

CSC561 NoSQL Databases

Lab 3: Document Database / MongoDB

Document Databases:

A Document DB is made up of databases which contain collections. A collection is made up of documents. Each document is made up of fields. Collections, like RDBs, can be indexed, which improves lookup and sorting performance.

These concepts are similar to their relational database counterparts, but they are not identical. The core difference comes from the fact that relational databases define columns at the table level whereas a document-oriented database defines its fields at the document level. That is to say that each document within a collection can have its own unique set of fields. As such, a collection is a “weaker” container in comparison to a table (but shares enough in common with a traditional table that you can safely think of the two as the same thing), while a document has a lot more information than a row.

MongoDB uses Cursors. The important thing to understand about cursors is that when you ask MongoDB for data, it returns a pointer to the result set called a cursor, which we can do things to, such as counting or skipping ahead, before actually pulling down data. A cursor’s actual execution is delayed until necessary.

Why MongoDB:

MongoDB’s strength lies in versatility, power, ease of use, and ability to handle jobs both large and small. First publicly released in 2009, MongoDB was designed as a scalable database—the name Mongo comes from “humongous”—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 (similar to Riak but unlike Postgres), so documents can optionally contain fields or types that no other document in the collection contains.

It is used in industry; notable deployments include Metlife, CitiGroup, eBay, Foursquare, McAfee, and Forbes. (Enterprise Customer lists can be found here: <https://www.mongodb.com/industries>).

A Word of Caution

Mongo is not very friendly when it comes to misspellings, so be warned. You can draw parallels between static and dynamic programming languages. You define static up front, while dynamic will accept values you may not have intended, even nonsensical types like `person_name = 5`.

Documents are schemaless, so Mongo has no way of knowing if you intended on inserting “pipulation” instead of “population” into your city collection or if you meant to querying on “list_census”, instead of “last_census”; it will happily insert those fields or return no matching values. It is up to you to watch your return statements from the information you enter to ensure proper execution.

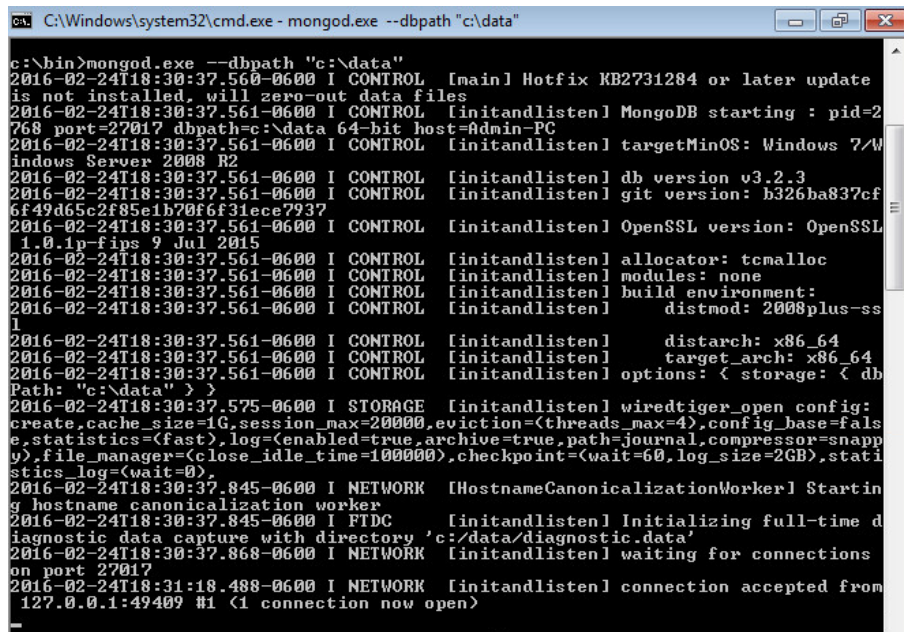
Flexibility has its price.

