This summary was designed in order to extract the main concepts of the book “MongoDB”: The Definitive Guide

# CHAPTER 2: Getting Started

**Documents:**
At the heart of MongoDB is the concept of a document: an ordered set of keys with associated values.
The keys in a document are strings. Any UTF-8 character is allowed in a key, with a few notable exceptions:
• Keys must not contain the character \0 (the null character). This character is used to signify the end of a key.
• The . and $ characters have some special properties and should be used only in certain circumstances, as described in later chapters. In general, they should be considered reserved, and drivers will complain if they are used inappropriately.
• Keys starting with _ should be considered reserved; although this is not strictly enforced.
A final important thing to note is that documents in MongoDB cannot contain duplicate keys.

**Collections:**
A collection is a group of documents

*Schema Free*
Collections are schema-free. This means that the documents within a single collection can have any number of different “shapes.”
Because any document can be put into any collection, the question often arises: “Why do we need separate collections at all?

I) Keeping different kinds of documents in the same collection can be a nightmare for developers and admins.
II) It is much faster to get a list of collections than to extract a list of the types in a collection.
III) Grouping documents of the same kind together in the same collection allows for data locality
IV) We begin to impose some structure on our documents when we create indexes. (This is especially true in the case of unique indexes.) These indexes are defined per collection. By putting only documents of a single type into the same collection, we can index our collections more efficiently.

*Naming*

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

x)The empty string ("") is not a valid collection name.

x)Collection names may not contain the character \0 (the null character) because this delineates the end of a collection name.

x)You should not create any collections that start with system., a prefix reserved for system collections

x)User-created collections should not contain the reserved character $ in the name

*Subcollections*

One convention for organizing collections is to use namespaced subcollections separated by the . character. **Subcollections are a great way to organize data in MongoDB, and their use is highly recommended.**

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

• GridFS, a protocol for storing large files, uses subcollections to store file metadata separately from content chunks (see Chapter 7 for more information about GridFS).

• The MongoDB web console organizes the data in its DBTOP section by subcollection (see Chapter 8 for more information on administration).

• Most drivers provide some syntactic sugar for accessing a subcollection of a given collection.

**Databases:**

In addition to grouping documents by collection, MongoDB groups collections into databases. A single instance of MongoDB can host several databases, each of which can be thought of as completely independent. A database has its own permissions, and each database is stored in separate files on disk.

One thing to remember about database names is that they will actually end up as files on your filesystem.

Database names can be any UTF-8 string, with the following restrictions:

• The empty string ("") is not a valid database name.

• A database name cannot contain any of these characters: ' ' (a single space), ., $, /, \, or \0 (the null character).

• Database names should be all lowercase.

• Database names are limited to a maximum of 64 bytes.

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

admin
This is the “root” database, in terms of authentication. If a user is added to the admin database, the user automatically inherits permissions for all databases. There are also certain server-wide commands that can be run only from the admin database, such as listing all of the databases or shutting down the server.
local
This database will never be replicated and can be used to store any collections that should be local to a single server (see Chapter 9 for more information about replication and the local database).
config
When Mongo is being used in a sharded setup (see Chapter 10), the config database is used internally to store information about the shards.

**Autogenerated_id:**
Is better to generate the id on the client side. **“Work should be pushed out of server and to the drivers whenever possible”**

**Date:**
Be carefull with using Date() against new Date()

**Arrays:**
**Embedded Documents:**

**Numbers:**
JavaScript has one “number” type. Because MongoDB has three number types (4-byte integer, 8-byte integer, and 8-byte float), the shell has to hack around JavaScript’s limitations a bit. By default, any number in the shell is treated as a double by MongoDB.

# CHAPTER 3: Creating, updating and Deleting

# Documents

**Inserting and Saving Documents**

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

**Batch Insert**

Batch inserts are intended to be used in applications, such as for inserting a couple hundred sensor data points into an analytics collection at once. They are useful only if you are inserting multiple documents into a single collection: you cannot use batch inserts to insert into multiple collections with a single request. If you are just importing

**Inserts: Internals and Implications**

When you perform an insert, the driver you are using converts the data structure into BSON, which it then sends to the database (see Appendix C for more on BSON). The database understands BSON and checks for an "_id" key and that the document's size does not exceed 4MB, but other than that, it doesn't do data validation; it just saves

**Removing Documents**

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

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

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

**Remove Speed**

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

**Updating Documents**

```
> db.users.update({"name" : "joe"}, joe);
```

**Using Modifiers**

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

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

**Getting started with the "$set" modifier**

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

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

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

"$set" can even change the type of the key it modifies.

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

**Array modifiers**

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

**Modifier speed**

MongoDB leaves some padding around a document to allow for changes in size (and,

in fact, figures out how much documents usually change in size and adjusts the amount of padding it leaves accordingly), but it will eventually have to allocate new space for a document if you make it much larger than it was originally.

**Upserts**

An upsert is a special type of update. If no document is found that matches the update criteria, a new document will be created by combining the criteria and update documents. If a matching document is found, it will be updated normally. Upserts can be very handy because they eliminate the need to “seed” your collection: you can have the same code create and update documents.
We can eliminate the race condition(in contrast with the same query using a javascript code) and cut down on the amount of code by just sending an upsert (the third parameter to update specifies that this should be an upsert):

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

**The save Shell Helper**

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

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

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

**Updating Multiple Documents**

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

```
> db.count.update({x : 1}, {$inc : {x : 1}}, false, true)
> db.runCommand({getLastError : 1})
```

**Returning Updated Documents**

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

*findAndModify*: A string, the collection name.
*query*: A query document, the criteria with which to search for documents.
*sort*: Criteria by which to sort results.
update: A modifier document, the update to perform on the document found.
*remove*: Boolean specifying whether the document should be removed.
*new*: Boolean specifying whether the document returned should be the updated document or the preupdate document. Defaults to the preupdate document.

