Chapter 2

Introduction

It's not my fault the chapters are short, MongoDB is just easy to learn.

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 at a rather slow pace. 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.

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

Even though these transitions seem to happen overnight, the reality is that they can take years to become accepted practice. The 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 more generalized NoSQL solution. It should be viewed as an alternative to relational databases. Like relational databases, it too can benefit from being paired with some of the more specialized NoSQL solutions. MongoDB has advantages and

drawbacks, which we'll cover in later parts of this book.

Chapter 3

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 as well as being a useful administrative tool, 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, NoRM is a C# library which implements LINQ, and MongoMapper is a Ruby library which is ActiveRecord-friendly. 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.

1. Head over to the official download page and grab the binaries from the first row (the recommended stable version) for your operating system of choice. For development purposes, you can pick either 32-bit or 64-bit.
2. Extract the archive (wherever you want) and navigate to the `bin` subfolder. Don't execute anything just yet, but know that `mongod` is the server process and `mongo` is the client shell - these are the two executables we'll be spending most of our time with.
3. Create a new text file in the `bin` subfolder named `mongodb.config`.
4. Add a single line to your mongodb.config: `dbpath=PATH_TO_WHERE_YOU_WANT_TO_STORE_YOUR_DATABASE_FILES`. For example, on Windows you might do `dbpath=c:\mongodb\data` and on Linux you might do `dbpath=/var/lib/mongodb/data`.
5. Make sure the `dbpath` you specified exists.
6. Launch mongod with the `--config /path/to/your/mongodb.config` parameter.

As an example for Windows users, if you extracted the downloaded file to `c:\mongodb\` and you created `c:\mongodb\data\` then within `c:\mongodb\bin\mongodb.config` you would specify `dbpath=c:\mongodb\data\`. You could then launch `mongod` from a command prompt via `c:\mongodb\bin\mongod --config c:\mongodb\bin\mongodb.config`.

Feel free to add the `bin` folder to your path to make all of this less verbose. MacOSX and Linux users can follow almost identical directions. The only thing you should have to change are the paths.

Hopefully you now have MongoDB up and running. If you get an error, read the output carefully - the server is quite good at explaining what's wrong.

You can now launch `mongo` (without the *d*) which will connect a shell to your running server. Try entering `db.version()` to make sure everything's working as it should. Hopefully you'll see the version number you installed.

Chapter 4

Chapter 1 - The Basics

We begin our journey by getting to know the basic mechanics 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 a `row`.
4. A document is made up of one or more `fields`, which you can probably guess are a lot like `columns`.
5. `Indexes` in MongoDB function mostly like their RDBMS counterparts.
6. `Cursors` are different than the other five concepts 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 `documents`. Each `document` is made up of `fields`. `Collections` can be `indexed`, which improves lookup and sorting performance. Finally, when we get data from MongoDB we 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 (it's schema-less). 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. 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.help` (without the parentheses), you'll see the internal implementation of the `help` 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 really 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 are schema-less, we don't explicitly need to 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.insert({name: 'Aurora',
    gender: 'f', weight: 450})
```

The above line is executing `insert` against the `unicorns` collection, passing it a single parameter. 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, the database and collection it was created against and the fields included in the index.

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