Starting MongoDB:

MongoDB has been installed on a Windows 7 Machine in your VM directory. Power on your CSC570_NoSQL_MongoDB_xx machine. (*****NOTE** You only need to power on the machine(s) we are using for each module. Please do not power-on your other machines*****) We need to start a localhost instance of the MongoDB by opening up a command line window, navigating to the /bin directory, and entering:

```
c:\bin>mongod.exe --dbpath "c:\data"
```

It should return:



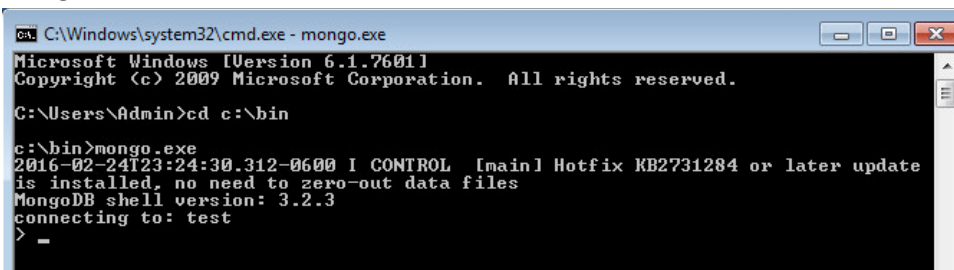
```
C:\Windows\system32\cmd.exe - mongod.exe --dbpath "c:\data"

c:\bin>mongod.exe --dbpath "c:\data"
2016-02-24T18:30:37.560-0600 I CONTROL [main] Hotfix KB2731284 or later update
is not installed, will zero-out data files
2016-02-24T18:30:37.561-0600 I CONTROL [initandlisten] MongoDB starting : pid=2
768 port=27017 dbpath=c:\data 64-bit host=Admin-PC
2016-02-24T18:30:37.561-0600 I CONTROL [initandlisten] targetMinOS: Windows 7/W
indows Server 2008 R2
2016-02-24T18:30:37.561-0600 I CONTROL [initandlisten] db version v3.2.3
2016-02-24T18:30:37.561-0600 I CONTROL [initandlisten] git version: b326ba837cf
6f49d65c2f85e1b70f6f31ece7937
2016-02-24T18:30:37.561-0600 I CONTROL [initandlisten] OpenSSL version: OpenSSL
1.0.1p-fips 9 Jul 2015
2016-02-24T18:30:37.561-0600 I CONTROL [initandlisten] allocator: tcmalloc
2016-02-24T18:30:37.561-0600 I CONTROL [initandlisten] modules: none
2016-02-24T18:30:37.561-0600 I CONTROL [initandlisten] build environment:
2016-02-24T18:30:37.561-0600 I CONTROL [initandlisten] distmod: 2008plus-ssl
2016-02-24T18:30:37.561-0600 I CONTROL [initandlisten] distarch: x86_64
2016-02-24T18:30:37.561-0600 I CONTROL [initandlisten] target_arch: x86_64
2016-02-24T18:30:37.561-0600 I CONTROL [initandlisten] options: { storage: { db
Path: "c:\data" } }
2016-02-24T18:30:37.575-0600 I STORAGE [initandlisten] wiredtiger_open config:
create,cache_size=1G,session_max=20000,eviction=(threads_max=4),config_base=fals
e,statistics=(fast),log=(enabled=true,archive=true,path=journal,compressor=snapp
y),file_manager=(close_idle_time=100000),checkpoint=(wait=60,log_size=2GB),stat
istics_log=(wait=0)
2016-02-24T18:30:37.845-0600 I NETWORK [HostnameCanonicalizationWorker] Startin
g hostname canonicalization worker
2016-02-24T18:30:37.845-0600 I FTDC [initandlisten] Initializing full-time d
iagnostic data capture with directory 'c:/data/diagnostic.data'
2016-02-24T18:30:37.868-0600 I NETWORK [initandlisten] waiting for connections
on port 27017
2016-02-24T18:31:18.488-0600 I NETWORK [initandlisten] connection accepted from
127.0.0.1:49409 #1 (1 connection now open)
```

This will show the 'waiting for connections' message on the console output indicating that the mongod.exe process is running successfully.

