Mongo Lab

1. If Mongo isn’t installed then install it from <https://www.mongodb.org/downloads> and create a location to store data at c:\data\db
2. Open command prompt and create a new mongo database called book (mongo book)
3. Insert

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"* }

})

1. List the new entry; db.towns.find()
2. Create a script

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 });}

1. Insert

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

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

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

1. Find a document (replace with your own document ID)

db.towns.find({ "\_id" : ObjectId(*"5ba21618e575595ed51050de"*) })

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

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

1. Find using Perl-Compatible Regular Expressions (PCRE)

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

)

9 Find using your own criteria

**var** population\_range = {}

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

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

db.towns.find(

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

{ name: 1 } )

db.towns.find(

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

{ \_id : 0, name: 1 }

)

db.towns.find(

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

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

)

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

)

db.towns.find(

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

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

)

db.towns.find(

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

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

)

10 Query sub-documents

find towns with independent mayors

db.towns.find(

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

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

)

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

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

)

11 Use find with elemMatch

db.countries.insert({

\_id : *"us"*,

name : *"United States"*,

exports : {

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

{ name : *"burgers"* }

] }

})

db.countries.insert({

\_id : *"ca"*,

name : *"Canada"*,

exports : {

foods : [

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

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

db.countries.insert({

\_id : *"mx"*,

name : *"Mexico"*,

exports : {

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

}] } })

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 }

)

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 }  
)

$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 }  
)

12. Find using 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 } )

13 Let’s update our towns collection to add some U.S. states (use your own objectr ID)

db.towns.update({ \_id : ObjectId(*"5d839e513af85d1d3b72dcb4"*) }, { $set : { "state" : *"OR"* } } );

db.towns.findOne({ \_id : ObjectId(*"5d839e513af85d1d3b72dcb4"*) })

14 Delete documents (add a document and remove it)

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

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

} }

}

db.countries.find( bad\_bacon )

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

14 Indexing

Create 100000 phone numbers

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 )

To see what they look like:

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

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()

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: *"+6 800-5550018"*}).explain(*"*executionStats*"*)

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*"*)

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.

15. Aggregate queries

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

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 ]

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

16 Mapreduce and finalize

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).

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})

With all that in, we can do a quick test:

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

map = **function**() {

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

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

}

newMap = **function**() {

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

}

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'* })

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

map: newMap,

reduce: reduce,

out: {inline: 1}})

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

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

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 }})

16 See lecture notes on creating replica sets and try to create some yourself

17 see lecture notes on sharding and try it yourself

18 Try some geospatial queries on the sharded data

19 try the gridFS example in the lecture notes.