THE LITTLE BOOK OF

MongoDB

BY KARL SEGUIN
UPDATED FOR 6.0

About This Book

License

About The Original Author

Karl Seguin is a developer with experience across various fields and technologies. He's an expert .NET and Ruby developer. He's a semi-active contributor to OSS projects, a technical writer and an occasional speaker. With respect to MongoDB, he was a core contributor to the C# MongoDB library NoRM, wrote the interactive tutorial `mongly` as well as the Mongo Web Admin. Karl has since written The Little Redis Book.

His blog can be found at http://openmymind.net, and he tweets via `@karlseguin`.

About MongoDB Inc

MongoDB Inc is the company behind the MongoDB database platform. From its first pull request by Dwight Merriman on October 19, 2007, MongoDB has grown into an international organization with a wide range of products and services. MongoDB Inc maintains and enhances MongoDB database platform, MongoDB drivers, provides MongoDB Atlas Database as a Service, and other products.

Latest Version

This version was updated for MongoDB 6.0 by Asya Kamsky. The latest source of this book is available at: github.com/mongodb-developer/the-little-mongodb-book.

Table of Contents

Introduction

It is often said that technology moves at a blazing pace. It's true that there is an ever growing list of new technologies and techniques being released. However, I've long been of the opinion that the fundamental technologies used by programmers move slowly. One could spend years learning little yet remain relevant. What is striking though is the speed at which established technologies get replaced. Seemingly overnight, long-established technologies find themselves threatened by shifts in developer focus and advances in developer tooling.

Nothing could be more representative of this sudden shift than the progress of NoSQL technologies. It almost seems like one day the web was being driven by a few RDBMSs, and the next, NoSQL solutions had established themselves as worthy alternatives.

Even though these transitions seemed to happen overnight, the reality is that they can take years to become accepted practice. Initial enthusiasm is driven by a relatively small set of developers and companies. Solutions are refined, lessons learned and seeing that a new technology is here to stay, others slowly try it for themselves. Again, this is particularly true in the case of NoSQL where many solutions aren't replacements for more traditional storage solutions, but rather address a specific need in addition to what one might get from traditional offerings.

Having said all of that, the first thing we ought to do is explain what is meant by NoSQL. It's a broad term that means different things to different people. Personally, I use it very broadly to mean a system that plays a part in the storage of data. Put another way, NoSQL (again, for me), is the belief that your persistence layer isn't necessarily the responsibility of a single system. Where relational database vendors have historically tried to position their software as a one-size-fits-all solution, NoSQL leans towards smaller units of responsibility where the best tool for a given job can be leveraged. So, your NoSQL stack might still leverage a relational database, say MySQL, but it'll also contain Redis as a persistence lookup for specific parts of the system as well as Hadoop for your intensive data processing. Put simply, NoSQL is about being open and aware of alternative, existing and additional patterns and tools for managing your data.

You might be wondering where MongoDB fits into all of this. As a document-oriented database, MongoDB is a generalized NoSQL solution. It should be viewed as an alternative or a companion to relational databases. MongoDB has advantages and drawbacks, which we'll cover in later parts of this book.

Getting Started

Most of this book will focus on core MongoDB functionality. We'll therefore rely on the MongoDB shell. While the shell is useful to learn, your code will use a MongoDB driver.

This does bring up the first thing you should know about MongoDB: its drivers. MongoDB has a number of official drivers for various languages. These drivers can be thought of as the various database drivers you are probably already familiar with. On top of these drivers, the development community has built more language/framework-specific libraries. For example, Mongoose is a Node.js Object Document Mapper (ODM) and Spring Data MongoDB provides a POJO centric model for interacting with MongoDB collection in Java. Whether you choose to program directly against the core MongoDB drivers or some higher-level library is up to you. I point this out only because many people new to MongoDB are confused as to why there are both official drivers and community libraries - the former generally focuses on core communication/connectivity with MongoDB and the latter with more language and framework-specific implementations.

As you read through this, I encourage you to play with MongoDB to replicate what I demonstrate as well as to explore questions that you might come up with on your own. It's easy to get up and running with MongoDB, so let's take a few minutes now to set things up. You'll need to have a MongoDB server running somewhere, as well as a MongoDB client (CLI or GUI) running locally.

For the client part, install either the MongoDB Shell from the official page or if you prefer GUI to command line shell, you can use MongoDB Compass, an open source GUI for MongoDB. MongoDB Compass has the shell built-in so you can switch between using GUI and command line as you wish.

To run the server, if you don't want to (or can't) install MongoDB server locally, you can sign up for a free hosted MongoDB cluster in MongoDB Atlas - follow the getting started directions there to connect to your cluster.

If you can run MongoDB server locally and prefer to do that, follow instructions for your operating system on the official installation manual page.

Once you have `mongod` running locally or a connection string to your Atlas MongoDB cluster, connect to it from your MongoDB Shell or Compass.

Try entering `db.version()` at the prompt to make sure everything's working as it should. Hopefully you'll see the server version number you connected to.

Chapter 1 - The Basics

We begin our journey by learning the basics of working with MongoDB. Obviously this is core to understanding MongoDB, but it should also help us answer higher-level questions about where MongoDB fits.

To get started, there are six simple concepts we need to understand.

1. MongoDB has the same concept of a `database` with which you are likely already familiar (or a schema for you Oracle folks). Within a MongoDB instance you can have zero or more databases, each acting as high-level containers for everything else.
2. A database can have zero or more `collections`. A collection shares enough in common with a traditional `table` that you can safely think of the two as the same thing.
3. Collections are made up of zero or more `documents`. Again, a document can safely be thought of as analogous to a `row`.
4. A document is made up of one or more `fields`, which you can probably guess are somewhat like `columns`.
5. `Columns` in MongoDB function mostly like their RDBMS counterparts.
6. `Cursors` are like relational database cursors, but they are important enough, and often overlooked, that I think they are worthy of their own discussion. The important thing to understand about cursors is that when you ask MongoDB for data, it returns a pointer to the result set called a cursor, which we can do things to, such as counting or skipping ahead, before actually pulling down data.