Now open a 2nd command-line window to connect a local client to the DB and verify Mongo is up and running by entering:

```
cd c:\bin
mongo.exe
```



```
C:\Windows\system32\cmd.exe - mongo.exe

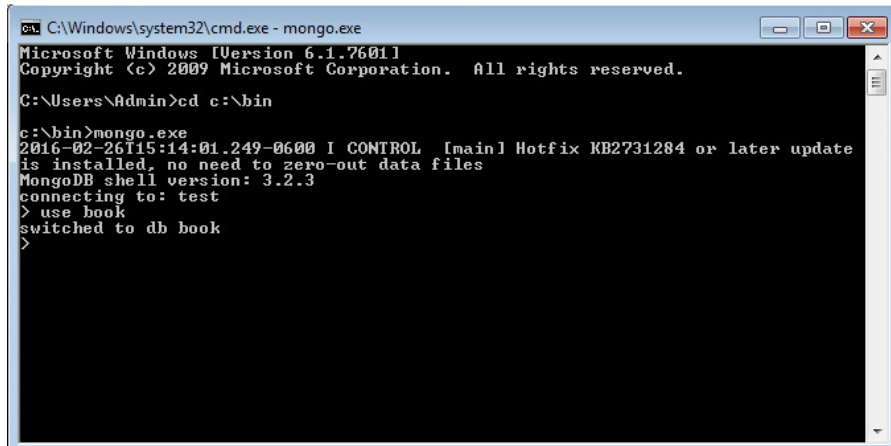
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Admin>cd c:\bin

c:\bin>mongo.exe
2016-02-24T23:24:30.312-0600 I CONTROL [main] Hotfix KB2731284 or later update
is installed, no need to zero-out data files
MongoDB shell version: 3.2.3
connecting to: test
> -
```

Change the working database to “book” by entering:

>use book



```
C:\Windows\system32\cmd.exe - mongo.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Admin>cd c:\bin

c:\bin>mongo.exe
2016-02-26T15:14:01.249-0600 I CONTROL [main] Hotfix KB2731284 or later update
is installed, no need to zero-out data files
MongoDB shell version: 3.2.3
connecting to: test
> use book
switched to db book
>
```

The “USE” instruction is powerful in MongoDB; if the database you request doesn’t exist in the workspace, it will be created.

Anytime you take a break (shutdown your VM) while performing the Mongo Lab, you will need to do these tasks:

1. Open a command line window and start mongod.exe
2. Open a 2nd command line window and start mongo.exe
3. Enter “use” to select which database you want to connect to

MongoDB CRUD:

Mongo is a JSON document database (though technically data is stored in a binary form of JSON known as BSON), thus our CRUD work in Mongo will look similar to RIAK. 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.

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 schemaless, there is no need to define anything up front; merely using it is enough.

The following code creates/inserts a towns collection:

```
> db.towns.insert({
name: "New York",
population: 8406000,
last_census: ISODate("2012-07-31"),
famous_for: [ "statue of liberty", "broadway", "food" ],
mayor : {
name : "Bill de Blasio",
party : "D"}
})
```

```
> db.towns.insert({ name:"New York", population: 8406000, last_census: ISODate("2012-07-31"), famous_for: ["statue of liberty", "broadway", "food"], mayor : { name: "Bill de Blasio", party:"D"}})
WriteResult({ "nInserted" : 1 })
>
```

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

```
> show collections
```

```
> show collections
towns
>
```

We can list the contents of a collection via find(). (The output will be unformatted as a single wrapped line.)

```
> db.towns.find()
```

```
> db.towns.find()
{ "_id" : ObjectId("56d0e7c640f012a0dad9ae50"), "name" : "New York", "population" : 8406000, "last_census" : ISODate("2012-07-31T00:00:00Z"), "famous_for" : [ "statue of liberty", "broadway", "food" ], "mayor" : { "name" : "Bill de Blasio", "party" : "D" } }
>
```

You may have noticed that the JSON output of your newly inserted town contains an `_id` field of ObjectId. This is similar to SERIAL incrementing a numeric primary key in PostgreSQL. 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 auto-numbering scheme is that each process on every machine can handle its own ID generation without colliding with other mongod instances. This design choice is indicative of Mongo's distributed nature.

Mongo's native language is JavaScript. Everything you want to do in Mongo can be done with JavaScript – even complex code like mapreduce queries.

Populate a Collection using JavaScript

For our first task, let's write a javascript function to populate our `towns` collection (type this into your shell):

```
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
});
}
```

```
> 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});
... }

```

Now we can call it to insert some additional cities:

```
insertCity("Springfield", 116250, "2010-07-10", ["Abraham Lincoln", "route 66"], {name: "Jim Langfelder"})
```

```
insertCity("Chicago", 2695600, "2010-07-10", ["The Cubs", "pizza"], {name: "Rahm Emanuel", party: "D"})
```

```
> insertCity("Springfield", 116250, "2010-07-10", ["Abraham Lincoln", "route 66"], {name: "Jim Langfelder"})
> insertCity("Chicago", 2695600, "2010-07-10", ["The Cubs", "pizza"], {name: "Rahm Emanuel", party: "D"})