**The Fastest Write This Side of Mississippi**

The three operations that this chapter focused on (inserts, removes, and updates) seem instantaneous because none of them waits for a database response. They are not asynchronous; they can be thought of as “fire-and-forget” functions.
speed of your network. This works well most of the time; however, sometimes something goes wrong: a server crashes, a rat chews through a network cable, or a data center is in a flood zone. If the server disappears, the client will happily send some writes to a server that isn’t there, entirely unaware of its absence...

**Safe Operations**

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

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

The price of performing “safe” operations is performance: waiting for a database response takes an order of magnitude longer than sending the message, ignoring the client-side cost of handling

exceptions. (This cost varies by language but is usually fairly

**Catching "Normal" Errors**

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

**Requests and Connections**

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

Note that this is a per-connection queue: if we open two shells, we will have two connections to the database. If we perform an insert in one shell, a subsequent query in the other shell might not return the inserted document. However, within a single shell, if we query for the document after inserting, the document will be returned. This behavior can be difficult to duplicate by hand, but on a busy server, interleaved inserts/queries are very likely to occur. Often developers run into this when they insert data in one thread and then check that it was successfully inserted in another. For a second or two, it looks like the data was not inserted, and then it suddenly appears.

This behavior is especially worth keeping in mind when using the Ruby, Python, and Java drivers, because all three drivers use connection pooling. For efficiency, these drivers open multiple connections (a pool) to the server and distribute requests across them.

# CHAPTER 4: Queries

**Introduction to find**

> db.c.find()
returns everything in the collection c.
> db.users.find({"age" : 27})

**Specifying Which Keys to Return(!)**

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

> db.users.find({}, {"fatal_weakness" : 0})
//returns all the users without the fatal_weakness attribute
> db.users.find({}, {"username" : 1, "_id" : 0})
//returns all the usernames without the _id attribute
```

**Limitations**

There are some restrictions on queries. The value of a query document must be a constant as far as the database is concerned. (It can be a normal variable in your own code.)

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

**Query Criteria**

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

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

> db.users.find({"username" : {"$ne" : "joe"}} ) //users whose name isn’t Joe
```

**OR Queries**

```
> db.raffle.find({"ticket_no" : {"$in" : [725, 542, 390]}})
> db.users.find({"user_id" : {"$in" : [12345, "joe"]})

> db.raffle.find({"ticket_no" : {"$nin" : [725, 542, 390]}})
//This query returns everyone who did not have tickets with those numbers.

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

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

With a normal AND-type query, you want to narrow your results down as far as possible in as

few arguments as possible. OR-type queries are the opposite: they are most efficient if the first arguments match as many documents as possible.

**$not**

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

//We want, to return users with "id_num"s of 2, 3, 4, 5, 7, 8, 9, 10, 12, and so on.

**Rules for Conditionals**

-Position:

This generally holds true: conditionals are an inner document key, and modifiers are always a key in the outer document

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

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

-Multiple uses

Any number of conditionals can be used with a single key. Multiple update modifiers cannot be used on a single key.

~~{"$inc" : {"age" : 1}, "$set" : {age : 40}}~~ //Wrong, it is modifing “age” attribute twice

**Type-Specific Queries**

However, null not only matches itself but also matches “does not exist.”

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

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

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

**Regular Expressions**

MongoDB uses the Perl Compatible Regular Expression (PCRE) library to match regular expressions; any regular expression syntax allowed by PCRE is allowed in

MongoDB.

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

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

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

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

**Querying Arrays(!)**
Querying for elements of an array is simple. An array can mostly be treated as though each element is the value of the overall key.

> db.food.insert({"fruit" : ["apple", "banana", "peach"]})
> db.food.find({"fruit" : "banana"})//Note that we directly use the key inside the array

If you want to query for a specific element of an array, you can specify an index using the syntax key.index:
> db.food.find({"fruit.2" : "peach"})
Arrays are always 0-indexed, so this would match the third array element against the string "peach".

***$all***
Then we can find all documents with both "apple" and "banana" elements by querying with "$all":
> db.food.find({fruit : {$all : ["apple", "banana"]}})

***$size***
A useful conditional for querying arrays is "$size", which allows you to query for arrays of a given size. Here's an example
> db.food.find({"fruit" : {"$size" : 3}})

***The $slice operator***
The keys to be returned. The special "$slice" operator can be used to return a subset f elements for an array key.

> db.blog.posts.findOne(criteria, {"comments" : {"$slice" : [23, 10]}}
//This would skip the first 23 elements and return the 24th through 34th.
For example, suppose we had a blog post document and we wanted to return the first 10 comments
> db.blog.posts.findOne(criteria, {"comments" : {"$slice" : 10}})
Alternatively, if we wanted the last 10 comments, we could use -10:
> db.blog.posts.findOne(criteria, {"comments" : {"$slice" : -10}})

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

Querying for an entire embedded document works identically to a normal query:

```
{
        "name" : {
                first" : "Joe",
                last" : "Schmoe"
        },
        "age" : 45
}
> db.people.find({"name" : {"first" : "Joe", "last" : "Schmoe"}})
```
However, if Joe decides to add a middle name key, suddenly this query won't work anymore; it doesn't match the entire embedded document! This type of query is also order-sensitive; {"last" : "Schmoe", "first" : "Joe"} would not be a match.

If possible, it's usually a good idea to query for just a specific key or keys of an embedded document. Then, if your schema changes, all of your queries won't suddenly break because they're no longer exact matches. You can query for embedded keys using dotnotation:
```
> db.people.find({"name.first" : "Joe", "name.last" : "Schmoe"})
```

***"$elemMatch"***
```
> db.blog.find({"comments" : {"$elemMatch" : {"author" : "joe", "score" : {"$gte" : 5}}}})
```
"$elemMatch" allows us to "group" our criteria. As such, it's only needed when you have more than one key you want to match on in an embedded document

**$where Queries**

Key/value pairs are a fairly expressive way to query, but there are some queries that they cannot represent. For queries that cannot be done any other way, there are "$where" clauses, which allow you to execute arbitrary JavaScript as part of your query. This allows you to do (almost) anything within a query.The most common case for this is wanting to compare the values for two keys in a document, for instance, if we had a list of items and wanted to return documents where

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

**Cursors**

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