To recap, MongoDB is made up of `databases` which contain `collections`. A `collection` is made up of `collections`. Each `document` is made up of `fields`. `Collections` can be `indexed`, which improves query and sorting performance. Finally, when we get data from MongoDB we usually do so through a `cursor` whose actual execution is delayed until necessary.

Why use new terminology (collection vs. table, document vs. row, and field vs. column)? Is it just to make things more complicated? The truth is that while these concepts are similar to their relational database counterparts, they are not identical. The core difference comes from the fact that relational databases define `columns` at the `table` level whereas a document-oriented database defines its `fields` at the `document` level. That is to say that each `document` within a `collection` can have its own unique set of `fields`. As such, a `collection` is a dumbed down container in comparison to a `table`, while a `document` has a lot more information than a `row`.

Although this is important to understand, don't worry if things aren't yet clear. It won't take more than a couple of inserts to see what this truly means. Ultimately, the point is that a collection isn't strict about what goes in it (its schema is flexible/dynamic). Fields are tracked with each individual document. The benefits and drawbacks of this will be explored in a future chapter.

Let's get hands-on. If you don't have it running already, go ahead and start the `mongod` server as well as a mongo shell or Compass. The shell runs JavaScript. There are some global commands you can execute, like `help` or `exit`. Commands that you execute against the current database are executed against the `db` object, such as `db.help()` or `db.stats()`. Commands that you execute against a specific collection, which is what we'll be doing a lot of, are executed against the `db.COLLECTION_NAME` object, such as `db.unicorns.help()` or `db.unicorns.count()`.

Go ahead and enter `db.help()`, you'll get a list of commands that you can execute against the `db` object.

A small side note: Because this is a JavaScript shell, if you execute a method and omit the parentheses `()`, you'll see the method body rather than executing the method. I only mention it so that the first time you do it and get a response that starts with `[Function: ...` you won't be surprised. For example, if you enter `db.stats` (without the parentheses), you'll see the details of implementation of the `stats` method.

First we'll use the global `use` helper to switch databases, so go ahead and enter `use learn`. It doesn't matter that the database doesn't exist yet. The first collection that we create will also create the actual `learn` database. Now that you are inside a database, you can start issuing database commands, like `db.getCollectionNames()`. If you do so, you should get an empty array `([ ])`. Since collections don't have schema, we don't usually need to explicitly create them. We can simply insert a document into a new collection. To do so, use the `insert` command, supplying it with the document to insert:

```
db.unicorns.insertOne({name: 'Aurora', gender: 'f', weight: 450})
```

The above line is executing `insert` against the `unicorns` collection, passing it a single document. Internally MongoDB uses a binary serialized JSON format called BSON. Externally, this means that we use JSON a lot, as is the case with our parameters. If we execute `db.getCollectionNames()` now, we'll see a `unicorns` collection.

You can now use the `find` command against `unicorns` to return a list of documents:

```
db.unicorns.find()
```

Notice that, in addition to the data you specified, there's an `_id` field. Every document must have a unique `_id` field. You can either generate one yourself or let MongoDB generate a value for you which has the type `ObjectId`. Most of the time you'll probably want to let MongoDB generate it for you. By default, the `_id` field is indexed. You can verify this through the `getIndexes` command:

```
db.unicorns.getIndexes()
```

What you're seeing is the name of the index and the fields included in the index.

Now, back to our discussion about "schemaless" collections. Insert a totally different document into `unicorns`, such as:

```
db.unicorns.insertOne({name: 'Leto', gender: 'm', home: 'Arrakeen',
worm: false})
```

And, again use `find` to list the documents. Once we know a bit more, we'll discuss this interesting behavior of MongoDB, but hopefully you are starting to understand why the more traditional terminology wasn't a good fit.

Mastering Selectors

In addition to the six concepts we've explored, there's one practical aspect of MongoDB you need to have a good grasp of before moving to more advanced topics: query selectors or predicates. A MongoDB query predicate is like the `WHERE` clause of an SQL statement. As such, you use it when finding, counting, updating or removing documents from collections. A selector is a JSON object, the simplest of which is `{}` which matches all documents. If we want to find all female unicorns, we use `{gender:'f'}`.

Before delving too deeply into selectors, let's set up some data to play with. First, remove what we've put so far in the `unicorns` collection via: `db.unicorns.remove({})`. Now, copy and paste following inserts to get some data we can play with:

```
db.unicorns.insertMany([
  {name: 'Horny', dob: ISODate("1992-03-13T07:47"), weight: 600,
      loves: ['carrot','papaya'], gender: 'm', vampires: 63},
  {name: 'Aurora', dob: ISODate("1991-01-24T13:00"), weight: 450,
      loves: ['carrot', 'grape'], gender: 'f', vampires: 43},
  {name: 'Unicrom', dob: ISODate("1973-02-09T22:10"), weight: 984,
      loves: ['energon', 'redbull'], gender: 'm', vampires: 182},
  {name: 'Roooooodles', dob: ISODate("1979-08-18T18:44"), weight: 575,
      loves: ['apple'], gender: 'm', vampires: 99},
  {name: 'Solnara', dob: ISODate("1985-07-04T02:01"), weight:550,
      loves:['apple', 'carrot', 'chocolate'], gender:'f', vampires:80},
  {name:'Ayna', dob: ISODate("1998-03-07T08:30"), weight: 733,
      loves: ['strawberry', 'lemon'], gender: 'f', vampires: 40},
  {name:'Kenny', dob: ISODate("1997-07-01T10:42"), weight: 690,
      loves: ['grape', 'lemon'], gender: 'm', vampires: 39},
  {name: 'Raleigh', dob: ISODate("2005-05-03T00:57"), weight: 421,
      loves: ['apple', 'sugar'], gender: 'm', vampires: 2},
  {name: 'Leia', dob: ISODate("2001-10-08T14:53"), weight: 601,
      loves: ['apple', 'watermelon'], gender: 'f', vampires: 33},
  {name: 'Pilot', dob: ISODate("1997-03-01T05:03"), weight: 650,
      loves: ['apple', 'watermelon'], gender: 'm', vampires: 54},
  {name: 'Nimue', dob: ISODate("1999-12-20T00:16:15"), weight: 540,
      loves: ['grape', 'carrot'], gender: 'f'},
  {name: 'Dunx', dob: ISODate("1976-07-18T18:18"), weight: 704,
      loves: ['grape', 'watermelon'], gender: 'm', vampires: 165}]);
```