```
db.unicorns.insert({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. A MongoDB query selector is like the `where` clause of an SQL statement. As such, you use it when finding, counting, updating and removing documents from collections. A selector is a JSON object, the simplest of which is `{}` which matches all documents. If we wanted to find all female unicorns, we could 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, issue the following inserts to get some data we can play with (I suggest you copy and paste this):

```
db.unicorns.insert({name: 'Horny',
    dob: new Date(1992,2,13,7,47),
    loves: ['carrot','papaya'],
    weight: 600,
    gender: 'm',
    vampires: 63});
db.unicorns.insert({name: 'Aurora',
    dob: new Date(1991, 0, 24, 13, 0),
    loves: ['carrot', 'grape'],
    weight: 450,
    gender: 'f',
    vampires: 43});
db.unicorns.insert({name: 'Unicrom',
    dob: new Date(1973, 1, 9, 22, 10),
    loves: ['energon', 'redbull'],
    weight: 984,
    gender: 'm',
    vampires: 182});
db.unicorns.insert({name: 'Roooooodles',
    dob: new Date(1979, 7, 18, 18, 44),
    loves: ['apple'],
    weight: 575,
    gender: 'm',
    vampires: 99});
db.unicorns.insert({name: 'Solnara',
    dob: new Date(1985, 6, 4, 2, 1),
    loves:['apple', 'carrot',
        'chocolate'],
    weight:550,
    gender:'f',
    vampires:80});
db.unicorns.insert({name:'Ayna',
```

```
    dob: new Date(1998, 2, 7, 8, 30),
    loves: ['strawberry', 'lemon'],
    weight: 733,
    gender: 'f',
    vampires: 40});
db.unicorns.insert({name:'Kenny',
    dob: new Date(1997, 6, 1, 10, 42),
    loves: ['grape', 'lemon'],
    weight: 690,
    gender: 'm',
    vampires: 39});
db.unicorns.insert({name: 'Raleigh',
    dob: new Date(2005, 4, 3, 0, 57),
    loves: ['apple', 'sugar'],
    weight: 421,
    gender: 'm',
    vampires: 2});
db.unicorns.insert({name: 'Leia',
    dob: new Date(2001, 9, 8, 14, 53),
    loves: ['apple', 'watermelon'],
    weight: 601,
    gender: 'f',
    vampires: 33});
db.unicorns.insert({name: 'Pilot',
    dob: new Date(1997, 2, 1, 5, 3),
    loves: ['apple', 'watermelon'],
    weight: 650,
    gender: 'm',
    vampires: 54});
db.unicorns.insert({name: 'Nimue',
    dob: new Date(1999, 11, 20, 16, 15),
    loves: ['grape', 'carrot'],
    weight: 540,
    gender: 'f'});
db.unicorns.insert({name: 'Dunx',
    dob: new Date(1976, 6, 18, 18, 18),
    loves: ['grape', 'watermelon'],
    weight: 704,
    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: {$gt: 700}})
//or (not quite the same thing, but for
//demonstration purposes)
db.unicorns.find({gender: {$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'.

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 or'd:

```
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("TheObjectId")})
```

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 5

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 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})
```

(If you've played with your unicorns collection and it doesn't have the original data anymore, go ahead and remove all documents and re-insert from the code in chapter 1.)

Now, if we look at the updated record:

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

You 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: new Date(1979, 7, 18, 18, 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.update({name: 'Roooooodles'},
    {$set: {weight: 590}})
```

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

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

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

Since no documents exists 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.update({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. So far, for the examples we've looked at, this might seem logical. However, if you executed something like:

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

You might expect to find all of your precious unicorns to be vaccinated. To get the behavior you desire, the `multi` option must be set to true:

```
db.unicorns.update({},
    {$set: {vaccinated: true }},
    {multi: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 the `multi` option when you want to update all matching documents.

Chapter 6

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.

Ordering

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. We can observe the true behavior of `cursors` by looking at one of the methods we can chain to `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 unicorn name then vampire kills:
db.unicorns.find().sort({name: 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.)

Paging

Paging results can be accomplished via the `limit` and `skip` cursor methods. To get the second and third heaviest unicorn, we could do:

```
db.unicorns.find()
    .sort({weight: -1})
    .limit(2)
    .skip(1)
```

Using `limit` in conjunction with `sort`, 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.count({vampires: {$gt: 50}})
```

In reality, `count` is actually a `cursor` method, the shell simply provides a shortcut. Drivers which don't provide such a shortcut need to be executed like this (which will also work in the shell):

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

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 7

Chapter 4 - 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.

No Joins

The first and most fundamental difference that you'll need to get comfortable with is MongoDB's lack of joins. I don't know the specific reason why some type of join syntax isn't 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 data *is* relational, and MongoDB doesn't support joins.

Without knowing anything else, to live in a join-less world, we have to do 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.insert({_id: ObjectId(
    "4d85c7039ab0fd70a117d730"),
    name: 'Leto'})
```

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

```
db.employees.insert({_id: ObjectId(
    "4d85c7039ab0fd70a117d731"),
    name: 'Duncan',
    manager: ObjectId(
    "4d85c7039ab0fd70a117d730")});
db.employees.insert({_id: ObjectId(
    "4d85c7039ab0fd70a117d732"),
    name: 'Moneo',
    manager: ObjectId(
    "4d85c7039ab0fd70a117d730")});
```

(It's worth repeating that the `_id` can be any unique value. Since you'd likely use an `ObjectId` in real life, we'll use them here as well.)