```
> for(i=0; i<100; i++) {
.. db.c.insert({x : i});
... }
> var cursor = db.collection.find();
> while (cursor.hasNext()) {
.. obj = cursor.next();
... // do stuff
... }
```

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

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

At this point, the query has not been executed yet. All of these functions merely build

the query. Now, suppose we call the following:

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

**Limits, Skips, and Sorts**

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

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

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

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

Comparison order

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

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

**Avoiding Large Skips(!)**

Using skip for a small number of documents is fine. For a large number of results, skip can be slow (this is true in nearly every database, not just MongoDB) and should be avoided. Usually you can build criteria into the documents themselves to avoid it.(read the example on the book)

**Finding a random document**

*Efficient way:*

```
> db.people.insert({"name" : "joe", "random" : Math.random()})
> db.people.insert({"name" : "john", "random" : Math.random()})
> db.people.insert({"name" : "jim", "random" : Math.random()})
> var random = Math.random()
> result = db.foo.findOne({"random" : {"$gt" : random}})
> if (result == null) {
... result = db.foo.findOne({"random" : {"$lt" : random}})
... }
```

**Advanced Query Options**

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

$maxscan : integer

$max : document

**Getting Consistent Results**
*We recommend to read it directly from the book because it possesses drawing which help the understanding of this topic*

**Cursor Internals**

There are two sides to a cursor: the client-facing cursor and the database cursor that the client-side one represents. We have been talking about the client-side one up untilOn the server side, a cursor takes up memory and resources. Once a cursor runs out of esults or the client sends a message telling it to die, the database can free the resources it was using. Freeing these resources lets the database use them for other things, which is good, so we want to make sure that cursors can be freed quickly (within reason).

There are a couple of conditions that can cause the death (and subsequent cleanup) of a cursor. First, when a cursor finishes iterating through the matching results, it will clean itself up. Another way is that, when a cursor goes out of scope on the client side, the drivers send the database a special message to let it know that it can kill that cursor. Finally, even if the user hasn't iterated through all the results and the cursor is still in scope, after 10 minutes of inactivity, a database cursor will automatically "die."
This "death by timeout" is usually the desired behavior: very few applications expect their users to sit around for minutes at a time waiting for results. However, sometimes you might know that you need a cursor to last for a long time. In that case, many drivers have implemented a function called immortal, or a similar mechanism, which tells the database not to time out the cursor. If you turn off a cursor's timeout, you must iterate through all of its results or make sure it gets closed. Otherwise, it will sit around in the database hogging resources forever.

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

# CHAPTER 5: Indexing

**Introduction to Indexing**

MongoDB's indexes work almost identically to typical relational database indexes, so if you are familiar with those, you can skim this section for syntax specifics. We'll go over some indexing basics.

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

- The document you pass to ensureIndex is of the same form as the document passed to the sort function: a set of keys with value 1 or -1, depending on the direction you want the index to go. If you have only a single key in the index, direction is irrelevant.
  The MongoDB query optimizer will reorder query terms to take advantage of indexes: if you query for {"x" : "foo", "y" : "bar"} and you have an index on {"y" : 1, "x" : 1}, MongoDB will figure it out.
- The disadvantage to creating an index is that it puts a little bit of overhead on every insert, update, and remove.Sometimes the most efficient solution is actually not to use an index. In general, if a query is returning a half or more of the collection, it will be more efficient for the database to just do a table scan instead of having to look up the index and then the value for almost every single document

**Scaling Indexes**

Suppose we have a collection of status messages from users. We want to query by user and date to pull up all of a user's recent statuses.It is completely different and much better to index by {date : -1, user : 1} because of it

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

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

**Indexing Keys in Embedded Documents(!)**

Indexes can be created on keys in embedded documents in the same way that they are created on normal keys:

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

**Indexing for Sorts(!)**

As your collection grows, you'll need to create indexes for any large sorts your queries are doing. If you call sort on a key that is not indexed, MongoDB needs to pull all of that data into memory to sort it. Thus, there's a limit on the amount you can sort

**Uniquely Identifying Indexes**

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

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

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

**Unique Indexes(!)**

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

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

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

**Dropping Duplicates**

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

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

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

**Compound Unique Indexes**

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

**Using explain(!)**

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

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

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

{
   "cursor" : "BasicCursor",
    "indexBounds" : [ ],
    "nscanned" : 64,
    "nscannedObjects" : 64,
    "n" : 64,
    "millis" : 0,
    "allPlans" : [
      {
          "cursor" : "BasicCursor",
          "indexBounds" : [ ]
       }
   ]
}
```

The important parts of this result are as follows:
*"cursor" : "BasicCursor"*
This means that the query did not use an index (unsurprisingly, because there was no query criteria). We'll see what this value looks like for an indexed query in a moment.
*"nscanned" : 64*
This is the number of documents that the database looked through. You want to make sure this is as close to the number returned as possible.
*"n" : 64*
This is the number of documents returned. We're doing pretty well here, because the number of documents scanned exactly matches the number returned. Of course, given that we're returning the entire collection, it would be difficult to do otherwise.
*"millis" : 0*
The number of milliseconds it Wtook the database to execute the query. 0 is a good time to shoot for.

We could query the system.indexes collection to find out more about this index (e.g., whether it is unique or what keys it includes):

```
> db.system.indexes.find({"ns" : "test.c", "name" : "age_1"})
```

**Use hints**

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

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

**Index Administration**

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

The system.indexes collection has detailed information about each index, but the system.namespaces collection also lists their names.

It is important to remember that the size of the collection name plus the index name cannot exceed 127 bytes.

