MongoDB

MongoDB’s strength lies in versatility, power, ease of use, and ability to handle jobs both large and small

It was designed as a scalable database—the name Mongo comes from “hu*mongo*us”—with performance and easy data access as core design goals.

It is a document database, which allows data to persist in a nested state, and importantly, it can query that nested data in an ad hoc fashion.

It enforces no schema so documents can optionally contain fields or types that no other document in the collection contains.

There are some huge production MongoDB (often just called *Mongo*) deployments out there, like Foursquare, bit.ly, and CERN, for collecting Large Hadron Collider data.

Hu(mongo)us

Mongo hits a sweet spot between the powerful queryability of a relational database and the distributed nature of other datastores like Riak or HBase

Mongo is a JSON document database (though technically data is stored in a binary form of JSON known as BSON).

A Mongo document can be likened to a relational table row without a schema, whose values can nest to an arbitrary depth.

To get an idea of what a JSON document is…

> printjson(db.towns.findOne())

{

"\_id" : ObjectId("54182d320dba70519f9a4b4a"),

"name" : "New York",

"population" : 22200000,

"last\_census" : ISODate("2009-07-31T00:00:00Z"),

"famous\_for" : [

"statue of liberty",

"food"

],

"mayor" : {

"name" : "Michael Bloomberg",

"party" : "I"

}

}

Mongo is an excellent choice for an ever-growing class of web projects with large-scale data storage requirements but very little budget to buy big-iron hardware.

Thanks to its lack of structured schema, Mongo can grow and change along with your data model.

If you’re in a web startup with dreams of enormity or are already large with the need to scale servers horizontally, consider MongoDB.

CRUD and Nesting

We will use a Windows installation (https://www.mongodb.org/downloads)

To prevent typos, Mongo requires you to first create the directory where mongod will store its data.

A common location is /data/db (c:/data/db)

If it’s not already running, you can fire up the Mongo service by running mongod (c:/Program Files/MongoDB/Server/4.0/bin)

Command-Line Fun

To create a new database named book, first run this command in your terminal (from c:/Program Files/MongoDB/Server/4.0/bin)

**$ mongo book**

Can also use intellij plugin for Mongo

It will connect to the MySQL-inspired command-line interface.

Machine generated alternative text:
Administrator: C:NWindcwsN.system32Ncmd.exe - mongo book 
reserved. 
Microsoft Windows LUersion 6 
opyright (c) 2øø9 Microsoft Corporation . 
RI I rights 
: XUsers 
: X)cd "MongoDB 2.6 Standard" 
:XMongoDB 2.6 Standard)cd bin 
: XMongoDB 2 .6 Standard Win )mongo 
ongoDB shell version: 2.6.3 
onnecting to: book 
elcome to the MongoDB shell. 
or interactive help. type "he Ip". 
book 
or more comprehensive documentation. see 
http:/'docs . mongodb.org/ 
uestions? Try the support group 
http://groups . google.com/group/mongodb—user 

Typing help in the console is a good start.

We’re currently in the book database, but you can view others via show dbs and switch databases with the use command.

Creating a collection (similar to a *bucket* in Riak nomenclature) in Mongo is as easy as adding an initial record to the collection.

Since Mongo is schema-less, there is no need to define anything up front; merely using it is enough.

What’s more, our book database doesn’t really exist until we first add values into it.

The following code creates/inserts a towns collection:

db.towns.insert({

name: *"New York"*,

population: 22200000,

last\_census: ISODate(*"2009-07-31"*),

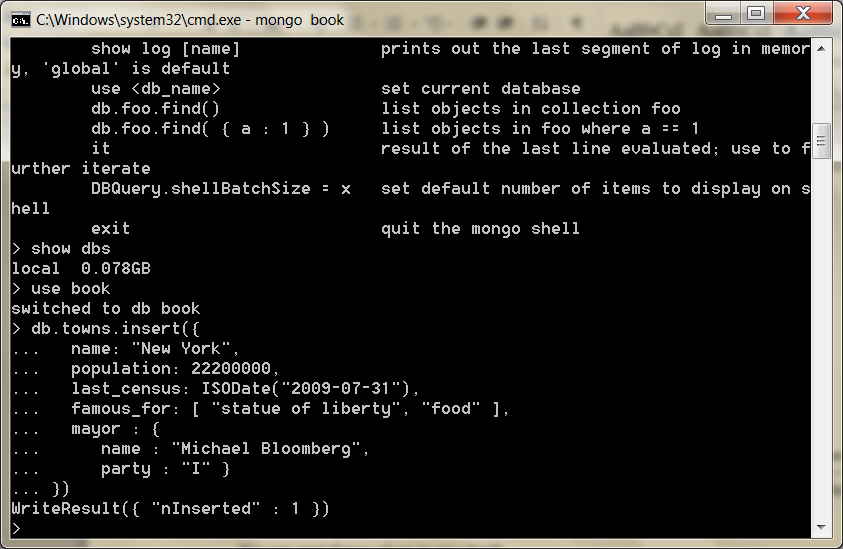
famous\_for: [ *"statue of liberty"*, *"food"* ],

mayor : {

name : *"Michael Bloomberg"*,

party : *"I"* }

})



In the previous section, we said documents were JSON (well, really BSON: <http://en.wikipedia.org/wiki/BSON>), so we add new documents in JSON format, where brackets like {...} denote an object (aka a hashtable or Map) with keyed values and where brackets like [...] denote an array.

You can nest these values to any depth.

With the show collections command, you can verify the collection now exists.

show collections

system.indexes  
towns

We just created towns, whereas system.indexes always exists.

Machine generated alternative text:
Administrator: - 
MongoDB shell version: 2.6.3 
connecting to: book 
elcome to the MongoDB shell. 
or interactive help. type "he Ip". 
mongo book 
or more comprehensive documentation. see 
http:/'docs . mongodb.org/ 
uestions? Try the support group 
http://groups . google.com/group/mongodb—user 
db. towns. 
"New York". 
name : 
population: 222øøøøø. 
last_census: 
famous_for: "statue of liberty". 
mayor 
name 
"Michael Bloomberg". 
"n Inserted" 
show collections 
ystem. indexes 

We can list the contents of a collection via find().

We formatted the output here for readability, but yours may just output as a single wrapped line.

db.towns.find()

{

"\_id" : ObjectId(*"4d0ad975bb30773266f39fe3"*),

"name" : *"New York"*,

"population": 22200000,

"last\_census": *"Fri Jul 31 2009 00:00:00 GMT-0700 (PDT)"*,

"famous\_for" : [ *"statue of liberty"*, *"food"* ],

"mayor" :

{ "name" : *"Michael Bloomberg"*,

"party" : *"I"* } }

Machine generated alternative text:
Administrator: - mongo book 
Welcome to the MongoDB shell. 
For interactive help. type "he Ip". 
or more comprehensive documentation. see 
http:/'docs . mongodb.org/ 
uestions? Try the support group 
http://groups . google.com/group/mongodb—user 
db. towns. 
"New York". 
name : 
population: 222øøøøø. 
last_census: 
famous_for: "statue of liberty". 
"New York". 
'population 
mayor 
name 
"Michael Bloomberg". 
"n Inserted" 
show collections 
system. indexes 
towns 
db. towns. find() 
. Objectld(" 
. 222øøøøø. 
"last 
"statue of liberty". 
54182d32ødbavø519f9a4b4a"). 
'name " 
census " 
"famous_for" 
'mayor" 
"name" 
"Michael Bloomberg". 

Unlike a relational database, Mongo does not support server-side joins.

A single JavaScript call will retrieve a document *and* all of its nested content, free of charge.

You may have noticed that the JSON output of your newly inserted town contains an \_id field of ObjectId.

This is akin to SERIAL incrementing a numeric primary key

The ObjectId is always 12 bytes, composed of a timestamp, client machine ID, client process ID, and a 3-byte incremented counter.

What’s great about this autonumbering scheme is that each process on every machine can handle its own ID generation without colliding with other mongod instances.

This design choice gives a hint of Mongo’s distributed nature.

**JavaScript**

Mongo’s native tongue is JavaScript, be it as complex as mapreduce queries or as simple as asking for help.

> db.help()

> db.towns.help()

These commands will list available functions related to the given object.

db is a JavaScript object that contains information about the current database.

db.x is a JavaScript object representing a collection (named x).

Commands are just JavaScript functions.

> **typeof** db

object

> **typeof** db.towns

object

> **typeof** db.towns.insert **function**

If you want to inspect the source code for a function, call it without parameters or parentheses (think more Python than Ruby).

db.towns.insert

**function** (obj, \_allow\_dot) {

**if** (!obj) {

**throw** *"no object passed to insert!"*;

}

**if** (!\_allow\_dot) {

this.\_validateForStorage(obj);  
 }

**if** (**typeof** obj.\_id == *"undefined"*) { **var** tmp = obj;

obj = {\_id:**new** ObjectId}; **for** (**var** key **in** tmp) {

obj[key] = tmp[key];  
 }

}  
 this.\_mongo.insert(this.\_fullName, obj);  
 this.\_lastID = obj.\_id;

}

Let’s populate a few more documents into our towns collection by creating our own JavaScript function.

mongo/insert\_city.js

function insertCity(name,population,last\_census,famous\_for, mayor\_info ) { db.towns.insert({name:name,population:population,last\_census: ISODate(last\_census),famous\_for:famous\_for,mayor:mayor\_info });}

You can just paste the code into the shell.

Then we can call it.

insertCity(*"Punxsutawney"*, 6200, *'2008-31-01'*, [*"phil the groundhog"*], { name : *"Jim Wehrle"* }

)

Machine generated alternative text:
Administrator: CAWindcwsN.system32Ncmd.exe - mongo book 
2ø14-ø9-16T13:46 .?12.ø1øø SyntaxError: Unexpected token 
function insertCity(name. population. last census. db. t 
owns . name :name.population : population. last census census). f 
mayor_info));) 
2ø14-ø9-16T13:46 .9?2.ø1øø SyntaxError: Unexpected token 
62øø. 
' 2øø8-31-ø1' . 
L "phil the groundhog"). name 
"Jim Wehrle" 
.242 ReferenceError: insertCity is not def ined 
function insertCity(name.popuIation. last census-famous_for. mayor_info ) db. 
owns . :name.population : population. last census: ISODate(Iast census). 
info ) ) ; ) 
L "phil the groundhog"). name 
insertCity(' 
'Punxsutawney". 
62øø. 
' 2øø8-31-ø1' . 
"Jim Wehrle" 
insertCity( 
L "beer". 
"Port land". 
"food"). 
582øøø. 
' 2øø?-2ø-øv . 
"Sam Adams". 
name 

insertCity(*"Portland"*, 582000, *'2007-20-09'*,

[*"beer"*, *"food"*], { name : *"Sam Adams"*, party : *"D"* }

)

We should now have three towns in our collection, which you can confirm by calling db.towns.find() as before.

Machine generated alternative text:
Administrator: CAWindcwsN.system32Ncmd.exe - mongo book 
.242 ReferenceError: insertCity is not def ined 
function insertCity(name.popuIation. last census-famous_for. mayor_info ) db. 
owns . :name.population : population. last census: ISODate(Iast census). 
info ) ) ; ) 
62øø. 
' 2øø8-31-ø1' . 
582øøø. 
' 2øø?-2ø-øv . 
"Sam Adams". 
name 
L "phil the groundhog"). name 
"Jim Wehrle" 
L "beer". 
"food"). 
'name " 
"New York". 
'population 
db. towns. find() 
. Objectld(" 
. 222øøøøø. 
"last 
"statue of liberty". 
. Objectld(" 
. 62øø. 
"last 
"phil the groundhog" 
. Objectld(" 
54182d32ødbavø519f9a4b4a"). 
census " 
"famous_for" 
'mayor" 
"name" . 
"Michael Bloomberg". 
54183281ødbavø519f9a4b4b"). 
'name " 
"Punxsutawney". 
'popula 
census " 
"famous_for" 
'mayor" 
"name" . 
"Jim Wehrle" 
'name " 
"Port land". 
'population 
"famous_for" 
. 582øøø. 
"last census" 
'mayor" 
"name" . 
"Sam Adams". 

Reading

Earlier we called the find() function without params to get all documents.

To access a specific one, you only need to set an \_id property. \_id is of type ObjectId, and so to query, you must convert a string by wrapping it in an ObjectId(str) function.

db.towns.find({ "\_id" : ObjectId(*"5d810465dc0d0ce556706291"*) })

should find:

{

"\_id" : ObjectId(*"4d0ada1fbb30773266f39fe4"*),

"name" : *"Punxsutawney"*,

"population" : 6200,

"last\_census" : *"Thu Jan 31 2008 00:00:00 GMT-0800 (PST)"*, "famous\_for" : [ *"phil the groundhog"* ],

"mayor" : { "name" : *"Jim Wehrle"* } }

The find() function also accepts an optional second parameter: a fields object we can use to filter which fields are retrieved.

If we want only the town name (along with \_id), pass in name with a value resolving to 1 (or true).

db.towns.find({ \_id : ObjectId(*"4d0ada1fbb30773266f39fe4"*) }, { name : 1 })

{

"\_id" : ObjectId(*"4d0ada1fbb30773266f39fe4"*),

"name" : *"Punxsutawney"* }

Machine generated alternative text:
Administrator: CAWindcwsN.system32Ncmd.exe - mongo book 
582øøø. 
' 2øø?-2ø-øv . 
L "beer". 
' 'food" I. name : 
"Sam Adams". 
id : 
id : 
id : 
"New York". 
'population 
db. towns. find() 
. Objectld(" 
. 222øøøøø. 
"last 
"statue of liberty". 
. Objectld(" 
. 62øø. 
"last 
"phil the groundhog" 
. Objectld(" 
54182d32ødbavø519f9a4b4a"). 
'name " 
census " 
"famous_for" 
'mayor" 
"name" 
"Michael Bloomberg". 
54183281ødbavø519f9a4b4b"). 
'name " 
"Punxsutawney". 
'popula 
census " 
"famous_for" 
'mayor" 
"name" . 
"Jim Wehrle" 
"Port land". 
'population 
"famous_for" 
'name " 
. 582øøø. 
"last census" 
'mayor" 
db. towns. 
"name" 
"Sam Adams". 
'name " 
"New York". 
'population 
. 222øøøøø. 
"last census" 
"famous_for" 
"statue of liberty". 
db. towns. 
db. towns. 
db. towns. 
'mayor" 
"name" 
"Michael Bloomberg". 
name : 1 » 
name : 1 » 
name : 1 » 
'name " 
"New York" 

To retrieve all fields *except* name, set name to 0 (or false or null).

db.towns.find({ \_id : ObjectId(*"4d0ada1fbb30773266f39fe4"*) }, { name : 0 })

{

"\_id" : ObjectId(*"4d0ada1fbb30773266f39fe4"*),

"population" : 6200,

"last\_census" : *"Thu Jan 31 2008 00:00:00 GMT-0800 (PST)"*, "famous\_for" : [ *"phil the groundhog"* ]

}

Machine generated alternative text:
Administrator: CAWindcwsN.system32Ncmd.exe - mongo book 
db. towns. find() 
'name " 
id : 
id : 
id : 
"New York". 
'population 
"famous_for" 
. 222øøøøø. 
"last census" 
"statue of liberty". 
'mayor" 
"name" 
"Michael Bloomberg". 
'name " 
"Punxsutawney". 
'popula 
. 62øø. 
"last census" 
"famous_for" 
"phil the groundhog" 
'mayor" 
"name" . 
"Jim Wehrle" 
'name " 
"Port land". 
'population 
"famous_for" 
. 582øøø. 
"last census" 
'mayor" 
db. towns. 
"name" . 
"Sam Adams". 
'name " 
"New York". 
'population 
. 222øøøøø. 
"last census" . 
"famous_for" 
"statue of liberty". 
db. towns. 
db. towns. 
db. towns. 
'mayor" 
"name" . 
"Michael Bloomberg". 
name : 1 » 
name : 1 » 
name : 1 » 
db. 
nsus " 
db. 
'name " 
"New York" 
towns. find (4 id 
) name : 
'populat ion " 
. 222øøøøø. 
"last c 
"famous_for" 
"statue of liberty". 
'mayor" 
"name" 
"Michael Bloomberg". 
towns. find (4 id : 
0b 
) name : 

You CAN construct ad hoc queries by field values, ranges, or a combination of criteria.

