EECS 484 Project #3: MongoDB

Due on March 24, 2022 by 11:55PM

Overview

In this project, we will use the same dataset as in Project 2 FakeBook and explore the capabilities of MongoDB (a NoSQL type DBMS). This spec will give you an introduction to MongoDB syntax.

There are two parts to the project. Part A of the project does not use MongoDB. You will be extracting data from tables in the Fakebook database and exporting a JSON file that contains information about Users. In Part B of the project, you will be importing the JSON file `output.json` (or a `sample.json` that we give you) into MongoDB to create a mongo collection called "users." You will then need to write 8 queries on the users collection. You can start on Part A right away without knowing anything about MongoDB, whereas Part B will require you to use MongoDB.

This project is to be done in teams of 2 students or individually. You may work with the same partner as project 1 and/or 2, or you may switch partners.

The autograder is located at autograder.io and you should follow the same instructions as before to form teams.

Do not make any submissions before joining your team! Once you click on “I’m working alone'', the autograder will not let you change team members. If you do need to make a correction, the teaching staff has the ability to modify teams.

The University of Michigan College of Engineering Honor Code strictly applies to this assignment, and we will be thoroughly checking to ensure that all submissions adhere to the Honor Code guidelines. Students whose submissions are found to be in violation of the Honor Code will be reported directly to the Honor Council. You may not share answers with other students actively enrolled in the course, nor may you consult with students who took the course in previous semesters. You are, however, allowed to discuss general approaches and class concepts with other students, and you are also permitted (and encouraged!) to post questions on Piazza.

1. The Environment

For **Part A**: Because some of our servers, including the Oracle server, are only accessible from the University network, you need to either work on this part in CAEN or be connected to the UM VPN if you want to work from your local machine.

For **Part B**, there are two ways of using MongoDB: either installing it on your local machine and creating a private MongoDB server, or using MongoDB via CAEN and interacting with the shared MongoDB server we have created.

To use MongoDB on your local machine: Refer to the MongoDB installation document for instructions. Once you have installed it, you should be able to execute 'mongod' (without a 'b') to start a private mongod server. To connect to your private server, you will generally type 'mongo' with the database name (which for this project, should be your uniqname) in a Terminal window:

```
$ mongo <uniqname>    # omit angle brackets when you actually run this
```

Note that the starter code `Makefile` does not work in a local environment unless properly modified. We may be unable to provide support in case you run into issues with your local `mongo` environment. Because of this, we recommend using MongoDB (v3.6) on CAEN.

To use MongoDB on CAEN: We have set up a MongoDB server on the host `eecs484.eecs.umich.edu`. To connect to this server, ssh into CAEN and type the following command (you should do this every time you start a new CAEN session):

```
$ module load mongodb
```

Then, you should update the uniqname and password fields of the `Makefile` from the starter files with the `mongo` Shell login you will create in part B. A few helpful commands from the `Makefile` are listed below:

```
$ make loginmongo      # mongo interactive mode
$ make setupsampledb   # drops all collections, then loads users collection using sample.json
$ make setupmydb       # drops all collections, then loads users collection using output.json
$ make mongotest       # runs test.js against all query*.js
```

2. Files Provided to You

On Canvas you will find a file called "`P3_starter_code.zip`", which contains the starter files for this project.

To complete **Part A: Export Oracle database to JSON,** start with the 2 Java files: `Main.java` and `GetData.java`. We have also provided 3 jar packages: `ojdbc6.jar`, `json_simple-1.1.jar`, and `json-20151123.jar`. Put all the above files in the same folder with `Makefile`.

To complete **Part B: MongoDB Queries**, set up your MongoDB database using the provided `Makefile`. Implement your MongoDB queries in the 8 JavaScript files `query[N].js`. The file `test.js` can be used to check partial correctness of your query results. To be clear, this file will not cover all Autograder tests.

3. Part A: Export Oracle database to JSON

1) Introduction to JSON

JSON (JavaScript Object Notation) is a way to represent data in a key-value format, much like an std::map in C++. JSON differs from maps in C++ in that the values do not have to be consistent in terms of data type. Here is an example of a JSON object (initialized in JavaScript):

```
var student1 = {"Name": "John Doe", "Age": 21, "Major": ["CS", "Math"]}
```

In student1, Name, Age and Major are the keys. Their corresponding value types are string, integer and array of strings. Note that JSON objects themselves can be values for other JSON objects. Below is an example of retrieving the value for a key:

```
student1[ "Name"];              // returns "John Doe"
```