**Changing Indexes**

*Creating:*

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

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

*Deleting:*

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

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

**Geospatial Indexing(!)**

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

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

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

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

{ "gps" : [ 0, 100 ] }

{ "gps" : { "x" : -30, "y" : 30 } }
{ "gps" : { "latitude" : -180, "longitude" : 180 } }
The keys can be anything; for example, {"gps" : {"foo" : 0, "bar" : 1}} is valid.

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

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

This finds all of the documents in the map collection, in order by distance from the point (40, -73). A default limit of 100 documents is applied if no limit is specified.

"$near" conditional. "$within" can take an increasing number of shapes; check the online documentation for geospatial indexing to see the most up-to-date list of what's supported (http://www.mongodb.org/display/DOCS/Geospatial+Indexing).

**Compound Geospatial Indexes**

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

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

Then we can quickly find the nearest coffee shop:

> db.map.find({"location" : {"$near" : [-70, 30]}, "desc" : "coffeeshop"}).limit(1)

# CHAPTER 6: Aggregation

**count**

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

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

**distint**

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

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

**group(!)**

If you are familiar with SQL, group is similar to SQL's GROUP BY.
group allows you to perform more complex aggregation. You choose a key to group by, and MongoDB divides the collection into separate groups for each value of the chosen key. For each group, you can create a result document by aggregating the documents that are members of that group.
*We recommend to read the book example carefully.*

**Using a Finalizer**

Finalizers can be used to minimize the amount of data that needs to be transferred from the database to the user, which is important, because the group command's output needs to fit in a single database response.

*Group and finalizer example*

```
> db.runCommand({"group" : {
... "ns" : "posts",
... "key" : {"tags" : true},
... "initial" : {"tags" : {}},
... "$reduce" : function(doc, prev) {
...     for (i in doc.tags) {
```

```
...         if (doc.tags[i] in prev.tags) {
...             prev.tags[doc.tags[i]]++;
...         } else {
...             prev.tags[doc.tags[i]] = 1;
...         }
..     },
... "finalize" : function(prev) {
...     var mostPopular = 0;
...     for (i in prev.tags) {
..         if (prev.tags[i] > mostPopular) {
...             prev.tag = i;
...             mostPopular = prev.tags[i];
...         }
...     }
..     delete prev.tags
... }}})
```

**MapReduce(!)**

MapReduce is the Uzi of aggregation tools. Everything described with count, distinct, and group can be done with MapReduce, and more. It is a method of aggregation that can be easily parallelized across multiple servers. It splits up a problem,

The price of using MapReduce is speed: group is not particularly speedy, but MapReduce is slower and is not supposed to be used in “real time.”

MapReduce as a background job, it creates a collection of results, and then you can query that collection in real time.

Example 2: Categorizing Web Pages

Suppose we have a site where people can submit links to other pages, such as reddit.com. Submitters can tag a link as related to certain popular topics, e.g., “politics,” “geek,” or “icanhascheezburger.” We can use MapReduce to figure out which topics are the most popular, as a combination of recent and most-voted-for.

First, we need a map function that emits tags with a value based on the popularity and recency of a document:

```
map = function() {
    for (var i in this.tags) {
        var recency = 1/(new Date() - this.date);
        var score = recency * this.score;
        emit(this.tags[i], {"urls" : [this.url], "score" : score});
    }
};
```

Now we need to reduce all of the emitted values for a tag into a single score for that tag:

```
reduce = function(key, emits) {
    var total = {urls : [], score : 0}
```

```
    for (var i in emits) {
        emits[i].urls.forEach(function(url) {
            total.urls.push(url);
        }
        total.score += emits[i].score;
    }
    return total;
};
```

The final collection will end up with a full list of URLs for each tag and a score showing how popular that particular tag is.

# CHAPTER 7: Advanced Topics

**Commands**

**Database Commands**

In addition to these basic operations, MongoDB supports a wide range of advanced operations that are implemented as commands. Commands implement all

> db.runCommand({getLastError : 1})

**How Commands Work**

> db.runCommand({"drop" : "test"});

Commands in MongoDB are actually implemented as a special type of query that gets performed on the $cmd collection. When the MongoDB server gets a query on the $cmd collection, it handles it using special logic, rather than the normal code for handling queries. Almost all MongoDB drivers provide a helper method like runCommand for running commands, but commands can always be run using a simple query if necessary

**Command Reference(!)**

There are two ways to get an up-to-date list of all of the commands supported by a MongoDB server:

• Run db.listCommands() from the shell, or run the equivalent listCommands command from any other driver.

• Browse to the http://localhost:28017/_commands URL on the MongoDB admin interface (for more on the admin interface see Chapter 8).

**Capped Collections(!)**

MongoDB also supports a different type of collection, called a capped collection, which is created in advance and is fixed in size (see Figure 7-1). **in size (see Figure 7-1). Having** fixed-size collections brings up an interesting question: what happens when we try to insert into a capped collection that is already full? The answer is that capped collections behave like

circular queues: if we're out of space, the oldest document(s) will be deleted, and the new one will take its place (see Figure 7-2). This means that capped collections automatically age-out the oldest documents as new documents are inserted.

Certain operations are not allowed on capped collections. Documents cannot be removed or deleted (aside from the automatic age-out described earlier), and updates that would cause documents to move (in general updates that cause documents to grow in size) are disallowed.
A final difference between capped and normal collections is that in a capped collection, there are no indexes by default, not even an index on "_id".

**Properties and Use Cases**

First, inserts into a capped collection are extremely fast. When doing an insert, there is never a need to allocate additional space.The inserted document can always be placed directly at the "tail" of the collection, overwriting old documents if needed. By default, there are also no indexes to update on an insert, so an insert is essentially a single memcpy.
Another interesting property of capped collections is that queries retrieving documents in insertion order are very fast.
Finally, capped collections have the useful property of automatically aging-out old data as new data is inserted. The combination of fast inserts, fast queries for documents
Sorted by insertion order, and automatic age-out makes capped collections ideal for use cases like **logging**. In fact, the primary motivation for including capped collections

**Creating and Converting to Capped Collections**

```
> db.createCollection("my_collection", {capped: true, size: 100000, max: 100});
{ "ok" : true }

> db.runCommand({convertToCapped: "test", size: 10000});
{ "ok" : true }
```

**Sorting Au Naturel**

There is a special type of sort that you can do with capped collections, called a natural sort. Natural order is just the order that documents appear on disk.

```
> db.my_collection.find().sort({"$natural" : -1})
```

**Tailable Cursors**

Tailable cursors are a very special type of persistent cursor that are not closed when their results are exhausted. the command, will continue fetching output for as long as possible. Because the cursors do not die when they runs out of results, they can continue to fetch new results as they are added to the collection. Tailable cursors can be used only on capped collections.