To find all towns that begin with the letter *P* and have a population less than 10,000, you can use a Perl-compatible regular expression (PCRE) and a range operator.

db.towns.find(  
 { name : /^P/, population : { $lt : 10000 } },  
 { name : 1, population : 1 }

)

{ "name" : *"Punxsutawney"*, "population" : 6200 }

Machine generated alternative text:
Administrator: - mongo book 
db. towns. find (4 id : 
db. towns. find (4 id : 
db. towns. find (4 id : 
name : 1 » 
name : 1 » 
name : 1 » 
'name " 
"New York" 
db. towns. find (4 id : 
name : ø » 
'populat ion " 
. 222øøøøø. 
"last c 
"famous_for" 
"statue 
'mayor" 
"name" . 
"Michael Bloomberg". 
of liberty". 
name 
nsus " 
db. towns. find (4 id : 
db. towns. find( 
• / AP/ . population : 
name 
I. population . 
name 
'name " 
'name " 
"New York" 
"Punxsutawney". 
. løøøø 
t ion" 
. 62øø 
'popula 

Conditional operators in Mongo follow the format of field : { $op : value }, where $op is an operation like $ne (not equal to).

The good news about the query language being JavaScript is you can construct operations as you would objects.

Here, we build criteria where the population must be between 10,000 and 1 million people.

**var** population\_range = {}

population\_range[*'$lt'*] = 1000000

population\_range[*'$gt'*] = 10000

db.towns.find(

{ name : /^P/, population : population\_range },

{ name: 1 } )

{ "\_id" : ObjectId(*"4d0ada87bb30773266f39fe5"*), "name" : *"Portland"* }

Machine generated alternative text:
Administrator: - 
mongo book 
var population_range 
population_rangel 
øøøøøø 
'$gt'l 
population_rangel 
øøøø 
db. towns. find( 
name 
. Objectld(" 
- løøøøøø 
- løøøø 
populat ion 
population_range 
54183295ødbavø519f9a4b4c"). 
'name " 
"Portland" 

We are not limited to number ranges but can also retrieve date ranges.

We can find all names with a *last\_census* less than or equal to January 31, 2008, like this:

db.towns.find(

{ last\_census : { $lte : ISODate(*'2008-31-01'*) } },

{ \_id : 0, name: 1 }

)

{ "name" : *"Punxsutawney"* }

{ "name" : *"Portland"* }

Machine generated alternative text:
Administrator: - 
mongo book 
db. towns. 
. last _ 
'name " 
'name " 
'name " 
f ind( 
census 
. $ Ite : 
2øøe- 
31- 
ISODate(' 
"New York" 
"Punxsutawney" 
"Portland" 

Notice how we suppressed the \_id field in the output explicitly by setting it to 0

Digging Deep

Mongo loves nested array data.

You can query by matching exact values...

db.towns.find(

{ famous\_for : *'food'* },

{ \_id : 0, name : 1, famous\_for : 1 }

)

{ "name" : *"New York"*,

"famous\_for" : [ *"statue of liberty"*, *"food"* ] }

{ "name" : *"Portland"*,

"famous\_for" : [ *"beer"*, *"food"* ] }

Machine generated alternative text:
Administrator: - 
mongo book 
db. towns. 
. last _ 
'name " 
'name " 
'name " 
db. towns. 
f ind( 
census 
. $ Ite : 
2øøe- 
31- 
ISODate(' 
"New York" 
"Punxsutawney" 
"Portland" 
f ind( 
"food" 
. famous_for . 
'name " 
'name " 
name 
"New York". 
"Port land". 
famous_for . 
"famous_for" 
"famous_for" 
"statue 
liberty". 
"food" 

...as well as matching partial values...

db.towns.find(  
 { famous\_for : /statue/ },  
 { \_id : 0, name : 1, famous\_for : 1 }

)

{ "name" : *"New York"*,

"famous\_for" : [ *"statue of liberty"*, *"food"* ] }

Machine generated alternative text:
Administrator: - 
mongo book 
db. towns. 
. last _ 
'name " 
'name " 
'name " 
db. towns. 
f ind( 
census 
. $ Ite : 
2øøe- 
31- 
ISODate(' 
"New York" 
"Punxsutawney" 
"Portland" 
f ind( 
. famous_for . 
'name " 
'name " 
name 
"New York". 
"Port land". 
famous_for . 
"famous_for" 
"famous_for" 
"food" 
"food" 
"statue 
"statue 
liberty". 
"food" 
liberty". 
db. towns. find( 
famous_for . 
/ statue/ 
I. famous_for . 
"famous_for" 
'name " 
name 
"New York". 

...or query by all matching values…

db.towns.find(

{ famous\_for : { $all : [*'food'*, *'beer'*] } },

{ \_id : 0, name:1, famous\_for:1 }

)

{ "name" : *"Portland"*,

"famous\_for" : [ *"beer"*, *"food"* ] }

Machine generated alternative text:
Administrator: - mongo book 
db. towns. find( 
last census 
. $ Ite : 
ISODate(' 2øø8-31- 
'name " 
'name " 
'name " 
"New York" 
"Punxsutawney" 
"Portland" 
db. towns. find( 
. famous_for . 
'name " 
'name " 
name 
"New York". 
"Port land". 
famous_for . 
"famous_for" 
"famous_for" 
"food" 
"food" 
"statue 
"statue 
liberty". 
"food" 
liberty". 
"food" 
db. towns. find( 
famous_for . 
/ statue/ 
I. famous_for . 
name 
'name " 
"New York". 
"famous_for" 
$all : 
L 'food'. 
"famous_for" 
db. towns. find( 
. famous_for . 
'name " 
name: 
"Port land". 

...or the lack of matching values:

db.towns.find(

{ famous\_for : { $nin : [*'food'*, *'beer'*] } },

{ \_id : 0, name : 1, famous\_for : 1 }

)

{ "name" : *"Punxsutawney"*, "famous\_for" : [ *"phil the groundhog"* ] }

Machine generated alternative text:
Administrator: - 
mongo book 
of liberty". 
"food" 
liberty". 
"food" 
"food" 
"food" 
'name " 
'name " 
'name " 
"New York" 
"Punxsutawney" 
"Portland" 
"statue 
"statue 
db. towns. find( 
. famous_for . 
'name " 
'name " 
name 
"New York". 
"Port land". 
famous_for . 
"famous_for" 
"famous_for" 
db. towns. find( 
famous_for . 
/ statue/ 
I. famous_for . 
name 
'name " 
"New York". 
"famous_for" 
$all : 
L 'food'. 
"famous_for" 
L 'food'. 
db. towns. find( 
. famous_for . 
'name " 
name: 
"Port land". 
db. towns. find( 
. famous_for . 
name 
'name " 
the 
I. famous_for . 
"Punxsutawney". 
"famous_for" 
"phil 
groundhog" 

But the true power of Mongo stems from its ability to dig down into a document and return the results of deeply nested subdocuments.

To query a subdocument, your field name is a string separating nested layers with a dot.

For instance, you can find towns with independent mayors. Look back at the JSON document you created – note that mayor is a nested JSON object with *name* and *party* fields.

db.towns.find(

{ 'mayor.party' : *'I'* },

{ \_id : 0, name : 1, 'mayor.name' : 1 }

)

{

name" : *"New York"*,

"mayor" : {

"name" : *"Michael Bloomberg"*,

"party" : *"I"* }

}

Machine generated alternative text:
Administrator: - 
db. towns. find( 
' mayor. party' 
mongo book 
'name " 
mayor 
'mayor" 
"Michael Bloomberg". 
'name " 
name 
"New York". 

or those with mayors who don’t have a party:

db.towns.find(  
 { 'mayor.party' : { $exists : false } },  
 { \_id : 0, name : 1, mayor : 1 }

)

{ "name" : *"Punxsutawney"*,

"mayor" : {

"name" : *"Jim Wehrle"* } }

Machine generated alternative text:
Administrator: - 
db. towns. find( 
' mayor. party' 
mongo book 
'name " 
false 
mayor 
'mayor" 
"Michael Bloomberg". 
"name" 
name 
"New York". 
$exists . 
mayor 
'mayor" 
"Jim Wehrle" 
db. towns. find( 
' mayor. party' 
name 
'name " 
'name " 
"Punxsutawney". 

The previous queries are great if you want to find documents with a single matching field, but what if we need to match several fields of a subdocument?

elemMatch

We’ll round out our dig with the $elemMatch directive.

Let’s create another collection that stores countries

This time we’ll override each \_id to be a string of our choosing.

db.countries.insert({

\_id : *"us"*,

name : *"United States"*,

exports : {

foods : [{ name : *"bacon"*, tasty : true },

{ name : *"burgers"* }

] }

})

Machine generated alternative text:
Administrator: - 
db. towns. find( 
' mayor. party' 
mongo book 
'name " 
false 
Bloomberg". 
Wehrle" 
mayor 
'mayor" 
"Michael 
"Jim 
'name " 
name 
"New York". 
db. towns. find( 
' mayor. party' 
name 
'name " 
"Punxsutawney". 
db. countries. 
$exists . 
mayor 
'mayor" 
'name " 
. true 
name 
"United States". 
. exports 
foods . 
riteResuIt (4 
name . 
"bacon 
"burgers" 
name 
"n Inserted" 

db.countries.insert({

\_id : *"ca"*,

name : *"Canada"*,

exports : {

foods : [

{ name : *"bacon"*, tasty : false },

{ name : *"syrup"*, tasty : true } ] }})

Machine generated alternative text:
Administrator: - 
mongo book 
false 
'name " 
. true 
"Jim 
Wehrle" 
db. towns. find( 
' mayor. party' 
name 
'name " 
"Punxsutawney". 
db. countries. 
$exists . 
mayor 
'mayor" 
name 
"United States". 
. exports 
foods . 
name . 
"bacon 
name 
riteResuIt (4 
"n Inserted" 
db. countries. 
insert (4 
"burgers" 
false 
true 
name 
"Canada". 
"bacon 
's yrup" 
"n Inserted" 
. exports 
foods . 
name 
name 
Write Result (4 

db.countries.insert({

\_id : *"mx"*,

name : *"Mexico"*,

exports : {

foods : [{name : *"salsa"*, tasty : true, condiment : true

}] } })

Machine generated alternative text:
Administrator: - 
db. countries. 
mongo book 
. true. 
condiment 
"Mexico 
name 
. exports 
foods . 
'(name . 
"salsa". 
"n Inserted" 

To validate the countries were added, we can execute the count function, expecting the number 3.

print( db.countries.count() )  
3

Machine generated alternative text:
Administrator: - 
db. countries. 
mongo book 
. true. 
condiment 
"Mexico 
name 
. exports 
foods . 
'(name . 
"salsa". 
"n Inserted" 
print( db. countries . count() 

Let’s find a country that not only exports bacon but exports *tasty* bacon.

db.countries.find(

{ 'exports.foods.name' : *'bacon'*, 'exports.foods.tasty' : true }, { \_id : 0, name : 1 }

)

{ "name" : *"United States"* }

{ "name" : *"Canada"* }

Machine generated alternative text:
Administrator: - 
mongo book 
db. countries. 
id : 
. true 
. true. 
condiment 
"Mexico 
name 
. exports 
foods . 
'(name . 
"salsa". 
foods. 
tasty' 
"n Inserted" 
print( db. countries . count() 
db. countries. 
' exports. 
name 
f ind( 
foods . name' 
' bacon' 
exports . 
States" 
'name " 
'name " 
"United 
"Canada" 

But this isn’t what we wanted.

Mongo returned *Canada* because it exports bacon and exports tasty syrup.

$elemMatch helps us here.

It specifies that if a document (or nested document) matches *all* of our criteria, the document counts as a match.

db.countries.find(  
 {

'exports.foods' : {  
 $elemMatch : {

name : *'bacon'*,

tasty : true }

} },

{ \_id : 0, name : 1 }  
)

{ "name" : *"United States"* }

Machine generated alternative text:
Administrator: - 
"n Inserted" 
print( db. countries. count ( ) ) 
id : 
mongo book 
' bacon' 
exports . 
foods. 
tasty' 
. true 
db. countries. 
' exports. 
f ind( 
foods . name' 
name 
'name " 
'name " 
"United States" 
"Canada" 
db. countries. find( 
exports . foods' 
$eIemMatch : 
name 
'name " 
' bacon' 
true 
name 
"United States" 

$elemMatch criteria can utilize advanced operators, too.

You can find any country that exports a tasty food that also has a condiment label:

db.countries.find(  
 {

'exports.foods' : {  
 $elemMatch : {

tasty : true,

condiment : { $exists : true }  
 }

} },

{ \_id : 0, name : 1, 'exports.foods.name':1 }  
)

{ "name" : *"Mexico"* }

Machine generated alternative text:
Administrator: - 
db. countries. find( 
exports . foods' 
$eIemMatch : 
mongo book 
true 
$exists . 
name 
'name " 
' bacon' 
true 
name 
"United States" 
db. countries. find( 
exports . foods' 
'name " 
$eIemMatch : 
condiment 
name 
"Mexico" 

Mexico is just what we wanted.

Boolean Ops

So far, all of our criteria are implicitly *and* operations.

If you try to find a country with the name *United States* and an \_id of *mx*, Mongo will yield no results.

db.countries.find(

{ \_id : *"mx"*, name : *"United States"* }, { \_id : 1 }

)

However, searching for one *or* the other with $or will return two results.

Think of this layout like *prefix notation*: OR A B.

db.countries.find(  
 {

$or : [

{ \_id : *"mx"* },

{ name : *"United States"* }

] },

{ \_id:1 } )

{ "\_id" : *"us"* } { "\_id" : *"mx"* }

Machine generated alternative text:
Administrator: - 
mongo book 
exports . foods' 
id : 
$exists . 
true 
'name " 
$eIemMatch : 
condiment 
name 
"Mexico" 
f ind( 
name 
f ind( 
"United 
db. countries. 
db. countries. 
States" 
name 
"United 
States" 

There are so many operators we can’t cover them all here, but we hope this has given you a taste of MongoDB’s powerful query ability (See book p 145 for more)

You can find all the commands on the MongoDB online documentation or grab a cheat sheet : <http://myqc.wordpress.com/2012/08/13/mongodb-cheatsheet/>

Updating

We have a problem.

New York and Punxsutawney are unique enough, but did we add Portland, Oregon, or Portland, Maine?

Let’s update our towns collection to add some U.S. states.

The update(criteria,operation) function requires two parameters.

The first is a criteria query—the same sort of object you would pass to find().

The second parameter is either an object whose fields will replace the matched document(s) or a modifier operation.

In this case, the modifier is to $set the field state with the string *OR*.

Note use the correct ObjectId

db.towns.update({ \_id : ObjectId(*"5ba3bcc69d59bfb1b88716dd"*) }, { $set : { "state" : *"OR"* } } );

Machine generated alternative text:
Administrator: - mongo book 
"nModif ied" 
"nMatched" 
db. towns . update( 
"nUpserted" 
. $set : 
"state" . 
. $set : 
. db. towns. update ( id 
"state" 
db. towns . update( 
db. towns. update (4 id : 
"nMatched" 
$set : 
"nUpserted" 
"nModif ied" 

We can verify our update was successful by finding it (note our use of findOne() to retrieve only one matching object).

db.towns.findOne({ \_id : ObjectId(*"5ba3bcc69d59bfb1b88716dd"*) })

{

"\_id" : ObjectId(*"5ba3bcc69d59bfb1b88716dd"*), "famous\_for" : [

*"beer"*,

*"food"*

],

"last\_census" : *"Thu Sep 20 2007 00:00:00 GMT-0700 (PDT)"*,

"mayor" : {

"name" : *"Sam Adams"*,

"party" : *"D"* },

"name" : *"Portland"*,

"population" : 582000,

"state" : *"OR"*

}

Machine generated alternative text:
Administrator: - mongo book 
vø?.øløø SyntaxError: Unexpected token 
db. towns. findOne« id : 
ILLEGAL 
» 
'name " 
"Port land". 
. 582øøø. 
"Sam Adams". 
'populat ion " 
"last census" 
"famous_for" 
"beer". 
'mayor" 
'name " 
"state" 

You can do more than $set a value. $inc (increment a number) is a pretty useful one.

Let’s increment Portland’s population by 1,000.

db.towns.update({ \_id : ObjectId(*"5ba3bcc69d59bfb1b88716dd"*) }, { $inc : { population : 1000} } )

Machine generated alternative text:
Administrator: - mongo book 
.?95.ø1øø SyntaxError: Unexpected token ) 
populat ion 
$inc : 
. løøø 
db. towns. 
update ( 
) $inc : 
"nModif ied" 
"nModif ied" 
"nMatched" 
"nUpserted" 
"nUpserted" 
Write Result (4 
db. towns. update (4 id : 
u lat ion 
. løøø» ) 
"nMatched" 

There are more directives than this, such as the $ positional operator for arrays.

