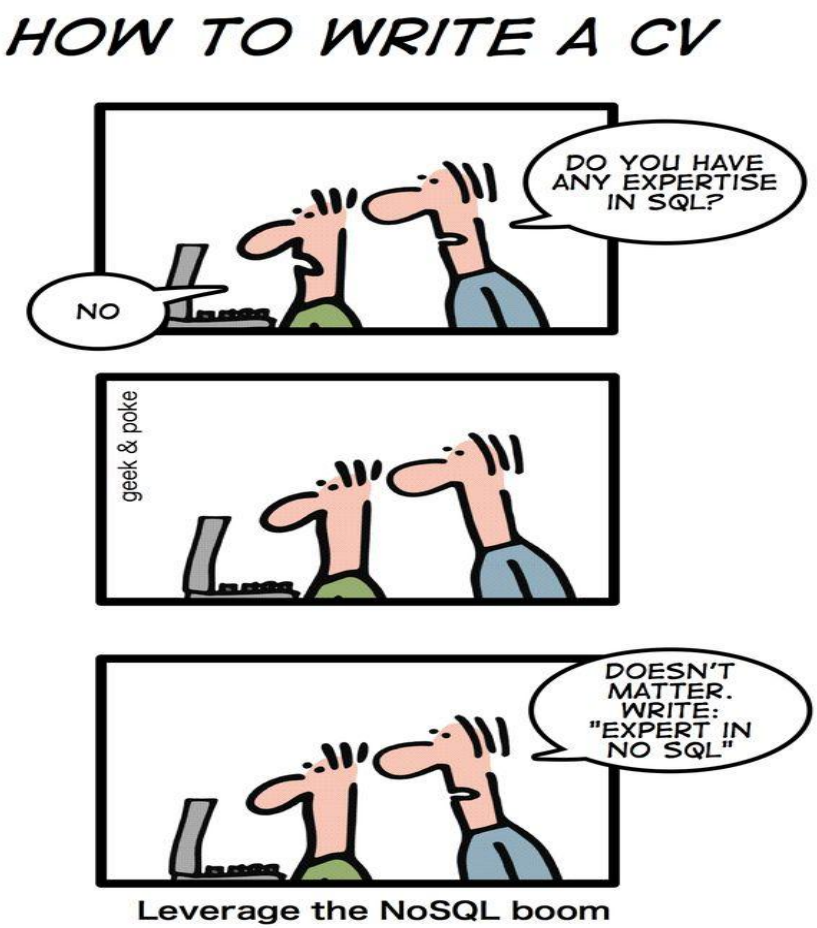
NYC DATA SCIENCE ACADEMY

Introduction to MongoDB

NYC Data Science Academy

Why NoSQL database?



http://geekandpoke.typepad.com/geekandpoke/2011/01/nosql.html

Why NoSQL database?

- Relational Databases Management System(RDBMS) like MySQl is good for structured data.
- NoSQL database uses dynamic schema, meaning that you can create records without defining the structure, such as the fields or the types of their values. MongoDB is the most popular one.
- You can change the structure of the records(which is called document in MongoDB) simply by adding new fields or deleting existing ones.

Terms in MongoDB

- A document is the basic unit of data for MongoDB and is roughly equivalent to a row in a relational database management system.
- A collection can be thought of as a table with a dynamic schema.
- Every document has a special key, "_id", that is unique within a collection.

MySQL	**MongoDB**
Table	Collection
Row	Document
Column	Field

Documents

- Document is an ordered set of keys with associated values.
- Most programming languages have a data structure that is a natural fit, such as a map, hash, or dictionary.

```
{'greeting': 'Hello world!'}

{'even': [2, 4, 6, 8]}

{'geolocation': {'long':74.3421, 'lat':40.75}}
```

Collection

- A collection is a group of documents. If a document is the MongoDB analog of a row in a relational database, then a collection can be thought of as the analog to a table.
- Collections have dynamic schema. The documents within a single collection can have any number of different “shapes”. For example, all of the following documents could be stored in a single collection.

```
{'greeting': 'Hello world!'}

{'even': [2, 4, 6, 8]}

{'geolocation': {'long':74.3421, 'lat':40.75}}
```

Collection

- Wait! Then why do we need separate collections at all?
- Keeping different kinds of documents in the same collection can be nightmare for developers. Developers need to make sure that each query is only returning documents of a certain type.
- For example, if you want to design a blog website and using MongoDB store all the information about blog posts and authors. How would you design your database?