Now that we have data, we can master selectors. `{field: value}` is used to find any documents where `field` is equal to `value`. `{field1: value1, field2: value2}` is how we do an `and` statement. The special `$lt`, `$lte`, `$gt`, `$gte` and `$ne` are used for less than, less than or equal, greater than, greater than or equal and not equal operations. For example, to get all male unicorns that weigh more than 700 pounds, we could do:

```
db.unicorns.find({gender: 'm', weight: {$gt: 700}})
// or similar but not quite the same:
db.unicorns.find({gender: {$ne: 'f'}, weight: {$gte: 701}})
```

The `$exists` operator is used for matching the presence or absence of a field, for example:

```
db.unicorns.find({vampires: {$exists: false}})
```

should return a single document. The `$in` operator is used for matching one of several values that we pass as an array, for example:

```
db.unicorns.find({loves: {$in:['apple','orange']}})
```

This returns any unicorn who loves 'apple' or 'orange'.

Multiple conditions separated by `,` are implicitly connected by `AND`. If we want to `OR` rather than `AND` several conditions on different fields, we use the `$or` operator and assign to it an array of selectors we want to `OR`:

```
db.unicorns.find({gender:'f',$or:[{loves:'apple'},{weight:{$lt:500}}]})
```

The above will return all female unicorns which either love apples or weigh less than 500 pounds.

There's something pretty neat going on in our last two examples. You might have already noticed, but the `loves` field is an array. MongoDB supports arrays as first class objects. This is an incredibly handy feature. Once you start using it, you wonder how you ever lived without it. What's more interesting is how easy selecting based on an array value is: `{loves: 'watermelon'}` will return any document where `watermelon` is a value of `loves`.

There are more available operators than what we've seen so far. These are all described in the Query Selectors section of the MongoDB manual. What we've covered so far though is the basics you'll need to get started. It's also what you'll end up using most of the time.

We've seen how these selectors can be used with the `find` command. They can also be used with the `remove` command which we've briefly looked at, the `count` command, which we haven't looked at but you can probably figure out, and the `update` command which we'll spend more time with later on.

The `ObjectId` which MongoDB generated for our `_id` field can be selected like so:

```
db.unicorns.find({_id: ObjectId("TheObjectIdString")})
```

In This Chapter

We haven't looked at the `update` command yet, or some of the fancier things we can do with `find`. However, we did get MongoDB up and running, looked briefly at the `insert` and `remove` commands (there isn't much more than what we've seen). We also introduced `find` and saw what MongoDB `selectors` were all about. We've had a good start and laid a solid foundation for things to come. Believe it or not, you actually know most of what you need to know to get started with MongoDB - it really is meant to be quick to learn and easy to use. I strongly urge you to play with your local copy before moving on. Insert different documents, possibly in new collections, and get familiar with different selectors. Use `find`, `count` and `remove`. After a few tries on your own, things that might have seemed awkward at first will hopefully fall into place.

Chapter 2 - Updating

In chapter 1 we introduced three of the four CRUD (create, read, update and delete) operations. This chapter is dedicated to the one we skipped over: `update`. `Update` has a few surprising behaviors, which is why we dedicate a chapter to it.

Update: Replace Versus `$set`

In its simplest form, `update` command takes two parameters: the selector (where) to use and what updates to apply to fields. If Roooooodles had gained a bit of weight, you might expect that we should execute:

```
db.unicorns.update({name: 'Roooooodles'}, {weight: 590});
db.unicorns.find({name: 'Roooooodles'});
```

Now, when we look at the updated record, we should discover the first surprise of `update`. No document is found because the second parameter we supplied didn't have any update operators, and therefore it was used to **replace** the original document. In other words, the `update` found a document by `name` and replaced the entire document with the new document (the second parameter). There is no equivalent functionality to this in SQL's `update` command. In some situations, this is ideal and can be leveraged for some truly dynamic updates. However, when you want to change the value of one, or a few fields, you must use MongoDB's `$set` operator. Go ahead and run this update to reset the lost fields:

```
db.unicorns.update({weight: 590}, {$set: {name: 'Roooooodles',
    dob: ISODate("1979-08-18T18:44"), loves: ['apple'], gender: 'm',
vampires: 99}})
```

This won't overwrite the new `weight` since we didn't specify it. Now if we execute:

```
db.unicorns.find({name: 'Roooooodles'})
```

We get the expected result. Therefore, the correct way to have updated the weight in the first place is:

```
db.unicorns.updateOne({name: 'Roooooodles'}, {$set: {weight: 590}})
```

Stick to using shell helper `updateOne` to prevent accidental replacement operations, and use `replaceOne` to replace the entire document.

Update Operators

In addition to `$set` we can leverage other operators to do some nifty things. All update operators work on fields - so your entire document won't be wiped out. For example, the `$inc` operator is used to increment a field by a certain positive or negative amount. If Pilot was incorrectly awarded a couple vampire kills, we could correct the mistake by executing:

```
db.unicorns.updateOne({name: 'Pilot'}, {$inc: {vampires: -2}})
```

If Aurora suddenly developed a sweet tooth, we could add a value to her `loves` field via the `$push` operator:

```
db.unicorns.updateOne({name: 'Aurora'}, {$push: {loves: 'sugar'}})
```

The Update Operators section of the MongoDB manual has more information on the other available update operators.

Upserts

One of the more pleasant surprises of using `update` is that it fully supports `upserts`. An `upsert` updates the document if found or inserts it if not. Upserts are handy to have in certain situations and when you run into one, you'll know it. To enable upserting, we pass a third parameter to update `{upsert:true}`.