With multiple JSON objects, we can create a JSON array in JavaScript:

```
var students = [
        {"Name": "John Doe", "Age": 21, "Major": ["CS", "Math"]},
        {"Name": "Richard Roe", "Age": 22, "Major": ["CS"]},
        {"Name": "Joe Public", "Age": 21, "Major": ["CE"]} ];
```

students [0] ["Name"]; // gives “John Doe”

2) Export to JSON

You will need to use JDBC from a Java program to query the USERS, FRIENDS, CITIES and other relevant tables in the Project 2 Fakebook Oracle database to export comprehensive information on each user. The results should be stored in a JSONArray called users_info, containing 800 JSONObjects for 800 users. It is suggested that you use multiple queries to retrieve all the information. Each JSONObject should include:

- user_id
- first_name
- last_name
- gender
- YOB
- MOB
- DOB
- hometown (JSONObject) that contains:
    - city
    - state
    - country
- current (JSONObject) that contains:
    - city
    - state
    - country
- friends (JSONArray) that contains: all of the user_ids of users who are friends with the current user, and has a **larger** user_id than the current user

Below is an example of one element of this JSON array.


```
{
   "MOB":2,
   "current":{
      "country":"Middle Earth",
      "city":"Pelargir",
      "state":"Gondor"
   },
   "hometown":{
      "country":"Middle Earth",
      "city":"Minas Tirith",
      "state":"Gondor"
   },
   "gender":"female",
   "user_id":582,
   "DOB":13,
   "last_name":"JOHNSON",
   "first_name":"Ornella",
   "YOB":41,
   "friends":[
      597,
      598,
      631,
      632,
      645,
      669,
      687,
      714,
      738,
      739,
      742,
      746,
      751,
      768,
      780,
      789
   ]
```


Note: It is possible that a user might have no hometown, current city, or list of friends. In this case, **put an empty JSONObject**({ }) as the value for the "hometown" or "current" key of that user or **an empty JSONArray([ ])** as the value for the "friends" key of that user. See `sample.json` for the correct output. More descriptions can be found in **subsection 4)**.

Here are the relevant files to get you started for this part of the project, which you can find in the Starter code that is provided to you.

1) Main.java

This file provides the main function for running Part A. You can use it to run your program, but you don't need to turn it in. Please only modify the `oracleUserName` and `password` static variables, replacing them with your own **Oracle** username and password.

```
13  public class Main {
14
15      static String dataType = "PUBLIC";
16      static String oracleUserName = "username"; //replace with your Oracle account name
17      static String password = "password"; //replace with your Oracle password
```

2) GetData.java

This file contains the function you need to implement for Part A. Query the USERS, FRIENDS, and CITIES tables to retrieve data from the Oracle Database and then convert them into JSON format. When Main.java is run, an output file named `output.json` should be generated in the folder where your Java files are. Your `output.json` is expected to contain the same data as in the provided `sample.json` file, but it can be in entirely different order from `sample.json`.

3) Makefile

To compile your program, execute `make compile` in the terminal.

To run your program, execute `make run` in the terminal.

4) sample.json

This file contains the JSON data from running our official implementation of GetData. Please DO NOT validate your output using `diff output.json sample.json` because JSON arrays are likely to come in entirely different order between any two runs. However, `output.json` and `sample.json` should contain the same elements in the JSON array. There are command line json processors that allow you to diff the contents properly.

Part A and Part B in this project do not depend on each other. You may set up your database for Part B using `sample.json` to test your MongoDB queries. The autograder testing on Part B **does not** rely on correct results from Part A.

If you'd like, you can submit the Java file from Part A to be graded without completing part B. See submission instructions at the end of part B.

4. Part B: MongoDB Queries

1) Introduction to MongoDB

MongoDB is a document-oriented database program. It is comparable to SQL Oracle Databases in many aspects. Each document in MongoDB is one JSON object, with key-value pairs of data, just like a tuple in SQL has fields of data; each collection in MongoDB is one JSON array of multiple documents, just like a table/relation in SQL has multiple tuples. Refer to the following table for concepts of document and collection in MongoDB, as well as queries to select certain columns and rows.

SQL	**MongoDB**
Tuple	Document. JSON object
Relation/Table	Collection. Initialized using a JSON array
SELECT * FROM users;	db.users.find();
SELECT * FROM users WHERE name = 'John' and age = 50;	db.users.find({name: 'John', age: 50});
SELECT user_id, addr FROM users WHERE name = 'John';	db.users.find({name: 'John'}, {user_id: 1, addr: 1, _id: 0});

https://docs.mongodb.com/manual/reference/sql-comparison/ is a document comparing MongoDB with SQL.

