Querying with MongoDB

In the last week, we have learned how to create basic databases, collection and documents. Let's explore more querying examples in MongoDB.

Creating a database

We can switch to a database in Mongo with the use command.

```
use petshop
```

This will switch to writing to the petshop database. It doesn't matter if the database doesn't exist yet. It will be brought into existence when you first write a document to it.

You can find which database you are using simply by typing db. You can drop the current database and everything in it using **db.dropDatabase**.

```
db
> petshop
db.dropDatabase()
```

Collections

Collections are sets of (usually) related documents. Your database can have as many collections as you like.

Because Mongo has no joins, a Mongo query can pull data from only one collection at a time. You will need to take this into account when deciding how to arrange your data.

You can create a collection using the createCollection command.

```
> use petshop
> db.createCollection('mammals')
```

Collections will also be created automatically. If you write a document to a collection that doesn't exist that collection will be brought into being for you.

View your databases and collections using the show command, like this:

```
show dbs
show collections
```

Documents

Documents are JSON objects that live inside a collection. They can be any valid JSON format, with the caveat that they can't contain functions.

The size limit for a document is **16Mb** which is more than ample for most use cases.

Creating a document

You can create a document by inserting it into a collection

```
db.mammals.insert({name: "Polar Bear"})
db.mammals.insert({name: "Star Nosed Mole"})
```

Finding a document

You can find a document or documents matching a particular pattern using the find function.

If you want to find all the mammals in the mammals collection, you can do this easily.

```
db.mammals.find()
```

Finding documents

Mongo comes with a set of convenience functions for performing common operations. Find is one such function. It allows us to find documents by providing a partial match, or an expression.

Uses
You **can** use find to:

- Find a document by id
- Find a user by email
- Find a list of all users with the same first name
- Find all cats who are more than 12 years old

Limitations

You **can't** use find to chain complex operators. You can do a few simple things like counting, but if you want real power you need the aggregate pipeline, which is actually not at all scary and is quite easy to use.

The Aggregate pipeline allows us to chain operations together and pipe a set of documents from one operation to the next.

Using find
You can use find with no arguments to list documents in a collection.

```
db.entrycodes.find()
```

This will list all of the codes, 20 at a time. (Make sure you have some data in the entrycodes collection.)

Or
Take your instructor help to load the open source datasets

https://github.com/ozlerhakan/mongodb-json-files/tree/master/datasets

You can get the same result by passing an empty object, like so:

```
db.entrycodes.find({})
```

Finding by ID

Assuming you know the object ID of a document. You can pull that document by id like so:

```
db.entrycodes.find(ObjectId("557afc91c0b20703009f7edf"))
```

- The _id field of any collection is automatically indexed.
- IDs are 12 byte BSON objects, not Strings which is why we need the ObectId function. If you want to read more on ObjectId, you can do so here.
- http://docs.mongodb.org/manual/reference/object-id/

Finding by partial match

Say you have a list of users and you want to find by name, you might do:

```
db.people.find({name: "dave"})
```

You can match on more than one field:

```
db.people.find({
  name: "dave",
  email: "davey@aol.com"
})
```

You can match on numbers:

```
db.people.find({
  name: "dave",
  age: 69,
  email: "davey@aol.com"
})
```

You also match using a regex (although be aware this is slow on large data sets):

```
db.people.find({
  name: /dave/
})
```

Finding with Expressions and comparison queries

We have seen how we can find elements by passing Mongo a partial match, like so:

```
db.people.find({name: 'Yolanda Sasquatch'})
```

We can also find using expressions. We define these using JSON, like so:

```
db.people.find({
  age: {
    $gt: 65
  }
})
```

We can use operators like this:

- $gt - Greater than
- $lt - Less than
- $exists - The field exists

Reference : http://docs.mongodb.org/manual/reference/operator/query/

$exists

We can use exists to filter on the existence of non-existence of a field. We might find all the breakfasts with eggs:

```
db.breakfast.find({
  eggs: {
    $exists: true
  }
})
```

$gt and $lt

We can use $gt and $lt to find documents that have fields which are greater than or less than a value:

```
db.breakfast.find({
  starRating: {
    $gt: 5
  }
})
```

Projection

Find takes a second parameter which allows you to whitelist fields to pass into the output document. We call this projection.