A mundane example is a hit counter for a website. If we wanted to keep an aggregate count in real time, we'd have to see if the record already existed for the page, and based on that decide to run an update or insert. With the upsert option omitted (or set to false), executing the following won't do anything:

```
db.hits.updateOne({page: 'unicorns'}, {$inc: {hits: 1}});
db.hits.find();
```

However, if we add the upsert option, the results are quite different:

```
db.hits.updateOne({page:'unicorns'}, {$inc:{hits:1}}, {upsert:true});
db.hits.find();
```

Since no documents exist with a field `page` equal to `unicorns`, a new document is inserted. If we execute it a second time, the existing document is updated and `hits` is incremented to 2.

```
db.hits.updateOne({page:'unicorns'}, {$inc:{hits:1}}, {upsert:true});
db.hits.find();
```

Multiple Updates

The final surprise `update` has to offer is that, by default, it'll update a single document, which is why the shell helper is called `updateOne`. For the examples we've looked at so far, this might seem logical. However, if you have an update like this:

```
db.unicorns.updateOne({}, {$set:{vaccinated:true}});
```

you might want to find all of your precious unicorns to be vaccinated. To get that behavior, use `updateMany` helper which executes the `update` command with the `multi` option set to true:

```
db.unicorns.updateMany({}, {$set:{vaccinated:true}});
db.unicorns.find({vaccinated: true});
```

In This Chapter

This chapter concluded our introduction to the basic CRUD operations available against a collection. We looked at `update` in detail and observed three interesting behaviors. First, if you pass it a document without update operators, MongoDB's `update` will replace the existing document. Because of this, normally you will use the `$set` operator (or one of the many other available operators that modify the document). Secondly, `update` supports an intuitive `upsert` option which is particularly useful when you don't know if the document already exists. Finally, by default, `update` updates only the first matching document, so use `updateOne` helper if you want to update one document, and use `updateMany` when you want to update all matching documents.

Chapter 3 - Mastering Find

Chapter 1 provided a superficial look at the `find` command. There's more to `find` than understanding `selectors` though. We already mentioned that the result from `find` is a `cursor`. We'll now look at exactly what this means in more detail.

Field Selection

Before we jump into `cursors`, you should know that `find` takes a second optional parameter called "projection". This parameter is the list of fields we want to retrieve or exclude. For example, we can get all of the unicorns' names without getting back other fields by executing:

```
db.unicorns.find({}, {name: 1});
```

By default, the `_id` field is always returned. We can explicitly exclude it by specifying `{name:1, _id: 0}`.

Aside from the `_id` field, you cannot mix and match inclusion and exclusion. If you think about it, that actually makes sense. You either want to select or exclude one or more fields explicitly.

Cursors

A few times now I've mentioned that `find` returns a cursor whose execution is delayed until needed. However, what you've no doubt observed from the shell is that `find` executes immediately. This is a behavior of the shell only which automatically iterates over the cursor and returns documents. In drivers, you would have to use `getNext()`method to fetch documents from the cursor yourself.

Ordering

We can chain various methods to the cursor returned from `find`. The first that we'll look at is `sort`. We specify the fields we want to sort on as a JSON document, using 1 for ascending and -1 for descending. For example:

```
//heaviest unicorns first
db.unicorns.find().sort({weight: -1})
```

```
//by gender and then vampire kills:
db.unicorns.find().sort({gender: 1, vampires: -1})
```

As with a relational database, MongoDB can use an index for sorting. We'll look at indexes in more detail later on. However, you should know that MongoDB limits the size of your sort without an index. That is, if you try to sort a very large result set which can't use an index, you'll get an error. Some people see this as a limitation. In truth, I wish more databases had the capability to refuse to run unoptimized queries. (I won't turn every MongoDB drawback into a positive, but I've seen enough poorly optimized databases that I sincerely wish they had a strict-mode.)

Limiting Results

Limiting results can be accomplished via the `limit` cursor method. To get the top three heaviest unicorn, we could do:

```
db.unicorns.find().sort({weight: -1}).limit(3)
```

Using `limit` can be a way to avoid running into problems when sorting on non-indexed fields.

Count

The shell makes it possible to execute a `count` directly on a collection, such as:

```
db.unicorns.countDocuments({vampires: {$gt: 50}})
```

In reality, `countDocuments` is actually a special method that translates into an aggregation. We will learn more about aggregations in the next chapter. All drivers provide the same helper methods for such common operations.

In This Chapter

Using `find` and `cursors` is a straightforward proposition. There are a few additional commands that we'll either cover in later chapters or which only serve edge cases, but, by now, you should be getting pretty comfortable working in the mongo shell and understanding the fundamentals of MongoDB.

Chapter 4 - Aggregating Data

Aggregation Pipeline

Aggregation pipeline gives you a way to transform and combine documents in your collection. You do it by passing the documents through a pipeline that's somewhat analogous to the Unix "pipe" where you send output from one command to another to a third, etc.

The simplest aggregation you are probably already familiar with is the SQL `group by` expression. We already saw the simple `countDocuments()` method which turns out to be equivalent to grouping all documents to get their count, but what if we want to see how many unicorns are male and how many are female?

```
db.unicorns.aggregate([{$group:{_id:'$gender', total: {$sum:1}}}])
```

In the shell we have the `aggregate` helper which takes an array of pipeline operators that are called "stages". For a simple count grouped by something, we only need one such stage and it's called `$group`. This is the exact analog of `GROUP BY` in SQL where we create a new document with `_id` field indicating what field we are grouping by (here it's `gender`) and other fields usually getting assigned results of some aggregation, in this case we `$sum` 1 for each document that matches a particular gender. You probably noticed that the `_id` field was assigned `'$gender'` and not `'gender'` - the `'$'` before a field name indicates that the value of this field from the incoming document will be substituted.

What are some of the other pipeline operators or stages that we can use? The most common one to use before (and frequently after) `$group` would be `$match` - this is exactly like the `find` method and it allows us to aggregate only a matching subset of our documents, or to exclude some documents from our result.

```
db.unicorns.aggregate([{$match: {weight:{$lt:600}}},
   {$group: {_id:'$gender',total:{$sum:1},avgVamp:{$avg:'$vampires'}}},
   {$sort:{avgVamp:-1}} ])
```

Here we introduced another pipeline operator `$sort` which does exactly what you would expect, along with it we also have stages `$skip` and `$limit`. We also used a `$group` operator `$avg` along with `$sum` which we already saw in the first example.