```

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

```
> db.towns.find()
{ "_id" : ObjectId("56d0e7c640f012a0dad9ae50"), "name" : "New York", "population" : 8406000, "last_census" : ISODate("2012-07-31T00:00:00Z"), "famous_for" : [ "statue of liberty", "broadway", "food" ], "mayor" : { "name" : "Bill de Blasio", "party" : "D" } }
{ "_id" : ObjectId("56d34343084bfde6c9242288"), "name" : "Springfield", "population" : 116250, "last_census" : ISODate("2010-07-10T00:00:00Z"), "famous_for" : [ "Abraham Lincoln", "route 66" ], "mayor" : { "name" : "Jim Langfelder" } }
{ "_id" : ObjectId("56d34439084bfde6c9242289"), "name" : "Chicago", "population" : 2695600, "last_census" : ISODate("2010-07-10T00:00:00Z"), "famous_for" : [ "The Cubs", "pizza" ], "mayor" : { "name" : "Rahm Emanuel", "party" : "D" } }

```

Reading from a Collection

We called the `find()` function without params to get all documents in our collection. 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. (You also need to use one of the IDs that your system assigned – typing the ones from the screenshot above will probably not work because your system created its own unique ID)

For my database, the query is:

```
db.towns.find({"_id": ObjectId("56d34343084bfde6c9242288") })
```

```
> db.towns.find({"_id": ObjectId("56d34343084bfde6c9242288")})
{ "_id" : ObjectId("56d34343084bfde6c9242288"), "name" : "Springfield", "population" : 116250, "last_census" : ISODate("2010-07-10T00:00:00Z"), "famous_for" : [ "Abraham Lincoln", "route 66" ], "mayor" : { "name" : "Jim Langfelder" } }
>

```

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("56d34343084bfde6c9242288")}, {name: 1})
```

```
> db.towns.find(<{"_id": ObjectId("56d34343084bfde6c9242288")}, {name: 1}>)
{ "_id" : ObjectId("56d34343084bfde6c9242288"), "name" : "Springfield" }
>
```

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

```
db.towns.find({"_id": ObjectId("56d34343084bfde6c9242288")}, {name: 0})
```

```
> db.towns.find(<{"_id": ObjectId("56d34343084bfde6c9242288")}, {name: 0}>)
{ "_id" : ObjectId("56d34343084bfde6c9242288"), "population" : 116250, "last_census" : ISODate("2010-07-10T00:00:00Z"), "famous_for" : [ "Abraham Lincoln", "route 66" ], "mayor" : { "name" : "Jim Langfelder" } }
>
```

Ad-hoc Query

Like PostgreSQL, in Mongo you can construct ad-hoc queries by field values, ranges, or a combination of criteria. To find all towns that begin with the letter S and have a population less than 1,000,000, you can use a Perl-compatible regular expression (PCRE)² and a range operator.

```
db.towns.find(
{ name : /^S/, population : { $lt : 1000000 } },
{ name : 1, population : 1 }
)
```

```
> db.towns.find(
... {name:/^S/,population:<$lt:1000000}},
... {name:1, population:1})
{ "_id" : ObjectId("56d34343084bfde6c9242288"), "name" : "Springfield", "population" : 116250 }
>
```

Conditional operators in Mongo (Like the \$lt above) follow the format of *field* : { \$op : value }, where \$op is an operation like \$ne (not equal to). You may want a terser syntax, like *field* < value. But this is JavaScript code, not a domain-specific query language, so queries must comply with JavaScript syntax rules. While this may seem “clunky”, with JavaScript is you can construct operations as you would objects.

For example, we can build criteria where the population must be between 1 million and 5 million people and use those objects in our query:

```
var population_range = {}
population_range['$lt'] = 5000000
population_range['$gt'] = 1000000
```

```
db.towns.find(
{ population : population_range },
{ name: 1 })
```

```
> var population_range={}
> population_range["$lt"]=5000000
5000000
> population_range["$gt"]=1000000
1000000
> db.towns.find( {population:population_range}, {name:1, population:1})
< "_id" : ObjectId<"56d34439084bfde6c9242289">, "name" : "Chicago", "population"
: 2695600 >
>
>
```

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, 2011, with this:

```
db.towns.find(
{ last_census : { $lte : ISODate('2011-31-01') } },
{ _id : 0, name: 1 }
)
```