New operations are added frequently and are updated in the online documentation.

Here are the major directives:

* + $unset Removes the field
  + $pop Removes the last (or first) element from an array
  + $push Adds the value to an array
  + $pushAll Adds all values to an array
  + $addToSet Similar to push, but won’t duplicate values
  + $pull Removes matching value from an array
  + $pullAll Removes all matching values from an array

Deleting

Removing documents from a collection is simple.

Merely replace the find function with a call to remove(), and all matched criteria will be removed.

It’s important to note that the entire matching document will be removed, not simply a matching element or a matching subdocument.

Let’s remove all countries that export bacon that isn’t tasty.

**var bad\_bacon = { "exports.foods" : {**

**$elemMatch : { name : "bacon", tasty : false**

**} }**

**}**

db.countries.find( bad\_bacon )

{

"\_id" : ObjectId(*"4d0b7b84bb30773266f39fef"*), "name" : *"Canada"*,

"exports" : {

"foods" : [ {

"name" : *"bacon"*,

"tasty" : false  
 },

{

"name" : *"syrup"*, "tasty" : true

} ]

} }

Machine generated alternative text:
Administrator: - mongo book 
dbl port land. country.$ref . findOne« id: port land. country.$id 
'name " 
'exports " 
"United States". 
"bacon 
"burgers " 
false 
var bad_bacon 
$eIemMatch : 
"foods" 
'name " 
'name " 
— 'exports. foods' 
name 
' bacon' 
db. countries . find( bad_bacon ) 
'name " 
'name " 
"bacon 
"Canada". 
'exports " 
's yrup" 
false "name" . 

Everything looks good. Let’s remove it.

db.countries.remove( bad\_bacon )  
db.countries.count()

2

Now when you run count(), verify we are left with only two countries.

If so, our delete was successful!

Reading with Code

You can request that MongoDB run a decision function across your documents.

It should always be a last resort.

These queries run quite slowly, you can’t index them, and Mongo can’t optimize them.

Say we’re looking for a population between 6,000 and 600,000 people.

db.towns.find( **function**() {

**return** this.population > 6000 && this.population < 600000;

})

Mongo even has a shortcut for simple decision functions.

db.towns.find(*"this.population > 6000 && this.population < 600000"*)

Machine generated alternative text:
Administrator: - 
mongo book 
false 
'exports " 
u ar bad_bacon 
$eIemMatch : 
— 'exports. foods' 
name 
' bacon' 
'name " 
"bacon 
'popula 
db. countries . find( bad_bacon ) 
'name " 
"Canada". 
false "name" 
db. towns. find( function() 
's yrup" 
. return this . population 6øøø && this . population 6øøøøø; 
'name " 
"Punxsutawney". 
. 62øø. 
"last census" . 
"famous_for" 
"Jim Wehrle" 
'name " 
"phil the groundhog" 
'mayor" 
"name" 
"Port land". 
'population 
. 583øøø. 
"last census" . 
"famous_for" 
'country" 
'mayor" 
"name" . 
"Sam Adams". 
"state" 

You can run custom code with other criteria using the $where clause.

In this example, the query also filters for towns famous for groundhogs.

db.towns.find( {

$where : *"this.population > 6000 && this.population < 600000"*, famous\_for : /groundhog/

})

Machine generated alternative text:
Administrator: - 
db. towns. find( function() 
mongo book 
. return this . population 6øøø && this . population 6øøøøø; 
'name " 
"Punxsutawney". 
'popula 
t ion" 
. 62øø. 
"last census" 
"famous_for" 
"phil the groundhog" 
'mayor" 
"name" . 
"Jim Wehrle" 
"Port land". 
'population 
"famous_for" 
"state" 
'name " 
. 583øøø. 
"last census" 
'country" 
db. towns. find( 
$ where : 
"this . population 6øøø && this. 
'mayor" 
"name" . 
"Sam Adams". 
6øøøøø". 
famous_for . 
populat ion 
'name " 
groundhog/ 
. Objectld(" 
. 62øø. 
"last 
"phil the groundhog" 
54183281 ødbavø519f9a4b4b"). 
"Punxsutawney". 
'popula 
øøz"). 
"famous_for" 
census " 
'mayor" 
"name" 
"Jim Wehrle" 

A word of warning: Mongo will brutishly run this function against each document, and there is no guarantee that the given field exists.

For example, if you assume a *population* field exists and *population* is missing in even a single document, the entire query will fail, since the JavaScript cannot properly execute.

Be careful when you write custom JavaScript functions, and be comfortable using JavaScript before attempting custom code.

ReCap

We saw how we can store nested structured data as JSON objects and query that data at any depth.

You learned that a *document* can be envisioned as a schema-less row in the relational model, keyed by a generated \_id.

Mongo stores complex, denormalized documents, stored and retrieved as collections of arbitrary JSON structures.

Mongo tops off this flexible storage strategy with a powerful query mechanism not constrained by any predefined schema.

Its denormalized nature makes a document datastore a superb choice for storing data with unknown qualities, while other styles (such as relational or columnar) prefer you know in advance and require schema migrations to add or edit fields.

Indexing:

One of Mongo’s useful built-in features is indexing to increase query performance—something that’s not available on all NoSQL databases.

MongoDB provides several of the best data structures for indexing, such as the classic B-tree (<http://en.wikipedia.org/wiki/B-tree>), and other additions such as two-dimensional and spherical GeoSpatial indexes.

For now we’re going to do a little experiment to see the power of MongoDB’s B-tree index by populating a series of phone numbers with a random country prefix

Enter the following code into your console.

This will generate 100,000 phone numbers (it may take a while), between *1-800-555-0000* and *1-800-565-9999*.

mongo/populate\_phones.js

populatePhones = **function**(area,start,stop) {

**for**(**var** i=start; i < stop; i++) {

**var** country = 1 + ((Math.random() \* 8) << 0);

**var** num = (country \* 1e10) + (area \* 1e7) + i;

db.phones.insert({

\_id: num,  
 components: {

country: country,  
 area: area,  
 prefix: (i \* 1e-4) << 0,  
 number: i

},

display: *"+"* + country + *" "* + area + *"-"* + i });

} }

Run the function with a three-digit area code (like 800) and a range of seven-digit numbers (5,550,000 to 5,650,000)

populatePhones( 800, 5550000, 5650000 )

This will take approx. 1 minutes to complete

db.phones.find().limit(2).pretty()

{

"\_id" : 18005550000,

"components" : { "country" : 1, "area" : 800, "prefix" : 555, "number" : 5550000 },

"display" : *"+1 800-5550000"* }

{ "\_id" : 88005550001, "components" : { "country" : 8, "area" : 800, "prefix" : 555, "number" : 5550001 }, "display" : *"+8 800-5550001"* }

Machine generated alternative text:
Administrator: - mongo 
, db. phones. 
insert (4 
populatePhones 
— function (area-start 
for(uar i:start; i stop; i 
I * ( (Math. random() 
var country 
-stop) 
(country 
leiØ) 
id: num. 
components: 
country: country. 
area: area. 
pref ix: (i 
le-4) 
number: i 
* (area 
« 
« 
D; 
. display: 
* country 
unction (area-start-stop) 
or(uar i:start; i stop; i 
I * ( (Math. random() 
ar country 
(country 
leiØ) 
id: num. 
components: 
country: country. 
area: area. 
pref ix: (i 
le-4) 
number: i 
* (area 
« 
, db. phones. 
insert (4 
. eøø. " 
. eøø. " 
isplay: 
* country 
populatePhones( eøø. sssøøøø. 565øøøø ) 
db. phones. 
. 88øøsssøøøø. 
'components" 
'number" 
. sssøøøø 
"display" 
. 48øøsssøøø1. 
'components" 
'number" . 
sssøøøl 
"display" 
'country" 
eøø-sssøøøi" 
'country" 
eøø-sssøøøi" 
pref ix" 
pref ix" 

Whenever a new collection is created, Mongo automatically creates an index by the \_id.

These indexes can be found in the system.indexes collection.

The following query shows all indexes in the database:

db.phones.getIndexes()

{ "name" : *"\_id\_"*, "ns" : *"book.phones"*, "key" : { "\_id" : 1 } }

Machine generated alternative text:
Administrator: - mongo 
, db. phones. 
insert (4 
populatePhones 
— function (area-start 
for(uar i:start; i stop; i 
I * ( (Math. random() 
var country 
-stop) 
(country 
leiØ) 
id: num. 
components: 
country: country. 
area: area. 
pref ix: (i 
le-4) 
number: i 
* (area 
« 
« 
D; 
. display: 
* country 
unction (area-start-stop) 
or(uar i:start; i stop; i 
I * ( (Math. random() 
var country 
(country 
leiØ) 
id: num. 
components: 
country: country. 
area: area. 
pref ix: (i 
le-4) 
number: i 
* (area 
« 
pref ix" 
pref ix" 
, db. phones. 
insert (4 
. eøø. " 
. eøø. " 
isplay: 
* country 
populatePhones( eøø. sssøøøø. 565øøøø ) 
db. phones. 
. 88øøsssøøøø. 
'components" 
'number" 
. sssøøøø 
"display" 
. 48øøsssøøø1. 
'components" 
'number" 
. sssøøøl 
"display" 
db. system. indexes. find() 
'name " 
'name " 
'name " 
"book. 
"book. 
"book. 
'country" 
eøø-sssøøøi" 
'country" 
eøø-sssøøøi" 
towns" 
countries " 
phones" 

Most queries will include more fields than just the \_id, so we need to make indexes on those fields.

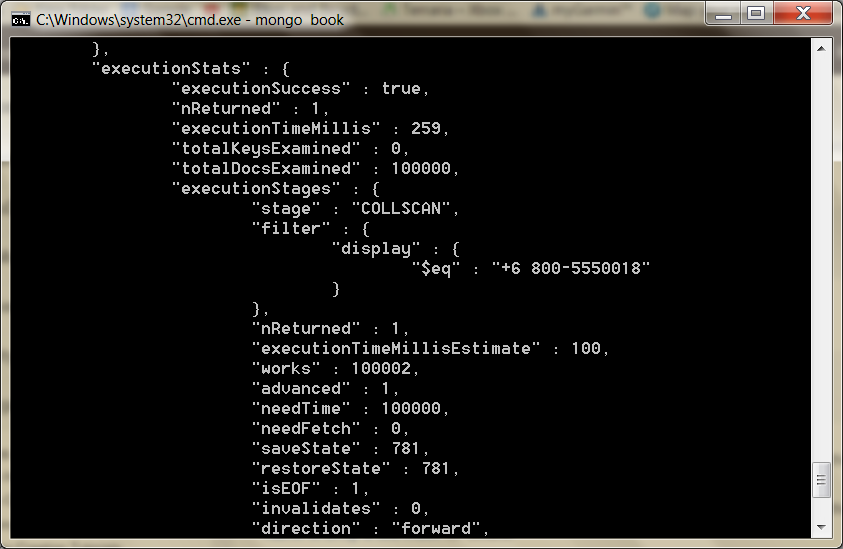
We’re going to create a B-tree index on the display field.

But first, let’s verify that the index will improve speed.

To do this, we’ll first check a query without an index.

The explain(“executionStats”) method is used to output details of a given operation.

db.phones.find({display: *"* *+7 800-5550000"*}).explain(*"*executionStats*"*)



Your output will not equal ours, but note the millis field—milliseconds to complete the query—will likely be double digits.

We create an index by calling ensureIndex(fields,options) on the collection.

The fields parameter is an object containing the fields to be indexed against.

The options parameter describes the type of index to make.

In this case, we’re building a unique index on display that should just drop duplicate entries.

db.phones.ensureIndex(  
 { display : 1 },  
 { unique : true, dropDups : true }

)



Now try find() again, and check explain() to see whether the situation improves.

db.phones.find({display: *"+6 800-5550018"*}).explain(*"*executionStats*"*)

{

"cursor" : *"BtreeCursor display\_1"*, "nscanned" : 1,

"nscannedObjects" : 1,

"n" : 1,

"millis" : 0,

"indexBounds" : {

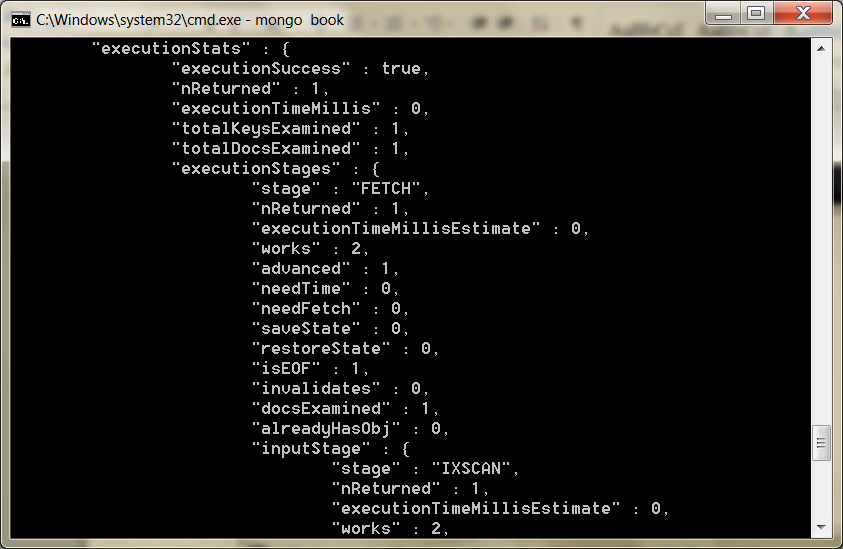
"display" : [  
 [

*"+1 800-5650001"*,

*"+1 800-5650001"*

] ]

} }



The millis value changed from 100 to 0 an orders of magnitude speedup.

Mongo is no longer doing a full collection scan but instead walking the tree to retrieve the value.

Importantly, scanned objects dropped from 109999 to 1—since it has become just a unique lookup.

Like nested queries, Mongo can build your index on nested values.

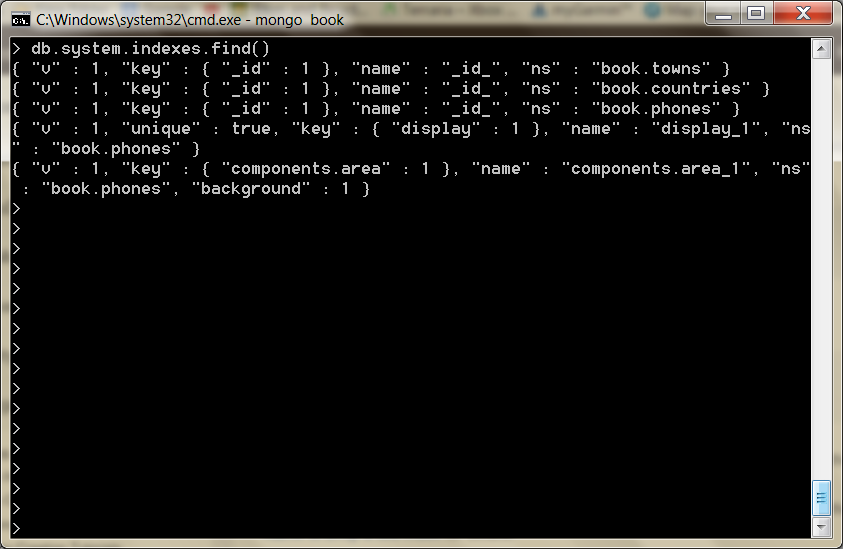
If you wanted to index on all area codes, use the dot-notated field representation: components.area.

In production, you should always build indexes in the background using the { background : 1 } option.

db.phones.ensureIndex({ "components.area": 1 }, { background : 1 })

If we find() all of the system indexes for our phones collection, the new one should appear last.

 db.phones.getIndexes()



The first index is always automatically created to quickly look up by \_id, and the second is the unique index we made previously.

db.phones.getIndexes()

Machine generated alternative text:
Administrator: - 
mongo 
background . 
"book. phones" 
'name " 
"display_l 
'components . area_l 
db. phones . ensure "components . area": 
"createdCoIIectionRutomaticaIIy" 
false. 
'numIndexesBefore" . 
"numIndexesRfter" . 
db. system. indexes. "ns" 
'un Ique " 
"book phones " 
'name " 
"display" 
true 
» 
"book. phones 
"book. phones 
"dropDups " 
"components . area" 
"background" 
'name " 

To delete an index use:

[db.collection.dropIndex(“indexname”)](http://docs.mongodb.org/manual/reference/method/db.collection.dropIndex/#_blank) method.

We should close this section by noting that creating an index on a large collection can be slow and resource-intensive.