MongoDB arrays are powerful and they don't stop us from being able to aggregate on values that are stored inside of them. We do sometimes need to be able to "flatten" them to properly count everything:

```
db.unicorns.aggregate([{$unwind:'$loves'},
   {$group:{_id:'$loves',total:{$sum:1},unicorns:{$addToSet:'$name'}}},
   {$sort:{total:-1}}, {$limit:1} ])
```

Here we will find out which single food item is loved by the most unicorns and we will also get the list of names of all the unicorns that love it. `$sort` and `$limit` in combination allow you to get answers to "top N" types of questions.

There are other powerful pipeline operator which allow you to transform values of fields as well as add (or remove) fields. Aggregation stages can use many different expressions which allows you to create or calculate new fields based on values in existing fields. For example, you can use math operators to add together values of several fields before finding out the average, or you can use string operators to create a new field that's a concatenation of some existing fields. There are even expressions that allow you to execute nearly arbitrary Javascript - though doing so would be less performant than sticking to native server expressions and operators.

This just barely scratches the surface of what you can do with aggregations. Aggregation is how you can do limited joins in MongoDB via the `$lookup` stage, as well as transitive closure expressions via recursive lookup called `$graphlookup`. You can also combine data from multiple pipelines using `$unionWith` - somewhat analogous with `SQL UNION`.

Aggregation Output

Aggregate command returns either a cursor to the result set (which you already know how to work with from Chapter 3) or it can write your results into a new collection using the `$out` pipeline operator. Using `$merge` pipeline stage you can output to an existing collection with powerful controls that let you specify exactly how the new output is merged with existing documents. You can see a lot more examples as well as all of the supported pipeline and expression operators in the MongoDB manual. For a more extensive collection of examples, take a look at the Practical MongoDB Aggregations eBook. Keep in mind that both Compass GUI and the Atlas web-based UI include an aggregation pipeline builder to help you write and debug powerful pipelines, and that often MongoDB drivers include aggregation builder helpers (Java and .Net/C# driver, for example).

In This Chapter

In this chapter we covered MongoDB's aggregation capabilities. Aggregation Pipeline is relatively simple to write once you understand how it's structured and it's a powerful way to group and analyze data. With the addition of user defined functions, its capabilities can be as boundless as any code you can write in JavaScript.

Chapter 5 - Data Modeling

Let's shift gears and have a more abstract conversation about MongoDB. Explaining a few new terms and some new syntax is a trivial task. Having a conversation about modeling with a new paradigm isn't as easy. The truth is that most of us are still finding out what works and what doesn't when it comes to modeling with these new technologies. It's a conversation we can start having, but ultimately you'll have to practice and learn on real code.

Out of all NoSQL databases, document-oriented databases are probably the most similar to relational databases - at least when it comes to modeling. However, the differences that exist are important.

Limited Joins

The first and most fundamental difference that you'll need to get comfortable with is MongoDB's minimal support of joins. I don't know the specific reason why some all types of joins aren't fully supported in MongoDB, but I do know that joins are generally seen as non-scalable. That is, once you start to split your data horizontally, you end up performing your joins on the client (the application server) anyway. Regardless of the reasons, the fact remains that if your data is fully normalized, MongoDB won't perform as well as RDBMS.

Without knowing anything else, to live in a world with fewer joins, we have to do some joins ourselves within our application's code. Essentially we need to issue a second query to `find` the relevant data in a second collection. Setting our data up isn't any different than declaring a foreign key in a relational database. Let's give a little less focus to our beautiful `unicorns` and a bit more time to our `employees`. The first thing we'll do is create an employee (I'm providing an explicit `_id` so that we can build coherent examples)

```
db.employees.insertOne({_id: 485, name: 'Leto'})
```

Now let's add a couple employees and set their manager as `Leto`:

```
db.employees.insertMany([
  {_id: 698, name: 'Duncan', manager: 485},
  {_id: 703, name: 'Moneo', manager: 485} ]);
```

(It's worth repeating that the `_id` can be any unique value. You will frequently use an `ObjectID` in real life, but here we will use numeric `_id`.)

Of course, to find all of Leto's employees, one simply executes:

```
db.employees.find({manager: 485})
```

There's nothing magical here. In the worst cases, most of the time, not using a join will merely require an extra query (likely indexed). Having said all that, MongoDB does provide `$lookup` stage in its aggregation framework which we saw in the previous chapter.

Arrays and Embedded Documents

Just because MongoDB doesn't have extensive support for joins doesn't mean it doesn't have a few tricks up its sleeve. Remember when we saw that MongoDB supports arrays as first class objects of a document? It turns out that this is incredibly handy when dealing with many-to-one or many-to-many relationships. As a simple example, if an employee could have two managers, we could simply store these in an array:

```
db.employees.insertOne({_id: 811, name: 'Siona', manager:[ 485, 698]})
```

Of particular interest is that, for some documents, `manager` can be a scalar value, while for others it can be an array. Our original `find` query will work for both:

```
db.employees.find({manager: 485})
```

You'll quickly find that arrays of values are much more convenient to deal with than many-to-many join-tables.

Besides arrays, MongoDB also supports embedded documents. Go ahead and try inserting a document with a nested document, such as:

```
db.employees.insertOne({_id: 501, name: 'Ghanima',
 family: {mother: 'Chani', father: 'Paul', brother: 485}})
```

In case you are wondering, embedded documents can be queried using a dot-notation:

```
db.employees.find({'family.mother': 'Chani'})
```

We'll briefly talk about where embedded documents fit and how you should use them.

