COMP-3704 / 4704 NoSQL Databases

Lecture 07
MongoDB Intro and CRUD Operations

Daniel Pittman, Ph.D., CISSP
4/23/2019

MongoDB - Flexible and Hu(mongo)us

- To begin our coverage of MongoDB let's review the book's introduction:

  MongoDB is in many ways like a power drill. Your ability to complete a task is framed largely by the components you choose to use (from drill bits of varying size to sander adapters). MongoDB's strength lies in its versatility, power, ease of use, and ability to handle jobs both large and small. Although it's a much newer invention than the hammer, it is a tool that builders reach for more and more often.

MongoDB - Flexible and Hu(mongo)us

- MongoDB (Mongo) was first released in 2009, and is one of the most heavily used NoSQL databases in existence today
- It is designed to scale (its name was derived from hu**mongo**us)
  - Two of the main design goals of Mongo were performance and ease of data access
- MongoDB is a **document database**, allowing for storage of objects nested to whichever depth you'd like
  - You can also query on that nested data in an ad-hoc fashion!
- MongoDB does not enforce a schema
  - Documents can contain fields or types unique to that document (even though you most likely will not use it that way!)
- Mongo is a great choice for anything from a small local project to a huge production deployment
  - Foursquare, Comcast, Adobe, and CERN (Large Hadron Collider), to name a few, all use MongoDB!

MongoDB Overview

- Mongo is a **JSON document database**
  - Data is stored in a binary form of JSON known as BSON
- Each document stored in MongoDB gets assigned a unique identifier, **_id**
- Documents can contain attributes that are **subdocuments**
  - Instead of creating separate tables and joining them, you store all of the data together!

```
{
  _id : ObjectId("52279effc62ca8b0c1000007")
  name: 'Starcups',
  facilities: ['Hot drinks', 'Food', 'Premium wifi'],
  openingTimes: [{
      days: 'Monday - Friday',
      opening: '7:00am',
      closing: '7:00pm'
    },{
      days: 'Saturday',
      opening: '8:00am',
      closing: '5:00pm'
    }]
}
```

MongoDB Overview

- MongoDB is the sweet spot between the flexible queryability of a relational database, and the distributed nature of other NoSQL databases like HBase
  - Dwight Merriman, the MongoDB project founder, has said that Mongo is the database he wishes he'd had at DoubleClick where he was CTO and had to house large-scale data that could be queried in an ad-hoc manner
- If you compare MongoDB with a relational database such as PostgreSQL you will see a fundamental difference in how you approach using them
- With relational databases, it is assumed that you know what data you wish to store, but not necessarily how you will use it
  - Query flexibility is paid upfront on storage
- Document databases require you to make some assumptions on how you wish to use your data, but very few assumptions of exactly what kind of data you want to store

Today’s Agenda

- We will start off by installing MongoDB!
- We will then go over CRUD operations in Mongo and demonstrate how to perform nested queries
    - We will also look at table administration
- I will show you how to interact with Mongo using both the command-line MongoDB client utility and a GUI tool, Robo 3T

MongoDB Installation

- Download mongodb.zip from Canvas
- Unzip the file into your repository in the directory **docker/mongodb**
- Build and run the Docker container for MongoDB:
  - **cd docker/mongodb**
  - **docker-compose build**
  - **docker-compose up**
- In a new terminal, attach to the running Docker container:
  - **docker exec -it nosql-node /bin/bash**
- When you are done working with MongoDB for the day, stop the container by running **Ctrl + C** in the terminal you ran **docker-compose up**

MongoDB Command-Line

- Now that you have MongoDB running, let's create a database named **book**:
  - **mongo --host nosql-mongodb book**
- Interested in what you can do through the command-line utility? The **help** command is a great place to start
  - You can also run **show dbs** to see what databases are available in your local instance and then switch between them using the **use** command!
- Creating a **collection** (think table) in Mongo is as easy as adding a record to it!
  - Since there are no schemas in MongoDB there is nothing to define up front
    - Using a collection is enough!
  - In fact, the **book** database we "created" doesn't really exist until we add values to it!
- Let's insert some data into a collection named **towns**:

MongoDB CRUD

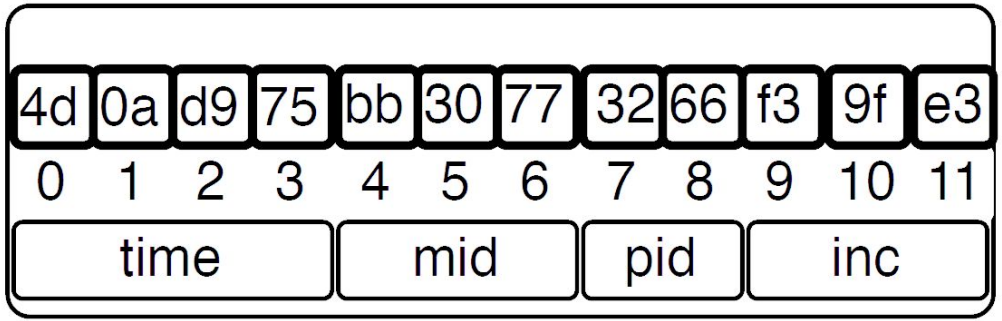
```
db.towns.insert({
    name: "New York",
    population: 22200000,
    lastCensus: ISODate("2016-07-01"),
    famousFor: [ "the MOMA", "food", "Derek Jeter" ],
    mayor : {
        name : "Bill de Blasio",
        party : "D"
    }
})
```

MongoDB CRUD

- As we've mentioned before, **documents** in MongoDB are JSON objects (stored as BSON)
- This means that when we add new documents to our database, we will be providing JSON to do so!
  - This will also be true for CouchDB later in the course
- Now that we've inserted some data we can verify the collection exists via the **show collections** command
  - The **towns** collection was created by storing data in it!
- We can list the contents of the collection via the **find** command:
  - **db.towns.find()**
- Unlike a relational database, Mongo does not support server-side joins
  - Thankfully, a single **find()** call will return a document **and** all of its nested content by default!

MongoDB CRUD

- Did you notice the _id field or type ObjectId in the documents returned from our **find()** query?
- A unique _id field is appended to every created document automatically in MongoDB
  - You can think of this field similar to how SERIAL worked in PostgreSQL
- The ObjectId is always 12 bytes, and is composed of a timestamp, client machine ID, client process ID, and a 3-byte increment counter, as illustrated below:



MongoDB and JavaScript

- Mongo uses JavaScript as its native language
- You will use JavaScript to do complex things like mapreduce queries, or simple things like asking for help
  - **db.help()**
  - **db.towns.help()**
  - **db** is a JavaScript object that contains information about the current database
    - **typeof db**
  - **db.x** is a nested JavaScript object that contains information about a collection named **x**
    - **typeof db.towns**
  - All commands are just JavaScript functions
    - **typeof db.towns.insert**
- If you want to inspect the source code for a function, call it without parameters or parentheses:
  - **db.towns.insert**

MongoDB and JavaScript

- Let’s insert some data! Since the Mongo shell supports JavaScript functions, we can take the function defined in **insertCity.js** and paste it right into the shell
- We can then call it to insert some data:

```
insertCity("Punxsutawney", 6200, '2016-01-31',
  ["Punxsutawney Phil"], { name : "Richard Alexander" })
insertCity("Portland", 582000, '2016-09-20',
  ["beer", "food", "Portlandia"], { name : "Ted Wheeler", party : "D" })
```

- We can confirm we now have three towns via **db.towns.find()**

MongoDB GUI

- In class our examples will be using either the Mongo shell or JavaScript functions
- There are great visual tools out there also, however, if that helps you!
- Want to see what's in your MongoDB instance in a GUI? Use Robo 3T!
  - https://robomongo.org/download
- It's a well written and feature-rich UI that I'm sure you'll like if you want to try one out
- I'll briefly demo it now!

Querying for Data in MongoDB

- Up until now we've been querying for data using the **find()** method with no arguments
- If you want to query for a specific object, you just need to set the **_id** property
  - **_id** is of type ObjectId, which means to query for it you must convert a string using the **ObjectId(str)** function

```
db.towns.find({ "_id" : ObjectId("59094288afbc9350ada6b807") })
```

- **find()** 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**), we pass in **name** with a value of 1 (or true)

```
db.towns.find({ _id : ObjectId("59094288afbc9350ada6b807") }, { name : 1 })
```

Querying for Data in MongoDB

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

```
db.towns.find({ _id : ObjectId("59094288afbc9350ada6b807") }, { name : 0 })
```

- As in PostgreSQL, MongoDB allows you to construct ad-hoc queries on the basis of field values, ranges, or a combination of criteria
- To find all towns that begin with the letter **P** and have 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 } },

  { _id: 0, name : 1, population : 1 })
```

Querying for Data in MongoDB

- Conditional operators in Mongo follow the format of **field: { $op : value }**, where **$op** is an operation (such as **$ne or $gt**)
  - Since we are working with JavaScript we can't use shorthand syntax to define criteria
    - No **field < value** like you might see in SQL (except in very simple cases)
- Even though the syntax is a little more verbose, JavaScript allows you to create complex query criteria as JavaScript objects!
- Here's a criteria where the population must be between 10,000 and 1 million people:

```
var population_range = { $lt: 1000000, $gt: 10000 }

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

Querying for Data in MongoDB

- We can also retrieve things like data ranges! Let's find all names with a **lastCensus** greater than or equal to June 1, 2016:

```
db.towns.find({ lastCensus : { $gte : ISODate('2016-06-01') } }, { _id : 0, name: 1 })
```

- Nested array data is easy to store and query for in Mongo! You can query by exact values:

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

- As well as partial values!