You should always consider these impacts when building an index by creating indexes off-peak times, running index creation in the background, and running them manually rather than using automated index creation.

**Aggregated Queries**

Say we wanted to count the phone numbers greater than 559–9999; we would prefer the database perform such a count on the back end.

count() is the most basic aggregator. It takes a query and returns a number (of matches).

db.phones.count({'components.number': { $gt : 5599999 } })

50000

To see the power of the next few aggregating queries, let’s add another 100,000 phone numbers to our phones collection, this time with a different area code.

populatePhones( 855, 5550000, 5650000 )

The distinct() command returns each matching value (not a full document) where one or more exists.

We can get the distinct component numbers that are less than 5,550,005 in this way:

db.phones.distinct(*'components.number'*, {'components.number': { $lt : 5550005 } })

[ 5550000, 5550001, 5550002, 5550003, 5550004 ]

Machine generated alternative text:
Administrator: - 
mongo 
. 5599999 » 
db. phones . components . number' : 
øøøø 
populatePhones( 855. SSSøøøø. 565øøøø ) 
db . phones . distinct components . number' 
4' components . number' : 
sssøøøø. sssøøøl. sssøøø2. sssøøø3. sssøøø4 
. sssøøøs 

Although we have two 5,550,000 numbers (one with an 800 area code and one with 855), it appears in the list only once.

Server-Side Commands

If you were to run the following function through a command line (or through a driver), the client will pull each phone locally, all 100,000 of them, and save each phone document one by one to the server.

mongo/update\_area.js

update\_area = **function**() { db.phones.find().forEach( **function**(phone) { phone.components.area++; phone.display = *"+"*+ phone.components.country+*" "*+ phone.components.area+*"-"*+ phone.components.number; db.phone.update({ \_id : phone.\_id },phone, false);}) }

Machine generated alternative text:
Select Administrator: C:NWindowsN.system32Ncmd.exe - 
. db. phones . find() .forEach( 
id : 
mongo 
components . 
phone . 
components . 
* phone. 
phone . 
function (phone) 
phone . components . area* * ; 
. phone . display 
. phone . components . country*" 
db . phone . update (4 
unction ( ) 
b. phones . f ind() . forEach( 
function (phone) 
phone . components . area* * ; 
phone . display 
hone . components . country*" 
db. phone . update (4 id : 
db. phones. 
id phone. false); 
phone . 
* phone. 
phone . 
. 88øøsssøøøø. " 
components" 
'number" . 
sssøøøø 
"display" 
. 48øøsssøøø1. 
'components" 
'number" 
. sssøøøl 
"display" 
components . 
id phone. false); 
'country" 
eøø-sssøøøi" 
'country" 
eøø-sssøøøi" 
components . 
number 
. eøø. " 
. eøø. " 
pref ix" 
pref ix" 

(Try saving this as a file in mongo/bin directory) and load("update\_area.js")

 (try using pwd() to see what directory your shell is looking at and cd(“directory path”) to change directories

You can run the function like this

update\_area()

However, the Mongo db object provides a command named eval(), which passes the given function to the server.

This dramatically reduces chatter between the client and server since the code is executed remotely.

> db.eval(update\_area)

In addition to evaluating JavaScript functions, there are several other prebuilt commands in Mongo, most of which are executed on the server, although some require executing only under the admin database (which you can access by entering use admin).

> use admin

> db.runCommand(*"top"*)

Machine generated alternative text:
Administrator: CAWindcwsN.system32Ncmd.exe - mongo 
"getmore " 
"t ime " 
'count" 
'insert " 
"t ime " 
'count" 
"update " 
"t ime " 
'count" 
'remove " 
"t ime " 
'count" 
'commands " 
"t ime " 
'count" 

The top command will output access details about all collections on the server.

> use book

> db.listCommands()

Machine generated alternative text:
Administrator: CAWindcwsN.system32Ncmd.exe - mongo 
updateUser: 
for example 
Used to 
users Info: 
Returns 
validate: 
update a user. 
change 
its 
its 
password 
data structures 
for correctne 
information about users . 
Ualidate contents of 
Slow. 
Add full: true option 
a namespace by scanning 
to do a more thorough check 
hatsmyuri : 
(whatsmyuri : I) 
riteBacksQueued: 
internal 
ritebacklisten : 
internal 
admin On Iy s lave Ok 
admin On Iy s lave Ok 

On running listCommands(), you may notice a lot of commands we’ve used.

In fact, you can execute many common commands through the runCommand() method, such as counting the number of phones.

However, you may notice a slightly different output.

db.runCommand({ "count" : *"phones"* })

{ "n" : 200000, "ok" : 1 }

Machine generated alternative text:
Administrator: CAWindcwsN.system32Ncmd.exe - mongo 
updateUser: 
its 
its 
password 
data structures 
Used to 
users Info: 
Returns 
validate: 
update a user. for example to change 
information about users . 
a namespace by scanning 
for correctne 
Ualidate contents of 
Slow. 
Add full: true option 
to do a more thorough check 
whatsmyuri : 
(whatsmyuri : I) 
write Backs Queued: 
internal 
ritebacklisten : 
internal 
admin On Iy s lave Ok 
admin On Iy s lave Ok 
"phones " 
» 
db . "count" 
. 2øøøøø. 'ok 

The number (*n*) returned is correct (200,000), but the format is an object with an *ok* field.

That’s because db.phones.count() is a wrapper function created for our convenience by the shell’s JavaScript interface, whereas runCommand() is a count executed on the server.

Remember that we can play detective on how a function like count() works by leaving off the calling parentheses.

> db.phones.count

**function** (x) {

**return** this.find(x).count(); }

Interesting! collection.count() is just a convenience wrapper for calling count() on the results of find() (which itself is just a wrapper for a native query object that returns a cursor pointing to results).

If you run *that* query...

> db.phones.find().count

you will get a much larger function (too much to print here).

But look in the code, and after a bunch of setup, you’ll find lines like this:

**var** res = this.\_db.runCommand(cmd);

**if** (res && res.n != null) {

**return** res.n; }

Machine generated alternative text:
Select Administrator: CAWindcwsN.system32Ncmd.exe - mongo 
if ( this ._query.$maxT imeMS ) 
— this ._query.$maxT imeMS; 
cmd . maxT imeMS 
if ( this ._query.$hint ) 
— this ._query.$hint; 
cmd . hint 
cmd . query 
— this. 
_query; 
limit; 
skip; 
cmd.f ie Ids 
— this._fields 
if ( applySkipLimit ) 
if ( this . _ limit ) 
cmd. limit 
— this. 
if ( this ._skip ) 
cmd . skip 
— this. 
— this ._db 
. runCommand( cmd 
res && res. n ! — null ) return res. n; 
throw "count failed: 
* tojson( res ) ; 

Double interesting! count() executes runCommand() and returns the value from the n field.

Diversion

We took this diversion for two reasons:

* + To drive home the idea that most of the magic you execute on the mongo console is executed on the server, not the client, which just provides convenient wrapper functions.
  + We can leverage the concept of executing server-side code for our own gain to create something in MongoDB that’s similar to the *stored procedures*

Any JavaScript function can be stored in a special collection named system.js.

This is a normal collection; you just save the function by setting the name as the \_id, and value is the function object.

db.system.js.save({ \_id:*'getLast'*,   
value:**function**(collection){  
**return** collection.find({}).sort({'\_id':1}).limit(1)[0]   
} })

The eval() function passes the string to the server, evaluates it as JavaScript code, and returns the results.

db.eval(*'getLast(db.phones)'*)

It should return the same values as calling getLast(collection) locally.

Machine generated alternative text:
Administrator: CAWindcwsN.system32Ncmd.exe - 
db. system. js . save (4 id:' getLast' 
. value : function 
mongo 
.1». 
"nModif ied" 
"getLast 
. return collection. find(O) . id' 
"nMatched" 
"nUpserted" 
db.eual(' getLast(db.phones)') 
. 18øøsssøøø2. 
'components" 
'country" 
. eøø. 
'pref ix" 
'number" . 
555iøø2 
"display" 
eøø-sssøøø2" 

> db.system.js.findOne({'\_id': *'getLast'*}).value(db.phones)

-- HERE – 20/9

Need to rebuild the 200000 phone records above

Mapreduce (and Finalize)

Mongo requires your mapper to call an emit() function with a key.

The benefit here is that you can emit more than once per document.

The reduce() function accepts a single key and a list of values that were emitted to that key.

Finally, Mongo provides an optional third step called finalize(), which is executed only once per mapped value after the reducers are run.

This allows you to perform any final calculations or cleanup you may need.

Let’s generate a report that counts all phone numbers that contain the same digits for each country.

First we’ll store a helper function that extracts an array of all distinct numbers (under- standing how this helper works is not imperative to understanding the overall mapreduce).

mongo/distinct\_digits.js

distinctDigits = **function**(phone){

**var** number = phone.components.number + *''*, seen = [],

result = [],

i = number.length;

**while**(i--) {

seen[+number[i]] = 1;

}

**for** (i=0; i<10; i++) {

**if** (seen[i]) { result[result.length] = i;

} }

**return** result; }

db.system.js.save({\_id: *'distinctDigits'*, value: distinctDigits})

Machine generated alternative text:
Administrator: CAWindcwsN.system32Ncmd.exe - mongo 
. seen L *number' i I I 
value: 
distinctDigits)) 
for (i:ø; i(lø; 
. if (seen Lil) 
. return result; 
unction (phone X 
var number 
phone. 
e sult 
— number. length; 
hile(i--) 
een L *number Lill 
or (i:ø; i(lø; i 
result Lresult . length) 
components . number 
if (seen Lil) 
eturn result; 
db. system.js. 
riteResuIt (4 
result Lresult . length) 
id: 
'distinctDigits' 
"nMatched" 
"nUpserted" 
"nModif ied" 
"distinctDigits" 

(OR save the function in a file and and load it as we did earlier)

With all that in, we can do a quick test (if you have some trouble, don’t feel

shy about adding a smattering of print() functions).

I

db.eval(*"distinctDigits(db.phones.findOne({ 'components.number' : 5551213 }))"*)

[ 1, 2, 3, 5 ]

Machine generated alternative text:
Administrator: CAWindcwsN.system32Ncmd.exe - mongo 
value: 
distinctDigits)) 
. if (seen Lil) 
. return result; 
function (phone X 
var number 
phone. 
result 
— number. length; 
while(i——) 
seen L *number Lill 
for (i:ø; i(lø; i 
result Lresult . length) 
components . number 
if (seen Lil) 
return result; 
db. system.js. 
Write Result (4 
result Lresult . length) 
id: 
'distinctDigits' 
number' 
. 5551213 
"nMatched" 
"nUpserted" 
"nModif ied" 
"distinctDigits" 
findOne« ' 
components . 

Now we can get to work on the mapper.

As with any mapreduce function, deciding what fields to map by is a crucial decision, since it dictates the aggregated values that you return.

Since our report is finding distinct numbers, the array of distinct values is one field.

But since we also need to query by country, that is another field.

We add both values as a compound key: {digits : X, country : Y}.

Our goal is to simply count these values, so we emit the value 1 (each document represents one item to count).

The reducer’s job is to sum all those 1s together.

mongo/map\_1.js

map = **function**() {

**var** digits = distinctDigits(this);

emit({digits : digits, country : this.components.country}, {count : 1});

}

mongo/reduce\_1.js

reduce = **function**(key, values) {

**var** total = 0;

**for**(**var** i=0; i<values.length; i++) {

total += values[i].count;  
 }

**return** { count : total }; }

results = db.runCommand({ mapReduce: *'phones'*,

map: map,

reduce: reduce,

out: *'phones.report'* })

Machine generated alternative text:
HOME 
INSERT 
HISTO 
EVIEW 
MongoDB - OneNote 
themat 
mongo 
Windows Task Mana 
File Options View Help 
Peter Given 
earch (Ctrl—El 
Add Page 
Llsers 
Ill 
Administrator: C:NWindcwsN.system32vcmd .exe - 
LIZ: peter's Note 
total ) ; 
. return count 
function (key. values) 
ar total — 
or(uar i:ø; 
total 
' phones' 
Now we can get to 
As with any 
aggregated values 
Since our report is 
But since we also n 
We add both values 
Our goal is to simpl 
coun 
The reducer's job is 
map — function') 
digits distirctDisits(this); 
emitl(digits : digits, 
total —O; 
cwlt : total 
i(ualues. length; i 
— values Lil.count; 
eturn count 
total ) ; 
results 
db. mapReduce: 
. map: map. 
. reduce: reduce. 
' phones . report' ) ) 
'result " 
"phones . report 
"timeMiIIis" : 
4788. 
'counts" 
'input" 
. 2øøøøø. 
. 2øøøøø. 
"reduce " 
. 48493. 
"output " 
. 3488 
. thi 
Applications Prcn:esses Services Performance Networking 
CPLI usage 
Memory 
Physical Memory (ME) 
Total 
Cached 
Available 
Free 
Kernel Memory (ME) 
CPU usage History 
Physical Memory usage History 
8134 
2952 
5167 
2461 
S ystem 
Handles 
Threads 
Pr c. sses 
LIP Time 
Commit (GE) 
1182 
3/15 
Paged 
Processes: 89 
* 
Resource Monitor... 
Physical Memory: 36% 
10:33 
17/09/2014 
red u. : reduce, 
Since we set the collection name via the parameter : vou can query the results like 
any other _ 
It's a materialized view that vou can see in the list. 
idxm.ntN' : 
CPU Usage: 

Since we set the collection name via the out parameter (out : 'phones.report'), you can query the results like any other.

Machine generated alternative text:
Administrator: CAWindcwsN.system32Ncmd.exe - 
. total ) ; 
. return count 
unction (key. values) 
ar total — 
or(uar i:ø; 
i(ualues . length; 
total 
— values Lil.count; 
total ) ; 
eturn count 
mongo 
' phones' 
results 
db. mapReduce: 
. map: map. 
. reduce: reduce. 
' phones . report' ) ) 
'result " 
"phones . report 
"timeMiIIis" : 
4788. 
'counts" 
'input" 
. 2øøøøø. 
. 2øøøøø. 
"reduce" . 
48493. 
"output " 
. 3488 

It’s a materialized view that you can see in the show tables list.

db.phones.report.find({'\_id.country' : 8})

{

"\_id" : { "digits" : [ 0, 1, 2, 3, 4, 5, 6 ], "country" : 8 },

"value" : { "count" : 19 }  
}

{  
 "\_id" : { "digits" : [ 0, 1, 2, 3, 5 ], "country" : 8 },  
 "value" : { "count" : 3 }

} {

"value" : { "count" : 48 }  
}

{  
 "\_id" : { "digits" : [ 0, 1, 2, 3, 5, 6, 7 ], "country" : 8 },  
 "value" : { "count" : 12 }

}

has more

Machine generated alternative text:
Administrator: CAWindcwsN.system32Ncmd.exe - 
mongo 
'country" 
'country" 
'country" 
'country" 
'country" 
'country" 
'country" 
'country" 
'country" 
"value" . 
'count" 
db. phones. 
"reduce" . 
48493. 
"output " 
. 3488 
report . f 
id.country' 
"digits" 
"digits" 
"digits" 
"digits" 
"digits" 
"digits" 
"digits" 
"digits" 
"digits" 
"value " 
'count 
'count 
'count 
'count 
"value" . 
"value" . 
"value" . 
"value" . 
"value" . 
"value" . 
"value " 

Type it to continue iterating through the results.

Note the unique emitted keys are under the field \_ids, and all of the data returned from the reducers are under the field value.