**GridFS: Storing Files(!)**

GridFS is a mechanism for storing large binary files in MongoDB. There are several reasons why you might consider using GridFS for file storage:

• Using GridFS can simplify your stack. If you're already using MongoDB, GridFS obviates the need for a separate file storage architecture.
• GridFS will leverage any existing replication or autosharding that you've set up for MongoDB, so getting failover and scale-out for file storage is easy.
• GridFS can alleviate some of the issues that certain filesystems can exhibit when being used to store user uploads. For example, GridFS does not have issues with storing large numbers of files in the same directory.

• You can get great disk locality with GridFS, because MongoDB allocates data files in 2GB chunks.

**Getting Started with GridFS: mongofiles**

The easiest way to get up and running with GridFS is by using the mongofiles utility. mongofiles is included with all MongoDB distributions and can be used to upload, download, list, search for, or delete files in GridFS.

```
$ ./mongofiles put foo.txt
$ ./mongofiles list
$ ./mongofiles get foo.txt
$ cat foo.txt
```

**Under the Hood**

The basic idea behind GridFS is that we can store large files by splitting them up into chunks and storing each chunk as a separate document. In addition to storing each chunk of a file, we store a single document that groups the chunks together and contains metadata about the file.

The chunks for GridFS are stored in their own collection. By default chunks will use the col.lection fs.chunks, but this can be overridden if needed. Within the chunks collection the structure of the individual documents is pretty simple:

```
{
    "_id" : ObjectId("..."),
    "n" : 0,
    "data" : BinData("..."),
    "files_id" : ObjectId("...")
}
```

**Server-Side Scripting**

JavaScript can be executed on the server using the db.eval function. It can also be stored

in the database and is used in some database commands.
*db.eval*
db.eval is a function that allows you to execute arbitrary JavaScript on the MongoDB server. It takes a string of JavaScript, sends it to MongoDB (which executes it), and returns the result.

**Stored JavaScript(!)**
MongoDB has a special collection for each database called system.js, which can store JavaScript variables. These variables can then be used in any of MongoDB's JavaScript contexts, including "$where" clauses, db.eval calls, and MapReduce jobs. You can add variables to system.js with a simple insert:
> db.system.js.insert({"_id" : "x", "value" : 1})
System.js can be used to store JavaScript code as well as simple values.
( - ) There are downsides to using stored JavaScript: it keeps portions of your code out of source control, and it can obfuscate JavaScript sent from the client

( + ) The best reason for storing JavaScript is if you have multiple parts of your code (or code in different programs or languages) using a single JavaScript function.

**Security(!)**
Executing JavaScript is one of the few times you must be careful about security with MongoDB. If done incorrectly, server-side JavaScript is susceptible to injection attacks
To prevent this, you should use a scope to pass in the username.
Most drivers have a special type for sending code to the database, since code can actually be a composite of a string and a scope. A scope is just a document mapping variable names to values.

**Database Referencest(!)**
DBRefs are like URLs: they are simply a specification for uniquely identifying a reference to document. They do not automatically load the document any
more than a URL automatically loads a web page into a site with a link.
looks like the following:
{"$ref" : collection, "$id" : id_value}
The DBRef references a specific collection and an id_value that we can use to find a single document by its "_id" within that collection. These two pieces of information allow us to use a DBRef to uniquely identify and reference any document within a MongoDB database. If we want to reference a document in a different database, DBRefs support an optional third key that we can use, "$db":
{"$ref" : collection, "$id" : id_value, "$db" : database}

In short, the best times to use DBRefs are when you're storing heterogeneous references to documents in different collections, like in the previous example or when you want to take advantage of some additional DBRef-specific functionality in a driver or tool.

```
{"_id" : 5, "author" : "mike", "text" : "MongoDB is fun!"}
{"_id" : 20, "author" : "kristina", "text" : "... and DBRefs are easy, too",
 "references": [{"$ref" : "users", "$id" : "mike"}, {"$ref" : "notes", "$id" : 5}]}
> var note = db.notes.findOne({"_id" : 20});
> note.references.forEach(function(ref) {
... printjson(db[ref.$ref].findOne({"_id" : ref.$id}));
... });
{ "_id" : "mike", "display_name" : "Mike D" }
{ "_id" : 5, "author" : "mike", "text" : "MongoDB is fun!" }
```

# CHAPTER 8: Administration

| CHAPTER 8:Administration We recommend to read directly from the book |
|---|

**Starting and stoping MongoDB**

_Starting from the Command Line

o --dbpath o --porto --forko --logpath o --config

_File-Based Configuration

_Stopping MongoDB

**Monitoring**

- Using the Admin Interface: To see the admin interface, start the database and go to http://localhost:28017 in a web
- ServerStatus:
  > db.runCommand({"serverStatus" : 1})
- mongostats
- Third-Part Plugs-ins

**Security and Authentication**

1. Authentication Basics:
   > use admin
   switched to db admin
   > db.addUser("root", "abcd");
   {
   "user" : "root",
   "readOnly" : false,

```
    "pwd" : "1a0f1c3c3aa1d592f490a2addc559383"
}
> use test
switched to db test
> db.addUser("test_user", "efgh");
{
    "user" : "test_user",
    "readOnly" : false,
    "pwd" : "6076b96fc3fe6002c810268702646eec"
}
> db.addUser("read_only", "ijkl", true);
{
    "user" : "read_only",
    "readOnly" : true,
    "pwd" : "f497e180c9dc0655292fee5893c162f1"
}
```

2. How Authentication Works
   Users of a given database are stored as documents in its system.users collection. The structure of a user document is {"user" : username, "readOnly" : true, "pwd" :password hash}. The password hash is a hash based on the username and password chosen.