Combining the two concepts, we can even embed arrays of documents:

```
db.employees.insertOne({_id: 502, name: 'Chani',
    family: [{relation:'mother', name: 'Chani'},
             {relation:'father', name: 'Paul'},
             {relation:'brother', name: 'Duncan'}]})
```

Denormalization

Yet another alternative to using joins is to denormalize your data. Historically, denormalization was reserved for performance-sensitive code, or when data should be snapshotted (like in an audit log). However, with the ever-growing popularity of NoSQL databases, many of which don't have any support for joins, denormalization as part of normal modeling is becoming increasingly common. This doesn't mean you should duplicate every piece of information in every document. However, rather than letting fear of duplicate data drive your design decisions, consider modeling your data based on what information belongs to what document.

For example, say you are writing a forum application. The traditional way to associate a specific `user` with a `post` is via a `userid` column within `posts`. With such a model, you can't display `posts` without retrieving (joining to) `users`. A possible alternative is simply to store the `name` as well as the `userid` with each `post`. You could even do so with an embedded document, like `user: {id: ObjectId('Something'), name: 'Karl'}`. Yes, if you let users change their name, you may have to update each document (which is one multi-update).

Adjusting to this kind of approach won't come easy to some. In a lot of cases, it won't even make sense. Don't be afraid to experiment with this approach though. It's not only suitable in some circumstances, but it can also be the best way to do things.

Which Should You Choose?

Arrays of ids can be a useful strategy when dealing with one-to-many or many-to-many scenarios. But more commonly, new developers are left deciding between using embedded documents versus doing "manual" referencing.

First, you should know that an individual document is currently limited to 16 megabytes in size. Knowing that documents have a size limit, though quite generous, gives you some idea of how they are intended to be used. At this point, it seems like most developers lean heavily on manual references for most of their relationships. Embedded documents are frequently leveraged, but mostly for smaller pieces of data which we want to always pull with the parent document. A real world example may be to store an `addresses` array of documents with each user, something like:

```
db.users.insert({name: 'leto', email: 'leto@dune.gov', addresses: [
  {street:"1633 Broadway", city:"New York", state:"NY",zip:"10019"},
  {street:"499 Hamilton", city:"Palo Alto", state:"CA",zip:"94301"}
]})
```

This doesn't mean you should underestimate the power of embedded documents or write them off as something of minor utility. Having your data model map directly to your objects makes things a lot simpler and often removes the need to join. This is especially true when you consider that MongoDB lets you query and index fields of embedded documents and arrays.

Few or Many Collections

Given that collections don't enforce any schema, it's entirely possible to build a system using a single collection with a mishmash of documents but it would be a very bad idea. Most MongoDB systems are laid out somewhat similarly to what you'd find in a relational system, though with fewer collections. In other words, if it would be a table in a relational database, there's a chance it'll be a collection in MongoDB (many-to-many join tables being an important exception as well as tables that exist only to enable one to many relationships with simple entities).

The conversation gets even more interesting when you consider embedded documents. The example that frequently comes up is a blog. Should you have a `posts` collection and a `comments` collection, or should each `post` have an array of `comments` embedded within it? Setting aside the 16MB document size limit for the time being (all of *Hamlet* is less than 200KB, so just how popular is your blog?), most developers prefer to separate things. It's simply cleaner, gives you better performance and is more explicit. MongoDB's flexible schema allows you to combine the two approaches by keeping comments in their own collection but embedding a few comments (maybe the first few) in the blog post to be able to display them with the post. This follows the principle of keeping together data that you want to get back in one query.

There's no hard rule (well, aside from 16MB). Play with different approaches and you'll get a sense of what does and does not feel right.

In This Chapter

Our goal in this chapter was to provide some helpful guidelines for modeling your data in MongoDB - a starting point, if you will. Modeling in a document-oriented system is different, but not too different, than in a relational world. You have more flexibility and one constraint, but for a new system, things tend to fit quite nicely. The only way you can go wrong is by not trying.

Chapter 6 - Performance

In this chapter, we look at a few performance topics as well as some of the tools available to MongoDB developers. We won't dive deeply into these topics, but we will examine the most important aspects of each.

Indexes

At the very beginning we discussed the `getIndexes` command which shows information on all the indexes in a collection. Indexes in MongoDB work a lot like indexes in a relational database: they help improve query and sorting performance. Indexes are created via

```
createIndex:
// where "name" is the field name
db.unicorns.createIndex({name: 1});
```

And dropped via `dropIndex`:

```
db.unicorns.dropIndex({name: 1});
```

A unique index can be created by supplying a second parameter and setting `unique` to `true`:

```
db.unicorns.createIndex({name: 1}, {unique: true});
```

Indexes can be created on embedded fields (again, using the dot-notation) and on array fields. We can also create compound indexes:

```
db.unicorns.createIndex({name: 1, vampires: -1});
```

The direction of your index (1 for ascending, -1 for descending) doesn't matter for a single key index, but it can make a difference for compound indexes when you are sorting on more than one indexed field.

The indexes page has additional information on indexes. You can create indexes in the shell, or use the UI provided by Compass or Atlas.

Explain

To see whether or not your queries are using an index, you can use the `explain` method:

```
db.unicorns.explain().find()
```

The output includes a field telling us what "plan" the optimizer used, `COLLSCAN` means the query was not indexed, how many objects were scanned, how long it took, and if the plan was `IXSCAN` what index was used as well as a few other pieces of useful information.

If we change our query to use an index, we'll see that the winning plan used `IXSCAN`, as well as which index was used to fulfill the request:

```
db.unicorns.explain().find({name: 'Pilot'})
```

The `explain()` method can be used with any command that could use an index, like `aggregate`, `update`, etc.

Stats

Obtain statistics on a MongoDB database by typing `db.stats()`. Most of the information deals with the size of your database. Statistics are also available per collection - for a collection named `unicorns`, typing `db.unicorns.stats()` will do the trick. Most of this information will relate to the size of your collection and its indexes. If you are using Atlas, there are multiple metrics screens showing you the same stats data in graphical format over time.

Profiler

MongoDB also has database profiling functionality - the output tells us what was run and when, and how many documents were scanned versus returned. You can enable the MongoDB profiler by executing:

```
db.setProfilingLevel(2);
```

With it enabled, we can run a command:

```
db.unicorns.find({weight: {$gt: 600}});
```

And then examine the profiler collection:

```
db.system.profile.find()
```

You disable the profiler by calling `setProfilingLevel` again but changing the parameter to `0`. Specifying `1` as the first parameter will profile queries that take more than 100 milliseconds.

