. However, some deployments may find it too expensive to run *mongos* processes that aren't being used. If you are in this situation, another way of getting more splits is to make the chunk size smaller than you actually want it to be. This will prompt splits to happen at a lower threshold.

You can turn off chunk splitting by starting every *mongos* with `--nosplit`.

## The Balancer

The balancer is responsible for migrating data. It regularly checks for imbalances between shards and, if it finds an imbalance, will begin migrating chunks. Although the balancer is often referred to as a single entity, each *mongos* plays the part of “the balancer” occasionally.

Every few seconds, a *mongos* will attempt to become the balancer. If there are no other balancers active, the *mongos* will take a cluster-wide lock from the config servers and do a balancing round.

Balancing doesn't affect a *mongos*'s normal routing operations, so clients using that *mongos* should be unaffected.

You can see which *mongos* is the balancer by looking at the the *config.locks* collection:

```
> db.locks.findOne({ "_id" : "balancer"})
{
  "_id" : "balancer",
  "process" : "router-23:27017:1355763351:1804289383",
  "state" : 0,
  "ts" : ObjectId("50cf939c051fcdb8139fc72c"),
  "when" : ISODate("2012-12-17T21:50:20.023Z"),
  "who" : "router-23:27017:1355763351:1804289383:Balancer:846930886",
  "why" : "doing balance round"
}
```

The *config.locks* collection keeps track of all cluster-wide locks. The balancer is the document with the `"_id"` of "balancer". The lock's `"who"` field tells you which *mongos* is—or was—balancing: *router-23:27017* in this case. The `"state"` field indicates whether the balancer is running; 0 means it is no longer active, 2 means it's still balancing. (1 means that the *mongos* is attempting to take the lock but has not yet acquired it—you won't usually see 1.)

Once a *mongos* has become the balancer, it checks its table of chunks for each collection to see if any shards have hit the balancing threshold. This is when one shard has significantly more chunks than the other shards (the exact threshold varies: larger collections tolerate larger imbalances than smaller ones). If an imbalance is detected, the balancer will redistribute chunks until all shards are within one chunk of one another. If no collections have hit the balancing threshold, the *mongos* stops being the balancer.

Assuming that some collections have hit the threshold, the balancer will begin migrating chunks. It chooses a chunk from the overloaded shard and asks the shard if it should split the chunk before migrating. Once it does any necessary splits, it migrates the chunk to a machine with fewer chunks.

An application using the cluster does not need be aware that the data is moving: all reads and writes are routed to the old chunk until the move is complete. Once the metadata is updated, all *mongos* processes attempting to access the data in the old location will get an error. These errors should not be visible to the client: the *mongos* will silently handle the error and retry the operation on the new shard.

This is a common cause of errors you might see in *mongos* logs that are about being “unable to `setShardVersion`.“ When *mongos* gets this type of error, it looks up the new location of the data from the config servers, updates its chunk table, and attempts the request again. If it successfully retrieves the data from the new location, it will return it to the client as though nothing went wrong (but it will print a message in the log that the error occurred).

If the *mongos* is unable to retrieve the new chunk location because the config servers are unavailable, it will return an error to the client. This is another reason why it is important to always have config servers up and healthy.

# Chapter 14. Choosing a Shard Key

---

The most important task when using sharding is choosing how your data will be distributed. To make intelligent choices about this, you have to understand how MongoDB distributes data. This chapter helps you make a good choice of shard key by covering:

- How to decide among multiple possible shard keys
- Shard keys for several use cases
- What you can't use as a shard key
- Some alternative strategies if you want to customize how data is distributed
- How to manually shard your data

This chapter assumes that you understand the basic components of sharding as covered in the previous chapters.

## Taking Stock of Your Usage

When you shard a collection you choose a field or two to use to split up the data. This key (or keys) is called a *shard key*. Once you shard a collection you cannot change your shard key, so it is important to choose correctly.

To choose a good shard key, you need to understand your workload and how your shard key is going to distribute your application's requests. This can be difficult to picture, so try to work out some examples or, even better, try it out on a backup data set with sample traffic. This section has lots of diagrams and explanations, but there is no substitute for trying it on your own data set.

For each collection that you're planning to shard, start by answering the following questions:

- How many shards are you planning to grow to? A three-shard cluster has a great deal more flexibility than a thousand-shard cluster. As a cluster gets larger, you should not plan to fire

off queries that can hit all shards, so almost all queries must include the shard key.

- Are you sharding to decrease read or write latency? (Latency refers to how long something takes, e.g., a write takes 20 ms, but we need it to take 10 ms.) Decreasing write latency usually involves sending requests to geographically closer or more powerful machines.
- Are you sharding to increase read or write throughput? (Throughput refers to how many requests the cluster can handle at the same time: the cluster can do 1,000 writes in 20 ms, but we need it to do 5,000 writes in 20 ms.) Increasing throughput usually involves adding more parallelization and making sure that requests are distributed evenly across the cluster.
- Are you sharding to increase system resources (e.g., give MongoDB more RAM per GB of data)? If so, you want to keep working set size as small possible.

Use these answers to evaluate the following shard key descriptions and decide whether the shard key you choose would work well in your situation. Does it give you the targeted queries that you need? Does it change the throughput or latency of your system in the ways you need? If you need a compact working set, does it provide that?

## Picturing Distributions

There are a number basic distributions that are the most common ways people choose to split their data: ascending key, random, and location-based. There are other types of keys that could be used, but most use cases fall into one of these categories. Each is discussed in the following sections.

### Ascending Shard Keys

Ascending shard keys are generally something like a "date" field or `ObjectId`—anything that steadily increases over time. An autoincrementing primary key is another example of an ascending field, albeit one that doesn't show up in MongoDB much (unless you're importing from another database).

Suppose that we shard on an ascending field, like "`_id`" on a collection using ObjectIds. If we shard on "`_id`", then this will be split into chunks of "`_id`" ranges, as in [Figure 14-1](#). These chunks will be distributed across our sharded cluster of, let's say, three shards, as shown in [Figure 14-2](#).

`$minKey -> ObjectId("5112fa61b4a4b396ff960262")`

`ObjectId("5112fa61b4a4b396ff960262") ->  
ObjectId("5112fa9bb4a4b396ff96671b")`

`ObjectId("5112fa9bb4a4b396ff96671b") ->  
ObjectId("5112faa0b4a4b396ff9732db")`

`ObjectId("5112faa0b4a4b396ff9732db") ->  
ObjectId("5112fabbb4a4b396ff97fb40")`

`ObjectId("5112fabbb4a4b396ff97fb40") ->  
ObjectId("5112fac0b4a4b396ff98c6f8")`

`ObjectId("5112fac0b4a4b396ff98c6f8") ->  
ObjectId("5112fac5b4a4b396ff998b59")`

`ObjectId("5112fac5b4a4b396ff998b59") ->  
ObjectId("5112facab4a4b396ff9a56c5")`

`ObjectId("5112facab4a4b396ff9a56c5") ->  
ObjectId("5112facfb4a4b396ff9b1b55")`

`ObjectId("5112facfb4a4b396ff9b1b55") ->  
ObjectId("5112fad4b4a4b396ff9bd69b")`

`ObjectId("5112fad4b4a4b396ff9bd69b") ->  
ObjectId("5112fae0b4a4b396ff9d0ee5")`

```
ObjectId("5112fae0b4a4b396ff9d0ee5") -> $maxKey
```

*Figure 14-1. The collection is split into ranges of ObjectIds. Each range is a chunk.*

Suppose we create a new document. Which chunk will it be in? The answer is the chunk with the range ObjectId ("5112fae0b4a4b396ff9d0ee5") through \$maxKey. This is called the max chunk, as it is the chunk containing \$maxKey.

If we insert another document, it will also be in the max chunk. In fact, every subsequent insert will be into the max chunk! Every insert's "\_id" field will be closer to infinity than the previous (because ObjectIds are always ascending), so they will all go to into the max chunk.

## shard0000

ObjectId("5112fa9bb4a4b396ff96671b") ->  
ObjectId("5112faa0b4a4b396ff9732db")

ObjectId("5112faa0b4a4b396ff9732db") ->  
ObjectId("5112fabb4a4b396ff97fb40")

ObjectId("5112fabb4a4b396ff97fb40") ->  
ObjectId("5112fac0b4a4b396ff98c6f8")

## shard0001

\$minKey -> ObjectId("5112fa61b4a4b396ff960262")

ObjectId("5112fa61b4a4b396ff960262") ->  
ObjectId("5112fa9bb4a4b396ff96671b")

ObjectId("5112fac0b4a4b396ff98c6f8") ->  
ObjectId("5112fac5b4a4b396ff998b59")

ObjectId("5112fac5b4a4b396ff998b59") ->  
ObjectId("5112facab4a4b396ff9a56c5")

## shard0002

ObjectId("5112facab4a4b396ff9a56c5") ->  
ObjectId("5112facfb4a4b396ff9b1b55")

```
ObjectId("5112facfb4a4b396ff9b1b55") ->  
ObjectId("5112fad4b4a4b396ff9bd69b")
```

```
ObjectId("5112fad4b4a4b396ff9bd69b") ->  
ObjectId("5112fae0b4a4b396ff9d0ee5")
```

```
ObjectId("5112fae0b4a4b396ff9d0ee5") -> $maxKey
```

Figure 14-2. Chunks are distributed across shards in a random order

This has a couple of interesting (and often undesirable) properties. First, all of your writes will be routed to one shard (shard0002, in this case). This chunk will be the only one growing and splitting, as it is the only one that receives inserts. As you insert data, new chunks will “fall off” of this chunk’s butt, as shown in Figure 14-3.

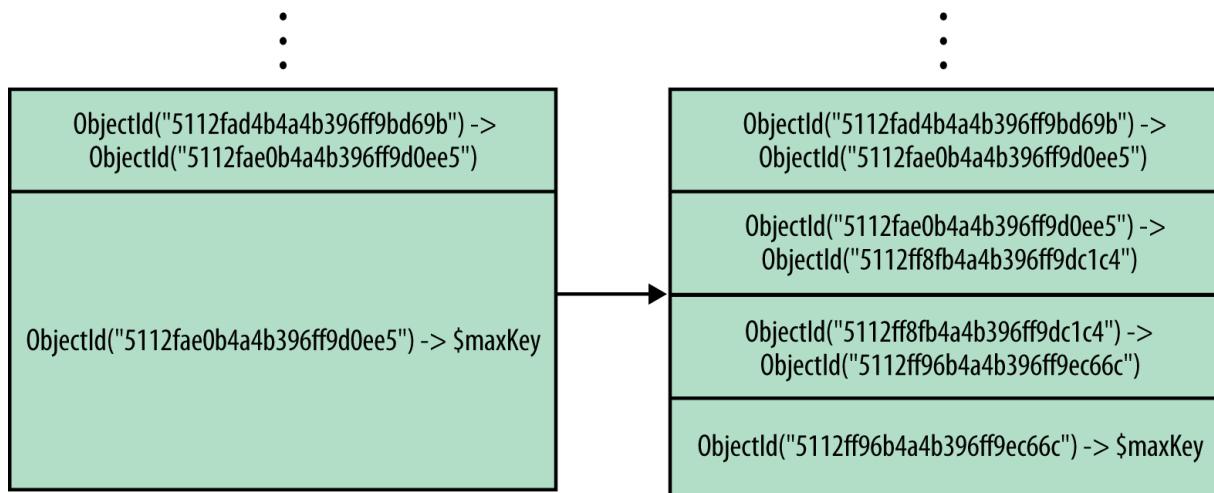


Figure 14-3. The max chunk continues growing and being split into multiple chunks

This pattern often makes it more difficult for MongoDB to keep chunks evenly balanced because all the chunks are being created by one shard. Therefore, MongoDB must constantly move chunks to other shards instead of correcting small imbalances that might occur in a more evenly distributed systems.

## **Randomly Distributed Shard Keys**

On the other end of the spectrum are randomly distributed shard keys. Randomly distributed keys could be usernames, email addresses, UUIDs, MD5 hashes, or any other key that has no identifiable pattern in your dataset.

Suppose the shard key is a random number between 0 and 1. We'll end up with a random distribution of chunks on the various shards, as shown in [Figure 14-4](#).

shard0000

\$minKey ->  
0.07152752857759748

0.5050852404345105 -> 0.5909494812833331

0.5909494812833331 -> 0.6969766499990353

shard0001

0.6969766499990353 -> 0.8400606470845913

0.8400606470845913 -> 0.9190519609736775

0.9190519609736775 -> 0.9999498302686232

0.9999498302686232 ->  
\$maxKey

shard0002

0.07152752857750748 -> 0.1542520872088625

```
0.07192752057759748 -> 0.15425320872988635
```

```
0.15425320872988635 -> 0.25743183243034107
```

```
0.25743183243034107 -> 0.3640577812240344
```

```
0.3640577812240344 -> 0.5050852404345105
```

Figure 14-4. As in the previous section, chunks are distributed randomly around the cluster

As more data is inserted, the data's random nature means that inserts should hit every chunk fairly evenly. You can prove this to yourself by inserting 10,000 documents and seeing where they end up:

```
> var servers = {}
> var findShard = function (id) {
...     var explain = db.random.find({_id:id}).explain();
...     for (var i in explain.shards) {
...         var server = explain.shards[i][0];
...         if (server.n == 1) {
...             if (server.server in servers) {
...                 servers[server.server]++;
...             } else {
...                 servers[server.server] = 1;
...             }
...         }
...     }
... }
> for (var i = 0; i < 10000; i++) {
...     var id = ObjectId();
...     db.random.insert({_id: id, "x": Math.random()});
...     findShard(id);
... }
> servers
{
    "spock:30001" : 2942,
    "spock:30002" : 4332,
```

```
"spock:30000" : 2726
```

```
}
```

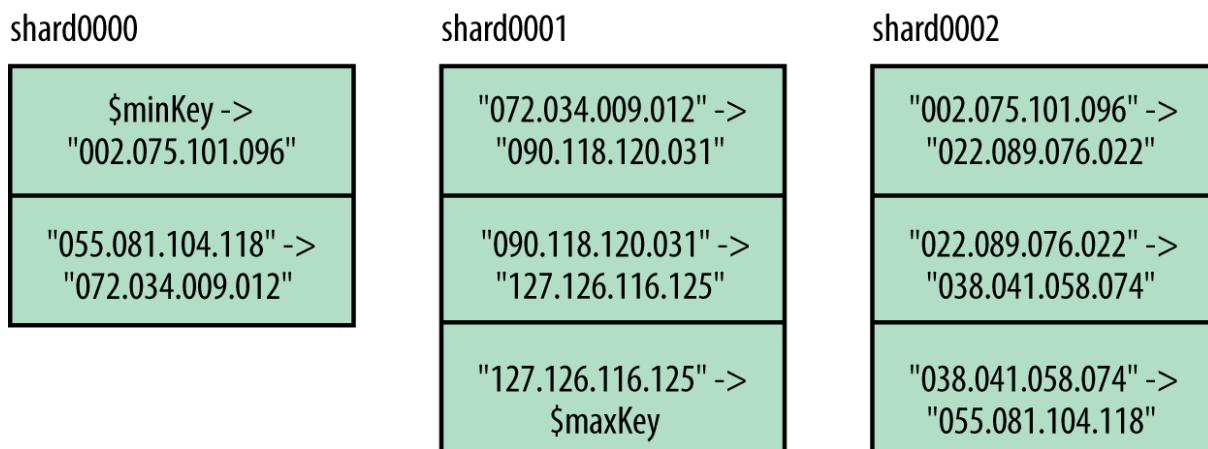
As writes are randomly distributed, shards should grow at roughly the same rate, limiting the number of migrates that need to occur.