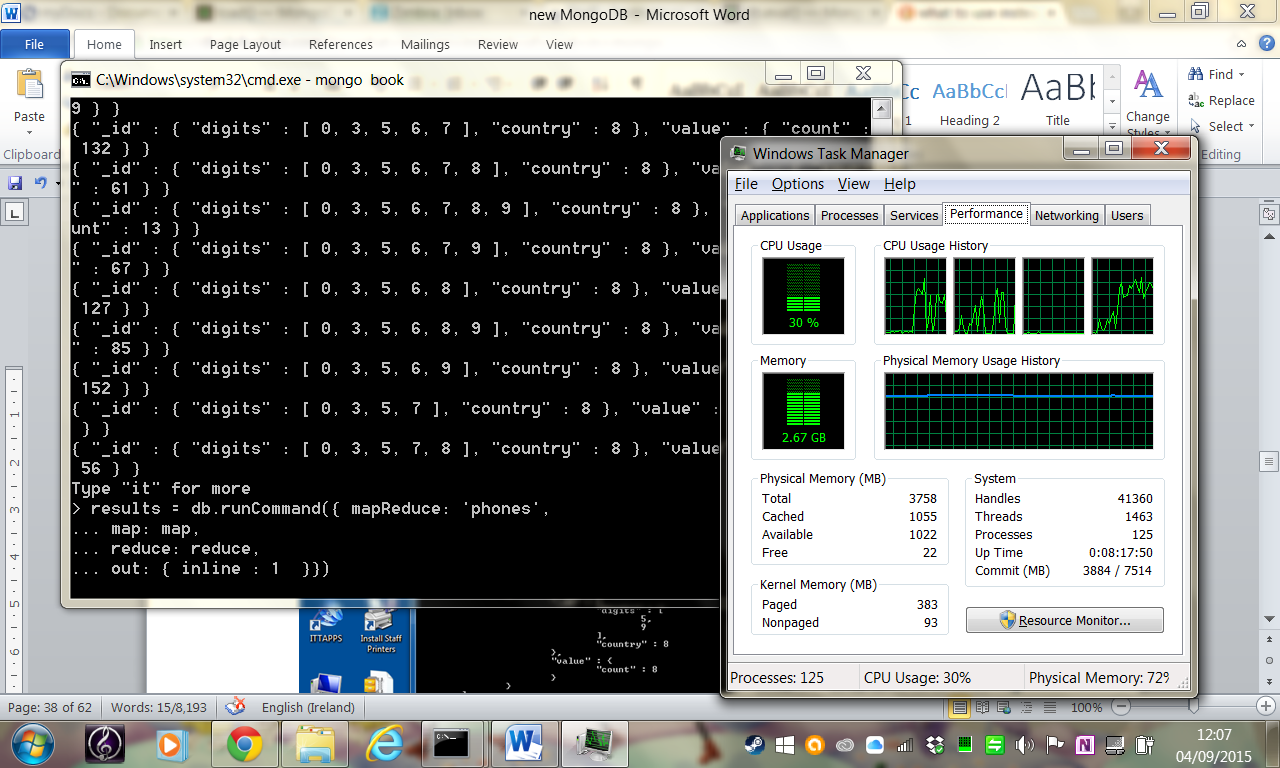
If you prefer that the mapreducer just output the results, rather than outputting to a collection, you can set the out value to { inline : 1 }, but bear in mind there is a limit to the size of a result you can output.

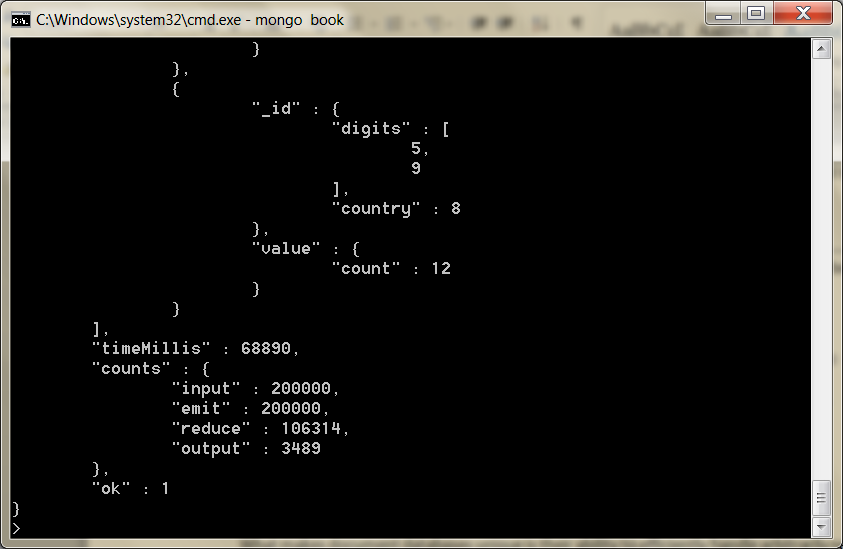
results = db.runCommand({ mapReduce: *'phones'*,

map: map,

reduce: reduce,

out: { inline : 1 }})





Recall from the CouchDB that reducers can have either mapped (emitted) results or other reducer results as inputs.

Why would the output of one reducer feed into the input of another if they are mapped to the same key?

Think of how this would look if run on separate servers, (see book page 164.)

Each server must run its own map() and reduce() functions and then push those results to be merged with the service that initiated the call, gathering them up.

ReCap

we’ve expanded our query power by including several aggregate queries: count(), distinct()

To speed up the response time of these queries, we used MongoDB’s indexing options.

When more power is required, the ever-present mapReduce() is available.

Replica Sets, Sharding, (GeoSpatial, and GridFS)

Mongo has a powerful ability to store and query data in a variety of ways.

What makes document databases unique is their ability to efficiently handle arbitrarily nested, schema-less data documents.

What makes Mongo special in the realm of document stores is its ability to scale across several servers, by replicating (copying data to other servers) or sharding collections (splitting a collection into pieces) and performing queries in parallel.

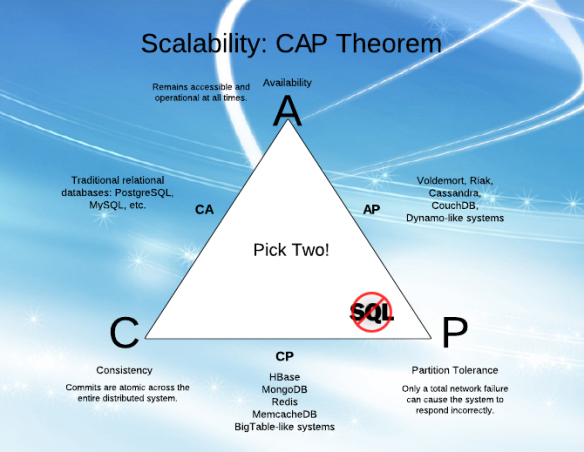
Both promote availability.

CAP Theorem

It’s very common to invoke the ‘CAP theorem’ when designing, or talking about designing, distributed data storage systems. The theorem, as commonly stated, gives system designers a choice between three competing guarantees:

<http://www.youtube.com/watch?v=Jw1iFr4v58M>

* Consistency: every read would get you the most recent write
* Availability: every node (if not failed) always executes queries
* Partition-tolerance: even if the connections between nodes are down, the other two (A & C) promises, are kept.



<http://architects.dzone.com/articles/better-explaining-cap-theorem>

Replica Sets

See: http://docs.mongodb.org/manual/core/replica-set-members/

Mongo was built to scale out, not to run stand-alone.

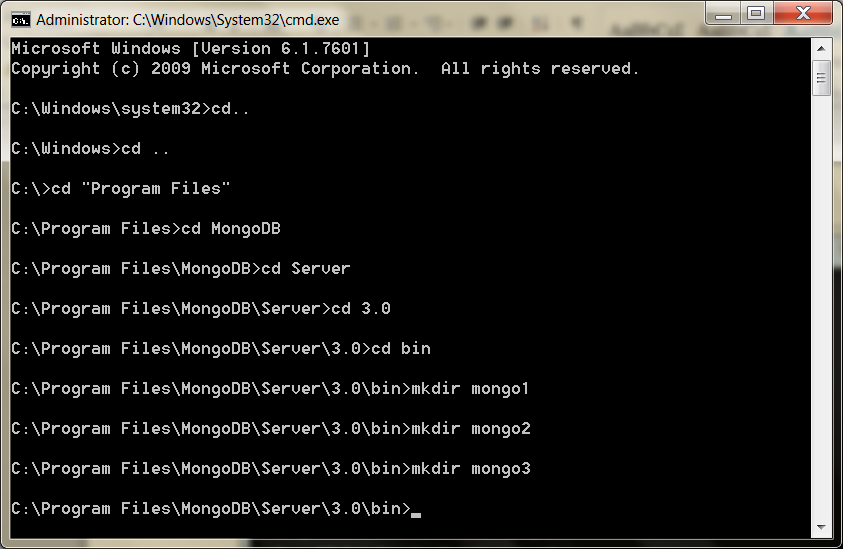
A replica set in MongoDB is a group of [mongod](https://docs.mongodb.com/manual/reference/program/mongod/" \l "bin.mongod) processes that provide redundancy and high availability.

It was built for data consistency and partition tolerance,

Mongo’s default port is 27017, so we’ll start up each server on other ports.

Recall you must create the data directories first, so create three of them:

$ mkdir ./mongo1 ./mongo2 ./mongo3



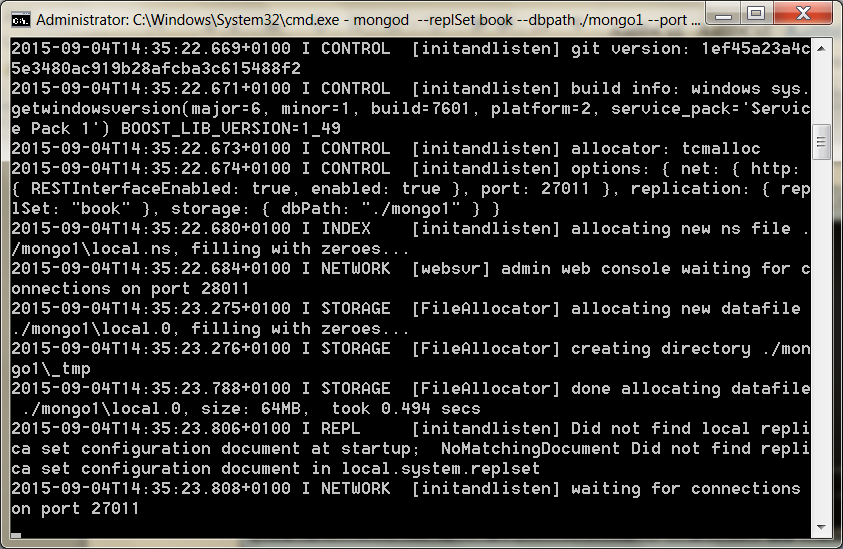
Next we’ll fire up the Mongo servers.

This time we’ll add the replSet flag with the name *book* and specify the ports.

Make sure to run in a command prompt as administrator

While we’re at it, let’s turn on the REST flag so we can use the web interface.

mongod --replSet book --dbpath ./mongo1 --port 27011

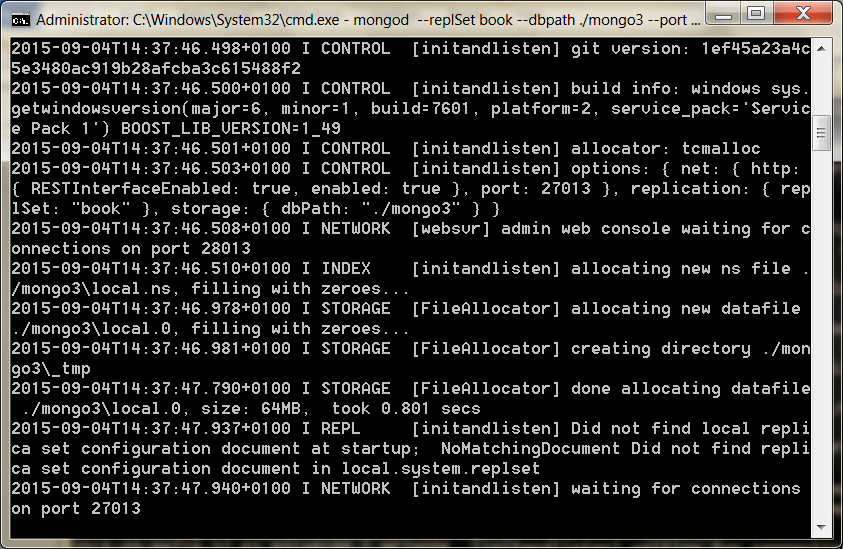


Open another terminal window, and run the next command, which launches another server, pointing to a different directory, available on another port.

 mongod --replSet book --dbpath ./mongo2 --port 27012

Then open a third terminal to start the third server.

mongod --replSet book --dbpath ./mongo3 --port 27013



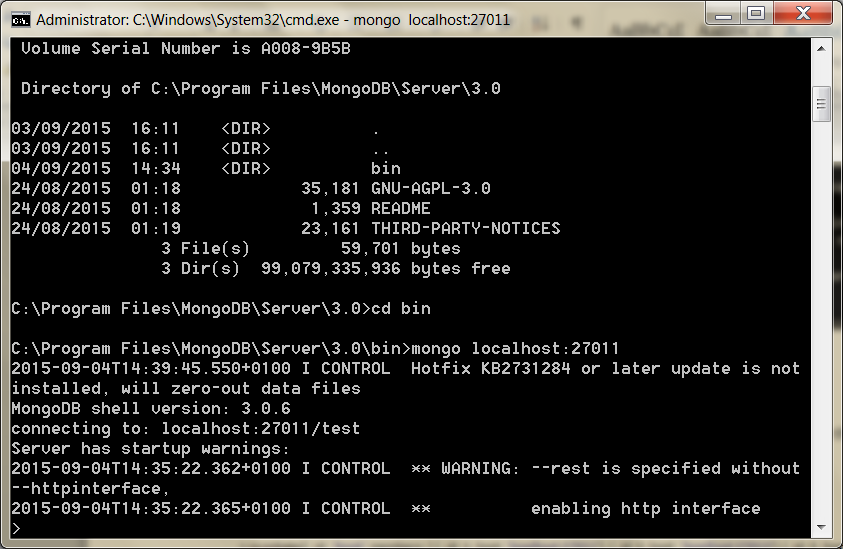
Notice that you get a lot of this noise on the output.

[startReplSets] replSet can't get local.system.replset config from self \  
 or any seed (EMPTYCONFIG)

That’s a good thing; we’ve yet to initialize our replica set, and Mongo is letting us know that.

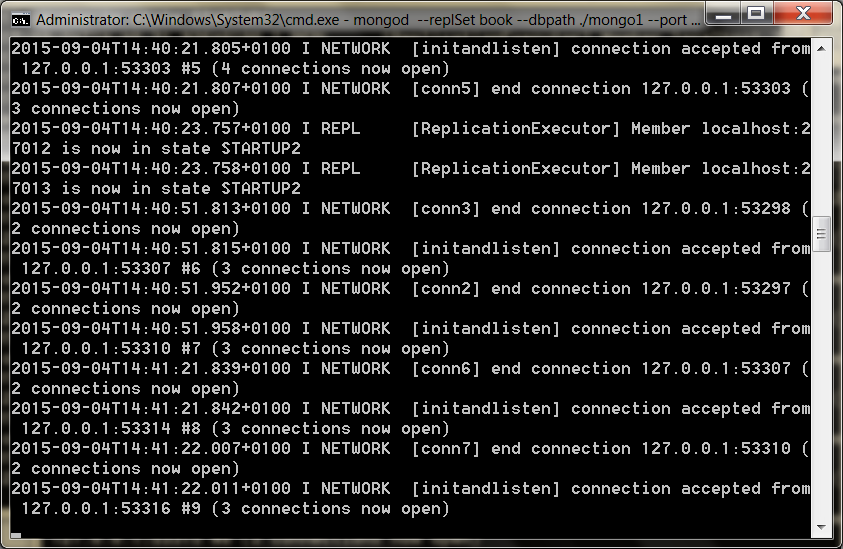
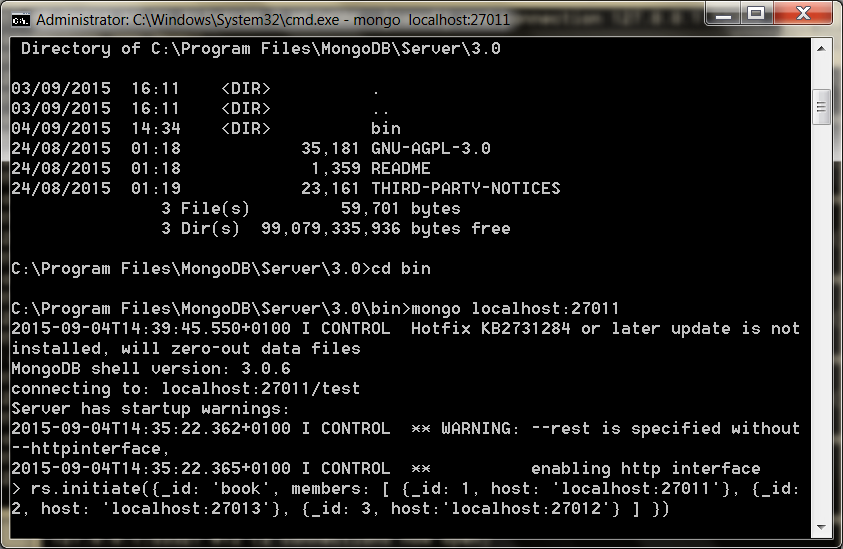
Fire up a mongo shell to one of the servers, and execute the rs.initiate() function.

