EECS 484 F16 Project 4

Due Date: Dec. 10th, 11:55 PM.
Late period: Dec. 13th, 11:55 PM (Note: this is the last day of classes). 10% late penalty.

The server will accept till 11:59:59 PM by its local clock, but it is best to submit prior to 11:55 PM to allow for clock drift, latencies, etc.

The project can be done individually or in a group of two. However, **each partner must submit separately**, but identify the partner they worked with.

Note: We may make minor tweaks to this draft to clarify the specifications. We recommend visiting this original page for the specs from time to time.

Overview

This project consists of two independent components:

1. Query planning in PostgresQL using a DVD store database.
2. Exporting FakeBook Project 2 data from Oracle to MongoDB, and then writing some MongoDB queries.

You may find the details of each component below. You will need to be on the University of Michigan network to do the assignment, since the database servers are only accessible from within the University. If you are off-campus, you must first connect to the University of Michigan VPN network. See http://www.itcom.itd.umich.edu/vpn/ for instructions on connecting to the VPN.

You can do 1. and 2. in either order -- they are totally independent activities. We have talked about MongoDB in the lecture on 11/21. So, you may consider starting on that first. Within MongoDB project, you can do Part I or Part 2 in either order as well. Part I is about writing a Java program to convert Oracle data from Project 2 to MongoDB format (JSON) and then importing it into MongoDB. Part 2 is about writing 7 queries on the imported data. We have given you a JSON file that you can use for Part 2, if you wish to do Part 2 first.

Our advice for F16: Do MongoDB part of of the project first. That is 80% of the project and is completely auto-graded. Do Part I or Part 2 of that in either order (or in parallel!). It is worth 8% of the overall grade.

Postgres Portion does not require coding and there is no auto-grader -- it requires you to use the Postgres database, read its manual, following the instructions, and fill out a Google form

based on the observations. It is almost like doing a homework (and since it is worth 20% of this project, it ends up being worth the same as other homeworks - 2% of the overall grade).

PostgreSQL (20 points)

We will have a couple of exercises using PostgreSQL to help you understand query planning and query optimization. The spec and questions can be found in the following Google form link. You need to fill in your answer in the Google form. You don't need to submit any files for this exercise.

https://goo.gl/forms/mqrz6jxjYmh0LAvr1

Following is an introduction on how to connect to Postgres, load data and some helpful links, before you answer the questions at the above google form.

Logging into Postgres

To login to postgres, login to a CAEN machine. Alternatively, download a postgres client program such as psql to your machine (first check - you may already have it if you have a Mac or Linux machine).

```
% psql -h eecs484.eecs.umich.edu -U  uniquename
```

Your default Postgres password is *postgres484student*. You need to connect from either a University machine, while you are connected to the University network via UM Wireless (not guest), or be on the University of Michigan VPN.

Once you login, you can change your password as follows on the postgres => prompt:

```
=> \password
```

Remember your new password. Resetting it is possible but will incur a delay (possibly even 24-48 hours) as it would be a manual process. If you need to reset it, then you should email the teaching staff and allow us 24-48 hours to respond. CAEN or ITD do not support this database system.

Initializing the Database

This project comes with a zip file (dvd_store_data.zip) that contains relevant commands to initialize the database. Download and unzip the file and you will find a file called setup_db.sql. You will use this file to populate the database.

Specifically, you can do the following steps on CAEN to initialize the database:

```
% unzip dvd_store_data.zip
% cd dvd_store_data
% psql -h eecs484.eecs.umich.edu -U  uniquename
Password for user  uniqname:
```

You are now connected to the Postgres interactive terminal. Run the following command (note the backslash) to execute commands from setup_db.sql:

```
\i setup_db.sql
```