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

```
db.employees.find({manager: ObjectId(
    "4d85c7039ab0fd70a117d730")})
```

There's nothing magical here. In the worst cases, most of the time, the lack of join will merely require an extra query (likely indexed).

Arrays and Embedded Documents

Just because MongoDB doesn't have 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.insert({_id: ObjectId(
    "4d85c7039ab0fd70a117d733"),
    name: 'Siona',
    manager: [ObjectId(
    "4d85c7039ab0fd70a117d730"),
    ObjectId(
    "4d85c7039ab0fd70a117d732")] })
```

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: ObjectId(
    "4d85c7039ab0fd70a117d730")})
```

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.insert({_id: ObjectId(
    "4d85c7039ab0fd70a117d734"),
    name: 'Ghanima',
    family: {mother: 'Chani',
        father: 'Paul',
        brother: ObjectId(
    "4d85c7039ab0fd70a117d730")}})
```

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.insert({_id: ObjectId(
    "4d85c7039ab0fd70a117d735"),
    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, many of which don't have 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: 'Leto'}`. 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 to do this. 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 it.

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` documents with each user, something like:

```
db.users.insert({name: 'leto',
    email: 'leto@dune.gov',
    addresses: [{street: "229 W. 43rd St",
                 city: "New York", state:"NY",zip:"10036"},
                {street: "555 University",
                 city: "Palo Alto", state:"CA",zip:"94107"}]})
```

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 an 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 should prefer to separate things out. It's simply cleaner, gives you better performance and 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 8

Chapter 5 - 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 projects - possibly even most - a single solution is the sensible approach. 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 database 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 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.

Writes

One area where MongoDB can fit a specialized role is in logging. There are two aspects of MongoDB which make writes quite fast. First, you have an option to send a write command and have it return immediately without waiting for the write to be acknowledged. Secondly, you can control the 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-write, giving you a great level of control over write performance and data durability.

In addition to these performance factors, log data is one of those data sets which can often take advantage of schema-less collections. Finally, MongoDB has something called a capped collection. So far, all of the implicitly created collections we've created are just normal collections. We can create a capped collection by using the `db.createCollection` command and flagging it as capped:

```
//limit our capped collection to 1 megabyte
db.createCollection('logs', {capped: true,
    size: 1048576})
```

When our capped collection reaches its 1MB limit, old documents are automatically purged. A limit on the number of documents, rather than the size, can be set using `max`. Capped collections have some interesting properties. For example, you can update a document but it can't change in size. The insertion order is preserved, so you don't need to add an extra index to get proper time-based sorting. You can "tail" a capped collection the way you tail a file in Unix via `tail -f <filename>` which allows you to get new data as it arrives, without having to re-query it.

If you want to "expire" your data based on time rather than overall collection size, you can use TTL Indexes where TTL stands for "time-to-live".

Durability

Prior to version 1.8, 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.

Durability is only mentioned here because a lot has been made around MongoDB's past lack of single-server durability. This'll likely show up in Google searches for some time to come. Information you find about journaling being a missing feature is simply out of date.

Full Text Search

True full text search capability is a recent addition to MongoDB. It supports fifteen languages with stemming and stop words. 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 doesn't have transactions. It has two alternatives, one which is great but with limited use, and the other that is 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.

The second, when atomic operations aren't enough, is 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 MongoDB website has an example illustrating the most typical example (a transfer of funds). 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.

MongoDB's support for nested documents and flexible schema design makes two-phase commits slightly less painful, but it still isn't a great process, especially when you are just getting started with it.

Data Processing

Before version 2.2 MongoDB relied on MapReduce for most data processing jobs. As of 2.2 it has added a powerful feature called aggregation framework or pipeline, so you'll only need to use MapReduce in rare cases where you need complex functions for aggregations that are not yet supported in the pipeline. In the next chapter we'll look at Aggregation Pipeline and MapReduce in detail. For now you can think of them as feature-rich and different ways to `group by` (which is an understatement). For parallel processing of very large data, you may need to rely on something else, such as Hadoop. Thankfully, since the two systems really do complement each other, there's a MongoDB connector for Hadoop.

Of course, parallelizing data processing isn't something relational databases excel at either. There are plans for future versions of MongoDB to be better at handling very large sets of data.

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 `$near` a set of coordinates or `$within` a box or circle. This is a feature best explained via some visual aids, so I invite you to try the 5 minute geospatial interactive tutorial, if you want to learn more.

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, the lack of support for base-10 floating point numbers will obviously be a concern (though not necessarily a show-stopper) for systems dealing with money.

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, are quickly becoming 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 lack of transactions can be a legitimate and serious concern. However, when people ask *where does MongoDB sit with respect to the new data storage landscape?* the answer is simple: **right in the middle**.

Chapter 9

Chapter 6 - 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 `count()` method, 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. For a simple count grouped by something, we only need one such operator 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 incoming document will be substituted.

What are some of the other pipeline operators 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 `$sort` which does exactly what you would expect, along with it we also get `$skip` and `$limit`. We also used a `$group` operator `$avg`.

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 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 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 is another powerful pipeline operator called `$project` (analogous to the projection we can specify to `find`) which allows you not just to include certain fields, but 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.

This just barely scratches the surface of what you can do with aggregations. In 2.6 aggregation got more powerful as the aggregate command returns either a cursor to the result set (which you already know how to work with from Chapter 1) or it can write your results into a new collection using the `$out` pipeline operator. You can see a lot more examples as well as all of the supported pipeline and expression operators in the MongoDB manual.

MapReduce

MapReduce is a two-step approach to data processing. First you map, and then you reduce. The mapping step transforms the inputted documents and emits a key=>value pair (the key and/or value can be complex). Then, key/value pairs are grouped by key, such that values for the same key end up in an array. The reduce gets a key and the array of values emitted for that key, and produces the final result. The map and reduce functions are written in JavaScript.

With MongoDB we use the `mapReduce` command on a collection. `mapReduce` takes a map function, a reduce function and an output directive. In our shell we can create and pass a JavaScript function. From most libraries you supply a string of your functions (which is a bit ugly). The third parameter sets additional options, for example we could filter, sort and limit the documents that we want analyzed. We can also supply a `finalize` method to be applied to the results after the `reduce` step.

You probably won't need to use MapReduce for most of your aggregations, but if you do, you can read more about it on my blog and in MongoDB manual.

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 data. MapReduce is more complicated to understand, but its capabilities can be as boundless as any code you can write in JavaScript.

Chapter 10

Chapter 7 - Performance and Tools

In this last 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 either topic, but we will examine the most important aspects of each.

Indexes

At the very beginning we saw 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 `ensureIndex`:

```
// where "name" is the field name
db.unicorns.ensureIndex({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.ensureIndex({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.ensureIndex({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.

Explain

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

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

The output tells us that a `BasicCursor` was used (which means non-indexed), that 12 objects were scanned, how long it took, what index, if any, 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 a `BtreeCursor` was used, as well as the index used to fulfill the request:

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

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.

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 perscribed approach to achieve scaling and high availability.

Stats

You can obtain statistics on a database by typing `db.stats()`. Most of the information deals with the size of your database. You can also get statistics on a collection, say `unicorns`, by typing `db.unicorns.stats()`. Most of this information relates to the size of your collection and its indexes.

Profiler

You 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:

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

The output tells us what was run and when, how many documents were scanned, and how much data was returned.

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);
```

Backups and Restore

Within the MongoDB `bin` folder is a `mongodump` executable. Simply executing `mongodump` will connect to localhost and backup all of your databases to a `dump` subfolder. You can type `mongodump --help` to see additional options. Common options are `--db DBNAME` to back up a specific database and `--collection COLLECTIONNAME` to back up a specific collection. You can then use the `mongorestore` executable, located in the same `bin` folder, to restore a previously made backup. Again, the `--db` and `--collection` can be specified to restore a specific database and/or collection. `mongodump` and `mongorestore` operate on BSON, which is MongoDB's native format.

For example, to back up our `learn` database to a `backup` folder, we'd execute (this is its own executable which you run in a command/terminal window, not within the mongo shell itself):

```
mongodump --db learn --out backup
```

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

```
mongorestore --db learn --collection 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 from JSON or CSV. For example, we can get a JSON output by doing:

```
mongoexport --db learn --collection unicorns
```

And a CSV output by doing:

```
mongoexport --db learn \
    --collection unicorns \
    --csv --fields name,weight,vampires
```

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

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 simple to use.

Chapter 11

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 MongoDB user group 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.