You can choose fields to pass though, like so:

```
{
  ham: 4,
  eggs: 2
}
{
  cheese: 6,
  lime: 0.5
}
db.breakfast.find({}, {
  eggs: true,
  lime: true
})
```

This will yield

```
{
  eggs: 2
},
{
  lime: 0.5
}
```

Excluding the id field

You will notice that the ID field is always passed through the project by default. This is often desirable, but you may wish to hide it, perhaps to conceal your implementation, or to keep your communication over the wire tight.

You can do this easily by passing _id: false:

```
db.breakfast.find({}, {
  eggs: true,
  lime: true,
  _id: false
})
```

Count

Count will convert our result set into a number. We can use it in two ways. We can either chain it:

```
db.people.find({sharks: 3}).count()
```

or we can use it in place of find:

```
db.people.count({sharks: 3})
```

To count the people who have exactly three sharks.

Don't confuse it with length(). Length will convert to an array, then count the length of that array. This is inefficient.

Limit and Skip

Limit will allow us to limit the results in the output set. Skip will allow us to offset the start. Between them they give us pagination.

For example

```
db.biscuits.find().limit(5)
```

will give us the first 5 biscuits. If we want the next 5 we can skip the first 5.

```
db.biscuits.find().limit(5).skip(5)
```

Sort

We can sort the results using the sort operator, like so:

```
db.spiders.find().sort({hairiness: 1})
```

This will sort the spiders in ascending order of hairiness. You can reverse the sort by passing -1.

```
db.spiders.find().sort({hairiness: -1})
```

This will get the most hairy spiders first.

We can sort by more than one field:

```
db.spiders.find().sort({
  hairiness: -1,
  scariness: -1
})
```

We might also sort by nested fields:

```
db.spiders.find().sort({
  'web.size': -1
})
```

will give the spiders with the largest webs.

Mongo CRUD Operations

CRUD is a basic function of any database. Crud stands for:

- Create

- Read
- Update
- Delete

The four basic things that any data store needs to give us.

Creating

We create using the insert command, like this:

```
db.people.insert({name: "Tony Stark", occupation: "Billionaire, playboy,
philantropist..."})
```

The JSON object will be created and saved.

Reading

We have many options for finding. We have already seen db.collection.find(). We can also use db.collection.findOne() which will return at most one result.

As we shall see soon, we also have the aggregate framework, and if we need maximum flexibility at the expense of a good deal of speed, we can also use map-reduce.

Updating

We save using the db.collection.save function. We pass the function a JSON object that contains the modified object to save, including the _id. The item will be found and updated.

```
var p = db.people.findOne()
p.age = 999
db.people.save(p)
```

(or)

We can also find and update in a single step using the update function:

```
db.people.update({name: 'dave'}, {name:'brunhilde'})
```

Deleting

We can remove people en-mass.

```
people = db.people.remove({name:'Dave'})
```

Reference https://www.tutorialspoint.com/mongodb/index.htm

The Mongo Aggregation Framework

The Mongo Aggregation framework gives you a document query pipeline. You can pipe a collection into the top and transform it though a series of operations, eventually popping a result out the bottom (snigger).

For example, you might take a result set, filter it, group by a particular field, then sum values in a particular group. You could find the total population of Iowa given an array of postcodes. You could find all the coupons that were used on Monday, and then count them.

We can compose a pipeline as a set of JSON objects, then run the pipeline on a collection.