Collection

- One document containing all the informations of blog posts and users and saved in one collection.
- Two documents for blog posts and users but saved in the same collection.
- Two documents for blog posts and users but saved in two collections.

Collection

- For the blog post and author collections, we might have the document structure like followings:

```
{
     'title': "Palming off the Earth" ,
     'author': "Wanda Wang",
     'comment': [
                    {
                         "name": "bob",
                         "email": bob@example.com
                         "content": "great post"
                    }
                ]
}
```

```
{
     'first name': "Wanda",
     'last name': "Wang",
     'email': "wanda@example.com"
}
```

Exercise 1

- Open your terminal and type `mongod` will start a MongoDB instance on your local machine.
  - Note: you might need to use `sudo mongod` if you get the permission denied error.
- Open a new tab on your terminal and type `mongo` will start the MongoDB shell. It will connect to the MongoDB instance you just started. Type `db` and check the output.
- Note: you **must** use **control + c** to terminate the mongoDB instance, otherwise it will throw an error when you start it next time.

Import files

- We can easily import files to mongoDB using the `mongoimport` command. It will automatically create the database and collection if they don't exist.
  - --db: name of database
  - --collection: name of collection
  - --file: name of the data file

```
$ mongoimport --db test --collection restaurants --file
restaurant.json
```

- For csv files, we need to pass an additional `--headerline` parameter to the `mongoimport` command.

```
$ mongoimport --db test --collection adults --type csv --file
adult.csv --headerline
```

Check database schema

- Check all the databases available on the machine.

```
> show databases
```

- Change the database

```
> use test
```

- Check all the collections available in the database.

```
> show collections
```

Query: find

- The find method is used to perform queries in MongoDB. Querying returns a subset of documents in a collection, from no document at all to the entire collection.
- Which documents get returned is determined by the first argument to find, which is a document specifying the query criteria.

```
> db.restaurants.find()
```

- Mongodb will only print the first 20 result on the terminal. Type `it` (`iterate`) to check the next 20 result. Or you can check the very first result by tying:

```
> db.restaurants.findOne()
```

ObjectIds

- Every document stored in MongoDB must have an "_id" key and it should be unique to all the other documents in the database.
- ObjectId is the default type of "_id" and it uses 12 bytes of storage, which gives them a string representation that is 24 hexadecimal digits: 2 digits for each byte.

  - **1-4:** 4-byte value representing the seconds since the Unix epoch,
  - **5-7:** 3-byte machine identifier,
  - **8-9:** 2-byte process id, and
  - **10-12:** 3-byte counter, starting with a random value.

Query: find

- You can further limit and format your result using `limit` and `pretty`.

```
> db.restaurants.find().limit(10).pretty()
```

- The previous query is the same as passing an empty document to the find function, which will match all the documents in the collection.

```
> db.restaurants.find({})
```

- We can check the number of observations using the count function.

```
> db.restaurants.find({}).count()
25359
```

Query: projection

- Sometimes we don't need all the key/value pairs in a document returned. If this is the case, you can pass a second argument to `find(or findOne)` specifying the keys you want.
- For example, if we only want the address and name of a restaurant, you can use the following query:

```
> db.restaurants.findOne({}, {"address":1, "name":1})
```

Query: AND Conditions

- When we start adding key/value pairs to the query document, we begin restricting our search. For example, to find all the restaurants in Queens, we can add that key/value pair to the query document.

```
> db.restaurants.find({"borough": "Queens"})
```

- Multiple conditions can be strung together by adding more key/value pairs to the query documents, which interpreted as “condition 1 AND condition2 AND ….condition N.” For instance:

```
> db.restaurants.find({"borough": "Manhattan", "cuisine":
"Italian"})
```

Query: OR Conditions

- There are two ways to do an OR query in MongoDB. “$in” can be used to query for a variety of values for a single key. “$or” is more general; it can be used to query for any of the given values across multiple keys.
- For example, we want to find restaurants either in Manhattan or in Queens, we might use the following query.

```
> db.restaurants.find({"borough": {"$in": ["Manhattan", "Queens"]}})
```

- Or we just want an Italian restaurant no matter where it is:

```
> db.restaurants.find({"$or":[{"borough": {"$in": ["Manhattan",
"Queens"]}}, {"cuisine":"Italian"}]})
```

Query Conditionals

- Queries can go beyond the exact matching described in the previous section; they can match more complex criteria, such as ranges, negation.
- “$lt”, “$lte”, “$gt” and “$gte” are all comparison operators, corresponding to <, <=, > and >=, respectively. They can be combined to look for a range of values.
- For example, to look for people who are between 18 and 30 years old from the adults collection, we can type the following query:

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

Exercise 2

- Use the adults collection and answer the following questions.
  - How many documents in class ‘>50K' where the age is less than 36.78?
  - What's the ratio of number of '<=50K' / number of '>50K' in column class?

