Introduction to MongoDb

Reading

- NoSQL Distilled[1] chapter 9
- MongoDb server docs[2]
- MongoDb Drivers for various languages[3]
- Mongo shell reference[4]

Reading questions

-

Notes

MongoDb is a premier document-based database with many features.

Setting up

You can run MongoDb in various ways:

- You can install[5] it locally to your computer.
- You can play around in an online shell[6].
- You can create an account with mlab[7].

We will follow this last method. You gain access to 500MB space for free. Here are the steps:

- Follow the above link for mlab[8] and sign up. Notice that you have an "account name", think of it as a group name, and it's separate from your username. For example your instructor used wranglingclass for the account name.
- You will also need to verify the email address. Check your email and click the link there.
- When you log in, you will be presented with your (empty) list of "deployments". We will create a new one by clicking on the "Create new" button.

[1] http://learning.acm.org/books/book_detail.cfm?id=2381014&type=safari
[2] https://docs.mongodb.com/manual/
[3] https://docs.mongodb.com/ecosystem/drivers
[4] https://docs.mongodb.com/manual/reference/method/
[5] https://docs.mongodb.com/manual/installation/
[6] https://www.tutorialspoint.com/codingground.htm
[7] https://mlab.com
[8] https://mlab.com

- You can choose between three “cloud providers”. They all offer a 500MB free “sandbox” option, choose whichever you like. Your instructor chose the Google Cloud Platform option.
- Make sure to choose “Single node” and Sandbox.
- Choose a database name. Your instructor named theirs “wrangling”.
- Click the Create button to finalize the setup.
- Congratulations! You now have your first cloud-storage-based database in place. Let’s learn how to access it.
- Click on the newly created deployment, and a new page will appear with information on how to access it.
- Click on the Users tab and add a new user for the database. You will use these credentials to remotely access the database later on.

Using the mongo shell

There are two standard ways to interact with your Mongodb database. In this section we’ll use the mongo shell.

- Open up a terminal.
- At the top of the mlab webpage with your deployment you’ll see a link like: mongo ds011168.mlab.com:11168/wranging –u <dbuser> –p <dbpassword>. Yours will have a different database number probably. You’ll need to paste that link and change the <dbuser> and <dbpassword> entries to your setup.
- You should now be presented with a welcome message. This is an interactive shell from where you can ask for details from the database. Start by typing:

  ```
  db
  ```

  which should show you the name of the database you are currently using. In general db is like an “object” that we will use to access the current database.
- We will now create a new **collection**. Collections are like tables in mysql. But since there is no strict form that documents in MongoDb need to follow, we don’t need to really specify anything about the collection. We just start using it. We’re going to call our collection “gpas”. Here is how we can add an entry to it:

  ```
  db.gpas.insert({ name: 'student0', gpa: _rand() * 4 })
  db.gpas.find()
  ```

  You should get back a response that shows you the new stored document, and it will also contain a “ObjectId”. This is an automatically generated by MongoDb. We could also manually generate it, but we will never have a need to do so.
- Let’s remove the entry we added:

  ```
  db.gpas.remove({ name: 'student0' })
  ```

- The MongoDb shell uses a mini programming language that looks a bit like Javascript, for those familiar with Javascript. For example we used _rand() to generate a random number. We will now insert a large number of values all at once. We first create them as a “Javascript” object:

  ```
  var d = []; for (var i = 0; i < 10000; i++) {
    d.push({ name: "student" + i, gpa: _rand()*4 })
  }
  db.gpas.insertMany(d)
  ```

- Let us now learn how to search for information in the documents. The main tool at our disposal is the find method. Its parameter is an object that describes the query. For example we can get the entries with a specific student name:

  ```
  db.gpas.find({ name: 'student100' })
  ```

  or we can perform more complex queries. For example, this asks for all entries whose gpa is over 3.95:

  ```
  db.gpas.find({ gpa: { $gt: 3.98 }})
  ```

  The shell will probably link only some of the results. We will discuss how to work with the result of a find, which is what is known as a *cursor*. In the meantime, if we only want to know how many results there are, we can use count:

  ```
  db.gpas.count({ gpa: { $gt: 3.98 }})
  ```

- Next we will do an update query: We will add an “atRisk” field to all students with a gpa of 2 or less. The query takes two parameters: The first specifies which entries to locate, the other specifies what changes to make.

  ```
  db.gpas.updateMany({ gpa: { $lte: 2 }}, { $set: { atRisk: true }})
  ```

  We’ll see that about half of the documents were updated. Now half the documents have this “atRisk” field, while the other half don’t have it at all.