Empty pipeline

If you provide an empty pipeline, Mongo will return all the results in the collection:

```
db.entrycodes.aggregate()
```

Filtering the pipeline with $match

We can use the aggregation pipeline to filter a result set. This is more or less analogous to find, and is probably the most common thing we want to do.

Say we want to list only people who have cats (where cat is a sub-document), we would probably do something like this this:

```
db.people.find({
  cat:{
    $exists: true
  }
})
```

We can get the same result in the aggregation framework using $match, like so:

```
db.people.aggregate({
  '$match' : {
    cat:{
      $exists: true
    }
  }
})
```

So why use aggregation over find? In this example they are the same, but the power comes when we start to chain additional functions as we shall soon see.

Modifying a stream with $project
The find function allowed us to do simple whitelist projection. The aggregate pipeline gives us many more options.

We can use $project to modify all the documents in the pipeline. We can remove elements, allow elements through, rename elements, and even create brand new elements based on expressions.

Say we have a set of voucher codes, like this:

```
{
  "firstName" : "Dave",
  "lastName" : "Jones",
  "code" : "74wy zphz",
  "usedAt" : ISODate("2015-06-13T17:47:20.423Z"),
  "email" : "123@gmail.com"
},
{
  "firstName" : "Stuart",
  "lastName" : "Hat",
  "code" : "7uwz e3cw",
  "usedAt" : ISODate("2015-06-13T17:47:50.489Z"),
  "email" : "456@gmail.com"
}
...
```

We can use **project** to restrict the fields that are passed through. We pick the fields we like and set true to pass them through unchanged.

```
db.entrycodes.aggregate(
  {
    '$project' : {
      email: true,
      code: true
    }
  }
)
```

Removing the id

Remove the id field bay passing _id: false.

This will yield a set something like the following:

```
[
  {
    "code" : "7uwy zphz",
    "email" : "123@gmail.com"
  },
  {
    "code" : "7uwz eccw",
    "email" : "123@gmail.com"
  }
]
```

Renaming Fields

We can use project to rename fields if we want to. We use an expression: $lastName to pull out the value from the lastName field, and project it forward into the surname field.

```
db.entrycodes.aggregate(
  {
    '$project' : {
      surname: "$lastName"
    }
  }
)
```

This will yield something like the following:

```
[
  {
    surname : "Jones"
  },
  {
    surname : "Hat"
  }
  ...
]
```

Creating new fields with $project
We can use project to add new fields to our documents based on expressions.

Say we had a list of people, like so:

```
{firstName: "Chinstrap", surname: "McGee"}
{firstName: "Bootle", surname: "Cheeserafter"}
{firstName: "Mangstrang", surname: "Fringlehoffen"}
```

We can use project to compose a name field, like so:

```
db.entrycodes.aggregate(
  {
    $project: {
      name: {$concat: ['$firstName', ' ', '$surname']}
    }
  }
)
```

This will give us results like this:

```
{ "name" : "Dave Smith" },
{ "name" : "Mike Moo" }
```

Mongo DB Relationships

https://docs.mongodb.com/manual/tutorial/model-embedded-one-to-one-relationships-between-documents/

https://docs.mongodb.com/manual/tutorial/model-embedded-one-to-many-relationships-between-documents/

https://docs.mongodb.com/manual/tutorial/model-referenced-one-to-many-relationships-between-documents/

Import and Export a MongoDB Database

Sample Datasets
https://github.com/ozlerhakan/mongodb-json-files/tree/master/datasets

Importing Information Into MongoDB

To learn how importing information into MongoDB works let's use a popular sample MongoDB database about restaurants. It's in .json format and can be downloaded using wget like this:

```
wget https://raw.githubusercontent.com/mongodb/docs-assets/primer-dataset/primer-dataset.json
```

Once the download completes you should have a file called primer-dataset.json (12 MB size) in the current directory. Let's import the data from this file into a new database called newdb and into a collection called restaurants. For importing we'll use the command mongoimport like this:

```
sudo mongoimport --db newdb --collection restaurants --file primer-dataset.json
```

The result should look like this:

Output of mongoimport

```
2016-01-17T14:27:04.806-0500    connected to: localhost
2016-01-17T14:27:07.315-0500    imported 25359 documents
```

As the above command shows, 25359 documents have been imported. Because we didn't have a database called newdb, MongoDB created it automatically.