The only downside to randomly distributed shard keys is that MongoDB isn't efficient at randomly accessing data beyond the size of RAM. However, if you have the capacity or don't mind the performance hit, random keys nicely distribute load across your cluster.

## Location-Based Shard Keys

Location-based shard keys may be things like a user's IP, latitude and longitude, or address. Location shard keys are not necessarily related to a physical location field: the "location" might be a more abstract way that data should be grouped together. In any case, it is a key where documents with some similarity fall into a range based on this field. This can be handy for both putting data close to its users and keeping related data together on disk.

For example, suppose we have a collection of documents that are sharded on an IP address. Documents will be organized into chunks based on their addresses and randomly spread across the cluster, as shown in [Figure 14-5](#).



*Figure 14-5. A sample distribution of chunks in the IP address collection*

If we wanted certain chunk ranges to be attached to certain shards, we could zone these shards and then assign chunk ranges to each zone. In this example, suppose that we wanted to keep certain IP blocks on certain shards: say, "56.\*.\*.\*" (the United States Postal Service's IP block) on shard0000 and "17.\*.\*.\*" (Apple's IP block) on either shard0000 or shard0002. We do not care where the other IPs live. We could request that the balancer do this by setting up zones:

```
> sh.addShardToZone("shard0000", "USPS")
> sh.addShardToZone("shard0000", "Apple")
> sh.addShardToZone("shard0002", "Apple")
```

Next, we create the rules:

```
> sh.updateZoneKeyRange("test.ips", {"ip" : "056.000.000.000"},
... {"ip" : "057.000.000.000"}, "USPS")
```

This attaches all IPs greater than or equal to 56.0.0.0 and less than 57.0.0.0 to the shard zoned as “USPS”. Next, we add a rule for Apple:

```
> sh.updateZoneKeyRange("test.ips", {"ip" : "017.000.000.000"},
... {"ip" : "018.000.000.000"}, "Apple")
```

When the balancer moves chunks, it will attempt to move chunks with those ranges to those shards. Note that this process is not immediate. Chunks that were not covered by a zone key range will be moved around normally. The balancer will continue attempting to distribute chunks evenly among shards.

## Shard Key Strategies

This section presents a number of shard key options for various types of applications.

### Hashed Shard Key

For loading data as fast as possible, hashed shard keys are the best option. A hashed shard key can make any field randomly distributed, so it is a good choice if you’re going to be using an ascending key a in a lot of queries but want writes to be random distributed.

The trade-off is that you can never do a targeted range query with a hashed shard key. If you will not be doing range queries, though, hashed shard keys are a good option.

To create a hashed shard key, first create a hashed index:

```
> db.users.createIndex({"username" : "hashed"})
```

Next, shard the collection with:

```
> sh.shardCollection("app.users", {"username" : "hashed"})
```

```
{ "collectionsharded" : "app.users", "ok" : 1 }
```

If you create a hashed shard key on a nonexistent collection, `shardCollection` behaves interestingly: it assumes that you want evenly distributed chunks, so it immediately creates a bunch of empty chunks and distributes them around your cluster. For example, suppose our cluster looked like this before creating the hashed shard key:

```
> sh.status()
--- Sharding Status ---
sharding version: { "_id" : 1, "version" : 3 }
shards:
{ "_id" : "shard0000", "host" : "localhost:30000" }
{ "_id" : "shard0001", "host" : "localhost:30001" }
{ "_id" : "shard0002", "host" : "localhost:30002" }
databases:
{ "_id" : "admin", "partitioned" : false, "primary" : "config" }
{ "_id" : "test", "partitioned" : true, "primary" : "shard0001" }
```

Immediately after `shardCollection` returns there are two chunks on each shard, evenly distributing the key space across the cluster:

```
> sh.status()
--- Sharding Status ---
sharding version: { "_id" : 1, "version" : 3 }
shards:
{ "_id" : "shard0000", "host" : "localhost:30000" }
{ "_id" : "shard0001", "host" : "localhost:30001" }
{ "_id" : "shard0002", "host" : "localhost:30002" }
databases:
{ "_id" : "admin", "partitioned" : false, "primary" : "config" }
{ "_id" : "test", "partitioned" : true, "primary" : "shard0001" }
    test.foo
        shard key: { "username" : "hashed" }
chunks:
    shard0000          2
    shard0001          2
    shard0002          2
{ "username" : { "$MinKey" : true } }
    -->> { "username" : NumberLong("-6148914691236517204") }
    on : shard0000 { "t" : 3000, "i" : 2 }
{ "username" : NumberLong("-6148914691236517204") }
    -->> { "username" : NumberLong("-3074457345618258602") }
    on : shard0000 { "t" : 3000, "i" : 3 }
{ "username" : NumberLong("-3074457345618258602") }
    -->> { "username" : NumberLong(0) }
    on : shard0001 { "t" : 3000, "i" : 4 }
{ "username" : NumberLong(0) }
```

```
-->> { "username" : NumberLong("3074457345618258602") }
      on : shard0001 { "t" : 3000, "i" : 5 }
      { "username" : NumberLong("3074457345618258602") }
      -->> { "username" : NumberLong("6148914691236517204") }
      on : shard0002 { "t" : 3000, "i" : 6 }
      { "username" : NumberLong("6148914691236517204") }
      -->> { "username" : { "$MaxKey" : true } }
      on : shard0002 { "t" : 3000, "i" : 7 }
```

Note that there are no documents in the collection yet, but when you start inserting them, writes should be evenly distributed across the shards from the get-go. Ordinarily, you would have to wait for chunks to grow, split, and move to start writing to other shards. With this automatic priming, you'll immediately have chunk ranges on all shards.

There are some limitations on what your shard key can be if you're using a hashed shard key. First, you cannot use the `unique` option. As with other shard keys, you cannot use array fields. Finally, be aware of is that floating point values will be rounded to whole numbers before hashing, so 1 and 1.999999 will both be hashed to the same value.

## Hashed Shard Keys for GridFS

Before attempting to shard GridFS collections, make sure that you understand how GridFS stores data (see [Chapter 6](#) for an explanation).

In the following explanation, the term “chunks” is overloaded since GridFS splits files into chunks and sharding splits collections into chunks. Thus, the two types of chunks are referred to as “GridFS chunks” and “sharding chunks” later in the chapter.

GridFS collections are generally excellent candidates for sharding, as they contain massive amounts of file data. However, neither of the indexes that are automatically created on `fs.chunks` are particularly good shard keys: `{"_id" : 1}` is an ascending key and `{"files_id" : 1, "n" : 1}` picks up `fs.files`' `_id` field, so it is also an ascending key.

However, if you create a hashed index on the `"files_id"` field, each file will be randomly distributed across the cluster. But a file will always be contained in a single chunk. This is the best of both worlds: writes will go to all shards evenly and reading a file's data will only ever have to hit a single shard.

To set this up, you must create a new index on `{"files_id" : "hashed"}` (as of this writing, `mongos` cannot use a subset of the compound index as a shard key). Then shard the collection on this field:

```
> db.fs.chunks.ensureIndex({ "files_id" : "hashed" })
> sh.shardCollection("test.fs.chunks", { "files_id" : "hashed" })
{ "collectionsharded" : "test.fs.chunks", "ok" : 1 }
```

As a side note, the *fs.files* collection may or may not need to be sharded, as it will be much smaller than *fs.chunks*. You can shard it if you would like, but it less likely to be necessary.

## The Firehose Strategy

If you have some servers that are more powerful than others, you might want to let them handle proportionally more load than your less-powerful servers. For example, suppose you have one shard that can handle 10 times the load of your other machines. Luckily, you have 10 other shards. You could force all inserts to go to the more powerful shard, and then allow the balancer to move older chunks to the other shards. This would give lower-latency writes.

To use this strategy, we have to pin the highest chunk to the more powerful shard. First, zone this shard:

```
> sh.addShardToZone("shard-name", "10x")
```

Now, pin the current value of the ascending key through infinity to that shard, so all new writes go to it:

```
> sh.updateZoneKeyRange("dbName.collName", { "_id" : ObjectId() },
... { "_id" : MaxKey}, "10x")
```

Now all inserts will be routed to this last chunk, which will always live on the shard zoned "10x".

However, ranges from now through infinity will be trapped on this shard unless we modify the zone key range. We could set up a cron job to update the key range once a day, like this:

```
> use config
> var zone = db.tags.findOne({ "ns" : "dbName.collName",
... "max" : { "shardKey" : MaxKey } })
> zone.min.shardKey = ObjectId()
> db.tags.save(zone)
```

Then all of the previous day's chunks would be able to move to other shards.

Another downside of this strategy is that it requires some changes to scale. If your most

powerful server can no longer handle the number of writes coming in, there is no trivial way to split the load between this server and another.

If you do not have a high-performance server to firehose into or you are not using zone sharding, do not use an ascending key as the shard key. If you do, all writes will go to a single shard.

## **Multi-Hotspot**

Standalone *mongod* servers are most efficient when doing ascending writes. This conflicts with sharding, in that sharding is most efficient when writes are spread over the cluster. This technique basically creates multiple hotspots—optimally several on each shard—so that writes are evenly balanced across the cluster but, within a shard, ascending.

To accomplish this, we use a compound shard key. The first value in the compound key is a rough, random value with low-ish cardinality. You can picture each value in the first part of the shard key as a chunk, as shown in [Figure 14-6](#). This will eventually work itself out as you insert more data, although it will probably never be divided up this neatly (right on the `$minKey` lines). However, if you insert enough data, you should eventually have approximately one chunk per random value. As you continue to insert data, you'll end up with multiple chunks with the same random value, which brings us to the second part of the shard key.

•  
•  
•

```
{"state": "KS", "_id": $minKey} ->  
 {"state": "KY", "_id": $minKey}
```

```
{"state": "KY", "_id": $minKey} ->  
 {"state": "LA", "_id": $minKey}
```

```
{"state": "LA", "_id": $minKey} ->  
 {"state": "MA", "_id": $minKey}
```

```
{"state": "MA", "_id": $minKey} ->  
 {"state": "MD", "_id": $minKey}
```

```
{"state": "MD", "_id": $minKey} ->  
 {"state": "ME", "_id": $minKey}
```

•  
•  
•

Figure 14-6. A subset of the chunks. Each chunk contains a single state and a range of \_ids

The second part of the shard key is an ascending key. This means that, within a chunk, values are always increasing, as shown in the sample documents in [Figure 14-7](#). Thus, if you had one chunk per shard, you'd have the perfect setup: ascending writes on every shard, as shown in [Figure 14-8](#). Of course having  $n$  chunks with  $n$  hotspots spread across  $n$  shards isn't very extensible: add a new shard and it won't get any writes because there's no hot chunk to put on it. Thus, you want a few hotspot chunks per shard (to give you room to grow). However, you don't want too many. A few hotspot chunks will keep the effectiveness of ascending writes. But having, say, a thousand "hotspots" on a shard will end up being equivalent to random writes.

```
{ "state" : "MA", "_id" : ObjectId("511bfb9e17d55c62b2371f1d") }
```

```
{ "state" : "NY", "_id" : ObjectId("511bfb9e17d55c62b2371f1e") }
```

```
{ "state" : "CA", "_id" : ObjectId("511bfb9e17d55c62b2371f1f") }
```

```
{ "state" : "NY", "_id" : ObjectId("511bfb9e17d55c62b2371f20") }
```

```
{ "state" : "MA", "_id" : ObjectId("511bfb9e17d55c62b2371f21") }
```

```
{ "state" : "MA", "_id" : ObjectId("511bfb9e17d55c62b2371f22") }
```

```
{ "state" : "NY", "_id" : ObjectId("511bfb9e17d55c62b2371f23") }
```

```
{ "state" : "CA", "_id" : ObjectId("511bfb9e17d55c62b2371f24") }
```

```
{ "state" : "CA", "_id" : ObjectId("511bfb9e17d55c62b2371f25") }
```

Figure 14-7. A sample list of inserted documents. Note that all \_ids are increasing.