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

Querying for Data in MongoDB

- You can also query by all matching values:

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

- Or the lack of matching values:

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

Querying for Data in MongoDB

- But what if you want to query for data **inside** a nested object in a document?
  - This is the true power of Mongo!
- To query a subdocument, the field name in the query is a string separating nested layers with a dot
- For instance, let's find towns with mayors from the Democratic Party:

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

- Or mayors with no party affiliation:

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

Querying for Data in MongoDB

- That was fun, but what if you want to match **several fields** in a subdocument?
  - Bring in the $elemMatch directive!
- Let’s create a new collection to store **countries** and insert data with custom _id strings

```
db.countries.insert({

    _id : "us",

    name : "United States",

    exports : { foods : [ { name : "bacon", tasty : true }, { name : "burgers" } ] }

})
```

Querying for Data in MongoDB

```
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 }]
}})
```

Querying for Data in MongoDB

- Let’s query for the countries we just added:

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

- Now let’s find a country that only exports **tasty** bacon

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

- Oh no! Canada was returned because it exports both bacon and tasty syrup! This is where **$elemMatch** helps us

Querying for Data in MongoDB

- **$elemMatch** specified that if a document (or nested document) matches **all** of our criteria, then the document counts as a match:

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

Querying for Data in MongoDB

- **$elemMatch** can also utilize advanced operators
- Let’s 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 })
```

Boolean Operations

- 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 find no results:

```
db.countries.find({ _id : "mx", name : "United States" },{ _id : 1 })
```

- However, we can search for one or the other using the **$or** operator

```
db.countries.find({$or : [{ _id : "mx" }, { name : "United States" } ]}, { _id:1 })
```

MongoDB Operator Command List

Command	Description
$regex	Match by any PCRE-compliant regular expression string (or just use the // delimiters as shown earlier)
$ne	Not equal to
$lt	Less than
$lte	Less than or equal to
$gt	Greater than
$gte	Greater than or equal to
$exists	Check for the existence of a field
$all	Match all elements in an array
$in	Match any elements in an array
$nin	Does not match any elements in an array
$elemMatch	Match all fields in an array of nested documents
$or	or
$nor	Not or
$size	Match array of given size
$mod	Modulus
$type	Match if field is a given datatype
$not	Negate the given operator check

Updating Data

- Uh oh! We added New York and Punxsutawney, which are obvious enough, but what Portland did we add? Oregon? Maine? Texas?
  - Let's update our **towns** collection to add some U.S states!
- The **update(criteria, operation)** function requires two parameters:
  - A criteria query of the same type you would pass to the **find()** method
  - Either an object whose fields will replace the matched document(s), or a modifier operation
- Let's try updating Portland and using a modifier of **$set** to set the field **state** with the string **OR**

```
db.towns.update(

   { _id : ObjectId("5b3c1b91e9499b9be241d30c") },

   { $set : { "state" : "OR" } } );
```

Updating Data

- Why is the **$set** operation required? Well, Mongo doesn't think in terms of attributes
- There is a simple understanding of attributes for optimization reasons, but nothing about the interface of Mongo is **attribute**-oriented
  - Mongo is **document**-oriented
- That means that you would rarely ever want to perform an update operation as follows:

```
db.towns.update(

  { _id : ObjectId("4d0ada87bb30773266f39fe5") }, { state : "OR" } );
```

- The above command would replace the **entire** document with { state: "OR"} !

Updating Data

- Let’s verify our update using the **findOne** function:

```
db.towns.findOne({ _id : ObjectId("5b3c1b91e9499b9be241d30c") })
```

- There’s many more functions available to you than just **$set**!
    - **$inc**, for instance, is pretty useful when you need to increment a number:

```
db.towns.update(
  { _id : ObjectId("5b3c1b91e9499b9be241d30c") },
  { $inc : { population : 1000} }
)
```

Update Directive Summary

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 values from an array
$pullAll	Removes all matching values from an array