100 milliseconds is the default threshold, you can specify a different minimum time, in milliseconds, with a second parameter:

```
//profile anything that takes more than 1 second
db.setProfilingLevel(1, 1000);
```

You can specify that only a sampling of all operations should be profiled, but even with sampling enabled, database profiling should be used very cautiously in production. More details about profiling and how to use it are in the MongoDB Documentation.

In This Chapter

In this chapter we looked at various commands, tools, and performance details of using MongoDB. We haven't touched on everything, but we've looked at some of the common ones. Indexing in MongoDB is similar to indexing with relational databases, as are many of the tools. However, with MongoDB, many of these are to the point and simpler to use.

Chapter 7 - Security and Backups

Security

When it was first released, MongoDB did not have security enabled by default. Rather, it relied on the individual performing installation to follow security best practices to lock down the data from malicious actors. A lot has changed, and many default settings now only allow open access to the data from `localhost` after installation. You should follow documented best practices to set up appropriate roles, users and permissions.

If you're not using MongoDB Atlas, you will need to follow these steps:

- enable access control and specify an authentication mechanism
- configure role based access-control
- encrypt communication (TLS/SSL)
- encrypt and protect data
- limit network exposure
- (optionally) audit system activity
- always run the latest version of MongoDB and the driver(s) to be sure any known issues are fixed

When using Atlas, security is already enforced for you, so fewer steps are required. You still must create an admin user and password, and then allow access to your data from specific IP addresses.

Backup and Restore

When you have production data, you want to make sure that you back it up on a regular basis (as well as restoring it). There are several approaches to doing backups with MongoDB, all described here. In production you will likely use disk level snapshots, but the simplest method for a small amount of data is using `mongodump` and `mongorestore` executables that are part of the MongoDB Tools package. Simply executing `mongodump` will connect to localhost or any connection string you pass to it, then backup all of your databases to a `dump` subfolder. You can type `mongodump --help` to see additional options. Common options are to backup only a specific database or collection. You can then use the `mongorestore` executable to restore a previously made backup. Both `mongodump` and `mongorestore` operate on BSON, which is MongoDB's native format.

To demonstrate, we could backup our `learn` database to a `backup` folder, by executing (this is its own executable which you run in a command/terminal window, not within the mongo shell itself):

```
mongodump -d learn -out backup
```

To restore only the `unicorns` collection, we could then do:

```
mongorestore -d learn -c unicorns backup/learn/unicorns.bson
```

It's worth pointing out that `mongoexport` and `mongoimport` are two other executables which can be used to export and import data to/from JSON or CSV. For example, we can get a JSON output by doing:

```
mongoexport -d learn -c unicorns
```

And a CSV output by doing:

```
mongoexport -d learn -c unicorns --csv --fields name,weight,vampires
```

Note that `mongoexport` and `mongoimport` cannot always represent your data fully. Only `mongodump` and `mongorestore` should ever be used for actual backups. You can read more about your backup options in the MongoDB documentation.

In This Chapter

In this chapter we very briefly listed security best practices and reviewed some basic options for backups.

Chapter 8 - When To Use MongoDB

By now you should have a feel for where and how MongoDB might fit into your existing system. There are enough new and competing storage technologies that it's easy to get overwhelmed by all of the choices.

For me, the most important lesson, which has nothing to do with MongoDB, is that you no longer have to rely on a single solution for dealing with your data. No doubt, a single solution has obvious advantages, and for a lot of projects - possibly even most - a single solution works. The idea isn't that you *must* use different technologies, but rather that you *can*. Only you know whether the benefits of introducing a new solution outweigh the costs.

With that said, I'm hopeful that what you've seen so far has made you see MongoDB as a general solution. It's been mentioned a couple times that document-oriented databases share a lot in common with relational databases. Therefore, rather than tiptoeing around it, let's simply state that MongoDB should be seen as a direct alternative to relational databases. Where one might see Lucene as enhancing a relational database with full text indexing, or Redis as a persistent key-value store, MongoDB is a central repository for your data.

Notice that I didn't call MongoDB a *replacement* for relational databases, but rather an *alternative*. It's a tool that can do what a lot of other tools can do. Some of it MongoDB does better, some of it MongoDB does worse. Let's dissect things a little further.

Flexible Schema

An oft-touted benefit of document-oriented databases is that they don't enforce a fixed schema. This makes them much more flexible than traditional database tables. I agree that flexible schema is a nice feature, but not for the main reason most people mention.

People talk about "schema-less" as though you'll suddenly start storing a crazy mishmash of data. There are domains and data sets which can really be a pain to model using relational databases, but I see those as edge cases. Schema-less is cool, but most of your data is going to be highly structured. It's true that having an occasional mismatch can be handy, especially when you introduce new features, but in reality it's nothing a nullable column probably wouldn't solve just as well.

For me, the real benefit of dynamic schema is the lack of setup and the reduced friction with OOP. This is particularly true when you're working with a static language. I've worked with MongoDB in both C# and Ruby, and the difference is striking. Ruby's dynamism and its popular ActiveRecord implementations already reduce much of the object-relational impedance mismatch. That isn't to say MongoDB isn't a good match for Ruby, it really is. Rather, I think most Ruby developers would see MongoDB as an incremental improvement, whereas C# or Java developers would see MongoDB as offering a fundamental shift in how they interact with their data.

Think about it from the perspective of a driver developer. You want to save an object? Serialize it to JSON (technically BSON, but close enough) and send it to MongoDB. There is no property mapping or type mapping. This straightforwardness definitely flows to you, the end developer.

Do keep in mind that MongoDB provides the ability to optionally enforce full or partial schemas via schema validation. This feature is powerful and lets you specify everything from types of fields, required and optional, ranges of values, and many other document constraints.