Chunk:

```
{"state": "CA", "_id": $minKey} ->  
{"state": "CO", "_id": $minKey}
```

```
{ "state": "CA", "_id": ObjectId("511bfb9e17d55c62b2371f1f") }
```

```
{ "state": "CA", "_id": ObjectId("511bfb9e17d55c62b2371f24") }
```

```
{ "state": "CA", "_id": ObjectId("511bfb9e17d55c62b2371f25") }
```

Chunk:

```
{"state": "MA", "_id": $minKey} ->  
{"state": "ME", "_id": $minKey}
```

```
{ "state": "MA", "_id": ObjectId("511bfb9e17d55c62b2371f1d") }
```

```
{ "state": "MA", "_id": ObjectId("511bfb9e17d55c62b2371f21") }
```

```
{ "state": "MA", "_id": ObjectId("511bfb9e17d55c62b2371f22") }
```

Chunk:

```
{"state": "NY", "_id": $minKey} ->  
{"state": "OH", "_id": $minKey}
```

```
{ "state": "NY", "_id": ObjectId("511bfb9e17d55c62b2371f1e") }
```

```
{ "state": "NY", "_id": ObjectId("511bfb9e17d55c62b2371f20") }
```

```
{ "state": "NY", "_id": ObjectId("511bfb9e17d55c62b2371f23") }
```

*Figure 14-8. The inserted documents, split into chunks. Note that, within each chunk, the \_ids are increasing.*

You can picture this setup as each chunk being a stack of ascending documents. There are multiple stacks on each shard, each ascending until the chunk is split. Once a chunk is split, only one of the new chunks will be a hotspot chunk: the other chunk will essentially be “dead” and never grow again. If the stacks are evenly distributed across the shards, writes will be evenly distributed.

## Shard Key Rules and Guidelines

There are several practical restrictions to be aware of before choosing a shard key.

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 may just be the index you use most often (or some variation on it).

## Shard Key Limitations

Shard keys cannot be arrays. `sh.shardCollection()` will fail if any key has an array value and inserting an array into that field is not allowed.

Once inserted, a document’s shard key value cannot be modified. To change a document’s shard key, you must remove the document, change the key, and reinsert it. Thus, you should choose a field that is unchangeable or changes infrequently.

Most special types of index cannot be used for shard keys. In particular, you cannot shard on a geospatial index. Using a hashed index for a shard key is allowed, as covered previously.

## Shard Key Cardinality

Whether your shard key jumps around or increases steadily, it is important to choose a key with values that will vary. As with indexes, sharding performs better on high cardinality fields. If, for example, we had a `"logLevel"` key that had only values `"DEBUG"`, `"WARN"`, or `"ERROR"`,

MongoDB wouldn't be able to break up your data into more than three chunks (because there would be only three different values for the shard key). If you have a key with 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". It is important that the combination of keys has high cardinality.

## Controlling Data Distribution

Sometimes, automatic data distribution will not fit your requirements. This section gives you some options beyond choosing a shard key and allowing MongoDB to do everything automatically.

As your cluster gets larger or busier, these solutions become less practical. However, for small clusters, you may want more control.

### Using a Cluster for Multiple Databases and Collections

MongoDB evenly distributes collections across every shard your cluster, which works well if you're storing homogeneous data. However, if you have a log collection that is "lower-value" than your other data, you might not want it taking up space on your more expensive servers. Or, if you have one powerful shard, you might want to use it for only a realtime collection and not allow other collections to use it. You can set up separate clusters, but you can also give MongoDB specific directions about where you want it to put certain data.

To set this up, use the `sh.addShardToZone()` helper in the shell:

```
> sh.addShardToZone("shard0000", "high")
> // shard0001 - no zone
> // shard0002 - no zone
> // shard0003 - no zone
> sh.addShardToZone("shard0004", "low")
> sh.addShardToZone("shard0005", "low")
```

Then we can assign different collections to different shards. For instance, our realtime collection:

```
> sh.updateZoneKeyRange("super.important", {"shardKey" : MinKey,
... {"shardKey" : MaxKey}, "high")
```

This says, "for negative infinity to infinity for this collection, store it on shards tagged 'high'." This means that no data from the important collection will be stored on any other server. Note that this does not effect how other collections are distributed: other collections will still be

evenly distributed between this shard and the others.

We can perform a similar operation to keep the log collection on a low-quality server:

```
> sh.updateZoneKeyRange("some.logs", {"shardKey" : MinKey},  
... {"shardKey" : MaxKey}, "low")
```

The log collections will now be split evenly between shard0004 and shard0005.

Assigning a zone key range to a collection does not affect it instantly. It is an instruction to the balancer that, when it runs, these are the viable targets to move the collection to. Thus, if the entire log collection is on shard0002 or evenly distributed among the shards, it will take a little while for all of the chunks to be migrated to shard0004 and shard0005.

As another example, perhaps we have a collection where we don't want it on the shard zoned "high" but do want it on any other one. We can zone all of the non-high-performance shards to create a new grouping. Shards can have as many zones as you need:

```
> sh.addShardToZone("shard0001", "whatever")  
> sh.addShardToZone("shard0002", "whatever")  
> sh.addShardToZone("shard0003", "whatever")  
> sh.addShardToZone("shard0004", "whatever")  
> sh.addShardToZone("shard0005", "whatever")
```

Now we can specify that we want this collection (call it "*normal.coll*") distributed across these five shards:

```
> sh.updateZoneKeyRange("normal.coll", {"shardKey" : MinKey},  
... {"shardKey" : MaxKey}, "whatever")
```

You cannot assign collections dynamically, i.e., "when a collection is created, randomly home it to a shard." However, you could have a cron job that went through and did it for you.

If you make a mistake or change your mind, you can remove a shard from a zone with *sh.removeShardFromZone()*:

```
> sh.removeShardFromZone("shard0005", "whatever")
```

If you remove all shards from zones described by a zone key range (for example, if you remove shard0000 from the zone "high") the balancer won't distribute the data anywhere because

there aren't any valid locations listed. All the data will still be readable and writable; it just won't be able to migrate until you modify your tags or tag ranges.

To remove a key range from a zone, use `sh.removeRangeFromZone()`: The following is an example. The range specified must be an exact match to a range previously defined for the namespace `some.logs` and a given zone.

```
> sh.removeRangeFromZone("some.logs", {"shardKey" : MinKey},  
... {"shardKey" : MaxKey})
```

## Manual Sharding

Sometimes, for complex requirements or special situations, you may prefer to have complete control over which data is distributed where. You can turn off the balancer if you don't want data to be automatically distributed and use the `moveChunk` command to manually distribute data.

To turn off the balancer, connect to a `mongos` (any `mongos` is fine) using the `mongo shell` and disable the balancer using the shell helper `sh.stopBalancer` namespace with the following:

```
> sh.stopBalancer()
```

If there is currently a migrate in progress, this setting will not take effect until the migrate has completed. However, once any in-flight migrations have finished, the balancer will stop moving data around. To verify no migrations are in progress after disabling, issue the following in the `mongo shell`.

```
use config  
while(sh.isBalancerRunning()) {  
    print("waiting...");  
    sleep(1000);  
}
```

Once the balancer is off, you can move data around manually (if necessary). First, find out which chunks are where by looking at `config.chunks`:

```
> db.chunks.find()
```

Now, use the `moveChunk` command to migrate chunks to other shards. Specify the lower bound of the chunk-to-be-migrated and give the name of the shard that you want to move the

chunk to:

```
> sh.moveChunk(  
...  "test.manual.stuff",  
...  {user_id: NumberLong("-1844674407370955160")},  
...  "test-rs1")
```

However, unless you are in an exceptional situation, you should use MongoDB's automatic sharding instead of doing it manually. If you end up with a hotspot on a shard that you weren't expecting, you might end up with most of your data on that shard.

In particular, do not combine setting up unusual distributions manually with running the balancer. If the balancer detects an uneven number of chunks it will simply reshuffle all of your work to get the collection evenly balanced again. If you want uneven distribution of chunks, use the zone sharding technique discussed in [“Using a Cluster for Multiple Databases and Collections”](#).

# Chapter 15. Sharding Administration

---

As with replica sets, you have a number of options for administering sharded clusters. Manual administration is one option. These days it is becoming increasingly common to use tools such as Ops Manager and Cloud Manager, and the Atlas database as a service (DBaaS) offering for all cluster administration. In this chapter, we will demonstrate how to administer a sharded cluster manually, including:

- Inspecting the cluster's state: who its members are, where data is held, and what connections are open
- How to add, remove, and change members of a cluster
- Administering data movement and manually moving data

## Seeing the Current State

There are several helpers available to find out what data is where, what the shards are, and what the cluster is doing.

### Getting a Summary with `sh.status`

`sh.status()` gives you an overview of your shards, databases, and sharded collections. If you have a small number of chunks, it will print a breakdown of which chunks are where as well. Otherwise it will simply give the collection's shard key and how many chunks each shard has:

```
> sh.status()
--- Sharding Status ---
sharding version: {
  "_id" : 1,
  "minCompatibleVersion" : 5,
  "currentVersion" : 6,
  "clusterId" : ObjectId("5bdf51ecf8c192ed922f3160")
}
shards:
```

```

{
  "_id" : "shard01", "host" : "shard01/localhost:27018,localhost:27019"
{
  "_id" : "shard02", "host" : "shard02/localhost:27021,localhost:27022"
{
  "_id" : "shard03", "host" : "shard03/localhost:27024,localhost:27025"

active mongoses:
  "4.0.3" : 1

autosplit:
  Currently enabled: yes

balancer:
  Currently enabled: yes
  Currently running: no
  Failed balancer rounds in last 5 attempts: 0
  Migration Results for the last 24 hours:
    6 : Success

databases:
  { "_id" : "config", "primary" : "config", "partitioned" : true }
    config.system.sessions
      shard key: { "_id" : 1 }
      unique: false
      balancing: true
      chunks:
        shard01 1
        { "_id" : { "$minKey" : 1 } } -->> { "_id" : { "$maxKey" : 1 } }

  { "_id" : "video", "primary" : "shard02", "partitioned" : true,
    "version" : { "uuid" : UUID("3d83d8b8-9260-4a6f-8d28-c3732d40d961") }
    video.movies
      shard key: { "imdbId" : "hashed" }
      unique: false
      balancing: true
      chunks:
        shard01 3
        shard02 4
        shard03 3
        { "imdbId" : { "$minKey" : 1 } } -->> { "imdbId" : NumberLong("8345072417171006784") }
        { "imdbId" : NumberLong("-7262221363006655132") } -->> { "imdbId" : NumberLong("524277486033652998") }
        { "imdbId" : NumberLong("-5315530662268120007") } -->> { "imdbId" : NumberLong("4436141279217488250") }
        { "imdbId" : NumberLong("-3362204802044524341") } -->> { "imdbId" : NumberLong("2484315172280977547") }
        { "imdbId" : NumberLong("-1412311662519947087") } -->> { "imdbId" : NumberLong("6386258634539951337") }
        { "imdbId" : NumberLong("524277486033652998") } -->> { "imdbId" : NumberLong("3d83d8b8-9260-4a6f-8d28-c3732d40d961") }
        { "imdbId" : NumberLong("2484315172280977547") } -->> { "imdbId" : NumberLong("3d83d8b8-9260-4a6f-8d28-c3732d40d961") }
        { "imdbId" : NumberLong("4436141279217488250") } -->> { "imdbId" : NumberLong("3d83d8b8-9260-4a6f-8d28-c3732d40d961") }
        { "imdbId" : NumberLong("6386258634539951337") } -->> { "imdbId" : NumberLong("3d83d8b8-9260-4a6f-8d28-c3732d40d961") }
        { "imdbId" : NumberLong("8345072417171006784") } -->> { "imdbId" : NumberLong("3d83d8b8-9260-4a6f-8d28-c3732d40d961") }

```

Once there are more than a few chunks, `sh.status()` will summarize the chunk stats instead of printing each chunk. To see all chunks, run `sh.status(true)` (the `true` tells `sh.status()` to be verbose).

All the information `sh.status()` shows is gathered from your *config* database.

## Seeing Configuration Information

All of the configuration information about your cluster is kept in collections in the *config* database on the config servers. You can access it directly, but the shell has several helpers for exposing this information in a more readable way. However, you can always directly query the *config* database for metadata about your cluster.

## WARNING

Never connect directly to your config servers, as you do not want to take the chance of accidentally changing or removing config server data. Instead, connect to the *mongos* and use the *config* database to see its data, as you would for any other database:

```
mongos> use config
```

If you manipulate config data through *mongos* (instead of connecting directly to the config servers), *mongos* will ensure that all of your config servers stay in sync and prevent various dangerous actions like accidentally dropping the *config* database.

In general, you should not directly change any data in the *config* database (exceptions are noted below). If you change anything, you will generally have to restart all of your *mongos* servers to see its effect.

There are several collections in the *config* database. This section covers what each one contains and how it can be used.

## CONFIG.SHARDS

The *shards* collection keeps track of all the shards in the cluster. A typical document in the *shards* collection might look something like this:

```
> db.shards.find()
{ "_id" : "shard01", "host" : "shard01/localhost:27018,localhost:27019,localhost:27020", "state" : 1 }
{ "_id" : "shard02", "host" : "shard02/localhost:27021,localhost:27022,localhost:27023", "state" : 1 }
{ "_id" : "shard03", "host" : "shard03/localhost:27024,localhost:27025,localhost:27026", "state" : 1 }
```

The shard's `_id` is picked up from the replica set name, so each replica set in your cluster

must have a unique name.

When you update your replica set configuration (e.g., adding or removing members), the "host" field will be updated automatically.

## CONFIG.DATABASES

The *databases* collection keeps track of all of the databases, sharded and non, that the cluster knows about:

```
> db.databases.find()
{ "_id" : "video", "primary" : "shard02", "partitioned" : true,
  "version" : { "uuid" : UUID("3d83d8b8-9260-4a6f-8d28-c3732d40d961"), "la:
```

If *enableSharding* has been run on a database, "partitioned" will be true. The "primary" is the database's "home base." By default, all new collections in that database will be created on the database's primary shard.