Query: arrays

- Querying for elements of an array is designed to behave the way querying for scalars does. For example, we can insert fruit arrays into the food collection.

```
> db.food.insert({"_id" : 1, "fruit" : ["apple", "banana",
"peach"]})
> db.food.insert({"_id" : 2, "fruit" : ["apple", "coconut",
"orange"]})
> db.food.insert({"_id" : 3, "fruit" : ["cherry", "banana",
"apple"]})
```

```
> db.food.find({"fruit": "coconut"})
{ "_id" : 2, "fruit" : [ "apple", "coconut", "orange" ] }
```

Query: arrays

- We can query for it in much the same way as we would if we had a document that looked like the (illegal) document: {“fruit”: “apple”, “fruit”: “banana”, “fruit”: “peach”}
- You can also query by exact match using the entire array. However, exact match a document if any elements are missing or superfluous. For example the following two queries won’t match anything.

```
> db.food.find({"fruit": ["apple", "banana"]})
```

```
> db.food.find({"fruit": ["banana", "apple", "peach"]})
```

Query: arrays

- If you need to match arrays by more than one element, you can use "$all" operator. This allows you to match a list of elements.

```
> db.food.find({"fruit": {"$all": ["apple", "banana"]}})
```

- Note: This time the order doesn't matter here.
- A useful conditional querying arrays is "$size", which allows you to query for arrays of a given size. Here is an example

```
> db.food.find({"fruit": {"$size":3}})
```

Query: embedded documents

- Query getting a little bit tricky when dealing with embedded documents. For example the grades field in the restaurants collection.

```
> db.restaurants.findOne({}, {"grades":1})
```

- It is an array but each element in the array is another document that has key/value pairs. According to what we have learned above, we might come up with the following query if we want to get all the restaurants that have grade A in the grades array.

```
> db.restaurants.find({"grades":{"grade": "A"}})
```

Query: embedded documents

- The reason is query embedded documents has to be exact match, which means we need to pass all the fields into the query document.

```
> db.restaurants.find({"grades":{"date" : ISODate("2013-03-
02T00:00:00Z"), "grade" : "A", "score" : 7}})
```

- A tricky solution would be using the dot notation in MongoDB. For example, if we want to find out how many restaurants in Brooklyn that has grade A in the grades field, we can use the following query:

```
> db.restaurants.find({"borough": "Brooklyn", "grades.grade":"
A"}).count()
5610
```

Update: document

- Once a document is stored in the database, it can be changed using the update method. Update takes two parameters: a query document, which locates documents to update, and a modifier document, which describes the changes to make to the documents found.
- For example, if I want to change the “coconut” in the food collection to “watermelon”, I could use the following query:

```
> db.food.update({"fruit": "coconut"}, {"fruit":
["apple", "watermelon", "orange"]})
```

Update: field

- But that is not convenient at all. What if you just want to update one field in the document? Then you can use the $set operator.

```
> db.restaurants.update({"cuisine": "Chinese"}, {"$set":
{"cuisine": "American Chinese"}})
```

- Note: Updates, by default, update only the first document found that matches the criteria. If there are more matching documents, they will remain unchanged. To modify all of the documents matching the criteria, you can pass true to the multi parameter.

```
> db.restaurants.update({"cuisine": "Chinese"}, {"$set":
{"cuisine": "American Chinese"}}, {"multi": true})
```

Remove

- To remove the whole collection, you can simply type:

```
> db.restaurants.drop()
```

- To remove documents that match some specific condition, you need to pass your document criteria to the remove function.
- For example, we want to remove all the documents in the adults collection that have age less than 30, we can use the following query:

```
> db.adults.remove({"age": {"$lt": 30}})
WriteResult({ "nRemoved" : 9711 })
```