```
> db.towns.find(
... {last_census:{lte: ISODate("2011-31-01")}},
... {_id:0, name:1})
< "name" : "New York" >
< "name" : "Springfield" >
< "name" : "Chicago" >
>
```

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

Mongo works well with nested array data. You can quickly query your collections:

by matching exact values:

```
db.towns.find(
{ famous_for : 'pizza' },
{ _id : 0, name : 1, famous_for : 1 }
)
```

Or matching partial values:

```
db.towns.find(
{ famous_for : /statue/ },
{ _id : 0, name : 1, famous_for : 1 }
)
```

or finding documents that lack of matching values:

```
db.towns.find(
{ famous_for : { $nin : ['food', 'beer'] } },
{ _id : 0, name : 1, famous_for : 1 }
)
```



```
C:\Windows\system32\cmd.exe - mongo.exe
> db.towns.find( {famous_for: "pizza"}, {_id:0, name:1, famous_for:1})
{ "name" : "Chicago", "famous_for" : [ "The Cubs", "pizza" ] }
> db.towns.find( {famous_for: /statue/}, {_id:0, name:1, famous_for:1})
{ "name" : "New York", "famous_for" : [ "statue of liberty", "broadway", "food" ] }
> db.towns.find( {famous_for:{ $nin: ["food","pizza"]}}, {_id:0, name:1, famous_for:1})
{ "name" : "Springfield", "famous_for" : [ "Abraham Lincoln", "route 66" ] }
```

Beyond the ease of ad-hoc querying, Mongo is also powerful in 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 example, you can find towns with democratic mayors...

```
db.towns.find(
{ 'mayor.party' : 'D' },
{ _id : 0, name : 1, mayor : 1 }
)
```

```
> db.towns.find( { "mayor.party": "D" }, { _id:0, name:1, mayor:1 })
{ "name" : "New York", "mayor" : { "name" : "Bill de Blasio", "party" : "D" } }
{ "name" : "Chicago", "mayor" : { "name" : "Rahm Emanuel", "party" : "D" } }
```

Or those with mayors who left the party affiliation blank:

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

```
> db.towns.find( { "mayor.party": { $exists: false } }, { _id:0, name:1, mayor:1 })
{ "name" : "Springfield", "mayor" : { "name" : "Jim Langfelder" } }
```

More Query Power: Using Directives

Let's create another collection that stores countries. This time we'll override each `_id` to be a string of our choosing and nest some documents concerning export foods :

```
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}]}
})

```

```

> db.countries.insert<<
... _id : "us",
... name : "United States",
... exports : {
... foods : [
... { name : "bacon", tasty : true },
... { name : "burgers" }
... ]
... }
... >>
WriteResult<< "nInserted" : 1 >>
> db.countries.insert<<
... _id : "ca",
... name : "Canada",
... exports : {
... foods : [
... { name : "bacon", tasty : false },
... { name : "syrup", tasty : true }
... ]
... }
... >>
WriteResult<< "nInserted" : 1 >>
> db.countries.insert<<
... _id : "mx",
... name : "Mexico",
... exports : {
... foods : [{
... name : "salsa",
... tasty : true,
... condiment : true
... }]
... }
... >>
WriteResult<< "nInserted" : 1 >>
> _

```

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

```

> print<db.countries.count<>>
3
> print( db.countries.count() )
>

```

Now let's query our DB to find the countries that export tasty bacon:

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 }
)
```

```
> db.countries.find( { 'exports.foods.name' : 'bacon', 'exports.foods.tasty' : true }, { _id : 0, name : 1 } )
{ "name" : "United States" }
{ "name" : "Canada" }
>
```

But this isn't what we wanted. Mongo returned Canada because it exports bacon and exports tasty syrup. Mongo has a directive, `$elemMatch` that can further help us filter the information we want. 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 })
```

```
> db.countries.find(
... {
...   'exports.foods' : {
...     $elemMatch : {
...       name : 'bacon',
...       tasty : true
...     }
...   }
... },
... { _id : 0, name : 1 }
... )
{ "name" : "United States" }
```

Now, we only return the 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 }  
)
```

Returns Mexico, which has the tasty condiment “salsa”.

```
> db.countries.find(  
... {  
...   'exports.foods' : {  
...     $elemMatch : {  
...       tasty : true,  
...       condiment : { $exists : true }  
...     }  
...   }  
... }  
... )  
... { _id : 0, name : 1 }  
... }  
< "name" : "Mexico" }  
> =
```

There are many operators and we won't cover them all in this lab, but you should have an appreciation for MongoDB's powerful query ability. You can find all the commands on the MongoDB online documentation or grab a cheat sheet from the Mongo website:

<https://docs.mongodb.org/manual/reference/operator/query/>

MongoDB Updating and Directives

We created our towns collection without considering if the Springfield we added was in Illinois or Missouri. Frequently, you may find that you need to update existing information within a collection.