## CONFIG.COLLECTIONS

The *collections* collection keeps track of all sharded collections (non-sharded collections are not shown). A typical document looks something like this:

```
> db.collections.find().pretty()
{
  "_id" : "config.system.sessions",
  "lastmodEpoch" : ObjectId("5bdf53122ad9c6907510c22d"),
  "lastmod" : ISODate("1970-02-19T17:02:47.296Z"),
  "dropped" : false,
  "key" : {
    "_id" : 1
  },
  "unique" : false,
  "uuid" : UUID("7584e4cd-fac4-4305-a9d4-bd73e93621bf")
}
{
  "_id" : "video.movies",
  "lastmodEpoch" : ObjectId("5bdf72c021b6e3be02fabe0c"),
  "lastmod" : ISODate("1970-02-19T17:02:47.305Z"),
  "dropped" : false,
  "key" : {
    "imdbID" : "hashed"
  },
  "unique" : false,
  "uuid" : UUID("e6580ffa-fcd3-418f-aa1a-0dfb71bc1c41")
}
```

The important fields are:

"\_id"

The namespace of the collection.

"key"

The shard key. In this case, it is a hashed shard key on "imdbId".

"unique"

Indicates that the shard key is not a unique index. By default, the shard key is not unique.

## CONFIG.CHUNKS

The *chunks* collection keeps a record of each chunk in all the collections. A typical document in the *chunks* collection looks something like this:

```
db.chunks.find().skip(1).limit(1).pretty()
{
    "_id" : "video.movies-imdbId_MinKey",
    "lastmod" : Timestamp(2, 0),
    "lastmodEpoch" : ObjectId("5bdf72c021b6e3be02fabe0c"),
    "ns" : "video.movies",
    "min" : {
        "imdbId" : { "$minKey" : 1 }
    },
    "max" : {
        "imdbId" : NumberLong("-7262221363006655132")
    },
    "shard" : "shard01",
    "history" : [
        {
            "validAfter" : Timestamp(1541370579, 3096),
            "shard" : "shard01"
        }
    ]
}
```

The most useful fields are:

"\_id"

The unique identifier for the chunk. Generally this is the namespace, shard key, and lower chunk boundary.

"ns"

The collection that this chunk is from.

"min"

The smallest value in the chunk's range (inclusive).

"max"

All values in the chunk are smaller than this value.

"shard"

Which shard the chunk resides on.

The "lastmod" field tracks chunk versioning. For example, if the chunk "video.movies-imdbId\_MinKey" split into two chunks, we'd want a way of distinguishing the new, smaller "video.movies-imdbId\_MinKey" chunks from their previous incarnation as a single chunk. Thus, the first component of the Timestamp value reflects the number of times a chunk has been migrated to a new shard. The second component of this value reflects splits. The "lastmodEpoch" field specifies the collection's creation epoch. It is used to differentiate requests for the same collection name in the cases where the collection was dropped and immediately recreated.

`sh.status()` uses the *config.chunks* collection to gather most of its information.

## CONFIG.CHANGELOG

The *changelog* collection is useful for keeping track of what a cluster is doing, since it records all of the splits and migrations that have occurred.

Splits are recorded in a document that looks like this:

```
> db.changelog.find({what: "split"}).pretty()
{
  "_id" : "router1-2018-11-05T09:58:58.915-0500-5be05ab2f8c192ed922f",
  "server" : "bob",
  "clientAddr" : "127.0.0.1:64621",
  "time" : ISODate("2018-11-05T14:58:58.915Z"),
  "what" : "split",
  "ns" : "video.movies",
  "details" : {
    "before" : {
      "min" : {
```

```

        "imdbID" : NumberLong("2484315172280977547"
    },
    "max" : {
        "imdbID" : NumberLong("4436141279217488250"
    },
    "lastmod" : Timestamp(9, 1),
    "lastmodEpoch" : ObjectId("5bdf72c021b6e3be02fabe0"
},
{
    "left" : {
        "min" : {
            "imdbID" : NumberLong("2484315172280977547"
        },
        "max" : {
            "imdbID" : NumberLong("3459137475094092005"
        },
        "lastmod" : Timestamp(9, 2),
        "lastmodEpoch" : ObjectId("5bdf72c021b6e3be02fabe0"
    },
    "right" : {
        "min" : {
            "imdbID" : NumberLong("3459137475094092005"
        },
        "max" : {
            "imdbID" : NumberLong("4436141279217488250"
        },
        "lastmod" : Timestamp(9, 3),
        "lastmodEpoch" : ObjectId("5bdf72c021b6e3be02fabe0"
    }
}
}

```

The "details" give information about what the original document looked like and what it split into.

This output is what the first chunk split of a collection looks like. Note that the second component of "lastmod" for each new chunk was updated so that the values are `Timestamp(9, 2)` and `Timestamp(9, 3)`, respectively.

Migrations are a bit more complicated and actually create four separate changelog documents: one noting the start of the migrate, one for the "from" shard, one for the "to" shard, and one for the commit that occurs when the migration is finalized. The middle two documents are of interest because these give a breakdown of how long each step in the process took. This can give you an idea whether it's the disk, network, or something else that is causing a bottleneck on migrates.

For example, the document created by the "from" shard looks like this:

```

> db.changelog.findOne({what: "moveChunk.to"})
{
    "_id" : "router1-2018-11-04T17:29:39.702-0500-5bdf72d32ad9c6907511",
    "server" : "bob",
    "clientAddr" : "",
    "time" : ISODate("2018-11-04T22:29:39.702Z"),
    "what" : "moveChunk.to",
    "ns" : "video.movies",
    "details" : {
        "min" : {
            "imdbId" : { "$minKey" : 1 }
        },
        "max" : {
            "imdbId" : NumberLong("-7262221363006655132")
        },
        "step 1 of 6" : 965,
        "step 2 of 6" : 608,
        "step 3 of 6" : 15424,
        "step 4 of 6" : 0,
        "step 5 of 6" : 72,
        "step 6 of 6" : 258,
        "note" : "success"
    }
}

```

Each of the steps listed in "details" is timed and the "stepN of 5" messages show how long the step took, in milliseconds.

When the "from" shard receives a *moveChunk* command from the *mongos*, it:

1. Checks the command parameters.
2. Confirms with the config servers that it can acquire a distributed lock for the migrate.
3. Tries to contact the "to" shard.
4. Copies the data. This is referred to and logged as “the critical section.”
5. Coordinates with the "to" shard and config servers to confirm the migration.

Note that the "to" and "from" shards must be in close communication starting at "step4 of 5": the shards directly talk to one another and the config server to perform the migration. If the "from" server has flaky network connectivity during the final steps, it may end up in a state where it cannot undo the migration and cannot move forward with it. In this case, the *mongod* will shut down.

The "to" shard's changelog document is similar to the "from" shard's, but the steps are a bit different. It looks like:

```
> db.changelog.find({what: "moveChunk.from", "details.max.imdbId": NumberLong(-7262221363006655132)}
```



```
{  
    "_id" : "router1-2018-11-04T17:29:39.753-0500-5bdf72d321b6e3be02fa",  
    "server" : "bob",  
    "clientAddr" : "127.0.0.1:64743",  
    "time" : ISODate("2018-11-04T22:29:39.753Z"),  
    "what" : "moveChunk.from",  
    "ns" : "video.movies",  
    "details" : {  
        "min" : {  
            "imdbId" : { "$minKey" : 1 }  
        },  
        "max" : {  
            "imdbId" : NumberLong("-7262221363006655132")  
        },  
        "step 1 of 6" : 0,  
        "step 2 of 6" : 4,  
        "step 3 of 6" : 191,  
        "step 4 of 6" : 17000,  
        "step 5 of 6" : 341,  
        "step 6 of 6" : 39,  
        "to" : "shard01",  
        "from" : "shard02",  
        "note" : "success"  
    }  
}
```

When the "to" shard receives a command from the "from" shard, it:

1. Migrates indexes. If this shard has never held chunks from the migrated collection before, it needs to know what fields are indexed. If this isn't the first time a chunk from this collection is being moved to this shard, then this should be a no-op.
2. Deletes any existing data in the chunk range. There might be data left over from a failed migration or restore procedure which we wouldn't want to interfere with the current data.
3. Copies all documents in the chunk to the "to" shard.
4. Replays any operations that happened to these document during the copy (on the "to" shard).
5. Waits for the "to" shard to have replicated the newly migrated data to a majority of servers.

6. Commits the migrate by changing the chunk's metadata to say that it lives on the "to" shardconfig.tags

## CONFIG.SETTINGS

This collection contains documents representing the current balancer settings and chunk size. By changing the documents in this collection, you can turn the balancer on or off or change the chunk size. Note that you should always connect to *mongos*, not the config servers directly, to change values in this collection.

## Tracking Network Connections

There are a lot of connections between the components of a cluster. This section covers some sharding-specific information.

### Getting Connection Statistics

The command, *connPoolStats*, returns information regarding the open outgoing connections from the current database instance to other members of the sharded cluster or replica set.

To avoid interference with any running operations, *connPoolStats* does not take any locks. As such, the counts may change slightly as *connPoolStats* gathers information, resulting in slight differences between the hosts and pools connection counts.

```
> db.adminCommand({ "connPoolStats": 1 })
{
  "numClientConnections" : 10,
  "numAScopedConnections" : 0,
  "totalInUse" : 0,
  "totalAvailable" : 13,
  "totalCreated" : 86,
  "totalRefreshing" : 0,
  "pools" : {
    "NetworkInterfaceTL-TaskExecutorPool-0" : {
      "poolInUse" : 0,
      "poolAvailable" : 2,
      "poolCreated" : 2,
      "poolRefreshing" : 0,
      "localhost:27027" : {
        "inUse" : 0,
        "available" : 1,
        "created" : 1,
        "refreshing" : 0
      },
      "localhost:27019" : {
        "inUse" : 0,
        "available" : 1,
        "created" : 1,
        "refreshing" : 0
      }
    }
}
```

```
        "refreshing" : 0
    }
},
"NetworkInterfaceTL-ShardRegistry" : {
    "poolInUse" : 0,
    "poolAvailable" : 1,
    "poolCreated" : 13,
    "poolRefreshing" : 0,
    "localhost:27027" : {
        "inUse" : 0,
        "available" : 1,
        "created" : 13,
        "refreshing" : 0
    }
},
"global" : {
    "poolInUse" : 0,
    "poolAvailable" : 10,
    "poolCreated" : 71,
    "poolRefreshing" : 0,
    "localhost:27026" : {
        "inUse" : 0,
        "available" : 1,
        "created" : 8,
        "refreshing" : 0
    },
    "localhost:27027" : {
        "inUse" : 0,
        "available" : 1,
        "created" : 1,
        "refreshing" : 0
    },
    "localhost:27023" : {
        "inUse" : 0,
        "available" : 1,
        "created" : 7,
        "refreshing" : 0
    },
    "localhost:27024" : {
        "inUse" : 0,
        "available" : 1,
        "created" : 6,
        "refreshing" : 0
    },
    "localhost:27022" : {
        "inUse" : 0,
        "available" : 1,
        "created" : 9,
        "refreshing" : 0
    },
    "localhost:27019" : {
        "inUse" : 0,
        "available" : 1,
        "created" : 8,
```

```
        "refreshing" : 0
    },
    "localhost:27021" : {
        "inUse" : 0,
        "available" : 1,
        "created" : 8,
        "refreshing" : 0
    },
    "localhost:27025" : {
        "inUse" : 0,
        "available" : 1,
        "created" : 9,
        "refreshing" : 0
    },
    "localhost:27020" : {
        "inUse" : 0,
        "available" : 1,
        "created" : 8,
        "refreshing" : 0
    },
    "localhost:27018" : {
        "inUse" : 0,
        "available" : 1,
        "created" : 7,
        "refreshing" : 0
    }
}
},
"hosts" : {
    "localhost:27026" : {
        "inUse" : 0,
        "available" : 1,
        "created" : 8,
        "refreshing" : 0
    },
    "localhost:27027" : {
        "inUse" : 0,
        "available" : 3,
        "created" : 15,
        "refreshing" : 0
    },
    "localhost:27023" : {
        "inUse" : 0,
        "available" : 1,
        "created" : 7,
        "refreshing" : 0
    },
    "localhost:27024" : {
        "inUse" : 0,
        "available" : 1,
        "created" : 6,
        "refreshing" : 0
    },
    "localhost:27022" : {
```

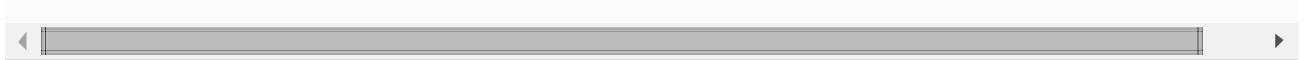
```
        "inUse" : 0,
        "available" : 1,
        "created" : 9,
        "refreshing" : 0
    },
    "localhost:27019" : {
        "inUse" : 0,
        "available" : 2,
        "created" : 9,
        "refreshing" : 0
    },
    "localhost:27021" : {
        "inUse" : 0,
        "available" : 1,
        "created" : 8,
        "refreshing" : 0
    },
    "localhost:27025" : {
        "inUse" : 0,
        "available" : 1,
        "created" : 9,
        "refreshing" : 0
    },
    "localhost:27020" : {
        "inUse" : 0,
        "available" : 1,
        "created" : 8,
        "refreshing" : 0
    },
    "localhost:27018" : {
        "inUse" : 0,
        "available" : 1,
        "created" : 7,
        "refreshing" : 0
    }
},
"replicaSets" : {
    "shard02" : {
        "hosts" : [
            {
                "addr" : "localhost:27021",
                "ok" : true,
                "ismaster" : true,
                "hidden" : false,
                "secondary" : false,
                "pingTimeMillis" : 0
            },
            {
                "addr" : "localhost:27022",
                "ok" : true,
                "ismaster" : false,
                "hidden" : false,
                "secondary" : true,
                "pingTimeMillis" : 0
            }
        ]
    }
}
```

```
        },
        {
            "addr" : "localhost:27023",
            "ok" : true,
            "ismaster" : false,
            "hidden" : false,
            "secondary" : true,
            "pingTimeMillis" : 0
        }
    ]
},
"shard03" : {
    "hosts" : [
        {
            "addr" : "localhost:27024",
            "ok" : true,
            "ismaster" : false,
            "hidden" : false,
            "secondary" : true,
            "pingTimeMillis" : 0
        },
        {
            "addr" : "localhost:27025",
            "ok" : true,
            "ismaster" : true,
            "hidden" : false,
            "secondary" : false,
            "pingTimeMillis" : 0
        },
        {
            "addr" : "localhost:27026",
            "ok" : true,
            "ismaster" : false,
            "hidden" : false,
            "secondary" : true,
            "pingTimeMillis" : 0
        }
    ]
},
"configRepl" : {
    "hosts" : [
        {
            "addr" : "localhost:27027",
            "ok" : true,
            "ismaster" : true,
            "hidden" : false,
            "secondary" : false,
            "pingTimeMillis" : 0
        }
    ]
},
"shard01" : {
    "hosts" : [
        {
```

```

        "addr" : "localhost:27018",
        "ok" : true,
        "ismaster" : false,
        "hidden" : false,
        "secondary" : true,
        "pingTimeMillis" : 0
    } ,
    {
        "addr" : "localhost:27019",
        "ok" : true,
        "ismaster" : true,
        "hidden" : false,
        "secondary" : false,
        "pingTimeMillis" : 0
    } ,
    {
        "addr" : "localhost:27020",
        "ok" : true,
        "ismaster" : false,
        "hidden" : false,
        "secondary" : true,
        "pingTimeMillis" : 0
    }
]
}
},
"ok" : 1,
"operationTime" : Timestamp(1541440424, 1),
"$clusterTime" : {
    "clusterTime" : Timestamp(1541440424, 1),
    "signature" : {
        "hash" : BinData(0, "AAAAAAAAAAAAAAAAAAAAA="),
        "keyId" : NumberLong(0)
    }
}
}

```



`totalAvailable` shows the total number of available outgoing connections from the current mongod/mongos instance to other members of the sharded cluster or replica set.