- Let us now do a more complex query, that captures all students whose gpa is less than 2.5, and adjusts that gpa by up to plus/minus 1 point.

  ```
  db.gpas.updateMany({ gpa: { $lte: 2.5 }}, { $inc: { gpa: _rand() * 2 - 1 }})
  ```

  Now we will look for students who were at-risk but whose gpa is now over 2. We will then mark all those to no-longer at risk:

  ```
  db.gpas.count({ gpa: { $gt: 2 }, atRisk: true })
  db.gpas.updateMany({ gpa: { $gt: 2 }, atRisk: true }, { $set: { atRisk: false }})
  ```

  Now we want to search for all students that are not at risk. We cannot simply look for atRisk: false because this doesn’t include those students where there is no atRisk entry at all. We can do this in two ways:

  ```
  db.gpas.count({
    $or: [
      { atRisk: { $exists: false } },
      { atRisk: true }
    ]
  ```

```
})

db.gpas.count({
  atRisk: { $ne: false }
})
```

- We will now arbitrarily assign all students into four groups. This may take a while as it has to update each entry:

  ```
  db.gpas.find().forEach(function(doc) {
    db.gpas.update(doc, { $set: { group: Math.ceil(_rand() * 4) }})
  })
  ```

  This is also the first time where we say the use of a *cursor method*: The result of the find call is a "cursor", which is basically a fancy word for something that we can iterate over. We therefore perform a forEach on it. That takes an arbitrary function as input, and it executes that function for every result. Let's see how the above worked. Everyone should be more or less equally divided into four "groups", identified by the numbers 1 through 4:

  ```
  for (var i = 1; i < 5; i++) {
    print(db.gpas.count({ group: i }))
  }
  ```

Some practice problems:

- Find out how many students in each group are at risk.
- Find out how many students have a gpa of 2.0 or below and are marked as at-risk. These students were at risk before we changed the grades, and still are.
- Find out how many students have a gpa of 2.0 or below and are not marked as at-risk.
- Mark those students from the previous part as being at-risk.
- Remove from the collection all students with a gpa less than 1.
- Find all at-risk students and put them in their own "group", with number 5.

Aggregation

A powerful part of Mongo is the aggregation framework. This allows us to set up a **pipeline** of operations to be performed. The collection of documents passes through these *stages* and produces a final result. For example some possible stages[9] are:

- $project reshapes each document, for example by adding or removing fields.
- $match filters the list of documents based on a query. Only documents that match continue.
- $group groups the documents based on some criteria, and returns one document for each group.

[9] https://docs.mongodb.com/v3.2/reference/operator/aggregation/#aggregation-pipeline-operator-reference

These first three are the most powerful tools. But there are some more:

- $limit set a limit on the number of returned documents.
- $skip skips through a number of documents.
- $sample randomly selects some number of results.
- $sort sorts the results.
- $out can be used to immediately write the results on another collection. It must be the last step if used.

Let's look at some examples. We want to count the number of documents for each value of the "group" field. We could do something like this:

```
db.gpas.aggregate([
  { $project: { group: "$group", n: { $literal: 1 } }},
  { $group: { _id: "$group", count: { $sum: "$n" } }}
])
```

Here's what's happening:

- The $project step goes through each document, and keeps only the group information (and automatically the id) and also sets a new field called n with value 1. We will use those in the next step to count. The $group tells it to populate the new field called group with the value from the "group" field in the documents.
- The $group step takes these documents from the previous step, and groups them under a new _id given by the "group" field of the documents from the previous step. For all the documents with the same "new" _id, i.e. for all the documents of the same group, we perform the aggregation operator described in the { $sum: "$n" }, namely we add the values in the field called n, namely all those 1s. This ends up counting how many cases there are. And the result is stored in a field named count.

Let's try another example. We will find for each group: The minimum gpa, the maximum gpa, and the average gpa. We will also write the results to a collection called "averages".

```
db.gpas.aggregate([
  { $project: { group: "$group", gpa: true }},
  { $group: {
    _id: "$group",
    avg: { $avg: "$gpa" },
    min: { $min: "$gpa" },
    max: { $max: "$gpa" }
  }},
  { $out: "summaries" }
])
db.summaries.find()
```

Notice the phrase gpa: true here. It tells mongo to include the "gpa" field to the document before moving on to the next group.

TODO