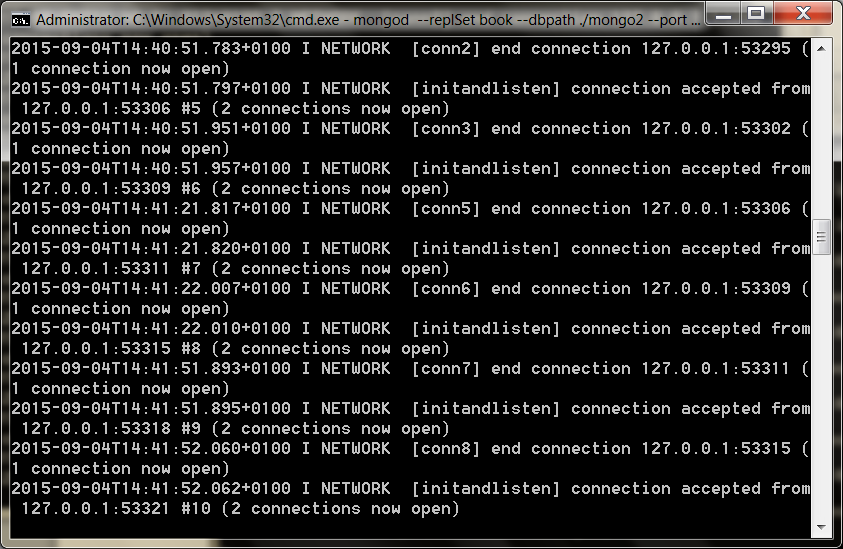
mongo localhost:27011

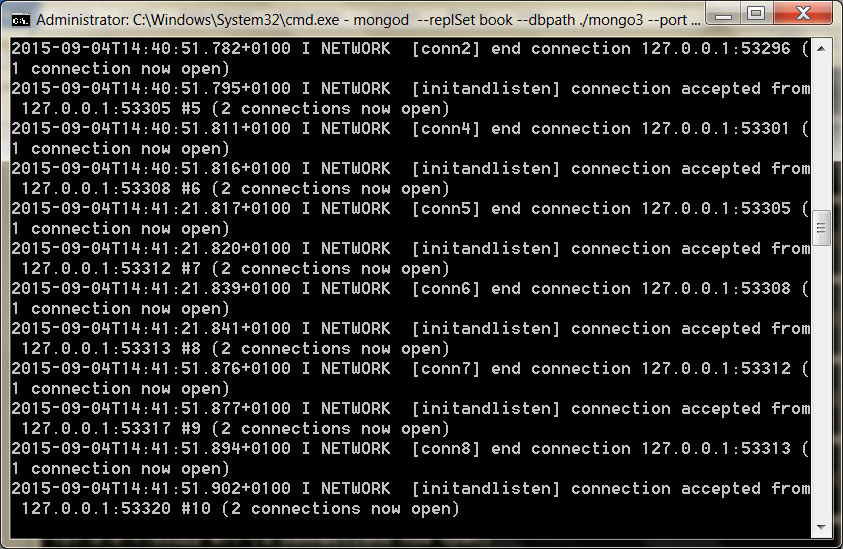


Run as Admin

rs.initiate({\_id: *'book'*, members: [ {\_id: 1, host: *'localhost:27011'*}, {\_id: 2, host: *'localhost:27013'*}, {\_id: 3, host:*'localhost:27012'*} ] })

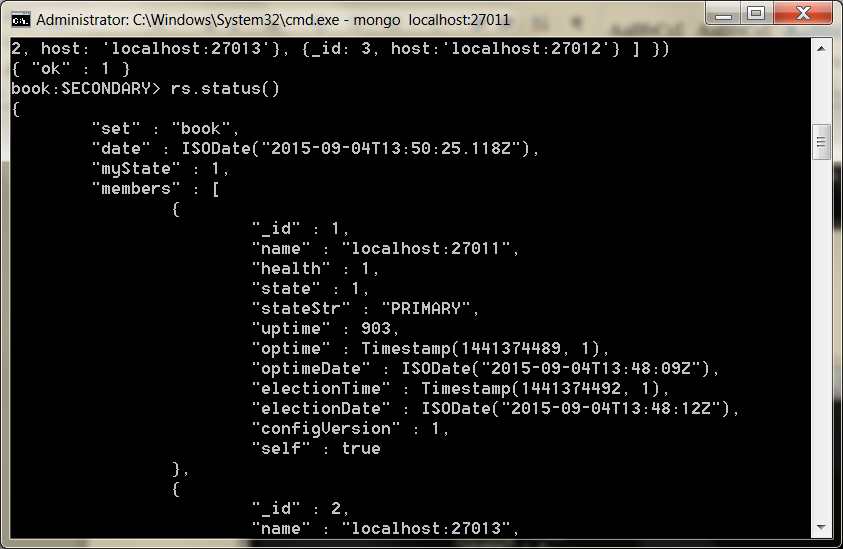






This could take 5-10 minutes to complete

 rs.status()



Notice we’re using a new object called rs (replica set).

Like other objects, it has a help() method you can call.

Running the status() command will let us know when our replica set is running, so just keep checking the status for completion before continuing.

If you watch the three server outputs, you should see that one server outputs this line:

[rs Manager] replSet PRIMARY

And two servers will have the following output:

[rs\_sync] replSet SECONDARY

Machine generated alternative text:
Administrator: - mangad --repISet book --dbpath ./mong02 
2ø14-ø9- 
2ø14-ø9- 
admin 
2ø14-ø9- 
2ø14-ø9- 
2ø14-ø9- 
ø14-ø9- 
17112: 
17112: 
17112: 
17112: 
17112: 
17112: 
24: 
24: 
24: 
24: 
24: 
24: 
ø4. 
ø4. 
ø4. 
ø4. 
ø4. 
ø4. 
85ø. 
85ø. 
855* 
855* 
973* 
974* 
øløø 
øløø 
øløø 
øløø 
øløø 
øløø 
øløø 
øløø 
øløø 
øløø 
øløø 
øløø 
øløø 
LrsSyncI replSet initial sync building indexes 
LrsSyncI replSet initial sync cloning indexes for . 
LrsSyncI oplog sync 3 of 3 
LrsSyncI replSet initial sync finishing up 
LrsBackgroundSyncI replSet syncing to: localhost : 2? 
LrsBackgroundSyncI replset setting syncSourceFeedba 
LrsSyncI replSet set 
LesSync1 replSet RECOUERING 
LrsSyncI replSet initial sync done 
LesSync1 replSet SECONDARY 
LrsHeaIthPoIII replSet member localhost : 2 WI 3 is no 
Lconn581 end connection (I connectio 
Linitandlistenl connection accepted from 12?.ø.ø.I: 
k to 
.94ø. 
.søo 
in state SECONDARY 
n now open) 
198 "6ø (2 connections 
now open) 
22øø "61 (2 connections 
n now open) 
now open) 
øIøø Lconn591 end connection (I connectio 
øIøø Linitandlistenl connection accepted from 12?.ø.ø.I: 
now open) 
øIøø Lconn6ØI end connection (I connectio 

Machine generated alternative text:
Administrator: - mangad --repISet book --dbpath ./mong03 
. vøø. 
Ck to localhost : 2 WII 
øløø 
øløø 
øløø 
øløø 
øløø 
øløø 
øløø 
øløø 
øløø 
øløø 
øløø 
LrsSyncI replSet initial sync finishing up 
LrsBackgroundSyncI replSet syncing to: localhost : 2? 
LrsBackgroundSyncI replset setting syncSourceFeedba 
LrsSyncI replSet set 
LesSync1 replSet RECOUERING 
LrsSyncI replSet initial sync done 
LrsHeaIthPoIII replSet member localhost : 2 WI 2 
LesSync1 replSet SECONDARY 
LrsHeaIthPoIII replSet member localhost : 2 WI 2 
Lconn591 end connection (I connectio 
Linitandlistenl connection accepted from 12?.ø.ø.I: 
2ø14- 
2ø14- 
2ø14- 
2ø14- 
W In 
2ø14- 
2ø14- 
W In 
2ø14- 
state RECOUERING 
.629* 
state SECONDARY 
n now open) 
2199 "61 (2 connections 
now open) 
2ø14-ø9-1?T12: 
2ø14-ø9-1?T12: 
2ø14-ø9-1?T12: 
2ø14-ø9-1?T12: 
2ø14-ø9-1?T12: 
n now open) 
2ø14-ø9-1?T12: 
24: 
24: 
24: 
24: 
24: 
24: 
16. 
16. 
16. 
16. 
25. 
25. 
344* 
344* 
344* 
344* 
924* 
924* 
øløø 
øløø 
øløø 
øløø 
øløø 
øløø 
'clientcursormonl mem (MB) res :44 :6994? 
'clientcursormonl mapped ( incl journal u iew) 
'clientcursormon connections : 2 
'clientcursormon replication threads : 32 
Lconn6ØI end connection (I connectio 
Linitandlistenl connection accepted from 12?.ø.ø.I: 

PRIMARY will be the master server.

Chances are, this will be the server on port 27011 (since it started first); however, if it’s not, go ahead and fire up a console to the primary.

mongo localhost:27011

Just insert any old thing on the command line, and we’ll try an experiment.

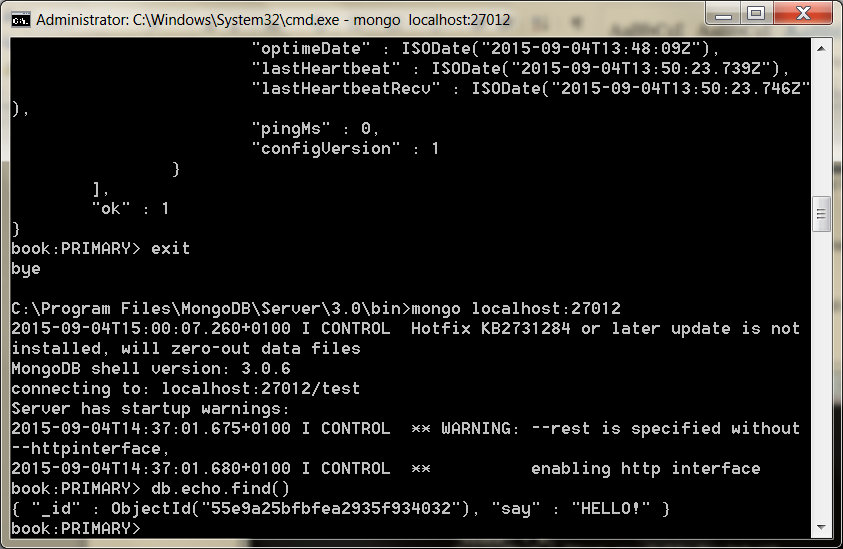
db.echo.insert({ say : *'HELLO!'* })

Machine generated alternative text:
Administrator: C.AWindcwsXsystern32,cmd .exe - 
mongo localhost:27DII 
book:PRIMRRY) 
book:PRIMRRY) 
book:PRIMRRY) 
book:PRIMRRY) 
book:PRIMRRY) 
book:PRIMRRY) 
book:PRIMRRY) 
db. echo . insert (4 
Write Result (4 
"n Inserted" 
book:PRIMRRY) _ 
say 
' HELLO! ' 
» 

After the insert, exit the console, and then let’s test that our change has been replicated by shutting down the master node; pressing CTRL+C is sufficient.

If you watch the logs of the remaining two servers, you should see that one of the two has now been promoted to master (it will output the replSet PRIMARY line).

Open a console into that machine (for us it was *localhost:27012*), and db.echo.find() should contain your value.



We’ll play one more round of our console-shuffle game.

Open a console into the remaining SECONDARY server.

Just to be sure, run the db.isMaster() function.

Ours looked like this:

$ mongo localhost:27011 (this was the secondary, 27012 was the master)  
  
> db.isMaster()  
{

"setName" : *"book"*,

"ismaster" : false,

"secondary" : true,

"hosts" : [

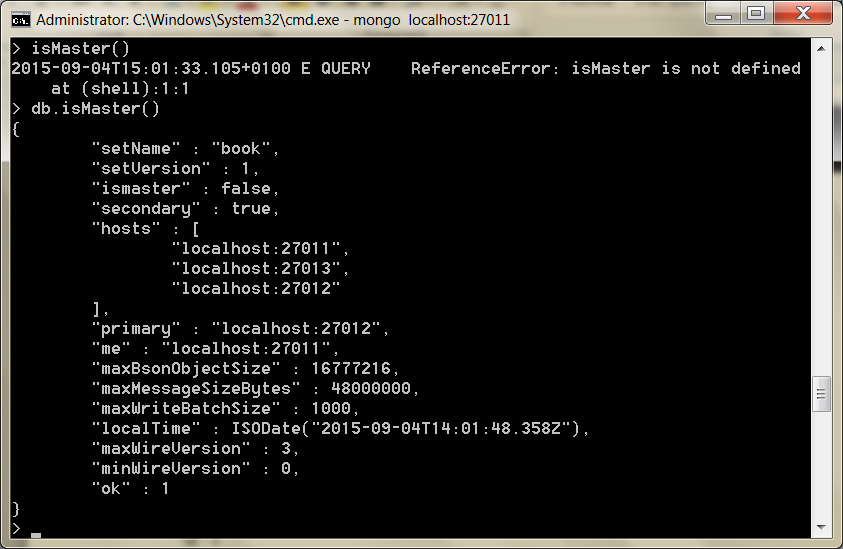
*"localhost:27013"*, *"localhost:27012"*, *"localhost:27011"*

],

"primary" : *"localhost:27012"*,

"ok" : 1

}



In this shell, let’s attempt to insert another value.

db.echo.insert({ say : *'is this thing on?'* })

You should see a writeError

not master

Machine generated alternative text:
Administrator: CAWindcwsN.system32Ncmd .exe - 
:XMongoDB 2.6 Standard)cd bin 
mongo localhost:27013 
——httpinter 
: XMongoDB 2 .6 Standard Xbin)mongo localhost : 2 WI 3 
MongoDB shell version: 2.6.3 
connecting to: 
Server has startup warnings: 
——rest is specif ied without 
enabling http interface 
WARNING: 
2ø14- 
face. 
2ø14- 
book: 
ø14- 
00k: 
00k: 
:54.612.ø1øø 
:54.612.ø1øø 
SECONDARY) isMaster() 
isMaster is not def ined 
ReferenceError: 
SECONDARY) db. 
"setName " 
"book". 
"setUersion" . 
'ismaster" 
'secondary" 
"hosts" . 
false. 
25. 
ø49Z") 
"localhost: 
"localhost: 
"localhost: 
'primary" 
2?ø13". 
2?ø12". 
2?ø11" 
"maxBsonObjectSize" 
'maxMessageSizeBytes " 
"maxWriteBatchS ize" . 
"local T ime" . 
. 48øøøøøø. 
1 øøø. 
'maxWireUers ion " 
'minWireUers ion " 
SECONDARY) db-echo . insert (4 
say 
» 
"not 
'is this thing on?' 
. undef ined- 
'errmsg" 
"writeError" . 
book:SECONDRRY) 
master" 

The message *not master* is letting us know that we cannot write to a secondary node.

Nor can you directly read from it.

There is only one master per replica set, and you must interact with it.

It is the gatekeeper to the set.

Replicating data has its own issues not found in single-source databases.

In the Mongo setup, one problem is deciding who gets promoted when a master node goes down.

Mongo deals with this by giving each mongod service a vote, and the one with the freshest data is elected the new master.

Right now you should still have two mongod services running.

Go ahead and shut down the current master.

Remember, when we did this with three nodes, one of the others just got promoted to be the new master.

BuT this time something different happened.

The output of the last remaining server will be something like this:

[ReplSetHealthPollTask] replSet info localhost:27012 is now down (or... [rs Manager] replSet can*'t see a majority, will not try to elect self*

Machine generated alternative text:
Administrator: CAWindowsIsystem32Ncmd.exe - mongo localhost:27013 
'members " 
'name " 
"localhost :wøll 
"health" : 
"state" 
"stateStr" : 
"(not reachable/ healthy)". 
"upt ime " 
"opt ime " 
Timestamp(141Ø95415?. 1). 
"opt imeDate " 
: 42 
"lastHeartbeat " 
: 55 
"lastHeartbeatRecu" . 
"pingMs" 
'name " 
"health" : 
"state" 
"stateStr" : 
"SECONDARY". 
"upt ime " 
. 322b 
"opt ime " 
Timestamp(141Ø95415?. 1). 
43 
42: 
"(not reachable/ healthy)". 
42: 
"opt imeDate " 
"self" 
'name " 
"health" : 
"state" 
"stateStr" : 
"upt ime " 
"opt ime " 
Timestamp(141Ø95415?. 1). 
"opt imeDate " 
"lastHeartbeat " 
55 
"lastHeartbeatRecu" 
: 52 
"pingMs" 
book:SECONDRRY) 

This comes down to the Mongo philosophy of server setups and the reason we should always have an odd number of servers (three, five, and so on).