`totalCreated` reports the total number of outgoing connections ever created by the current mongod/mongos instance to other members of the sharded cluster or replica set.

`totalInUse` provides the total number of outgoing connections from the current mongod/mongos instance to other members of the sharded cluster or replica set that are currently in use.

`totalRefreshing` displays the total number of outgoing connections from the current

`mongod/mongos` instance to other members of the sharded cluster or replica set that are currently being refreshed.

`numClientConnections` identifies the number of active and stored outgoing synchronous connections from the current `mongod/mongos` instance to other members of the sharded cluster or replica set. These connections are a part of a pool that is a subset of the data reported by `totalAvailable`, `totalCreated`, and `totalInUse`.

`numAScopedConnection` reports the number of active and stored outgoing scoped synchronous connections from the current `mongod/mongos` instance to other members of the sharded cluster or replica set. These connections are a part of a pool that is a subset of the data reported by `totalAvailable`, `totalCreated`, and `totalInUse`.

The `pools` field shows connection statistics (in use/available/created/refreshing) grouped by the connection pools. A `mongod` or `mongos` has two distinct families of outgoing connection pools:

- DBClient-based pools (the “write path”, identified by the field name, “global” in the “pools” document) and
- NetworkInterfaceTL-based pools (the “read path”).

The `hosts` field reports on connection statistics (in use/available/created/refreshing) grouped by the hosts. This field contains documents that represent a report of connections between the current `mongod/mongos` instance and each member of the sharded cluster or replica set.

You might see connections to other shards in the output of `connPoolStats`. These indicate that shards are connecting to other shards to migrate data. The primary of one shard will connect directly to the primary of another shard and “suck” its data.

When a migrate occurs, a shard sets up a `ReplicaSetMonitor` (a process that monitors replica set health) to track the health of the shard on the other side of the migrate. `mongod` never destroys this monitor, so you may see messages in one replica set’s log about the members of another replica set. This is totally normal and should have no effect on your application.

## **Limiting the Number of Connections**

When a client connects to a `mongos`, `mongos` creates a connection to at least one shard to pass along the client’s request. Thus, every client connection into a `mongos` yields at least one outgoing connection from `mongos` to the shards.

If you have many `mongos` processes, they may create more connections than your shards can

handle: by default a *mongos* will accept up to 65536 connections (same as *mongod*), so if you have 5 *mongos* processes with 10,000 client connections each, they may be attempting to create 50,000 connections to a shard!

To prevent this, you can use the `maxConns` option to your command line configuration for *mongos* to limit the number of connections it can create. The following formula can be used to calculate the maximum number of connections a shard can handle from a single *mongos*:

$$\text{maxConns} = 20,000 - (\text{numMongosProcesses} \times 3) - (\text{numMembersPerReplicaSet} \times 3) - (\text{other} / \text{numMongosProcesses})$$

Breaking down the pieces of this formula:

$$(\text{numMongosProcesses} \times 3)$$

Each *mongos* creates three connections per *mongod*: a connection to forward client requests, an error-tracking connection (the writeback listener), and a connection to monitor the replica set's status.

$$(\text{numMembersPerReplicaSet} \times 3)$$

The primary creates a connection to each secondary and each secondary creates two connections to the primary, for a total of three connections.

$$(\text{other} / \text{numMongosProcesses})$$

`other` is the number of miscellaneous processes that may connect to your *mongods*, such as monitoring or backup agents, direct shell connections (for administration), or connections to other shards for migrations.

Note that `maxConns` only prevents *mongos* from creating more than this many connections. It doesn't mean that it does anything particularly helpful when it runs out of connections: it will block requests, waiting for connections to be "freed." Thus, you must prevent your application from using this many connections, especially as your number of *mongos* processes grows.

When a MongoDB instance exits cleanly it closes all connections before stopping. The members who were connected to it will immediately get socket errors on those connections and be able to refresh them. However, if a MongoDB instance suddenly goes offline due to a power loss, crash, or network problems, it probably won't cleanly close all of its sockets. In this case, other servers in the cluster may be under the impression that their connection is healthy until they try to perform an operation on it. At that point, they will get an error and refresh the connection (if

the member is up again at that point).

This is a quick process when there are only a few connections. However, when there are thousands of connections that must be refreshed one by one you can get a lot of errors because each connection to the downed member must be tried, determined to be bad, and re-established. There isn't a particularly good way of preventing this aside from restarting processes that get bogged down in a reconnection storm.

## Server Administration

As your cluster grows, you'll need to add capacity or change configurations. This section covers how to add and remove servers from your cluster.

### Adding Servers

You can add new *mongos* processes at any time. Make sure their `--configdb` option specifies the correct set of config servers and they should be immediately available for clients to connect to.

To add new shards, use the "addShard" command as shown in [Chapter 13](#).

### Changing Servers in a Shard

As you use your sharded cluster, you may want to change the servers in individual shards. To change a shard's membership, connect directly to the shard's primary (not through the *mongos*) and issue a replica set reconfig. The cluster configuration will pick up the change and update *config.shards* automatically. Do not modify *config.shards* by hand.

The only exception to this is if you started your cluster with standalone servers as shards, not replica sets.

### CHANGING A SHARD FROM A STANDALONE SERVER TO REPLICA SET

The easiest way to do this is to add a new, empty replica set shard and then remove the standalone server shard (which we'll discuss in the next section). Migrations will take care of moving your data to the new shard.

### Removing a Shard

In general, shards should not be removed from a cluster. If you are regularly adding and removing shards, you are putting a lot more stress on the system than necessary. If you add too many shards it is better to let your system grow into it, not remove them and add them back later. However, if necessary, you can remove shards.

First make sure that the balancer is on. The balancer will be tasked with moving all the data on this shard to other shards in a process called draining. To start draining, run the `removeShard` command. `removeShard` takes the shard's name and drains all the chunks on a given shard to the other shards:

```
> db.adminCommand({ "removeShard" : "shard03" })
{
  "msg" : "draining started successfully",
  "state" : "started",
  "shard" : "shard03",
  "note" : "you need to drop or movePrimary these databases",
  "dbsToMove" : [ ],
  "ok" : 1,
  "operationTime" : Timestamp(1541450091, 2),
  "$clusterTime" : {
    "clusterTime" : Timestamp(1541450091, 2),
    "signature" : {
      "hash" : BinData(0, "AAAAAAAAAAAAAAAAAAAAAAA="),
      "keyId" : NumberLong(0)
    }
  }
}
```

Draining can take a long time if there are a lot of chunks or large chunks to move. If you have jumbo chunks (see “[Jumbo Chunks](#)”), you may have to temporarily raise the chunk size to allow draining to move them.

If you want to keep tabs on how much has been moved, run `removeShard` again to give you the current status:

```
> db.adminCommand({ "removeShard" : "shard02" })
{
  "msg" : "draining ongoing",
  "state" : "ongoing",
  "remaining" : {
    "chunks" : NumberLong(3),
    "dbs" : NumberLong(0)
  },
  "note" : "you need to drop or movePrimary these databases",
  "dbsToMove" : [
    "video"
  ],
  "ok" : 1,
  "operationTime" : Timestamp(1541450139, 1),
  "$clusterTime" : {
    "clusterTime" : Timestamp(1541450139, 1),
    "signature" : {
```

```
        "hash" : BinData(0,"AAAAAAAAAAAAAAA="),
        "keyId" : NumberLong(0)
    }
}

```

You can run `removeShard` as many times as you want.

Chunks may have to split to be moved, so you may see the number of chunks increase in the system during the drain. For example, suppose we have a 5-shard cluster with the following chunk distributions:

test-rs0	10
test-rs1	10
test-rs2	10
test-rs3	11
test-rs4	11

This cluster has a total of 52 chunks. If we remove `test-rs3`, we might end up with:

test-rs0	15
test-rs1	15
test-rs2	15
test-rs4	15

The cluster now has 60 chunks, 18 of which came from shard `test-rs3` (11 were there to start and 7 were created from draining splits).

Once all the chunks have been moved, if there are still databases that have the removed shard as their primary, you'll need to remove them before the shard can be removed. Each database in a sharded cluster has a primary shard. If the shard you want to remove is also the primary of one of the cluster's databases, `removeShard` lists the database in the `dbsToMove` field. To finish removing the shard, you must either move the database to a new shard after migrating all data from the shard or drop the database, deleting the associated data files. The output of `removeShard` will be something like:

```
> db.adminCommand({"removeShard" : "shard02"})
{
    "msg" : "draining ongoing",
    "state" : "ongoing",
    "remaining" : {
        "chunks" : NumberLong(3),
        "dbs" : NumberLong(0)
```

```
},
"note" : "you need to drop or movePrimary these databases",
"dbsToMove" : [
    "video"
],
"ok" : 1,
"operationTime" : Timestamp(1541450139, 1),
"$clusterTime" : {
    "clusterTime" : Timestamp(1541450139, 1),
    "signature" : {
        "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAA="),
        "keyId" : NumberLong(0)
    }
}
}
```

To finish the remove, move the homed databases with the *movePrimary* command:

```
> db.adminCommand({ "movePrimary" : "video", "to" : "shard01" })
{
    "ok" : 1,
    "operationTime" : Timestamp(1541450554, 12),
    "$clusterTime" : {
        "clusterTime" : Timestamp(1541450554, 12),
        "signature" : {
            "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAA="),
            "keyId" : NumberLong(0)
        }
    }
}
```

Once you have moved any databases, run *removeShard* one more time:

```
> db.adminCommand({ "removeShard" : "shard02" })
{
    "msg" : "removeshard completed successfully",
    "state" : "completed",
    "shard" : "shard03",
    "ok" : 1,
    "operationTime" : Timestamp(1541450619, 2),
    "$clusterTime" : {
        "clusterTime" : Timestamp(1541450619, 2),
        "signature" : {
            "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAA="),
            "keyId" : NumberLong(0)
        }
    }
}
```

This is not strictly necessary, but it confirms that you have completed the process. If there are no databases that have this shard as primary, you will get this response as soon as all chunks have been migrated off.

Once you have started a shard draining, there is no built-in way to stop it.

## Balancing Data

In general, MongoDB automatically takes care of balancing data. This section covers how to enable and disable this automatic balancing as well as how to intervene in the balancing process.

### The Balancer

Turning off the balancer is a prerequisite to nearly any administrative activity. There is a shell helper to make this easier:

```
> sh.setBalancerState(false)
{
    "ok" : 1,
    "operationTime" : Timestamp(1541450923, 2),
    "$clusterTime" : {
        "clusterTime" : Timestamp(1541450923, 2),
        "signature" : {
            "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAA="),
            "keyId" : NumberLong(0)
        }
    }
}
```

With the balancer off a new balancing round will not begin, but it will not force an ongoing balancing round to stop immediately: migrations generally cannot stop on a dime. Thus, you should check the *config.locks* collection to see whether or not a balancing round is still in progress:

```
> db.locks.find({"_id" : "balancer"})["state"]
0
```

0 means the balancer is off.

Balancing puts load on your system: the destination shard must query the source shard for all the documents in a chunk, insert them, and then the source shard must delete them. There are

two circumstances in particular where migrations can cause performance problems:

1. Using a hotspot shard key will force constant migrations (as all new chunks will be created on the hotspot). Your system must have the capacity to handle the flow of data coming off of your hotspot shard.
2. Adding a new shard will trigger a stream of migrations as the balancer attempts to populate it.