   Other Security Considerations(!)

   There are a couple of options besides authentication that should be considered when locking down a MongoDB instance. First, even when using authentication, th MongoDB wire protocol is not encrypted. If that is a requirement, consider using SSH tunneling or another similar mechanism to encrypt traffic between clients and the MongoDB server.
   We suggest always running your MongoDB servers behind a firewall or on a networ accessible only through your application servers. If you do have MongoDB on a machine accessible to the outside world, however, it is recommended that you start it with the --bindip option, which allows you to specify a local IP address that mongod will be bound to. For instance, to only allow connections from an application server running on the same machine, you could run mongod --bindip localhost.

**Backup (!)**

The directory to use as the data directory is configurable through the --dbpath option when starting MongoDB. Regardless of where the data directory is, its contents form a complete representation of the data stored in MongoDB. This suggests that making a backup of MongoDB is as simple as creating a copy of all of the files in the data directory.

It is not safe to create a copy of the data directory while MongoDB is running unless the server has done a full fsync and is not allowing writes. Such a backup will likely turn out to b corrupt and need repairing

It is not safe to create a copy of the data directory while MongoDB is running unless the server has done a full fsync and is not allowing writes. Such a backup will likely turn out to be corrupt and need repairing

**MongoDump**
utility that is included with all MongoDB distributions. mongodump works by querying against a running MongoDB server and writing all of the documents it contains to disk. Because mongodump is just a regular client, it can be run against a live instance of MongoDB, even one handling other requests and performing writes.

Because mongodump operates using the normal MongoDB query mechanism, the backups it produces are not necessarily point-in-time snapshots of the server's data. Another consequence of the fact that mongodump acts through the normal query mechanism is that it can cause some performance degradation for other clients throughout the duration of the backup.

*MongoRestore*
Along with mongodump, MongoDB distributions include a corresponding tool for restoring data from a backup, mongorestore. mongorestore takes the output from running mongodump and inserts the backed-up data into a running instance of MongoDB.
$ ./mongodump -d test -o backup
$ ./mongorestore -d foo --drop backup/test/

**fsync and Lock**
The fsync command will force the MongoDB server to flush all pending writes to disk.It will also, optionally, hold a lock preventing any further writes to the database until the server is unlocked. This write lock is what allows the fsync command to be useful for backups.
> db.runCommand({"fsync" : 1, "lock" : 1});
After performing the backup, we need to unlock the database again:
> db.$cmd.sys.unlock.findOne();
Here we run the currentOp command to ensure that the lock has been released. (It may>
db.currentOp();

**Slave Backups**
The slave will always have a copy of the data that is nearly in sync with the master. Because we're not depending on the performance of the slave or its availability for reads or writes, we are free to use any of the three options above: shutting down, the dump and restore tools, or the fsync command.

| |
|---|
| Backing up from a slave is the recommended way to handle data backups with MongoDB. |

**Repair**
There will unfortunately always be cases when a server with no backups (or slaves to failover to) fails. In the case of a power failure or a software crash, the disk will be fine when the machine comes back up. Because of the way MongoDB stores data, however, we are not guaranteed that the data on the disk is OK to use: corruption might have occurred
The easiest way to repair all of the databases for a given server is to start up a server

with --repair: mongod --repair. To repair a single database on a running server, you can use the repairDatabase method from the shell. If we wanted to repair the database test, we would do the following:

# CHAPTER 9: Replication

**Master-Slave Replication(!)**
This mode is very flexible and can be used for backup, failover, read scaling, and more
$ mkdir -p ~/dbs/master
$ ./mongod --dbpath ~/dbs/master --port 10000 --master
$ mkdir -p ~/dbs/slave
$ ./mongod --dbpath ~/dbs/slave --port 10001 --slave --source localhost:10000
All slaves must be replicated from a master node. There is currently no mechanism for replicating from a slave (daisy chaining), because slaves do not keep their own oplog

There is no explicit limit on the number of slaves in a cluster, but having a thousand slaves querying a single master will likely overwhelm the master node. In practice clusters with less than a dozen slaves tend to work well.

*Options*

| | | |
|---|---|---|
| --only | --slave delay | --fastsync |
| --autoresync | --oplogSize | |

*Adding and Removing Sources*
You can specify a master by starting your slave with the --source option, but you can also configure its source(s) from the shell.
> use local
> db.sources.insert({"host" : "localhost:27017"})

**Replica Sets(!)**

A replica set is basically a master-slave cluster with automatic failover. The biggest difference between a master-slave cluster and a replica set is that a replica set does not have a single master: one is elected by the cluster and may change to another node if the current master goes down. However, they look very similar: a replica set always has a single master node (called a primary) and one or more slaves (called secondaries). The nice thing about replica sets is how automatic everything is. First, the set itself does a lot of the administration for you, promoting slaves automatically and making sure you won't run into inconsistencies. For a developer, they are easy to use: you specify a few servers in a set, and the driver will automatically figure out all of the servers in the set and handle failover if the current master dies.

*Initializing a Set*

First, we create our data directories and choose ports for each server:

```
$ mkdir -p ~/dbs/node1 ~/dbs/node2
```

We have one more decision to make before we start up the servers: we must choose name for this replica set.

```
$ ./mongod --dbpath ~/dbs/node1 --port 10001 --replSet blort/morton:10002
$ ./mongod --dbpath ~/dbs/node2 --port 10002 --replSet blort/morton:10001
$ ./mongod --dbpath ~/dbs/node3 --port 10003 --replSet blort/morton:10001
```

This is because there's one more step: initializing the set in the shell.

```
$ ./mongo morton:10001/admin
```

*Nodes in a Replica Set*

At any point in time, one node in the cluster is primary, and the rest are secondary. The primary node is essentially the master, the difference being that which node is designated as primary can vary over time.

standard

This is a regular replica set node. It stores a full copy of the data being replicated, takes part in voting when a new primary is being elected, and is capable of becoming the primary node in the set.

passive

Passive nodes store a full copy of the data and participate in voting but will never become the primary node for the set.

arbiter

An arbiter node participates only in voting; it does not receive any of the data being replicated and cannot become the primary node.