Go ahead and relaunch the other servers and watch the logs.

When the nodes are brought back up, they go into a recovery state and attempt to resync their data with the new master node.

Machine generated alternative text:
Administrator: CAWindcwsN.system32Ncmd.exe - mongo localhost:27013 
00k:SECONDRRY) 
"book". 
"date" 
"myState" 
'members " 
'name " 
ø9- 
17111: 
57: 
2?ø13". 
36Z"). 
2?ø13". 
"localhost :wøll 
"SECONDARY". 
Timestamp(141Ø95415?. 1). 
"health" : 
"state" 
"stateStr" : 
"upt ime " 
"opt ime " 
"opt imeDate " 
: 42 
"lastHeartbeat " 
"lastHeartbeatRecu" . 
"pingMs" 
"lastHeartbeatMessage" 
'syncing to: localhost: 
'syncingTo" 
'name " 
"health" : 
"state" 
"stateStr" : 
"PRI 
"upt ime " 
. 3343. 
"opt ime " 
Timestamp(141Ø95415?. 1). 
: 42 
Timestamp(141Ø95498Ø. 1). 
"opt imeDate " 
"election T ime " 
"electionDate" . 
"self" 
'name " 
"health" : 
"state" 
"stateStr" : 
"upt ime " 
"SECONDARY". 
"opt ime " 
Timestamp(141Ø95415?. 1). 
"opt imeDate " 
: 42 
"lastHeartbeat " 
"lastHeartbeatRecu" . 
"pingMs" 
"lastHeartbeatMessage" 
'syncing to: localhost: 
'syncingTo" 

“What a minute!?” (we hear you cry). “So, what if the original master had data that did not yet propagate?”

Those operations are dropped.

A write in a Mongo replica set isn’t considered successful until most nodes have a copy of the data.

HERE 18/9

The Problem with Even Nodes

The concept of replication is easy enough to grasp: you write to one MongoDB server, and that data is duplicated across others within the replica set.

If one server is unavailable, then one of the others can be promoted and serve requests.

But there are more ways a server can be unavailable than a server crash.

Sometimes, the network connection between nodes is down.

In that case, Mongo dictates that *a majority of nodes that can still communicate make up the network*.

MongoDB expects an odd number of total nodes in the replica set.

Consider a five-node network, for example.

If connection issues split it into a three- node fragment and a two-node fragment, the larger fragment has a clear majority and can elect a master and continue servicing requests.

With no clear majority, a quorum couldn’t be reached.

To see why an odd number of nodes is preferred, consider what might happen to a four-node replica set.

Say a network partition causes two of the servers to lose connectivity from the other two.

One set will have the original master, but since it can’t see a *clear majority* of the network, the master steps down.

The other set will similarly be unable to elect a master because it too can’t communicate with a clear majority of nodes.

Both sets are now unable to process requests and the system is effectively down.

Having an odd number of total nodes would have made this particular scenario—a fragmented network where each fragment has less than a clear majority—less likely to occur.

Some databases (e.g., CouchDB) are built to allow multiple masters, but Mongo is not, and so it isn’t prepared to resolve data updates between them

MongoDB deals with conflicts between multiple masters by simply not allowing them.

Mongo always knows the most recent value; the client needn’t decide.

Mongo’s concern is strong consistency on writes, and preventing a multimaster scenario is not a bad method for achieving it.

You may not always want to have an odd number of servers replicating data.

In that case, you can either launch an arbiter (generally recommended) or increase voting rights on your servers (generally not recommended). - see book

Sharding

See Sharding at <http://docs.mongodb.org/manual/sharding/>

[Sharding](https://docs.mongodb.com/manual/reference/glossary/#term-sharding) is a method for distributing data across multiple machines. MongoDB uses sharding to support deployments with very large data sets and high throughput operations.

One of the central reasons for Mongo to exist is to safely and quickly handle very large datasets.

The clearest method of achieving this is through horizontal sharding by value ranges—or just *sharding* for brevity.

Rather than a single server hosting all values in a collection, some range of values are split (or in other words, sharded) onto other servers.

For example, in our phone numbers collection, we may put all phone numbers less than 1-500-000-0000 onto Mongo server A and put numbers greater than or equal to 1-500-000-0001 onto a server B.

Mongo makes this easier by autosharding, managing this division for you.

Like replica sets, there’s a special parameter necessary to be considered a shard server (which just means this server is capable of sharding).

$ mkdir ./mongo4 ./mongo5  
$ mongod --shardsvr --dbpath ./mongo4 --port 27014  
$ mongod --shardsvr --dbpath ./mongo5 --port 27015

Machine generated alternative text:
Administrator: - mongod --shardsvr --dbpath ./mong05 --port 27 . 
Linitandlistenl 
Linitandlistenl 
arding: clusterRoIe: 
"shardsur" storage: 
2ø14-ø9-17113 : ø3 : 54. .øløø 
Lin itandlistenl 
2ø14-ø9-17113 : ø3 : 54. .øløø 
Lin itandlistenl 
no recovery needed 
2ø14-ø9-1?T13 
L FileRIIocatorI 
cal.ns. filling with zeroes... 
2ø14-ø9-1?T13 
L FileRIIocatorI 
ø14-ø9-1?T13 .øløø 
L FileRIIocatorI 
ocal.ns. size: 16MB. 
took ø. 
ø45 secs 
ø14-ø9-1?T13 .øløø 
L FileRIIocatorI 
al.Ø. filling with zeroes... 
ø14-ø9-1?T13 :54.263.ø1øø 
L FileRIIocatorI 
ocaI.Ø. size: 64MB. 
took ø.ø98 secs 
allocator: system 
options: net: port: 2 WI 5 sh 
dbPath: 
'./mong05" 
journal dir—./mong05XjournaI 
. no journal files present. 
recover 
allocating new dataf ile . /mong05XIo 
creating directory . /mong05X_tmp 
done allocating dataf ile ./mong05XI 
allocating new dataf ile . /mong05XIo 
done allocating dataf ile ./mong05XI 
build index on: local. startup_log p 
"local. startup_log" 
added index to empty collection 
command local.$cmd command: create 
operties: u: I. key: 
"startup_log". 
create : 
numY ie Ids: reslen : 3? 
øIøø Lin itandlistenl 
id: I name: 
øIøø Lin itandlistenl 
øIøø Lin itandlistenl 
size: Iø485?6ø. capped: true ntoreturn:l keyUpdates 
14?ms 
øIøø Linitandlistenl waiting for connections on port 

Now we need a server to actually keep track of our keys.

Imagine we created a table to store city names alphabetically.

We need some way to know that (for example) cities starting with A–N go to server mongo4 and O–Z go to server mongo5.

In Mongo you create a *config server* (which is just a regular mongod) that keeps track of which server (mongo4 or mongo5) owns what values.

$ mkdir ./mongoconfig  
$ mongod --configsvr --replSet configSet --dbpath ./mongoconfig --port 27016

Machine generated alternative text:
Administrator: CAWindowsXsystem32,cmd.exe - mongod --configsvr --dbpath ./mongoconfig --p.. 
igXIocaI.ns. filling with zeroes... 
creating directory ./mongoconf igX_t 
fig X local.ns. size: 
igXIocaI.Ø. filling 
28ø.ø1øø 'FileR110cator1 
312.ø1øø 'FileR110cator1 
16MB. 
took ø.ø32 secs 
313.ø1øø 'FileR110cator1 
with zeroes... 
done allocating dataf ile 
allocating new datafile 
done allocating dataf ile 
. / mongocon 
. /mongoconf 
. / mongocon 
346* 
figXIocaI.Ø. size: 16MB. 
roperties: u: I. key: 
øløø 'FileR110cator1 
took ø.ø32 secs 
øIøø Lin itandlistenl 
id: I name: 
build index on: local. startup_log p 
"local. startup_log" 
added index to empty collection 
creating replication oplog of size: 
waiting for connections on port 
2ø14- 
2ø14- 
2ø14- 
2ø14- 
ø14- 
ø9- 
ø9- 
ø9- 
ø9- 
ø9- 
17113: 
17113: 
17113: 
17113: 
17113: 
ø4: 
ø4: 
ø4: 
ø4: 
ø4: 
51. 
51. 
51. 
51. 
51. 
348* 
348* 
348* 
eø6 
øløø 
øløø 
øløø 
øløø 
øløø 
Lin itandlistenl 
Lin itandlistenl 
Lin itandlistenl 
Lin itandlistenl 
Lin itandlistenl 

Now enter the Mongo shell for the config server by running mongo localhost:27016 and initiate the config server cluster (with just one member for this example):

rs.initiate({ \_id: 'configSet', configsvr: true, members: [ { \_id: 0, host: 'localhost:27016' } ] })

You should get

{ "ok" : 1}

Or type

rs.status().ok

1

Finally, we need to run a fourth server called mongos, which is the single point of entry for our clients.

The mongos server will connect to the mongoconfig config server to keep track of the sharding information stored there.

We’ll set it on port 27020 with a chunkSize of 1. (Our chunkSize is 1MB, which is the smallest value allowed.

This is just for our small dataset, so we can watch sharding take place. In production you’d use the default or a much bigger number.)

We point mongos to the config server:port with the --configdb flag.

$ mongos --configdb configSet/localhost:27016 --port 27020

Machine generated alternative text:
Administrator: CAWindowsIsystem32Ncmd.exe - mangos --configdb localhost:27016 --chunkSize 
6 port-2?Ø2Ø 64-bit (--help for usage) 
'mongosMain1 db version u2.6.3 
'mongosMain1 git version: 255f6?a66f96ø3c593Wb2a38 
9e38691øbbb52cb 
: 26 .291 *øløø LmongosMainI build info: windows sys .getwindowsuers 
ion (major:" minor—I. platform—2- service_pack—' Service Pack I') BOO 
:26. 
:26. 
ing: chunkSize: I. 
:26. 
3ØØØØms) 
:26. 
fully at Wed sep 1? 
:26. 
291 *øløø LmongosMainI allocator: system 
291 *øløø LmongosMainI options: net: port: 2?ø2ø shard 
conf igDB: 
525 L LockPingerI creating distributed lock ping thread 
and process (sleeping fo 
58? *øløø L LockPingerI cluster localhost pinged success 
13:ø5 : 26 2ø14 by distributed lock pinger ' 
141ø955526 :41' 
sleeping for 3ØØØØms 
58? *øløø LmongosMainI distributed lock ' conf igUpgrade/IXX341 
:41' acquired. ts : 
5419?9Ø66Øae45ea6a863ee8 
: 26 . "3 LmongosMainI starting upgrade of config server from 
uø to us 
: 26 . "3 LmongosMainI starting next upgrade step from uø to 
: 26 . "3 LmongosMainI about to log new metadata event: id 
. "IXX34 
clientRddr: 
. time: new Date(141Ø9555266Ø3). what: 
"starting u 
grade of config database". 
'conf ig . version 
details: from: to: 5 
: 26 . "3 LmongosMainI creating WriteBackListener for: localh 
ost :2vø16 seruer1D: øøøøøøøøøøøøøøøøøøøøøøøø 
: 27.133 *øløø LmongosMainI writing initial config version at us 
: 2?.36? *øløø LmongosMainI about to log new metadata event: id 
. "IXX34 
clientRddr: 
grade of config database". 
. time: new what: 
"finished u 
'conf ig . version 
details: from: to: 5 
*øløø LmongosMainI upgrade of config server to us success 
'mongosMain1 distributed lock 'configUpgrade/1XX341 
:41' unlocked. 
: 29 .2ø3 *øløø LmongosMainI scoped connection to localhost n 
t being returned to the pool 
.2ø6* 
.221* 
ssfuIIy 
.221* 
d at sep 1? 
89 :41' 
.229* 
:41' 
øIøø LmongosMainI waiting for connections on port 2?ø2ø 
øIøø 'Balancer) about to contact config servers and shar 
øIøø 'Balancer) config servers and shards contacted succ 
øløø 'Balancer) balancer id: start 
øløø 'Balancer) distributed lock 
acquired. ts 
. 5419?9Ø96Øae45ea6a863eed 
øløø 'Balancer) distributed lock 
un locked. 

A neat thing about mongos is that it is a lightweight clone of a full mongod server.

Nearly any command you can throw at a mongod, you can throw at a mongos, which makes it the perfect go-between for clients to connect to multiple sharded servers.

A picture of our server setup may help on page 171

Now let’s jump into the mongos server console on the admin database.

We’re going to configure some sharding.

$ mongo localhost:27020/admin

sh.addShard('localhost:27014')

sh.addShard('localhost:27015')

Machine generated alternative text:
Administrator: C:NWindcwsNsystem32Ncmd.exe - mango localhcst:27D2D/admin 
78 dbclient error communicating with server: localhost : 2 WI 3 
:38.81 ?.øløø 
.818 .øløø 
.818 .øløø 
2ø14-ø9-1?T13 .823 .øløø 
2ø14-ø9-1?T13 .823 .øløø 
2ø14-ø9-1?T13 .823 .øløø 
C: XMongoDB 2.6 Standard Win) 
'conso 
'conso 
'conso 
'conso 
'conso 
'conso 
shutdown: closing all files... 
closeRIIFiIes() finished 
journalCIeanup. 
remove JournalFiIes 
shutdown: removing f s lock... 
dbexit: really exiting now 
: XMongoDB 2 .6 Standard Win )mongo localhost 
ongoDB shell version: 2.6.3 
onnecting to: localhost 
ongos) db . runCommand( addshard . 
"shardRdded" : 
"shardøøøø". 
ongos) "shardRdded" 
"shardøøøø". 
.853 .øløø SyntaxError: Unexpected 
mongo. db. runCommand( addshard . 
øøøø". 
.164.ø1øø SyntaxError: Unexpected 
ongos) db. runCommand( addshard . 
'errmsg" 
"host already used" 
ongos) db. runCommand( addshard . 
"shardRdded" : 
"shardøøøl 
on os) 
token . 
"shardRdded" : 
token 
"shard 

With that set up, now we have to give it the database and collection to shard and the field to shard by (in our case, the city name).

db.runCommand( { enablesharding : *"test"* } )

{ "ok" : 1 }

db.runCommand( { shardcollection : *"test.cities"*, key : {name : 1} } )

{ "collectionsharded" : *"test.cities"*, "ok" : 1 }

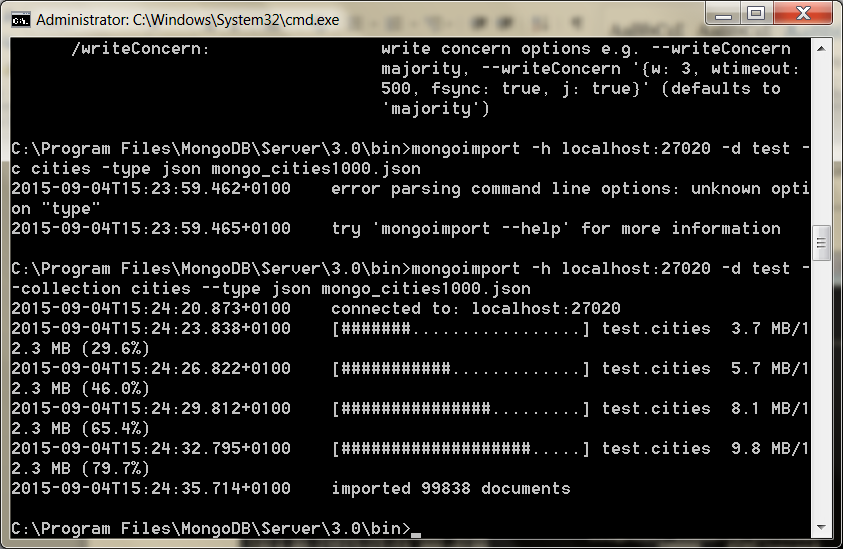
Machine generated alternative text:
Administrator: - mongo localhost:27D2D/admin 
'consolererminatel remoueJourna1Fi1es 
: 38 .823 *øløø 'console Terminate) shutdown: removing f s lock. 
: 38 .823*øIøø 'console Terminate) dbexit: really exiting now 
C: XMongoDB 2.6 Standard Win) 
C: XMongoDB 2 .6 Standard Win )mongo localhost 
MongoDB shell version: 2.6.3 
connecting to: localhost 
mongo s) db . runCommand( addshard . 
"shardRdded" : 
"shardøøøø". 
token . 
"shardRdded" : 
token 
mongo s) "shardRdded" 
"shardøøøø". 
.853 .øløø SyntaxError: Unexpected 
mongo. db. runCommand( addshard . 
øøøø". 
.164.ø1øø SyntaxError: Unexpected 
mongo. db. runCommand( addshard . 
'errmsg" 
"host already used" 
mongo. db. runCommand( addshard . 
"shard 
(name 
"shardRdded" : 
"shardøøøl 
mongo s) db . runCommand( enablesharding . 
ongos) db. runCommand( shardcollection . 
"collectionsharded" 
"test. cities". 
on os) 
"test" 
"test. 
cities". 