If you find that migrations are affecting your application's performance, you can schedule a window for balancing in the `config.settings` collection. Run the following update to only allow balancing between 1 p.m. and 4 p.m. First make sure the balancer is on. Then schedule the window.

```
> sh.setBalancerState( true )
{
  "ok" : 1,
  "operationTime" : Timestamp(1541451846, 4),
  "$clusterTime" : {
    "clusterTime" : Timestamp(1541451846, 4),
    "signature" : {
      "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAA="),
      "keyId" : NumberLong(0)
    }
  }
}
> db.settings.update(
  { _id: "balancer" },
  { $set: { activeWindow : { start : "13:00", stop : "16:00" } } },
  { upsert: true }
)
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

If you set a balancing window, monitor it closely to ensure that `mongos` can actually keep your cluster balanced in the time that you have allotted it.

You must be careful if you plan to combine manual balancing with the automatic balancer, since the automatic balancer always determines what to move based on the current state of the set and does not take into account the set's history. For example, suppose you have `shardA` and `shardB`, each holding 500 chunks. `shardA` is getting a lot of writes, so you turn off the balancer and move 30 of the most active chunks to `shardB`. If you turn the balancer back on at this point, it will immediately swoop in and move 30 chunks (possibly a different 30) back from `shardB` to `shardA` to balance the chunk counts.

To prevent this, move 30 quiescent chunks from *shardB* to *shardA* before starting the balancer. That way there will be no imbalance between the shards and the balancer will be happy to leave things as they are. Alternatively, you could perform 30 splits on *shardA*'s chunks to even out the chunk counts.

Note that the balancer only uses number of chunks as a metric, not size of data. Moving a chunk is called a migration and is how MongoDB balances data across your cluster. Thus, a shard with a few large chunks may end up as the target of a migration from a shard with many small chunks (but a smaller data size).

## Changing Chunk Size

There can be anywhere from zero to millions of documents in a chunk. Generally, the larger a chunk is, the longer it takes to migrate to another shard. In [Chapter 12](#), we used a chunk size of 1 MB, so that we could see chunk movement easily and quickly. This is generally impractical in a live system. MongoDB would be doing a lot of unnecessary work to keep shards within a few megabytes of each other in size. By default, chunks are 64 MB, which is generally a good balance between ease of migration and migratory churn.

Sometimes you may find that migrations are taking too long with 64 MB chunks. To speed them up, you can decrease your chunk size. To do this, connect to *mongos* through the shell and update the *config.settings* collection:

```
> db.settings.findOne()
{
  "_id" : "chunksize",
  "value" : 64
}
> db.settings.save({"_id" : "chunksize", "value" : 32})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

The previous update would change your chunk size to 32 MB. Existing chunks would not be changed immediately. Automatic splitting only occurs on insert or update. If you lower the chunk size, it may take time for all chunks to split to the new size. Splits cannot be undone. If you increase the chunk size, existing chunks grow only through insertion or updates until they reach the new size. The allowed range of the chunk size is between 1 and 1024 megabytes, inclusive.

Note that this is a cluster-wide setting: it affects all collections and databases. Thus, if you need a small chunk size for one collection and a large chunk size for another, you may have to compromise with a chunk size in between the two ideals (or put the collections in different clusters).

If MongoDB is doing too many migrations or your documents are large, you may want to increase your chunk size.

## Moving Chunks

As mentioned earlier, all the data in a chunk lives on a certain shard. If that shard ends up with more chunks than the other shards, MongoDB will move some chunks off it.

You can manually move chunks using the *moveChunk* shell helper:

```
> sh.moveChunk("video.movies", {imdbId: 500000}, "shard02")
{ "millis" : 4079, "ok" : 1 }
```

This would move the chunk containing the document with "imdbId" of 500000 to the shard named "shard02". You must use the shard key to find which chunk to move ("imdbId", in this case). Generally, the easiest way to specify a chunk is by its lower bound, although any value in the chunk will work (the upper bound will not, as it is not actually in the chunk). This command will move the chunk before returning, so it may take a while to run. The logs are the best place to see what it is doing if it takes a long time.

If a chunk is larger than the max chunk size, *mongos* will refuse to move it:

```
> sh.moveChunk("video.movies", {imdbId: NumberLong("8345072417171006784")})
{
  "cause" : {
    "chunkTooBig" : true,
    "estimatedChunkSize" : 2214960,
    "ok" : 0,
    "errmsg" : "chunk too big to move"
  },
  "ok" : 0,
  "errmsg" : "move failed"
}
```

In this case, you must manually split the chunk before moving it, using the *splitAt* command:

```
> db.chunks.find({ns: "video.movies", "min.imdbId": NumberLong("6386258634539951337")})
{
  "_id" : "video.movies-imdbId_6386258634539951337",
  "ns" : "video.movies",
  "min" : {
    "imdbId" : NumberLong("6386258634539951337")
  },
  "max" : {
```

```

        "imdbId" : NumberLong("8345072417171006784")
    },
    "shard" : "shard02",
    "lastmod" : Timestamp(1, 9),
    "lastmodEpoch" : ObjectId("5bdf72c021b6e3be02fabe0c"),
    "history" : [
        {
            "validAfter" : Timestamp(1541370559, 4),
            "shard" : "shard02"
        }
    ]
}
> sh.splitAt("video.movies", {"imdbId": NumberLong("70000000000000000000")})
{
    "ok" : 1,
    "operationTime" : Timestamp(1541453304, 1),
    "$clusterTime" : {
        "clusterTime" : Timestamp(1541453306, 5),
        "signature" : {
            "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAA="),
            "keyId" : NumberLong(0)
        }
    }
}
> db.chunks.find({ns: "video.movies", "min.imdbId": NumberLong("6386258634")})
{
    "_id" : "video.movies-imdbId_6386258634539951337",
    "lastmod" : Timestamp(15, 2),
    "lastmodEpoch" : ObjectId("5bdf72c021b6e3be02fabe0c"),
    "ns" : "video.movies",
    "min" : {
        "imdbId" : NumberLong("6386258634539951337")
    },
    "max" : {
        "imdbId" : NumberLong("70000000000000000000")
    },
    "shard" : "shard02",
    "history" : [
        {
            "validAfter" : Timestamp(1541370559, 4),
            "shard" : "shard02"
        }
    ]
}

```

Once the chunk has been split into smaller pieces, it should be movable. Alternatively, you can raise the max chunk size and then move it, but you should break up large chunks whenever possible. Sometimes, though, they cannot be broken up: these are called jumbo chunks.

## Jumbo Chunks

Suppose you choose the "date" field as your shard key. The "date" field in this collection is a string that looks like "year/month/day", which means that *mongos* can create at most one chunk per day. This works fine for a while, until your application suddenly goes viral and gets a thousand times its typical traffic for one day.

This day's chunk is going to be much larger than any other day's, but it is also completely unsplittable because every document has the same value for the shard key.

Once a chunk is larger than the max chunk size set in *config.settings*, the balancer will not be allowed to move the chunk. These unsplittable, unmovable chunks are called jumbo chunks and they are inconvenient to deal with.

Let's take an example. Suppose there are three shards, *shard1*, *shard2*, and *shard3*. If you use the hotspot shard key pattern described in “[Ascending Shard Keys](#)”, all your writes will be going to one shard, say *shard1*. *mongos* will try to balance the number of chunks evenly between the shards. But the only chunks that the balancer can move are the non-jumbo chunks, so it will migrate all the small chunks off the hot shard.

Now all the shards will have roughly the same number of chunks, but all of *shard2* and *shard3*'s chunks will be less than 64 MB in size. And if jumbo chunks are being created, more and more of *shard1*'s chunks will be more than 64 MB in size. Thus, *shard1* will fill up a lot faster than the other two shards, even though the number of chunks is perfectly balanced between the three.

Thus, one of the indicators that you have jumbo chunk problems is that one shard's size is growing much faster than the others. You can also look at `sh.status()` to see if you have jumbo chunks: they will be marked with a "jumbo" attribute:

```
> sh.status()
...
{
  "x" : -7 } --> { "x" : 5 } on : shard0001
{
  "x" : 5 } --> { "x" : 6 } on : shard0001 jumbo
{
  "x" : 6 } --> { "x" : 7 } on : shard0001 jumbo
{
  "x" : 7 } --> { "x" : 339 } on : shard0001
...
```

You can use the `dataSize` command to check chunk sizes.

First, we use the *config.chunks* collection to find the chunk ranges:

```
> use config
> var chunks = db.chunks.find({ "ns" : "acme.analytics" }).toArray()
```

Use these chunk ranges to find possible jumbo chunks:

```
> use dbName
> db.runCommand({ "dataSize" : "dbName.collName",
... "keyPattern" : {"date" : 1}, // shard key
... "min" : chunks[0].min,
... "max" : chunks[0].max})
{
    "size" : 33567917,
    "numObjects" : 108942,
    "millis" : 634,
    "ok" : 1,
    "operationTime" : Timestamp(1541455552, 10),
    "$clusterTime" : {
        "clusterTime" : Timestamp(1541455552, 10),
        "signature" : {
            "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAA="),
            "keyId" : NumberLong(0)
        }
    }
}
```

Be careful—the `dataSize` command does have to scan the chunk’s data to figure out how big it is. If you can, narrow down your search by using your knowledge of your data: were jumbo chunks created on certain date? For example, if there was a really busy day on November 1, look for chunks with that day in their shard key range. If you’re using GridFS and sharding by `"files_id"`, you can look at the `fs.files` collection to find a file’s size.

## DISTRIBUTING JUMBO CHUNKS

To fix a cluster thrown off-balance by jumbo chunks, you must evenly distribute them among the shards.

This is a complex manual process, but should not cause any downtime (it may cause slowness, as you’ll be migrating a lot of data). In the description below, the shard with the jumbo chunks is referred to as the “from” shard. The shards that the jumbo chunks are migrated to are called the “to” shards. Note that you may have multiple “from” shards that you wish to move chunks off of. Repeat these steps for each:

1. Turn off the balancer. You don’t want to the balancer trying to “help” during this process:

```
> sh.setBalancerState(false)
```

2. MongoDB will not allow you to move chunks larger than the max chunk size, so

temporarily raise the chunk size. Make a note of what your original chunk size is and then change it to something large, like 10,000. Chunk size is specified in megabytes:

```
> use config
> db.settings.findOne({ "_id" : "chunksize" })
{
  "_id" : "chunksize",
  "value" : 64
}
> db.settings.save({ "_id" : "chunksize", "value" : 10000 })
```

3. Use the `moveChunk` command to move jumbo chunks off the “from” shard.
4. Run `splitChunk` on the remaining chunks on the donor shard until it has a roughly even number of chunks as the other shards.
5. Set chunk size back to its original value:

```
> db.settings.save({ "_id" : "chunksize", "value" : 64 })
```

6. Turn on the balancer:

```
> sh.setBalancerState(true)
```

When the balancer is turned on again it cannot move the jumbo chunks again, as they are essentially held in place by their size.

## PREVENTING JUMBO CHUNKS

As the amount of data you are storing grows, the manual process described in the previous section becomes unsustainable. Thus, if you’re having problems with jumbo chunks, you should make it a priority to prevent them from forming.

To prevent jumbo chunks, modify your shard key to have more granularity. You want almost every document to have a unique value for the shard key, or at least never have more than `chunksize`-worth of data with a single shard key value.

For example, if you were using the year/month/day key described earlier it can quickly be made finer-grained by adding hours, minutes, and seconds. Similarly, if you’re sharding on something coarsely-grained like log level, add a second field to your shard key with a lot of granularity, such as an MD5 hash or UUID. Then you can always split a chunk, even if the first field is the same for many documents.

## Refreshing Configurations

As a final tip, sometimes *mongos* will not update its configuration correctly from the config servers. If you ever get a configuration that you don't expect or a *mongos* seems to be out of date or cannot find data that you know is there, use the *flushRouterConfig* command to manually clear all caches:

```
> db.adminCommand({ "flushRouterConfig" : 1 })
```

If *flushRouterConfig* does not work, restarting all your *mongos* or *mongod* processes clears any possible cache.

# Chapter 16. An Introduction to MongoDB Security

---

To protect your MongoDB cluster and the data it holds, you will want to employ the following security measures:

- Enable authorization and enforce authentication
- Encrypt communication
- Encrypt data

Throughout this chapter we will demonstrate how to address the first two security measures with a tutorial on using MongoDB's support for x.509 to configure authentication and transport layer encryption to ensure secure communications among clients and servers in a MongoDB replica set. We will touch on encrypting data at the storage layer in a later chapter.

## Authentication Mechanisms

Enabling authorization on a MongoDB cluster enforces authentication and ensures users can only perform actions they are authorized for as determined by their roles. The community version of MongoDB provides support for SCRAM (Salted Challenge Response Authentication Mechanism) and x.509 Certificate Authentication. In addition to SCRAM and x.509, MongoDB Enterprise supports Kerberos authentication and LDAP Proxy Authentication. See the MongoDB documentation for details on the various authentication mechanisms that MongoDB supports <https://docs.mongodb.com/manual/core/authentication/>. In this chapter, we will dive deeper on x.509 authentication. An x.509 digital certificate uses the widely accepted x.509 public key infrastructure (PKI) standard to verify that a public key belongs to the presenter.

## Using x.509 Certificates to Authenticate Both Members and Clients

Given that all production MongoDB clusters are composed of multiple members, to secure a cluster, it is essential that all services communicating within the cluster authenticate with one another. Each member of a replica set must authenticate with the others in order to exchange

data. Likewise, clients must authenticate with the primary and any secondaries that they communicate with.

For x.509, it's necessary that a trusted certification authority (CA) sign all certificates. Signing certifies that the named subject of a certificate owns the public key associated with that certificate. A CA acts as a trusted third party to prevent man-in-the-middle attacks.

The figure below depicts x.509 authentication used to secure a three member MongoDB replica set. Note the authentication among the client and members of the replica set and the trust relationships with the CA.

Note that member and the client each have their own certificate signed by the CA. For production use, your MongoDB deployment should use valid certificates generated and signed by a single certificate authority. You or your organization can generate and maintain an independent certificate authority or use certificates generated by a third-party TLS/SSL vendor.

Throughout the rest of this chapter we will perform the necessary steps to configure and deploy a cluster such as the one depicted in the figure above.

