EECS 484 Projects | p3-mongodb

Project 3: MangoDB

Worth	Released	Due
100 points (20 for Part A, 80 for Part B)	May 23rd	**June 6th at 11:55 PM EST**

Project 3 is due on **June 6th at 11:55 PM EST**. Please refer to the EECS 484 SP23 Course Policies for more information on penalties for late submissions, late day tokens, and sick days.

Introduction

In Project 3, we will use a similar dataset as in Project 2 and explore the capabilities of MongoDB (a NoSQL DBMS). 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 output.json that contains information about users. In Part B of the project, you will be importing output.json (or a sample.json that we give you) into MongoDB to create a mongoDB collection of 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.

Submissions

This project is to be done in teams of 2 students (recommended) or individually. Be sure to create your team on the Autograder.

Honor Code

By submitting this project, you are agreeing to abide by the Honor Code: "I have neither given nor received unauthorized aid on this assignment, nor have I concealed any violations of the Honor Code." You may not share answers with other students actively enrolled in the course outside of your teammate, 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.

Part A: Export Oracle Database to JSON

Introduction to JSON

JSON (JavaScript Object Notation) is a way to represent data in a key-value format, much like a `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"]; // returns "John Doe"
```

Export to JSON

Your job for Part A is to query the Project 3 Fakebook Oracle database (tables are prefixed with `project3.public_`) to export comprehensive information on each user. The results should be stored in a `JSONArray`, containing 800 `JSONObjects` for 800 users. It is suggested that you use multiple queries to retrieve all the information. This should feel very similar to Project 2. Each `JSONObject` should include:

- user_id (`int`)
- first_name (`String`)
- last_name (`String`)
- gender (`String`)
- YOB (`int`)
- MOB (`int`)
- DOB (`int`)

- 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. Note that Friends relationship is assumed to be symmetric. If user 700 is friends with user 25, it will show up on the list for user 25 but will not show up on the list for user 700.
- current (`JSONObject`) that contains:
  - city
  - state
  - country
- hometown (`JSONObject`) that contains:
  - city
  - state
  - country

Below is an example of one element of this `JSONArray` . It is possible that a user might have no list of friends, current city, or hometown city. In this case, put **an empty `JSONArray([])`** as the value for the "friends" key of that user or **an empty `JSONObject({})`** as the value for the "current" or "hometown" key of that user. See `sample.json` for the correct output.

```
{
  "MOB": 10,
  "hometown": {
    "country": "Middle Earth",
    "city": "Linhir",
    "state": "Gondor"
  },
  "current": {
    "country": "Middle Earth",
    "city": "Caras Galadhon",
    "state": "Lothlorien"
  },
  "gender": "female",
  "user_id": 744,
  "DOB": 14,
  "last_name": "MARTINEZ",
  "first_name": "Lily",
  "YOB": 516,
  "friends": [754, 760, 772, 782]
}
```

Starter Files

Download the starter files (p3-starter_files.tar.gz).

For Part A, you only need to be concerned about the following files

- GetData.java
- Main.java
- Makefile
- sample.json
- json_simple-1.1.jar, json-20151123.jar, ojdbc6.jar

GetData.java

Submit this file. Implement `toJSON()` by querying the users, friends, and cities tables to retrieve data from the Oracle Database. The `writeJSON()` function will take care of converting your output (a `JSONArray`) into a JSON string stored in `output.json`. Feel free to use the SQL*Plus CLI; the table names are listed in the beginning of this file.

Main.java

This file provides the main driver function for running Part A. You should use it to run your program, but you don't need to turn it in. When `Main.java` is run, an output file named `output.json` should be generated. Modify the `oracleUserName` and `password` static variables, replacing them with your own **Oracle** username and password.

```
1  static String oracleUserName = "uniqname"; // replace with your uniqname
2  static String password = "password"; // replace with your Oracle password (default: ee
```

Makefile

Once you have implemented `GetData.java` and modified `Main.java`, you can compile and run your program.

```
1  $ make compile
2  $ make run
```

If your username/password combination is incorrect in `Main.java`, you will get an error message `java.sql.SQLException: ORA-01017: invalid username/password; logon denied`. If you need to reset your password, refer to Tools.

sample.json

This file contains the JSON data from running the instructor implementation of `toJSON()` in `GetData.java`. Please **do not** validate your output using `diff output.json sample.json` because JSON arrays are likely to come out in different orderings 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. jd (github and online) and deepdiff are both valid options.

json_simple-1.1.jar, json-20151123.jar, ojdbc6.jar

These jar packages help compile your code. Do not modify them.

Wrapping Up

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 a correct `output.json` from Part A.

If you'd like, you can also submit `GetData.java` from Part A on the Autograder without completing Part B.

Part B: MongoDB Queries

Introduction to MongoDB

MongoDB is a document-oriented noSQL DBMS. Each document in MongoDB is one JSON object, with key-value pairs of data, just like how a tuple in SQL has fields of data. Each collection in MongoDB is one JSON array of multiple documents, just like how a table in SQL has multiple tuples. Refer to the following table for some high-level differences between SQL and MongoDB.

SQL	MongoDB
Tuple	Document. Represented as a JSON object
Relation/Table	Collection. Represented as 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});`

Logging into MongoDB

To run the MongoDB queries, you will need to login to a `mongo` shell. You may either download MongoDB(v3.6) on your personal computer with a private server, or access CAEN and use the shared server. We recommend the second approach, but both options are listed below.

Local MongoDB

To use MongoDB on your local machine, refer to MongoDB's installation 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 in a Terminal window:

```
$ mongo <database> # omit angle brackets
```

Note that the starter file `Makefile` does not work in a local environment unless properly modified. No hostname, userid, or password is required, so edit your `Makefile` such that these fields are removed from all of your make rules. In our `Makefile`, `uniqname` is the name of the database that `mongo` will use for commands. 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 on CAEN.

CAEN MongoDB

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. Double check that you have the `mongodb` module loaded (see Class Modules).

Then, fill in the `uniqname` and `password` fields in the `Makefile`. The default MongoDB password is your uniqname. If you have the wrong login credentials, you will get the error message `Error: Authentication failed`.

```
uniqname = uniqname # replace with your uniqname
password = password # replace with your mongoDB password (default: your uniqname)
```

Then, login into the mongo shell. You can use this interactive shell to test queries directly on your database, similar to the SQL*Plus CLI in Projects 1 and 2.

```
$ make loginmongo
```

The mongo shell will open up in your terminal. You can update your password with the following command, which will take effect when you log out.

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

Do not include the '$' symbol in your password. You may be unable to login again, and you will have to ask the staff to reset your account.

Import JSON to MongoDB

Now that you have access to a MongoDB database, the next step is to load data into it. Open a terminal in the folder where you have `sample.json` (or `output.json`) and `Makefile`. **Update the `Makefile` with your new password** and run ***either*** of the following

```
$ make setupsampledb  # load user collection using sample.json
```

```
$ make setupmydb  # load user collection using 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. As a reminder, `sample.json` is correct and given in the starter files. `output.json` is generated by your code from Part A.

Testing Your Queries

In the next section, 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.

```
let dbname = "uniqname"; // replace with your uniqname
```

To run `test.js`, use the following Makefile command

```
$ make mongotest
```

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. Each test also has a line you can uncomment to show the output for a specific test. For example, the test for query 1 looks like

```
print("=== Test 1 ===");
let test1 = find_user("Bucklebury", dbname);
// print(test1); // uncomment this line to print the query1 output
let ans1 = test1.length;
```

```
if (ans1 == 42) {
  print("Local test passed! Partially correct.");
} else {
  print("Local test failed!");
  print("Expected 42 users from Bucklebury, you found", ans1, "users.");
}
cleanUp();
```

Queries

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 `db.collection.find()` and `cursor.forEach()` helpful.

Query 2: Flatten Friends

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_id`s 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.

You may find `$unwind` helpful. You may use `$project` and `$out` to create the collection, or you may insert tuples into `flat_users` iteratively.

Query 3: City Dwellers

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_id`s who currently live in that city. The `user_id`s do not need to be sorted but should be distinct. 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.

You may find `$group` helpful.

Query 4: Matchmaker

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

- user A is `"male"` and user B is `"female"`
- the difference between their year of births (`YOB`) is less than the specified `year_diff`
- user A and user B are not friends
- user A and user B are from the same `hometown.city`

Your query should return a JSON array of pairs, where each pair is an array with two `user_id`s. In other words, you should return an array of arrays.

You may find `cursor.forEach()` helpful. You may also use `array.indexOf()` to check for the non-friend constraints.

Query 5: Oldest Friends

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

Notice in the users collection, each user only has 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. 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.**

Your query should return a JSON object: the keys should be `user_id`s and the value for each `user_id` is their oldest friend's `user_id`. The order of your results does not matter. The number of key-value pairs should be the same as the number of users who have friends. The schema should look like:

```
{user_id1: user_idx,
```

```
    user_id2: user_idy,
    ...}
```

You may find $or, $in, and cursor.sort() helpful. Again, you can also choose to do this query iteratively.

Query 6: 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: Birth Months 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.

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 allows you to bring up the collection showing the number of users born in each month. For example, if there are 66 users born in September, the document below would be in the collection:

```
{"_id": 9, "value": 66}
```

You may find these helpful: Map-Reduce, Map-Reduce Examples

Query 8: Birth Friendly Cities using MapReduce

In this query, 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` allows you to bring up the collection with per city average friend count. For example, if users whose hometown is Breredon have an average friend count 27.2, the document below would be in the collection:

```
{"_id": "Breredon", "value": 27.2}
```

Mapreduce Tips for Query 7 and 8

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:

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

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

Submitting

The deliverable for Part A is `GetData.java`. This is worth 20 points. The deliverables for Part B are `query[1-8].js`. Each query is worth 10 points, with a total of 80 points. The entire project is worth 100 points. There are no private tests.

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

Remember to remove any print statements, as your submission will fail on the Autograder even if it compiles on CAEN.

Each team will be allowed 3 submissions per day with feedback; any submissions made in excess of those 3 will be graded, but the results of those submissions will be hidden from the team. Your highest scoring submission will be used for grading, with ties favoring your latest submission.

Acknowledgements

This project was written and revised over the years by EECS 484 staff at the University of Michigan. The most recent version was updated and moved to Primer Spec by Owen Pang.