Secondary nodes will pull from the primary node's oplog and apply operations, just like a slave in a master-slave system. A secondary node will also write the operation to its own local oplog, however, so that it is capable of becoming the primary. Operations in the oplog also include a monotonically increasing ordinal. This ordinal is used to determine how up-to-date the data is on any node in the cluster.

*Failover and Primary Election*
Read directly from the book.

**Performing Operations on a Slave**

- The primary purpose and most common use case of a MongoDB slave is to function as failover mechanism in the case of data loss or downtime on the master node.
- A slave can be used as a source for taking backups (see Chapter 8).
- It can also be used for scaling out reads or for performing data processing jobs on.

**Read Scaling**

One way to scale reads with MongoDB is to issue queries against slave nodes. By issuing queries on slaves, the workload for the master is reduced.One important note about using slaves to scale reads in MongoDB is that replication is asynchronous. This means that when data is inserted or updated on the master, the data on the slave will be out-of-date momentarily. This is important to consider if you are serving some requests using queries to slaves.

**Using Slaves for Data Processing**

Another interesting technique is to use slaves as a mechanism for offloading intensive processing or aggregation to avoid degrading performance on the master.
Starting with both --slave and --master may seem like a bit of a paradox. What it means, however, is that you'll be able to write to the slave, query on it like usual, and basically treat it like you would a normal MongoDB master node. In addition, the slave will continue to replicate data from the actual master. This way, you can perform blocking operations on the slave without ever affecting the performance of the master

When using this technique, you should be sure never to write to any database on the slave that is being replicated from the master. The slave will not revert any such writes in order to properly mirror the master.

**How It Works**

At a very high level, a replicated MongoDB setup always consists of at least two servers, or nodes. One node is the master and is responsible for handling normal client requests.The other node(s) is a slave and is responsible for mirroring the data stored on the master. The master keeps a record of all operations that have been performed on it. The slave periodically polls the master for any new operations and then performs them on its copy of the data. By performing all of the same operations that have been performed on the master node, the slave keeps its copy of the data up-to-date with the master's.

| **The Oplog** |
| --- |
| |

The record of operations kept by the master is called the oplog, short for operation log. The oplog is stored in a special database called local, in the oplog.$main collection. Each document in the oplog represents a single operation performed on the master server. The documents contain several keys, including the following:

ts: Timestamp for the operation. The timestamp type is an internal type used to track when operations are performed. It is composed of a 4-byte timestamp and a 4-byte incrementing counter.

op: Type of operation performed as a 1-byte code (e.g., “i” for an insert).

ns: Namespace (collection name) where the operation was performed.

o: Document further specifying the operation to perform. For an insert, this would be the document to insert. One important note about the oplog is that it stores only operations that change the state of the database. A query, for example, would not be stored in the oplog. Thi makes sense because the oplog is intended only as a mechanism for keeping the data on slaves in sync with the master. The operations stored in the oplog are also not exactly those that were performed on the master server itself. The operations are transformed before being stored such that they are idempotent. This means that operations can be applied multiple times on a slave with no ill effects, so long as the operations are applied in the correct order (e.g., an incrementing update, using "$inc", will be transformed to a "$set" operation).
A final important note about the oplog is that it is stored in a capped collection (see

Syncing
When a slave first starts up, it will do a full sync of the data on the master node. The slave will copy every document from the master node, which is obviously an expensive operation. After the initial sync is complete, the slave will begin querying the master’s oplog and applying operations in order to stay up-to-date.
If the application of operations on the slave gets too far behind the actual operations being performed on the master, the slave will fall out of sync.

When a slave gets out of sync, replication will halt, and the slave will need to be fully resynced from the master. This resync can be performed manually by running the command {"resync" : 1} on the slave’s admin database or automatically by starting the slave with the --autoresync option. Either way, doing a resync is a very expensive operation, and it’s a situation that is best avoided by choosing a large enough oplog size.
**To avoid out of sync slaves, it’s important to have a large oplog so that the master can store a long history of operations**. A larger oplog will obviously use up more disk space, but in general this is a good trade-off to make

**Replication State and the Local Database**

- The local database is used for all internal replication state, on both the master and the slave. The local database’s name is local, and its contents will never be replicated.

- Other replication state stored on the master includes a list of its slaves. (Slaves perform a handshake using the handshake command when they connect to the master.) This list is stored in the slaves collection: > db.slaves.find()

- Slaves also store state in the local database. They store a unique slave identifier in the me collection, and a list of sources, or nodes, that they are slaving from, in the sources collection: > db.sources.find(){ "_id" : ObjectId("4c1287178e00e93d1858567b"), "host" : "localhost:27017", "source" : "main", "syncedTo" : { "t" : 1276283096000, "i" : 1 }, "localLogTs" : { "t" : 0, "i" : 0 } }

- Both the master and slave keep track of how up-to-date a slave is, using the timestamp stored in "syncedTo". Each time the slave queries the oplog for new operations, it uses "syncedTo" to specify which new operations it needs to apply or to find out if it is out of sync.