## Member Certificates

We will refer to certificates used for internal authentication to verify membership in a cluster as member certificates. Both member certificates and client certificates (used to authenticate clients) have a structure resembling the following.

```
Certificate:  
Data:  
    Version: 1 (0x0)  
    Serial Number: 1 (0x1)  
    Signature Algorithm: sha256WithRSAEncryption  
    Issuer: C=US, ST=NY, L=New York, O=MongoDB, CN=CA-SIGNER  
    Validity  
        Not Before: Nov 11 22:00:03 2018 GMT  
        Not After : Nov 11 22:00:03 2019 GMT  
    Subject: C=US, ST=NY, L=New York, O=MongoDB, OU=MyServers, CN=server  
    Subject Public Key Info:  
        Public Key Algorithm: rsaEncryption  
        Public-Key: (2048 bit)  
        Modulus:  
            00:d3:1c:29:ba:3d:29:44:3b:2b:75:60:95:c8:83:  
            fc:32:1a:fa:29:5c:56:f3:b3:66:88:7f:f9:f9:89:  
            ff:c2:51:b9:ca:1d:4c:d8:b8:5a:fd:76:f5:d3:c9:  
            95:9c:74:52:e9:8d:f5:2e:6b:ca:f8:6a:16:17:98:  
            dc:aa:bf:34:d0:44:33:33:f3:9d:4b:7e:dd:7a:19:  
            1b:eb:3b:9e:21:d9:d9:ba:01:9c:8b:16:86:a3:52:  
            a3:e6:e4:5c:f7:0c:ab:7a:1a:be:c6:42:d3:a6:01:
```

```

8e:0a:57:b2:cd:5b:28:ee:9d:f5:76:ca:75:7a:c1:
7c:42:d1:2a:7f:17:fe:69:17:49:91:4b:ca:2e:39:
b4:a5:e0:03:bf:64:86:ca:15:c7:b2:f7:54:00:f7:
02:fe:cf:3e:12:6b:28:58:1c:35:68:86:3f:63:46:
75:f1:fe:ac:1b:41:91:4f:f2:24:99:54:f2:ed:5b:
fd:01:98:65:ac:7a:7a:57:2f:a8:a5:5a:85:72:a6:
9e:fb:44:fb:3b:1c:79:88:3f:60:85:dd:d1:5c:1c:
db:62:8c:6a:f7:da:ab:2e:76:ac:af:6d:7d:b1:46:
69:c1:59:db:c6:fb:6f:e1:a3:21:0c:5f:2e:8e:a7:
d5:73:87:3e:60:26:75:eb:6f:10:c2:64:1d:a6:19:
f3:0b

Exponent: 65537 (0x10001)

Signature Algorithm: sha256WithRSAEncryption

5d:dd:b2:35:be:27:c2:41:4a:0d:c7:8c:c9:22:05:cd:eb:88:
9d:71:4f:28:c1:79:71:3c:6d:30:19:f4:9c:3d:48:3a:84:d0:
19:00:b1:ec:a9:11:02:c9:a6:9c:74:e7:4e:3c:3a:9f:23:30:
50:5a:d2:47:53:65:06:a7:22:0b:59:71:b0:47:61:62:89:3d:
cf:c6:d8:b3:d9:cc:70:20:35:bf:5a:2d:14:51:79:4b:7c:00:
30:39:2d:1d:af:2c:f3:32:fe:c2:c6:a5:b8:93:44:fa:7f:08:
85:f0:01:31:29:00:d4:be:75:7e:0d:f9:1a:f5:e9:75:00:9a:
7b:d0:eb:80:b1:01:00:c0:66:f8:c9:f0:35:6e:13:80:70:08:
5b:95:53:4b:34:ec:48:e3:02:88:5c:cd:a0:6c:b4:bc:65:15:
4d:c8:41:9d:00:f5:e7:f2:d7:f5:67:4a:32:82:2a:04:ae:d7:
25:31:0f:34:e8:63:a5:93:f2:b5:5a:90:71:ed:77:2a:a6:15:
eb:fc:c3:ac:ef:55:25:d1:a1:31:7a:2c:80:e3:42:c2:b3:7d:
5e:9a:fc:e4:73:a8:39:50:62:db:b1:85:aa:06:1f:42:27:25:
4b:24:cf:d0:40:ca:51:13:94:97:7f:65:3e:ed:d9:3a:67:08:
79:64:a1:ba

-----BEGIN CERTIFICATE-----
MIIDODCCAiACAQEwDQYJKoZIhvCNQELBQAwWTELMAkGA1UEBhMCQ04xCzAJBgNV
BAgMAkdEMREwDwYDVQQHDAhTaGVuemh1bjEWMBQGA1UECgwNTW9uZ29EQiBDaGlu
YTESMBAGA1UEAwwJQ0EtU01HTkVSMB4XDTE4MTExMTIyMDAwM1oXDTE5MTExMTIy
MDAwM1owazELMAkGA1UEBhMCQ04xCzAJBgNVBAGMAkdEMREwDwYDVQQHDAhTaGVu
emh1bjEWMBQGA1UECgwNTW9uZ29EQiBDaGluYTESBAGA1UECwwJTX1TZXJ2ZXJz
MRAwDgYDVQQDDAdzZXJ2ZXIxMIIBIjANBgkqhkiG9w0BAQEFAOCAQ8AMIBCgKC
AQEA0xwpuj0pRDsrWCvYIP8Mhr6KVxW87NmiH/5+Yn/wlG5yh1M2Lha/Xb108mV
nHRS6Y1fLmvK+GoWF5jcqr800EQzM/Ods37dehkb6zueIdnZugGcixaGo1Kj5uRc
9wyrehq+xkLTpgGOCliezVso7p31dsplesF8QtEqfxf+aRdJkUvKLjm0peADv2SG
yhXHsvdUAPcC/s8+EmsowBw1aIY/Y0Z18f6sG0GRT/IkmVTy7Vv9Azh1rHp6Vy+o
pVqFcqae+0T7Oxx5iD9ghd3RXBzbYoxq99qrLnasr219sUZpwVnbxvtv4aMhDF8u
jqfVc4c+YCZ1628QwmQdpfnzCwIDAQABMA0GCSqGSIb3DQEBCwUAA4IBAQBd3bI1
vifCQuoNx4zJigXN64idcU8owXlxPG0wGfScPUG6hNAZALHsqRECyaacdOdOPDqf
IzBQWtJHU2UGpyILWXGwR2FiiT3Pxtiz2cxwIDW/Wi0UUX1LfAAwOS0dryzzMv7C
xqW4k0T6fwf8AExKQDUvnV+Dfka9e11AJp70OuAsQEAwGb4yfA1bhOAcAhb1VNL
NOxI4wK1XM2gbLS8ZRVNyEGdAPXn8tf1Z0oygioErtclMQ806G0lk/K1WpBx7Xcq
phXr/MOs71Ul0aExeiyA40LCs31emvzkc6g5UGLbsYWqBh9CJyVLJM/QQMpRE5SX
f2U+7dk6Zwh5ZKG6

-----END CERTIFICATE-----

```

For use with x.509 authentication in MongoDB, member certificates must have the following properties:

- A single CA must issue all x.509 certificates for the members of the cluster.
- The Distinguished Name (DN), found in the subject of the member certificate must specify a non-empty value for at least one of the following attributes: Organization (O), Organizational Unit (OU), or Domain Component (DC).
- The Organization attributes, Organizational Unit attributes, and the Domain Components must match those from the certificates for the other cluster members.
- Either the Common Name (CN) or one of the Subject Alternative Name (SAN) entries must match the hostname of the server used by the other members of the cluster.

## Generating a Root CA

Before we can generate signed certificates for the members of our replica set, we must first address the issue of a certificate authority. Again, to do this, you can generate and maintain an independent certificate authority or use certificates generated by a third-party TLS/SSL vendor. We will generate our own certificate authority to use for the running example in this chapter. Note that you may access all examples in this chapter from the GitHub repository maintained for this book. The examples in this chapter are drawn from a script you can use to deploy a secure replica set. You'll see comments from this script throughout these examples.

To generate our certificate authority, we will use OpenSSL. To follow along, please make sure you have access to OpenSSL on your local machine.

A root CA (Certificate Authority) is at the top of the certificate chain. This is the ultimate source of trust. Ideally a third party CA should be used. However in the case of an isolated network (very typical in large enterprise environment), or for testing purpose, we need to use a local CA to test the functionality.

First we'll initialize some variables.

---

```
dn_prefix="/C=US/ST=NY/L=New York/O=MongoDB"
ou_member="MyServers"
ou_client="MyClients"
mongodb_server_hosts=( "server1" "server2" "server3" )
mongodb_client_hosts=( "client1" "client2" )
mongodb_port=27017
```

---

Then, we'll create a key pair and store it in the file, *root-ca.key*.

---

```
# !!! In production you will want to password protect the keys
```

---

```
# openssl genrsa -aes256 -out root-ca.key 2048
openssl genrsa -out root-ca.key 2048
```

Then using the `openssl req` command, we will create the root certificate. Since the root is the very top of the authority chain we'll self sign this certificate using the private key we created in the previous step (stored in `root-ca.key`). The `-x509` option tells the `openssl req` command we want to self-sign the certificate using the private key supplied to the `-key` option. The output is a file called `root-ca.crt`.

```
openssl req -new -x509 -days 3650 -key root-ca.key -out root-ca.crt -subj
    /CN=RootCA
mkdir -p RootCA/ca.db.certs
echo "01" >> RootCA/ca.db.serial
touch RootCA/ca.db.index
echo $RANDOM >> RootCA/ca.db.rand
mv root-ca* RootCA/
```

If you take a look at the `root-ca.crt` file, you'll find that it contains the public certificate for the root CA. You can verify the contents by taking a look at a human-readable version of the certificate produced by the following command.

```
openssl x509 -noout -text -in root-ca.crt
```

The output from this command will resemble the following.

```
Certificate:
Data:
Version: 3 (0x2)
Serial Number:
95:46:78:a7:ea:00:c2:ae
Signature Algorithm: sha256WithRSAEncryption
Issuer: C=US, ST=NY, L=New York, O=MongoDB, CN=ROOTCA
Validity
    Not Before: Nov 20 16:23:49 2018 GMT
    Not After : Nov 17 16:23:49 2028 GMT
Subject: C=US, ST=NY, L=New York, O=MongoDB, CN=ROOTCA
Subject Public Key Info:
    Public Key Algorithm: rsaEncryption
    Public-Key: (2048 bit)
        Modulus:
            00:e4:65:8c:c9:62:a7:b1:4d:a7:0d:2e:c3:e0:0e:
            0c:a1:74:4d:20:e3:68:32:65:f2:02:e6:36:69:55:
            08:2b:3b:a0:fb:22:ab:61:1c:f9:5b:fa:51:a3:ca:
            cd:62:82:e2:fb:eb:e3:fd:fc:90:30:e2:a5:18:bf:
            87:a5:c6:14:7d:0a:09:68:c1:70:93:bd:89:5a:7e:
```

```

1d:25:02:71:74:01:85:e5:90:cf:29:ad:ff:b3:58:
f0:f9:e8:45:db:e2:86:73:51:9c:11:e1:29:f4:80:
f4:eb:a6:94:39:51:c0:6e:c7:8e:7f:63:34:1f:c0:
22:66:0d:d8:23:da:b2:a2:ca:40:d3:76:55:f3:5b:
13:c0:c1:89:b8:a0:08:7c:80:6d:9b:1f:4a:7c:b4:
77:88:33:ef:cc:ef:61:75:e3:1a:6c:3f:8e:2c:53:
fb:77:28:3e:b6:cf:28:92:dd:c1:b8:17:5b:9b:38:
d1:b3:aa:fb:bd:cc:81:ea:b9:a8:f9:38:93:4c:ce:
b9:1f:03:a6:87:f8:66:77:40:74:f8:29:7f:b3:20:
a8:60:65:b4:11:1d:de:ae:64:f8:e3:d6:d4:2f:f4:
aa:dc:0e:b1:7d:4e:74:67:1f:e5:5b:90:b0:19:98:
9e:cc:ae:e2:3e:e3:86:a7:cc:95:1d:f4:9c:02:5c:
c8:63

Exponent: 65537 (0x10001)

X509v3 extensions:
    X509v3 Subject Key Identifier:
        2F:93:7D:44:02:E6:73:B9:A9:1D:D3:7A:7A:91:39:2C:3B:8C:68:21
    X509v3 Authority Key Identifier:
        keyid:2F:93:7D:44:02:E6:73:B9:A9:1D:D3:7A:7A:91:39:2C:3B:80

    X509v3 Basic Constraints:
        CA:TRUE
    Signature Algorithm: sha256WithRSAEncryption
        e2:a4:2b:36:81:99:ad:89:48:a2:ee:75:bb:b0:7b:8f:0c:38:
        22:65:8a:f4:62:dc:ee:ae:60:d0:e8:5d:81:1c:54:c9:04:c9:
        a4:0d:de:f7:bf:45:d3:c9:05:51:88:3d:71:4d:a9:0c:e7:66:
        e7:d9:05:75:fd:6a:60:f9:2f:2d:af:dd:cc:8d:99:88:d6:f7:
        22:99:c6:d5:ac:7c:1b:28:db:ce:7c:13:25:57:8d:91:e3:8f:
        52:1e:c5:ae:75:10:fd:dc:40:b2:2d:e1:b0:d9:2d:55:15:3b:
        88:fa:fb:2e:46:db:ce:42:f6:cc:95:57:08:ba:47:53:e1:ed:
        ac:98:0b:0c:3e:d7:c7:cb:27:07:bb:01:eb:10:41:b7:7b:63:
        f3:29:1c:a4:b4:c7:48:35:85:56:28:6b:e7:ef:07:61:90:58:
        f6:2f:02:f6:d9:64:f9:ed:e6:a5:bb:1d:6c:d2:78:96:fa:39:
        fd:ef:aa:17:b8:06:8a:20:ff:5e:1a:06:09:48:c7:22:2d:56:
        05:cf:d4:52:fe:ed:9a:33:6f:f8:c3:e3:ad:89:2d:46:1a:01:
        62:80:4a:04:c8:4f:3d:0a:00:80:7e:5c:bf:51:95:ef:71:03:
        3b:e7:3f:a6:59:ac:55:43:aa:a5:28:6e:68:99:f5:d8:66:52:
        f1:54:51:86

```

## Creating an Intermediate CA for Signing

Now that we've created our root CA, we will create an intermediate CA for signing member and client certificates. An intermediate CA is nothing more than a certificate signed using our root certificate. It is a best practice to use an intermediate CA to sign server (i.e. member) and client certificates. Typically, a CA will use different intermediate CA for signing different categories of certificates. If the intermediate CA is compromised and the certificate needs to be revoked, only a portion of the trust tree is affected instead of all certificates signed by the CA, as would be the case if the root CA were used to sign all certificates.

```
# again, in production you would want to password protect your signing key
# openssl genrsa -aes256 -out signing-ca.key 2048
openssl genrsa -out signing-ca.key 2048

openssl req -new -days 1460 -key signing-ca.key -out signing-ca.csr -subj /
openssl ca -batch -name RootCA -config root-ca.cfg -extensions v3_ca -out :

mkdir -p SigningCA/ca.db.certs
echo "01" >> SigningCA/ca.db.serial
touch SigningCA/ca.db.index
# Should use a better source of random here..
echo $RANDOM >> SigningCA/ca.db.rand
mv signing-ca* SigningCA/
```

Note that in the statements above we are using the *openssl req* command followed by the *openssl ca* command to sign our signing certificate using our root certificate. The *openssl req* command creates a signing request and the *openssl ca* command uses that request as input to create a signed intermediate (signing) certificate.