With all that setup out of the way, let’s load some data.

If you download the book code, you’ll find a 12MB data file named mongo\_cities1000.json that contains data for every city in the world with a population of more than 1,000 people.

Download that file to the working directory, and run the following import script (**from command line**) that imports the data into our mongos server:

mongoimport -h localhost:27020 -d test --collection cities --type json mongo\_cities1000.json



To see the number of chunks on each shard call

sh.status()

If all data is on one share try importing the data 2 more times.

to go to test

use test

To see distribution of collection

db.cities.getShardDistribution()

From the mongos console, type use test to go back to the test environment from the admin environment.

You may wonder why Mongo separates configuration and the mongos *point of entry* into two different servers.

This is because in production environments they will generally live on different physical servers.

The config server (itself replicated) manages the sharded information for other sharded servers, while mongos will likely live on your local application server where clients can easily connect (without needing to manage which shards to connect to).

GeoSpatial Queries

Mongo’s can quickly perform geospatial queries.

First connect to the mongos sharded server.

$ mongo localhost:27020

Switch to use test

The core of the geospatial secret lies in indexing.

It’s a special form of indexing geographic data called *geohash* that not only finds values of a specific value or range quickly but finds nearby values quickly in ad hoc queries.

Conveniently, at the end of our previous section, we installed a lot of geographic data.

So to query it, step 1 is to index the data on the location field.

The *2d* index must be set on any two value fields, in our case a hash (for example, { longitude:1.48453, latitude:42.57205 }), but it could easily have been an array (for example, [1.48453, 42.57205]).

db.cities.ensureIndex({ location : *"2d"* })

Machine generated alternative text:
Administrator: CAWindcwsXsystem32,cmd.exe - mongo localhost:27020/admin 
Json 
"collectionsharded" 
"test. cities". 
ongos) mongo import —h localhost : 2?ø2ø —db test 
——collection cities 
mongo_c it iesIØØØ . json 
:32.958.ø1øø SyntaxError: Unexpected identifier 
mongo. use test 
switched to db test 
ongos) db. cities . ensure location 
false. 
false. 
"createdCoIIectionRutomaticaIIy" 
'numIndexesBefore" . 
"numIndexesRfter" . 
"createdCoIIectionRutomaticaIIy" 
'numIndexesBefore " 
"numIndexesRfter" . 

Here is a sample of what it can return:

 db.runCommand({geoNear : *'cities'*, near : [52.26, -9.69], num : 20, maxDistance : 1})

{

"ns" : *"test.cities"*,

"near" : *"1000110001000000011100101011100011001001110001111110"*,

"results" : [

{  
 "dis" : 0.007105400003747849,  
 "obj" : {

"\_id" : ObjectId(*"4d81c216a5d037634ca98df6"*), "name" : *"Portland"*,

...

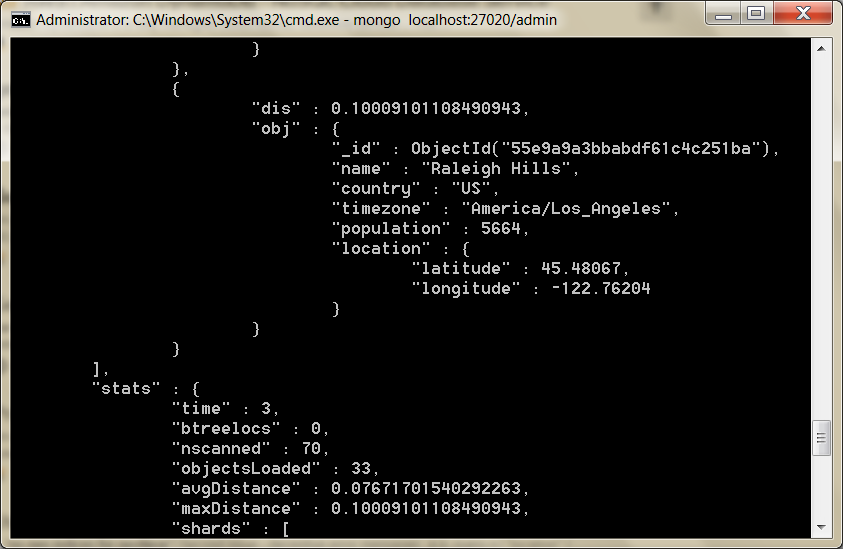
} },

... ],

"stats" : {  
 "time" : 0,  
 "btreelocs" : 53,  
 "nscanned" : 49,  
 "objectsLoaded" : 6,  
 "avgDistance" : 0.02166813996454613,  
 "maxDistance" : 0.07991909980773926

},

"ok" : 1 }



Try 53.34, -6.25

Machine generated alternative text:
Administrator: C.AWindowsN.system32,cmd.exe - mongo localhost:27020/admin 
"shards" : 
"shardøøøø". 
"shardøøøl " 
mongo. db. : 
stance 
"test. cities". 
'results" 
cities'. 
34. 
maxDi 
. ø.ø1?6469884116232?. 
'name " 
"Dublin 
"Europe / Dublin 
. 1ø24ø2?. 
'country" 
'populat ion " 
"location" . 
53 .34399. 
. -6.26719 
"latitude" : 
"longitude" 
. ø.ø3?844ø233ø62øø44. 
'name " 
"Donnybrook". 
"Europe / Dublin 
. 2øøø. 
'country" 
'populat ion " 
"location" . 
53 .313?5. 
. -6.22274 
"latitude" : 
"longitude" 

geoNear() also helps with troubleshooting geospatial commands

It returns a gold mine of useful information such as distance from the queried point, average and max distance of the returned set, and index information.

GridFS

One downside of a distributed system can be the lack of a single coherent filesystem.

Say you operate a website where users can upload images of themselves.

If you run several web servers on several different nodes, you must manually replicate the uploaded image to each web server’s disk or create some alternative central system.

Mongo handles this scenario by its own distributed filesystem called GridFS.

Mongo comes bundled with a command-line tool for interacting with the GridFS.

The great thing is we don’t have to set up anything special to use it.

If we list the files in the mongos managed shards using the command mongofiles, we get an empty list.

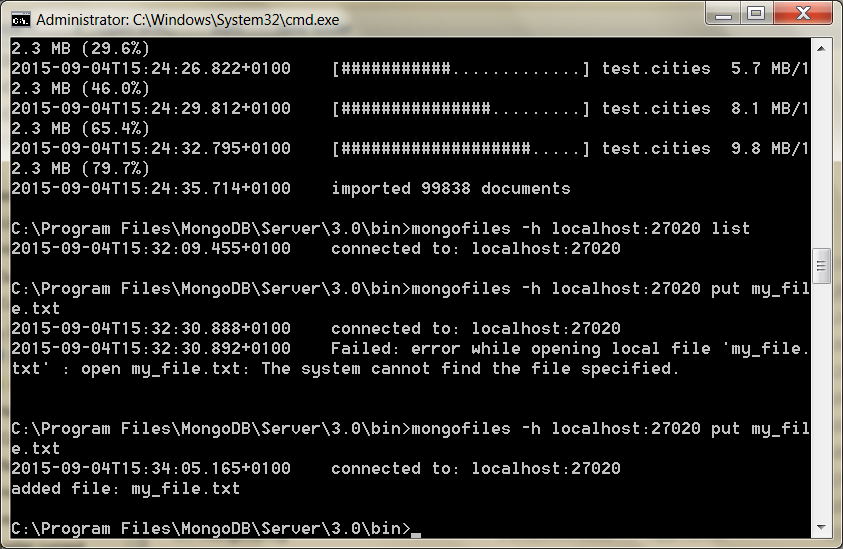
$ mongofiles -h localhost:27020 list

connected to: localhost:27020

Machine generated alternative text:
Administrator: CAWindowsXsystem32Ncmd.exe 
: XMongoDB 2 .6 Standard Win hongoimport —h localhost : 2?ø2ø —db test 
cities 
——type json mongo_citiesIØØØ.json 
——collection 
onnected to: 
localhost: 
2?ø2ø 
øløø 
øløø 
øløø 
øløø 
øløø 
øløø 
øløø 
øløø 
øløø 
øløø 
øløø 
øløø 
øløø 
øløø 
Progress: 637298/129356ø9 
48øø 
16ØØ/second 
Progress: 2616884/129356ø9 
2ø2øø 
3366/second 
Progress: 4626899/129356ø9 
358øø 
Progress: 6767116/129356ø9 
526øø 
4383/second 
Progress: 85ø8?88/129356ø9 
661øø 
44Ø6/second 
Progress: 9865493/129356ø9 
?66øø 
4255/second 
Progress: 
9 øøøø 
4285/second 
2ø14- 
2ø14- 
2ø14- 
2ø14- 
2ø14- 
2ø14- 
2ø14- 
2ø14- 
2ø14- 
2ø14- 
2ø14- 
2ø14- 
2ø14- 
2ø14- 
2ø14- 
ø14- 
ø9- 
ø9- 
ø9- 
ø9- 
ø9- 
ø9- 
ø9- 
ø9- 
ø9- 
ø9- 
ø9- 
ø9- 
ø9- 
ø9- 
ø9- 
ø9- 
17113: 
17113: 
17113: 
17113: 
17113: 
17113: 
17113: 
17113: 
17113: 
17113: 
17113: 
17113: 
17113: 
17113: 
17113: 
17113: 
12: 
12: 
12: 
12: 
12: 
12: 
12: 
12: 
12: 
12: 
12: 
12: 
12: 
12: 
13: 
13: 
41. 
41. 
44. 
44. 
53. 
53. 
56. 
56. 
59. 
59. 
øø. 
øø. 
øøs. 
øøs. 
øll* 
øll* 
øø2 
øø2 
ø13* 
ø13* 
øøe 
øøe 
646* 
øløø check 9 99838 
øIøø imported 99838 objects 
:xMongoDB 2. 
onnected to: 
:xMon oDB 2. 
6 Standard Win hongof i les —h localhost : 2?ø2ø list 
6 Standard Win) _ 

But upload any file (create the file first called my\_file.txt and save it in the working directory)

$ mongofiles -h localhost:27020 put my\_file.txt



And *voila*! If we list the contents of mongofiles, we’ll find the uploaded name name.

$ mongofiles -h localhost:27020 list

connected to: localhost:27020  
my\_file.txt 3067

Machine generated alternative text:
Administrator: CAWindcwsXsystem32,cmd.exe 
526øø 
4383/second 
Progress: 85ø8?88/129356ø9 
661øø 
44Ø6/second 
Progress: 9865493/129356ø9 
?66øø 
4255/second 
Progress: 
9 øøøø 
4285/second 
2ø14- 
2ø14- 
2ø14- 
2ø14- 
2ø14- 
2ø14- 
2ø14- 
2ø14- 
2ø14- 
ø9- 
ø9- 
ø9- 
ø9- 
ø9- 
ø9- 
ø9- 
ø9- 
ø9- 
17113: 
17113: 
17113: 
17113: 
17113: 
17113: 
17113: 
17113: 
17113: 
12: 
12: 
12: 
12: 
12: 
12: 
12: 
13: 
13: 
53. 
53. 
56. 
56. 
59. 
59. 
øø. 
øø. 
øø2 
øø2 
ø13* 
ø13* 
øøe 
øøe 
646* 
øløø 
øløø 
øløø 
øløø 
øløø 
øløø 
øløø 
øløø check 9 99838 
øIøø imported 99838 objects 
C: XMongoDB 2 .6 Standard Xbin)mongofiles —h localhost : 2?ø2ø list 
connected to: 
C:XMongoDB 2 .6 Standard Xbin)mongofiles —h localhost : 2?ø2ø put my_file . txt 
connected to: 
added file: id: filename: 
"my_f ile . txt" 
chunkSize: 26112b uploadDate: new Date(141Ø95654Ø546). md5: 
"Sd414ø2abc4b2a?6 
length: 5 
-h list 
done! 
C:xMongoDB 2. 
connected to: 
my_f ile . txt 
C:xMongoDB 2. 
6 Standard Win hongof i les 
6 Standard Win) 

Back in our mongo console, we can see the collections Mongo stores the data in.

> show collections

cities  
fs.chunks  
fs.files  
system.indexes

Machine generated alternative text:
Administrator: C.AWindowsXsystem32,cmd.exe 
- mongo localhost:27020/adrnl 
"latitude" 
"longitude" 
. 53. 
3ø15. 
'name " 
' 'Dén Laoghaire 
"Europe / Dublin 
. 1854øø. 
'country" 
'populat ion " 
"location" . 
53.29395. 
. -6.13586 
"latitude" : 
"longitude" 
"stats" 
"t ime " 
"btreelocs" . 
'nscanned" 
"objectsLoaded" 
. ø.ø6Éø41856593ø94. 
'augDistance " 
'maxDistance" 
"shards" : 
"shardøøøø". 
"shardøøøl " 
showCoIIections 
ongos) showCoIIections() 
.875 .øløø 
ongos) show collections 
s . chunks 
s. files 
ystem. indexes 
mon os) 
Reference Error: 
not 
def ined 

Since they’re just plain old collections, they ccan be replicated or queried like any other.

Summary

We focused on how Mongo enhances data durability with replica sets and supports horizontal scaling with sharding.

We looked at good server configurations and how Mongo provides the mongos server to act as a relay for handling autosharding between multiple nodes.

Finally, we toyed with some of Mongo’s built-in tools, such as geospatial queries and GridFS.

Mongo’s Strengths

Mongo’s primary strength lies in its ability to handle huge amounts of data (and huge amounts of requests) by replication and horizontal scaling.

But it also has an added benefit of a very flexible data model, since you needn’t ever conform to a schema and can simply nest any values you would generally join using SQL in an RDBMS anyway.

Finally, MongoDB was built to be easy to use.

You may have noticed the similarity between Mongo commands and SQL database concepts (minus the server-side joins). See: <https://docs.mongodb.org/manual/reference/sql-comparison/>

This is not by accident and is one reason Mongo is gaining so much mind share from former object-relational model (ORM) users.

It’s different enough to scratch a lot of developer itches but not so different it becomes a wholly different and scary monster.

Mongo’s Weaknesses

How Mongo encourages denormalization of schemas (by not having any) might be a bit too much for some to swallow.

Some developers find the cold, hard constraints of a relational database reassuring.

It can be dangerous to insert any old value of any type into any collection.

A single typo can cause hours of headache if you don’t think to look at field names and collection names as a possible culprit.

Mongo’s flexibility is generally not important if your data model is already fairly mature and locked down.

Because Mongo is focused on large datasets, it works best in large clusters, which can require some effort to design and manage.

Setting up a Mongo cluster requires a little forethought (see Riak later)

Parting Thoughts

Mongo is an excellent choice if you are currently using a relational database to store your data through an ORM out of habit.

We often recommend it to Rails, Django, and Model-View-Controller (MVC) developers, since they can then perform validations and field management through the models at the application layer (using Mongoose) and because schema migrations become a thing of the past (for the most part).

Adding new fields to a document is as easy as adding a new field to your data model, and Mongo will happily accept the new terms.

We find Mongo to be a much more natural answer to many common problem scopes for application-driven datasets than relational databases.

Author- I was on the fence about using a document datastore before making the switch in my own production code. Coming from the relational database world, I found Mongo to be an easy move with its ad hoc queries. And its ability to scale out mirrored my own web-scale dreams. But beyond the structure, I trusted the development team. They readily admitted that Mongo wasn’t perfect, but their clear plans (and general adherence to those plans) were based on general web infrastructure use cases, rather than idyllic debates on scalability and replication. This pragmatic focus on usability should shine as you use MongoDB. A trade-off of this evolutionary behavior is that there are several paths to performing any given function in Mongo.

References: Seven database in Seven Weeks