Additionally, https://docs.mongodb.com/manual/tutorial/getting-started/#getting-started contains a very basic MongoDB tutorial that is a good starting point to understand the basics of the system and become comfortable interacting with it.

2) Log in to MongoDB

To perform the MongoDB queries, you will need to login to the `mongo` Shell. There are 2 options here, depending on whether you are running the `mongo` shell on your personal computer or on CAEN and whether you are using a private mongod server or the shared server. Do whatever is convenient and works best for you.

Option 1: Login from your local machine to a private `mongo` server:

```
$ mongo <uniqname>
```

No hostname, userid, or password is required, so edit your `Makefile` such that these fields are removed from all of your make rules. "uniqname" is the name of the database that `mongo` will use for commands that follow.

Option 2: Login from your CAEN to the shared `mongo` server:

In your `Makefile`, set the uniqname and password fields equal to your uniqname (The default MongoDB password is your uniqname.) Then run the following commands in your terminal.

```
$ module load mongodb
$ make loginmongo
```

The `mongo` Shell will open up in your terminal.

You can update password with the following command in Mongo Shell:

```
> db.updateUser("<uniqname>", {pwd : "<newpassword>" })
```

The new password takes effect when you logout (Ctrl + D).

3) Import JSON to MongoDB

Open a terminal in the folder where you have `sample.json` (and/or `output.json`) and `Makefile`. Remember to `module load mongodb` each time you open a new terminal to perform any MongoDB operations. Modify your `Makefile` with your updated **MongoDB account** information and run `make setupsampledb` in the terminal to load the data from `sample.json`, or `make setupmydb` to load the data from your `output.json`.

Refer to the `Makefile` for the details on the actual commands. Please do not modify the `–collection users` field.

On success, you should have imported 800 user documents. See the Makefile contents on what this command does, in case you are using a private server.

To load data into your private database if you are using a private mongod server:

If you are using a private mongodb installation on your local machine for the project, omit the `--host...--username...--password <password>` portion from the `setupsampledb` and `setupmydb` commands.

4) Locally testing your queries using test.js

In Part B, you will implement 8 queries in the given JavaScript files. The file `test.js` contains one **simple** test on each of the queries. In `test.js`, you will need to set the "dbname" variable equal to your uniqname, as that will serve as the name of your database. You may use `test.js` to check **partial correctness** of your implementations. Note that an output saying "Local test passed! Partially correct." does not assure your queries will get a full score on the Autograder. Use `make mongotest` to feed the test file into MongoDB, or run the following command on a CAEN terminal (use your `mongo` Shell password):

```
$ mongo <uniqname> -u <uniqname> -p <password> --host
eecs484.eecs.umich.edu < test.js
```

Alternative: Again, if you are using a private installation of MongDB on your personal machine, and you have started a mongod server as instructed earlier, you can omit username, hostname, and password arguments and connect to your local MongDB server more simply as follows:

```
mongo <uniqname> < test.js
```

4) The eight queries you need to write

Query 1: Townspeople

In this query, we want to find all users whose hometown city is the specified 'city'. The result is to be returned as a JavaScript array of user_ids. The order of user_ids does not matter.

You may find `cursor.forEach()` helpful:
https://docs.mongodb.com/v3.0/reference/method/cursor.forEach/

Query 2: flat_users

In Part A, we created a `friends` array for every user using JDBC. Each user (JSON object) has `friends` (JSON array) that contains all the user_ids representing friends of the current user who have a larger user_id. In this query, we want to restore the friendship information into a friend pair table format.

Create a collection called `flat_users`. Documents in the collection follow this schema:
{"user_id": xxx, "friends": xxx}
For example, if we have the following user in the users collection:
{"user_id": 100, "first_name": "John" , … "friends": [120, 200, 300]}
The query would produce 3 documents (JSON objects) and **store them in the collection** `flat_users`:
{"user_id": 100, "friends": 120},
{"user_id": 100, "friends": 200},
{"user_id": 100, "friends": 300},

You do not need to return anything for this query.
Hint: You may find this link on MongoDB `$unwind` helpful:
https://docs.mongodb.org/manual/reference/operator/aggregation/unwind/
You may use `$project` and `$out` to create the collection, or you may insert tuples into `flat_users` iteratively.

Query 3: Current Cities collection

In this query, we want to create a collection named `cities`. Each document in the collection should contain two fields: a field called `_id` holding the city name, and a `users` field holding an array of user_ids who currently live in that city.

For example, if users 10, 20 and 30 live in Bucklebury, the following document will be in the collection `cities`:

{" _id": "Bucklebury", "users": [10, 20, 30]}

You do not need to return anything for this query.