**Blocking for Replication**

MongoDB's getLastError command allows developers to enforce guarantees about how up-to-date replication is by using the optional "w" parameter. Here we run a getLastError that will block until at least N servers have replicated the last write operation:

```
> db.runCommand({getLastError: 1, w: N});
```

**Administration**

**Diagnostics**

```
> db.printReplicationInfo();
> db.printSlaveReplicationInfo();
```

**Changing the Oplog Size**

--oplogSize. To change the oplog size to size, we shut down the master and run the following:

```
$ rm /data/db/local.*
$ ./mongod --master --oplogSize size
```

# CHAPTER 9: Sharding

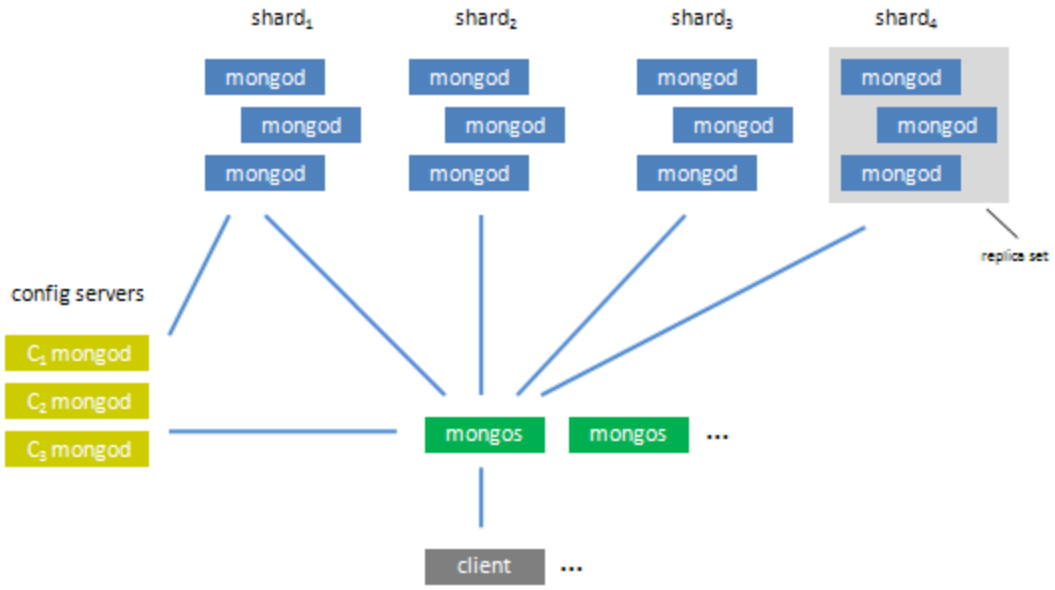
**Introduction**

Sharding refers to the process of splitting data up and storing different portions of the data on different machines; the term partitioning is also sometimes used to describe this concept. By splitting data up across machines, it becomes possible to store more Manual sharding can be done with almost any database software. It is when an application maintains connections to several different database servers, each of which are completely independent. The application code manages storing different data ondifferent servers and

querying against the appropriate server to get data back. This approach can work well but becomes difficult to maintain when adding or removing nodes

**Autosharding in MongoDB(!)**

The basic concept behind MongoDB's sharding is to break up collections into smaller chunks. These chunks can be distributed across shards so that each shard is responsible for a subset of the total data set. We don't want our application to have to know what shard has what data, or even that our data is broken up across multiple shards, so we run a routing process called mongos in front of the shards. This router knows where all of the data is located, so applications can connect to it and issue requests normally.




**When to Shard(!)**

One question people often have is when to start sharding. There are a couple of signs that sharding might be a good idea:

- You've run out of disk space on your current machine
- You want to write data faster than a single mongod can handle.
- You want to keep a larger proportion of data in memory to improve performance.

**The Key to Sharding: Shard Keys**

When you set up sharding, you choose a key from a collection and use that key's values to split up the data. This key is called a shard key.

**Incrementing Shard Keys Versus Random Shard Keys(!)**

The distribution of inserts across shards is very dependent on which key we're sharding on. If we have a high write load and want to evenly distribute writes across multiple shards, we should pick a shard key that will jump around more. This could be a hash of the

timestamp in the log example or a key like "logMessage", which won't have any particular pattern to it.

Whether your shard key jumps around or increases steadily, it is important to choose a key that will vary somewhat. If, for example, we had a "logLevel" key that had only values "DEBUG", "WARN", or "ERROR", MongoDB won't be able to break up your data into more than three chunks (because there are only three different values). If you have a key with very little variation and want to use it as a shard key anyway, you can do so by creating a compound shard key on that key and a key that varies more, like "logLevel" and "timestamp".

**Starting the Servers**

1) First we need to start up our config server and mongos. The config server needs to be started first, because mongos uses it to get its configuration. The config server can be started like any other mongod process:

```
$ mkdir -p ~/dbs/config
$ ./mongod --dbpath ~/db
```

(A config server does not need much space or resources. (A generous estimate is 1KB of config server space per 200MB of actual data.)

2)Add Mongos Process

Ruting servers don't even need a data directory, but they need to know where the config server is:

```
$ ./mongos --port 30000 --configdb localhost:20000
```

3)Adding a shard

A shard is just a normal mongod instance (or replica set):

```
$ mkdir -p ~/dbs/shard1
$ ./mongod --dbpath ~/dbs/shard1 --port 10000
```

4)Connect to the mongos process we started and add the shard to the cluster.

```
$ ./mongo localhost:30000/admin
> db.runCommand({addshard : "localhost:10000", allowLocal : true})
```

//The  "allowLocal" key is necessary only if you are running the shard on  localhost.

**Sharding Data**

MongoDB won't just distribute every piece of data you've ever stored: you have to explicitly turn sharding on at both the database and collection levels.

```
> db.runCommand({"enablesharding" : "foo"})
```

Once you've enabled sharding on the database level, you can shard a collection by running the shardcollection command:

```
> db.runCommand({"shardcollection" : "foo.bar", "key" : {"_id" : 1}})
```

**Production Configuration**

The example in the previous section is fine for trying sharding or for development. However, when you move an application into production, you'll want a more robust setup. To set up sharding with no points of failure, you'll need the following:

- Multiple config servers
- Multiple mongos servers
- Replica sets for each shard
- w set correctly (see the previous chapter for information on w and replication)