Writes

MongoDB allows you to control write behavior with respect to data durability. These settings, in addition to specifying how many servers should get your data before being considered successful, are configurable per-connection, per-collection, or per-write, giving you a great level of control over the trade-off between write latency and data durability. Since MongoDB 5.0, w:majority is the default write setting as it is considered a general best practice.

Special Collections and Indexes

MongoDB also supports many different types of special collections and indexes.

You can create read-only views by defining an aggregation pipeline on an existing collection or view.

If you have data that represents time series, MongoDB supports a special time series collection type which stores sequences of measurements over a period of time.

There are capped collections, which you create with pre-defined size. These collections automatically truncate old data as new data is appended and once they reach the specified maximum size. Capped collections do have some limitations, so as an alternative, if you want to "expire" your data based on time, you can use TTL Indexes where TTL stands for "time-to-live".

Durability

Prior to version 1.8 (released in 2010), MongoDB did not have single-server durability. That is, a server crash would likely result in lost or corrupt data. The solution had always been to run MongoDB in a multi-server setup (MongoDB supports replication). Journaling was one of the major features added in 1.8. Since version 2.0, MongoDB enables journaling by default, which allows fast recovery of the server in case of a crash or abrupt power loss. As of version 4.0, you no longer even have the option of disabling journaling.

Durability is only mentioned here because a lot has been made around MongoDB's previous lack of single-server durability. This may still show up in Google searches. Information you find about journaling being a missing feature is simply out of date.

Full Text Search

True full text search capability is a relatively recent addition to MongoDB. Text indexes in the server support fifteen languages with stemming and stop words. In MongoDB Atlas, there's full text search built on top of the Apache Lucene open source search engine. With MongoDB's support for arrays and full text search, you will only need to look to other solutions if you need a more powerful and full-featured full text search engine.

Transactions

MongoDB added full support for ACID transactions in 4.0 (extending it to sharded clusters in 4.2). Before that there were two alternatives, one which is great and still has its place, and the other that was cumbersome but flexible.

The first is its many atomic update operations. These are great, so long as they actually address your problem. We already saw some of the simpler ones, like `$inc` and `$set`. There are also commands like `findAndModify` which can update or delete a document and return it atomically. When atomicity can be ensured this way, it's preferable to use transactions for speed and overall scalability of the system.

The second, when atomic operations aren't enough, was to fall back to a two-phase commit. A two-phase commit is to transactions what manual dereferencing is to joins. It's a storage-agnostic solution that you do in code. Two-phase commits are actually quite popular in the relational world as a way to implement transactions across multiple databases. The general idea is that you store the state of the transaction within the actual document being updated atomically and go through the init-pending-commit/rollback steps manually. This is the case where using MongoDB multi-document transactions is a great option as they significantly simplify the application code.

Using transactions in MongoDB is as straightforward as in relational databases. You start a transaction which later you can commit or abort. To simplify your code even further, drivers automatically provide retry functionality on retryable errors.

Geospatial

A particularly powerful feature of MongoDB is its support for geospatial indexes. This allows you to store either geoJSON or x and y coordinates within documents and then find documents that are `$geoNear` a set of coordinates or `$geoWithin` a box or circle.

Replication

MongoDB replication works in some ways similarly to how relational database replication works. All production deployments should be replica sets, which consist of ideally three or more servers that hold the same data. Writes are sent to a single server, the primary, from where it's asynchronously replicated to every secondary. You can control whether you allow reads to happen on secondaries or not, which can help direct some special queries away from the primary, at the risk of reading slightly stale data. If the primary goes down, one of the secondaries will be automatically elected to be the new primary. Again, MongoDB replication is outside the scope of this book.

Change Streams

Change streams allow applications to subscribe to real-time data changes. Under the covers, change streams use MongoDB server replication and the aggregation pipeline. Applications can watch all data changes on a single collection, a database, or an entire deployment, and immediately react to them. Because change streams use the aggregation framework, applications can also filter for specific changes or transform the notifications at will.

Sharding

MongoDB supports auto-sharding. Sharding is an approach to scalability which partitions your data across multiple servers or clusters. A naive implementation might put all of the data for users with a name that starts with A-M on server 1 and the rest on server 2. Thankfully, MongoDB's sharding capabilities far exceed such a simple algorithm. Sharding is a topic well beyond the scope of this book, but you should know that it exists and that you should consider it, should your needs grow beyond a single replica set.

While replication can help performance somewhat (by isolating long running queries to secondaries, and reducing latency for some other types of queries), its main purpose is to provide high availability. Sharding is the primary method for scaling MongoDB clusters. Combining replication with sharding is the prescribed approach to achieve scaling and high availability.

Tools and Maturity

You probably already know the answer to this, but MongoDB is obviously younger than most relational database systems. This is absolutely something you should consider, though how much it matters depends on what you are doing and how you are doing it. Nevertheless, an honest assessment simply can't ignore the fact that MongoDB is younger and the available tooling around isn't great (although the tooling around a lot of very mature relational databases is pretty horrible too!). As an example, support for base-10 floating point numbers was only added in version 3.4, and there is still no support for `DATE` without time - MongoDB only supports the equivalent of `DATETIME` and `TIMESTAMP`.

On the positive side, drivers exist for a great many languages, the protocol is modern and simple, and development is happening at blinding speeds. MongoDB is in production at enough companies that concerns about maturity, while valid, have quickly become a thing of the past.

In This Chapter

The message from this chapter is that MongoDB, in most cases, can replace a relational database. It's much simpler and straightforward; it's faster and generally imposes fewer restrictions on application developers. The addition of transactions addressed a legitimate and serious concern. So, when people ask *where does MongoDB sit with respect to the new data storage landscape?* the answer is simple: **right in the middle**.

Conclusion

You should have enough information to start using MongoDB in a real project. There's more to MongoDB than what we've covered, but your next priority should be putting together what we've learned, and getting familiar with the driver you'll be using. The MongoDB website has a lot of useful information. The official community site MongoDB Community Forums is a great place to ask questions.

NoSQL was born not only out of necessity, but also out of an interest in trying new approaches. It is an acknowledgment that our field is ever-advancing and that if we don't try, and sometimes fail, we can never succeed. This, I think, is a good way to lead our professional lives.