As a last step in creating our signing CA, we will concatenate our root certificate (containing our root public key) and signing certificate (containing our signing public key) into a single pem file. This file will be supplied to our *mongod* or client process later as the value of `--sslCAFile` option.

```
cat RootCA/root-ca.crt SigningCA/signing-ca.crt > root-ca.pem
```

With the root CA and signing CA set up, we are now ready to create the member and client certificates used for authentication in our MongoDB cluster.

## Generate and Sign Member Certificates

Member certificates are what is typically referred to as x.509 server certificates. Use this type of certificate for *mongod* and *mongos* processes. Members of a MongoDB cluster use these certificates to verify membership in the cluster. Stated another way, one *mongod* authenticates itself with other members of a replica set using a server certificate.

To generate certificates for the members of our replica set, we will use a for loop to generate multiple certificates.

```
# Pay attention to the OU part of the subject in "openssl req" command
for host in "${mongodb_server_hosts[@]}"; do
    echo "Generating key for $host"
    openssl genrsa -out ${host}.key 2048
```

```
openssl req -new -days 365 -key ${host}.key -out ${host}.csr -subj "$d  
openssl ca -batch -name SigningCA -config root-ca.cfg -out ${host}.crt  
cat ${host}.crt ${host}.key > ${host}.pem  
done
```

Three steps are involved with each certificate:

- Use the *openssl genrsa* command to create a new key pair.
- Use the *openssl req* command to generate a signing request for the key.
- Use the *openssl ca* command to sign and output a certificate using the signing CA.

Notice the variable `$ou_member`. This signifies the difference between server certificates and client certificates. Server and client certificates must differ in the organization part of the Distinguished Names. More specifically, they must differ in at least in one of the O, OU, or DC values.

## Generate and Sign Client Certificates

Client certificates are used by the mongo shell, MongoDB Compass, MongoDB utilities and tools and, of course, by applications using a MongoDB driver. Generating client certificates follows essentially the same process as for member certificates. The one difference is our use of the variable `$ou_client`. This ensure that the combination of the O, OU, and DC values will be different from those of the server certificates generated above.

```
# Pay attention to the OU part of the subject in "openssl req" command  
for host in "${mongodb_client_hosts[@]}"; do  
    echo "Generating key for $host"  
    openssl genrsa -out ${host}.key 2048  
    openssl req -new -days 365 -key ${host}.key -out ${host}.csr -subj "$d  
    openssl ca -batch -name SigningCA -config root-ca.cfg -out ${host}.crt  
    cat ${host}.crt ${host}.key > ${host}.pem  
done
```

## A Primer on MongoDB Authentication and Authorization

We are nearly ready to launch our replica set; however, before we do, it is important to address some fundamentals of how authorization and authentication work in MongoDB. As a reminder, authentication verifies the identity of a user; authorization determines what level of access a verified user is permitted to the cluster, databases, collections, and operations on those resources.

When adding a user in MongoDB, you must create the user in a specific database. The database is the authentication database for the user. The username and authentication database serve as a unique identifier for a user. You can use any database as the authentication database for a user. However, a user's privileges are not limited to their authentication database. When creating a user, you can specify the operations the user may perform on any resources to which they should have access. Resources include the cluster, databases, and collections.

MongoDB provides a number of built-in roles that grant commonly needed permissions for database users. These include the following:

- **read** -- read data on all non-system collections and on the following system collections: system.indexes, system.js, and system.namespaces collections
- **readWrite** -- read role privileges plus the ability to modify data on all non-system collections and the system.js collection
- **dbAdmin** -- perform administrative tasks such as schema-related tasks, indexing, and gathering statistics (does not grant privileges for user and role management)
- **userAdmin** -- create and modify roles and users on the current database
- **dbOwner** -- combines the privileges granted by the readWrite, dbAdmin and userAdmin roles
- **clusterManager** -- perform management and monitoring actions on the cluster
- **clusterMonitor** -- provides read-only access to monitoring tools, such as the MongoDB Cloud Manager and Ops Manager monitoring agent
- **hostManager** -- monitor and manage servers
- **clusterAdmin** -- combines the privileges granted by the clusterManager, clusterMonitor, and hostManager roles plus the dropDatabase action.
- **backup** -- provides sufficient privileges to use the MongoDB Cloud Manager backup agent, Ops Manager backup agent, or to use mongodump to back up an entire mongod instance
- **restore** -- provides privileges needed to restore data from backups that do not include system.profile collection data
- **readAnyDatabase** -- same privileges as read except local and config plus the listDatabases action on the cluster as a whole

- `readWriteAnyDatabase` -- same privileges as `readWrite` on all databases except local and config plus the `listDatabases` action on the cluster as a whole
- `userAdminAnyDatabase` -- same privileges as `userAdmin` on all databases except local and config (effectively a superuser role)
- `dbAdminAnyDatabase` -- provides the same privileges as `dbAdmin` on all databases except local and config plus the `listDatabases` action on the cluster as a whole
- `root` -- provides access to the operations and all the resources of the `readWriteAnyDatabase`, `dbAdminAnyDatabase`, `userAdminAnyDatabase`, `clusterAdmin`, `restore`, and `backup` combined

You may also create what are known as “user-defined roles”, which are custom roles that group together authorization to perform specific operations and label them with a name so that you may grant this set of permissions to multiple users easily.

A deep dive on built-in roles or user-defined roles is beyond the scope of this chapter. However, with the above, you should have a pretty good idea of what’s possible with MongoDB authorization. For greater detail, please see the authorization section of the MongoDB documentation <https://docs.mongodb.com/manual/core/authorization/>.

To ensure that we can add new users as needed, we must first create an admin user. MongoDB does not create a default root or admin user when enabling authentication and authorization, regardless of the authentication mode we are using (x.509 is no exception).

In MongoDB, authentication and authorization are not enabled by default. You must explicitly enable them using the `--auth` option to the `mongod` command or by specifying a value of `enabled` for the `security.authorization` setting in a MongoDB config file.

To create an admin user for the replica set we are in the midst of configuring, we must first bring up our replica set without authentication and authorization enabled. We can then create the admin user and then the users we’ll need for each client.

## **Bring Up the Replica Set Without Authentication and Authorization Enabled**

We can start each member of our replica set without auth enabled as follows. Previously, when working with replica sets we’ve not enabled auth so this should look familiar. Here again we are making use of a few variables we defined in Section 1.4 (or see the full script for this chapter) and a loop to launch each member (`mongod`) of our replica set.

```

import=$mongodb_port
for host in "${mongodb_server_hosts[@]}"; do
    echo "Starting server $host in non-auth mode"
    mkdir -p ./db/${host}
    mongod --replSet set509 --port $import --dbpath ./db/${host} \
        --fork --logpath ./db/${host}.log
    let "import++"
done

```

Once each *mongod* has started, we can then initialize a replica set using these *mongods*.

```

myhostname=`hostname`
cat > init_set.js <<EOF
rs.initiate();
import=$mongodb_port;
import++;
rs.add("$myhostname:" + import);
import++;
rs.add("$myhostname:" + import);
EOF
mongo localhost:$mongodb_port init_set.js

```

Note that the code above simply constructs a series of commands, stores these commands in a JavaScript file and then runs the *mongo* shell to execute the small script that was created. Together, these commands, when executed in the *mongo* shell, will connect to the *mongod* running on port 27017 (value of the `$mongodb_port` variable set in Section 1.4), initiate the replica set, and then add each of the other two *mongods* (on ports 27018 and 27019) to the replica set.

## Create the Admin User

Now, we'll create an admin user based on one of the client certificates we created in Section 1.7. We will authenticate as this user when connecting from the *mongo* shell or another client to perform administrative tasks. To authenticate with a client certificate, you must first add the value of the subject from the client certificate as a MongoDB user. Each unique x.509 client certificate corresponds to a single MongoDB user; i.e. you cannot use a single client certificate to authenticate more than one MongoDB user. We must add the user in the `$external` database; i.e. the authentication database is the `$external` database.

First we'll get the subject from our client certificate using the *openssl x509* command.

```
openssl x509 -in client1.pem -inform PEM -subject -nameopt RFC2253 | grep :
```

This should result in the following output.

```
subject= CN=client1,OU=MyClients,O=MongoDB,L=New York,ST=NY,C=US
```

To create our admin user, we'll first connect to the primary of our replica set using the *mongo* shell.

```
mongo localhost:27017
```

From within the *mongo* shell, we will issue the following command.

```
db.getSiblingDB("$external").runCommand(  
{  
    createUser: "CN=client1,OU=MyClients,O=MongoDB,L=New York,ST=NY,C=US",  
    roles: [  
        { role: "readWrite", db: 'test' },  
        { role: "userAdminAnyDatabase", db: "admin" },  
        { role: "clusterAdmin", db: "admin" }  
    ],  
    writeConcern: { w: "majority" , wtimeout: 5000 }  
}  
) ;
```

Note the use of the \$external database in this command and the fact that we've specified the subject of our client certificate as the user name.

## Restart the Replica Set with Authentication and Authorization Enabled

Now that we have an admin user, we can restart the replica set with authentication and authorization enabled and connect as a client. Without a user of any kind, it would be impossible to connect to a replica set with auth enabled.

Let's stop the replica set in it's current form (without auth enabled).

```
kill $(ps -ef | grep mongod | grep set509 | awk '{print $2}')
```

We are now ready to restart the replica set with auth enabled. In a production environment, we would copy each of the certificate and key files to their corresponding hosts. Here we're doing everything on localhost to make things easier. To initiate a secure replica set we will add the following command-line options to each invocation of *mongod*:

- sslMode
- clusterAuthMode
- sslCAFile – root CA file (root-ca.key)
- sslPEMKeyFile – certificate file for the mongod
- sslAllowInvalidHostnames – only used for testing; allows invalid hostnames

Here the sslCAFile is used to establish a trust chain. As you recall the root-ca.key file contains the certificate of the root CA as well as the signing CA. By providing this file to mongod process, we are stating our desire to trust the certificate contained in this file as well as all other certificates signed by these certificates.

Okay, let's do this.

```
mport=$mongodb_port
for host in "${mongodb_server_hosts[@]}"; do
    echo "Starting server $host"
    mongod --replSet set509 --port $mport --dbpath ./db/$host \
        --sslMode requireSSL --clusterAuthMode x509 --sslCAFile root-ca.pem \
        --sslAllowInvalidHostnames --fork --logpath ./db/${host}.log \
        --sslPEMKeyFile ${host}.pem --sslClusterFile ${host}.pem
    let "mport++"
done
```

And with that, we have a three-member replica set secured using x.509 certificates for authentication and transport-layer encryption. The only thing left to do is to connect with the mongo shell. We'll use the client1 certificate to authenticate, because that is the certificate for which we created an admin user.

```
cat > do_login.js <<EOF
db.getSiblingDB("\$external").auth(
{
    mechanism: "MONGODB-X509",
    user: "$client_subject"
})
EOF
mongo --ssl --sslPEMKeyFile client1.pem --sslCAFile root-ca.pem --sslAllow:
```

Once connected, we encourage you to experiment by inserting some data to a collection. You

should also attempt to connect using any other user, for example, using the client2.pem. While you will be able to connect, that user is not authorized to do anything on our cluster. Attempts to perform operations will result in errors like the following.

```
mongo --ssl --sslPEMKeyFile client2.pem --sslCAFile root-ca.pem --sslAllowInvalidCertificates
MongoDB shell version v4.0.3
connecting to: mongodb://127.0.0.1:27017
2018-11-21T12:28:55.073-0500 W NETWORK  [js] The server certificate does not match the host name!
Implicit session: session { "id" : UUID("4737faed-adab-4ec7-9274-006acf9bc1")
MongoDB server version: 4.0.3
type "help" for help
Error: Username "CN=client1,OU=MyClients,O=MongoDB,L>New York,ST=NY,C=US" does not exist
MongoDB Enterprise set509:PRIMARY> show dbs
2018-11-21T12:29:10.859-0500 E QUERY    [js] Error: listDatabases failed: {
    "operationTime" : Timestamp(1542821348, 1),
    "ok" : 0,
    "errmsg" : "command listDatabases requires authentication",
    "code" : 13,
    "codeName" : "Unauthorized",
    "$clusterTime" : {
        "clusterTime" : Timestamp(1542821348, 1),
        "signature" : {
            "hash" : BinData(0,"wpU0r1mr8VQwVME45TXAPkispPM="),
            "keyId" : NumberLong("6626365429344370689")
        }
    }
} :
_getErrorWithCode@src/mongo/shell/utils.js:25:13
Mongo.prototype.getDBs@src/mongo/shell/mongo.js:67:1
shellHelper.show@src/mongo/shell/utils.js:876:19
shellHelper@src/mongo/shell/utils.js:766:15
@(shellhelp2):1:1
```

## Summary

In the tutorial in this chapter, we've looked at an example of using x.509 certificates as a basis for authentication and to encrypt communication among clients and members of a replica set. The same procedure works for sharded clusters as well. With respect to securing a MongoDB cluster, please keep the following in mind:

- The directories, Root CA and Signing CA, as well as the host itself where you generate and sign certificates for the member machines or clients, should be protected from unauthorized access.
- For simplicity, the Root CA and Signing CA keys are not password protected in this tutorial. In production it is necessary to use passwords to protect the key from unauthorized use.

We encourage you to download and experiment with the demo scripts we have provided for this chapter in the book GitHub repository.

## Colophon

The animal on the cover of *MongoDB: The Definitive Guide, Second Edition* 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 Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.