Let's update our towns collection to add the state that the town is in. 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 the following case, the modifier is to `$set` the field “state” with the string IL:

```
db.towns.update(  
{ _id : ObjectId("56d34343084bfde6c9242288") },  
{ $set : { "state" : "IL" } }  
);
```

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("56d34343084bfde6c9242288") })
```

```

> db.towns.update( { _id : ObjectId("56d34343084bfde6c9242288") }, { $set : { "state" : "IL" } } );
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 0 })
> db.towns.findOne( { _id : ObjectId("56d34343084bfde6c9242288") } )
{
  "_id" : ObjectId("56d34343084bfde6c9242288"),
  "name" : "Springfield",
  "population" : 116250,
  "last_census" : ISODate("2010-07-10T00:00:00Z"),
  "famous_for" : [
    "Abraham Lincoln",
    "route 66"
  ],
  "mayor" : {
    "name" : "Jim Langfelder"
  },
  "state" : "IL"
}
> _

```

You may wonder why the \$set operation is even required. Mongo doesn't think in terms of attributes; it has only an internal, implicit understanding of attributes for optimization reasons. But nothing about the interface is attribute-oriented. Mongo is document-oriented. You will rarely want something like this (notice the lack of \$set operation):

```

db.towns.update(
{ _id : ObjectId("4d0ada87bb30773266f39fe5") },
{ state : "NY" } );

```

This would replace the entire matching document with the document you gave it ({ state : "NY" }). Since you didn't give it a command like \$set, Mongo assumes you just want to replace the existing content, so be careful.

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:

Command	Description
\$set	Sets the given field with the given value
\$unset	Removes the field
\$inc	Adds the given field by the given number
\$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

References

As we mentioned previously, Mongo isn't built to perform joins. Because of its distributed nature, joins are pretty inefficient operations. Still, it's sometimes useful for documents to reference each other. In these cases, the Mongo you can use a construct like

```
{ $ref : "collection_name", $id: "reference_id" }.
```

For example, we can update the *towns* collection to contain a reference to a document in *countries*

```
db.towns.update(  
  { _id : ObjectId("56d34343084bfde6c9242288") },  
  { $set : { country: { $ref: "countries", $id: "us" } } }  
)
```

Now you can retrieve Portland from your towns collection.

```
var springfield = db.towns.findOne({ _id : ObjectId("56d34343084bfde6c9242288") })
```

Then, to retrieve the town's country, you can query the countries collection using the stored \$id:

```
db.countries.findOne({ _id: springfield.country.$id })
```

Better yet, in JavaScript, you can ask the town document the name of the collection stored in the fields reference.

```
db[ springfield.country.$ref ].findOne({ _id: springfield.country.$id })  
cls
```

The last two queries are equivalent; the second is just a bit more data-driven.