Hint: You may find this link on MongoDB `$group` helpful:

https://docs.mongodb.com/manual/reference/operator/aggregation/group/

Query 4: Suggest friends

Find all user_id pairs (A, B) that meet the following requirements:

i. user A is male and user B is female
ii. their Year_Of_Birth difference is less than year_diff, an argument passed in to the query
iii. user A and user B are not friends
iv. user A and user B are from the same hometown city

Your query should return a JSON array of "pairs"; each pair is an array with two user_ids. In other words, you should return an array of arrays.

Hint: You may find `cursor.forEach()` useful. You may also use `array.indexOf()` in JavaScript to check for the non-friend constraints.

Query 5: Find the oldest friend

Find the oldest friend for each user who has friends. For simplicity, use only the Year_Of_Birth field to determine age. In case of a tie, return the friend with the smallest user_id.

Notice in the `users` collection, each user has only information on friends whose user_id is greater than their user_id. You will need to consider all existing friendships. It may be helpful to go over some of the strategies you used in Queries 2 and 3 (see **Important Note** below). Your query should return a JSON object: the keys should be user_ids and the value for each user_id is their oldest friend's user_id. The order of your results does not matter. The schema should look like the following:

{ user_id1: user_idx,

user_id2: user_idy, …}

The number of key-value pairs should be the same as the number of users who have friends.

Important Note: Collections created by your queries such as `flat_user` and `cities` **will NOT** persist across test cases in the Autograder. If you want to re-use any of these collections, you should create them again in the corresponding queries.

Query 6: Find average friend count

Find the average number of friends a user has in the `users` collection and return a decimal number. The average friend count on users should also consider those who have 0 friends. In order to make this easier, we're treating the number of friends that a user has as equal to the number of friends in their friend list (we ARE NOT counting users with lower ids, since they aren't in the friend list). DO NOT round the result to an integer.

Query 7: Find count of user born in each month using MapReduce

MapReduce is a powerful parallel data processing paradigm. We have set up the MapReduce calling point in test.js and you need to implement the mapper, reducer and finalizer.

In this query, we are asking you to use MapReduce to find the number of users born in each month. Note that after running test.js, running `db.born_each_month.find()` in the `mongo` Shell (use `make loginmongo` to log into mongo shell) allows you to bring up the collection showing the number of users born in each month. For example, if there are 200 users born in September, the document below would be in the collection:

{" _id": 9, "value": 200}

Query 8: Find city-average friend count using MapReduce

In this query, we are asking you to use MapReduce to find the average friend count per user where the users have the same hometown city. Instead of getting only one number for all users' average friend count, we will have an average friend count for each hometown city.

The average calculation should be performed in the finalizer. Note that after running test.js, running `db.friend_city_population.find()` in Mongo Shell allows you to bring up the collection with per city average friend count. For example, if users whose hometown is Bucklebury have an average friend count 15.23, the document below would be in the collection:

{" _id": "Bucklebury", "value": 15.23}

5. Submission and Grading

The Autograder is available at https://autograder.io/. Before you submit to the Autograder, it is best to do some local testing using the provided test.js and Makefile since you have limited submissions per calendar day. Only your first three submits will receive feedback. We will grade any additional submissions but not display any feedback or score until after the deadline.

To submit on the Autograder, join a team first and submit the following files directly:

1. `GetData.java`

2. `query[1-8].js`

All test cases are graded separately, so you can submit just the files you want to have graded.

Late day policy:

Project 3 is due on March 24, 2022 at 11:55 pm EDT. Please refer to the course policy for information on the late submission penalty.

6. Appendix

6.1 Mapreduce tips for Query 7 & 8:

Understanding the concept with examples:

- https://docs.mongodb.com/v3.2/core/map-reduce/
- https://docs.mongodb.com/v3.2/tutorial/map-reduce-examples/

Since the output of a reducer can be fed into another reducer (reducers can take input from both mappers and reducers), the *value* emitted from your mapper (where the mapper emits (*key*, *value*)) should have **the exact same form** as what is returned by your reducer.

The reducer must satisfy the following conditions:

- the *type* of the return object must be **identical** to the type of the `value` emitted by the `map` function.
- the `reduce` function must be *associative*. The following statement must be true:

```
reduce(key, [ C, reduce(key, [ A, B ]) ] ) == reduce( key, [ C, A, B ] )
```

Source: https://docs.mongodb.com/manual/reference/command/mapReduce/

For query 8, the average calculation must be performed in the finalizer because the reducer function must be associative.

6.2 Tips for debugging

The `test.js` file will print out the output of your queries.

You can also add `print()` or `printjson()` inside your code for print debugging.