Exporting Information From MongoDB

As we have previously mentioned, by exporting MongoDB information you can acquire a human readable text file with your data. By default, information is exported in json format but you can also export to csv (comma separated value).

To export information from MongoDB, use the command mongoexport. It allows you to export a very fine-grained export so that you can specify a database, a collection, a field, and even use a query for the export.

A simple `mongoexport` example would be to export the restaurants collection from the `newdb` database which we have previously imported. It can be done like this:

```
sudo mongoexport --db newdb -c restaurants --out newdbexport.json
```

In the above command, we use --db to specify the database, -c for the collection and --out for the file in which the data will be saved.

The output of a successful `mongoexport` should look like this:

Output of mongoexport

```
2016-01-20T03:39:00.143-0500    connected to: localhost
2016-01-20T03:39:03.145-0500    exported 25359 records
```

The above output shows that 25359 documents have been exported— the same number as of the imported ones.

In some cases you might need to export only a part of your collection. Considering the structure and content of the restaurants json file, let's export all the restaurants which satisfy the criteria to be situated in the Bronx borough and to have Chinese cuisine. If we want to get this information directly while connected to MongoDB, connect to the database again:

```
$ sudo mongo newdb
```

Then, use this query:

```
> db.restaurants.find( { borough: "Bronx", cuisine: "Chinese" } )
```

The results are displayed to the terminal. To exit the MongoDB prompt, type `exit` at the prompt:

```
> exit
```

If you want to export the data from a sudo command line instead of while connected to the database, make the previous query part of the `mongoexport` command by specifying it for the -q argument like this:

```
$ sudo mongoexport --db newdb -c restaurants -q "{ borough: 'Bronx', cuisine: 'Chinese'
}" --out Bronx_Chinese_retaurants.json
```

Note that we are using single quotes inside the double quotes for the query conditions. If you use double quotes or special characters like $ you will have to escape them with backslash (\) in the query.

If the export has been successful, the result should look like this:

Output of mongoexport

```
2016-01-20T04:16:28.381-0500    connected to: localhost
2016-01-20T04:16:28.461-0500    exported 323 records
```

The above shows that 323 records have been exported, and you can find them in the Bronx_Chinese_retaurants.json file which we have specified.

MongoDB Performance

Performance Tuning with Indexes in MongoDB

Indexes
Indexes support the efficient execution of queries in MongoDB. Without indexes, MongoDB must perform a collection scan, i.e. scan every document in a collection, to select those documents that match the query statement. If an appropriate index exists for a query, MongoDB can use the index to limit the number of documents it must inspect.

Indexes are special data structures that store a small portion of the collection's data set in an easy to traverse form. The index stores the value of a specific field or set of fields, ordered by the value of the field. The ordering of the index entries supports efficient equality matches and range-based query operations. In addition, MongoDB can return sorted results by using the ordering in the index.

(Improves the querying performance)
- Without indexing, it searches in all the documents to match(one by one)
- With Indexing, (Binary Tree Index)

Before Indexing perform the below query and analyse the stats

```
db.student_data.find({"name":"Mariela Sherer"}).explain("executionStats");
```

After Indexing perform the below query and analyse the stats

```
db.student_data.createIndex({"name":1});
db.student_data.find({"name":"Mariela Sherer"}).explain("executionStats");
```

Make sure you do indexing from now on to improve querying performance.

Working with Geospatial Data and locations

https://docs.mongodb.com/manual/geospatial-queries/

https://docs.mongodb.com/manual/tutorial/geospatial-tutorial/

https://docs.mongodb.com/manual/reference/geojson/