```
> db.towns.update( { _id : ObjectId("56d34343084bfde6c9242288") }, { $set : { co  
country: { $ref: "countries", $id: "us" } } } )  
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 0 })  
> var springfield = db.towns.findOne( { _id : ObjectId("56d34343084bfde6c9242288"  
> })  
> db.countries.findOne( { _id: springfield.country.$id } )  
<  
  {  
    "_id" : "us",  
    "name" : "United States",  
    "exports" : {  
      "foods" : [  
        {  
          "name" : "bacon",  
          "tasty" : true  
        },  
        {  
          "name" : "burgers"  
        }  
      ]  
    }  
  }  
>  
>  
> db[ springfield.country.$ref ].findOne( { _id: springfield.country.$id } )  
<  
  {  
    "_id" : "us",  
    "name" : "United States",  
    "exports" : {  
      "foods" : [  
        {  
          "name" : "bacon",  
          "tasty" : true  
        },  
        {  
          "name" : "burgers"  
        }  
      ]  
    }  
  }  
>  
> _
```

Deleting

Removing documents from a collection is simple. All we have to do is replace the *find()* function with a call to *remove()*. All matched criteria will be removed. It's important to note that the entire matching document will be removed, not just a matching element or a matching subdocument.

It is generally recommended that you running *find()* to verify your criteria before running *remove()*. Mongo will not pause or give you time to correct an error before running your operation.

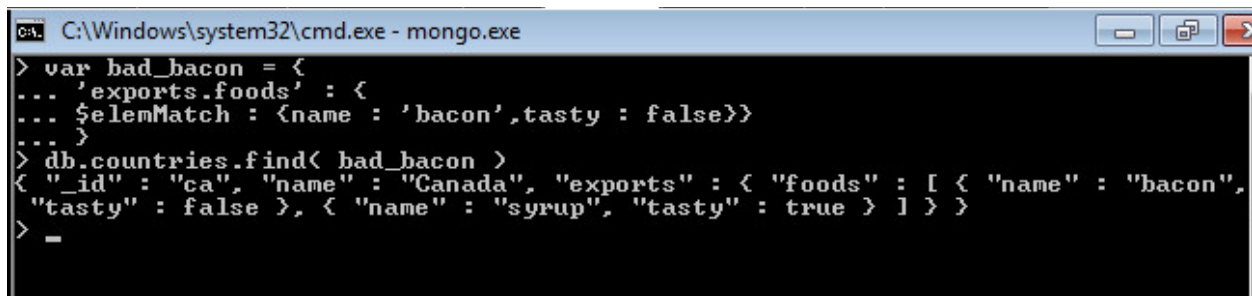
Let's remove all countries that export bacon that isn't tasty:

First create our javascript *bad_bacon* variable:

```
var bad_bacon = {  
'exports.foods' : {  
$elemMatch : {name : 'bacon',tasty : false}}  
}
```

Now find the matching documents:

```
db.countries.find( bad_bacon )
```



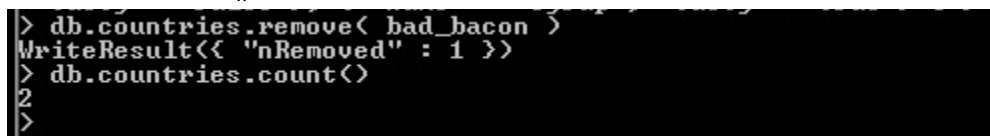
```
C:\Windows\system32\cmd.exe - mongo.exe  
> var bad_bacon = {  
... 'exports.foods' : {  
... $elemMatch : {name : 'bacon',tasty : false}}  
... }  
> db.countries.find( bad_bacon )  
{ "_id" : "ca", "name" : "Canada", "exports" : { "foods" : [ { "name" : "bacon",  
"tasty" : false }, { "name" : "syrup", "tasty" : true } ] } }  
> _
```

The *find()* returns the document for "Canada". This is the only country we had entered who exported bacon that was not "tasty". Now let's remove it:

```
db.countries.remove( bad_bacon )
```

We can verify that it is gone by checking our document count:

```
db.countries.count()
```



```
> db.countries.remove( bad_bacon )  
WriteResult({ "nRemoved" : 1 })  
> db.countries.count()  
2  
>
```

The count returned 2, so we know our document was removed.

Reading by Code

The last interesting Mongo query option we should look at is: query-by-code. You can request that MongoDB run a decision function across your documents. While this is very powerful, it should only be used in practice as a last resort. These queries run quite slowly, you can't index them, and Mongo can't optimize them. (But sometimes it's hard to create a solution without the power of custom code.)

So let's query for a city with a population between 6,000 and 600,000 people.

```
db.towns.find( function() {  
  return this.population > 6000 && this.population < 600000;  
})
```

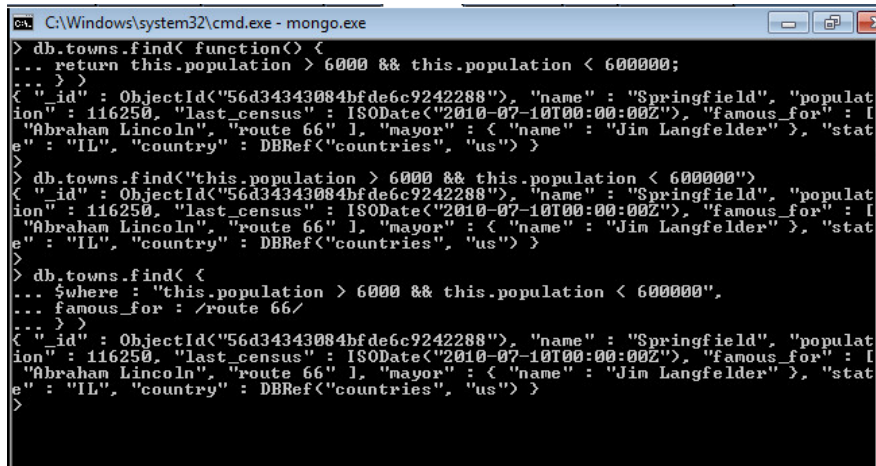
Mongo even has a shortcut for simple decision functions, use the object instance reference, *this*:

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

You can run custom code with other criteria using the *\$where* clause. In this example, the query also filters for towns famous for "route 66".

```
db.towns.find( {  
  $where : "this.population > 6000 && this.population < 600000",  
  famous_for : /route 66/ } )
```

All three query-by-code examples will return the correct document for Springfield, IL:



The screenshot shows a terminal window titled "C:\Windows\system32\cmd.exe - mongo.exe". It displays three MongoDB queries and their results. The first query uses a function to filter by population, the second uses a string query with 'this', and the third uses the '\$where' clause along with a regular expression for 'famous_for'. All three queries return the same document for Springfield, IL.

```
> db.towns.find( function() {  
...  return this.population > 6000 && this.population < 600000;  
...  } )  
{  
  "_id" : ObjectId("56d34343084bfde6c9242288"), "name" : "Springfield", "populat  
ion" : 116250, "last_census" : ISODate("2010-07-10T00:00:00Z"), "famous_for" : [ "Abraham Lincoln", "route 66" ], "mayor" : { "name" : "Jim Langfelder" }, "stat  
e" : "IL", "country" : DBRef("countries", "us") }  
>  
> db.towns.find("this.population > 6000 && this.population < 600000")  
{  
  "_id" : ObjectId("56d34343084bfde6c9242288"), "name" : "Springfield", "populat  
ion" : 116250, "last_census" : ISODate("2010-07-10T00:00:00Z"), "famous_for" : [ "Abraham Lincoln", "route 66" ], "mayor" : { "name" : "Jim Langfelder" }, "stat  
e" : "IL", "country" : DBRef("countries", "us") }  
>  
> db.towns.find( {  
...  $where : "this.population > 6000 && this.population < 600000",  
...  famous_for : /route 66/  
...  } )  
{  
  "_id" : ObjectId("56d34343084bfde6c9242288"), "name" : "Springfield", "populat  
ion" : 116250, "last_census" : ISODate("2010-07-10T00:00:00Z"), "famous_for" : [ "Abraham Lincoln", "route 66" ], "mayor" : { "name" : "Jim Langfelder" }, "stat  
e" : "IL", "country" : DBRef("countries", "us") }  
>
```

A word of warning: Mongo will brute-force run this function against every document in a collection—even if the given field doesn't exist. Sometimes there is no way of determining if the query code failed because there were no matches or if it failed because the JavaScript didn't execute.

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.

CONTINUE TO NEXT PAGE FOR WORK TO SUBMIT

Work to Submit

Lab 2 Part 2 (lab2b) must be successfully graded before the lab3 folder will be added in your repo.

Be sure to do a 'git pull' to sync your repo.

Create the following document database from any blogger website you frequent (or a news article site). Your submission will be a script file named lab3\lab3a.js with the MongoDB commands to do the following:

1. Create a new database named "blogger":
2. Create 3 users with `_id` = "5bb26043708926e438db6cad", "5bb26043708926e438db6cae", "5bb26043708926e438db6caf" The users collection should contain the fields name and email. For the field `_id`, use `ObjectId` instead of `String`. Ex: `"_id" : ObjectId("5bb26043708926e438db6cad")`
 - a. List the contents of the users collection in pretty form
 - b. Search for user 5bb26043708926e438db6cad
3. Create 3 blogs with fields: title, body, slug, author, comments (array with objects containing `user_id`, `comment`, `approved`, `created_at`), and category (array with objects containing name)
 - a. The `user_id` and `author` fields should be one of the 3 users `_id` found above
 - b. One of the posts should contain the word "framework" in the body
 - c. Get all comments by User 5bb26043708926e438db6caf across all posts displaying only the title and slug
4. Select a blog via a case-insensitive regular expression containing the word Framework in the body displaying only the title and body