It can take a few minutes for the database to be initialized. Here is what you may see when initializing (or re-initializing). The error messages about permission being denied on triggers can be ignored.

```
uniquename=> \i setup_db.sql
psql:pgsqlds2_delete_all.sql:2: ERROR:  permission denied:
"RI_ConstraintTrigger_19357" is a system trigger
psql:pgsqlds2_delete_all.sql:3: ERROR:  permission denied:
"RI_ConstraintTrigger_19359" is a system trigger
psql:pgsqlds2_delete_all.sql:4: ERROR:  permission denied:
"RI_ConstraintTrigger_19409" is a system trigger
psql:pgsqlds2_delete_all.sql:5: ERROR:  permission denied:
"RI_ConstraintTrigger_19391" is a system trigger
psql:pgsqlds2_delete_all.sql:6: ERROR:  permission denied:
"RI_ConstraintTrigger_19424" is a system trigger
psql:pgsqlds2_delete_all.sql:7: ERROR:  permission denied:
"RI_ConstraintTrigger_19383" is a system trigger
DROP TABLE
```

```
DROP TABLE
psql:pgsqlds2_create_tbl2.sql:30: NOTICE:  CREATE TABLE will create
implicit sequence "customers_customerid_seq" for serial column
"customers.customerid"
psql:pgsqlds2_create_tbl2.sql:30: NOTICE:  CREATE TABLE / PRIMARY KEY
will create implicit index "customers_pkey" for table "customers"
CREATE TABLE
psql:pgsqlds2_create_tbl2.sql:43: NOTICE:  CREATE TABLE will create
implicit sequence "orders_orderid_seq" for serial column
"orders.orderid"
psql:pgsqlds2_create_tbl2.sql:43: NOTICE:  CREATE TABLE / PRIMARY KEY
will create implicit index "orders_pkey" for table "orders"
CREATE TABLE
psql:pgsqlds2_create_tbl2.sql:51: NOTICE:  CREATE TABLE will create
implicit sequence "categories_category_seq" for serial column
"categories.category"
psql:pgsqlds2_create_tbl2.sql:51: NOTICE:  CREATE TABLE / PRIMARY KEY
will create implicit index "categories_pkey" for table "categories"
CREATE TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
```

```
psql:pgsqlds2_create_tbl2.sql:82: NOTICE:  CREATE TABLE will create
implicit sequence "products_prod_id_seq" for serial column
"products.prod_id"
psql:pgsqlds2_create_tbl2.sql:82: NOTICE:  CREATE TABLE / PRIMARY KEY
will create implicit index "products_pkey" for table "products"
CREATE TABLE
CREATE TABLE
CREATE TABLE
psql:pgsqlds2_create_tbl2.sql:115: NOTICE:  CREATE TABLE / PRIMARY
KEY will create implicit index "inventory_pkey" for table "inventory"
CREATE TABLE
CREATE TABLE
psql:pgsqlds2_load_orderlines.sql:1: ERROR:  permission denied:
"RI_ConstraintTrigger_20643" is a system trigger

psql:pgsqlds2_load_orderlines.sql:16: ERROR:  permission denied:
"RI_ConstraintTrigger_20643" is a system trigger
```

Helpful Links

Now, you should proceed to do the quiz at the Google Forms link given at the beginning. When you do the quiz, you may have to refer to Postgres documentation and run postgres commands to find the answers to the quiz questions.

Some of the tables that you will be using during the course are system catalog pages. In most databases, there are tables about tables that you create. For example, one of the tables in Postgres is pg_class that has general information about all the relations, indexes, etc., including their column names. Another table is pg_stats, which contains approximate number of tuples in each table, number of distinct values in each column, etc. These two tables are very useful in query optimization. The data in these tables helps the query optimizer estimate the cost of different ways of evaluating a query (e.g., whether to use an index or to ignore it).

For example, ignoring an index and just doing a regular file scan may be more efficient in some cases (e.g., see Lecture Notes where examples of SELECT queries on age being 18 for UM students were discussed).

The following are links to Postgres documentations relevant to this assignment. Be prepared to look up the documentation as you work on the exercises.

- Full PostgreSQL 8.4.16 documentation:
http://www.postgresql.org/docs/8.4/static/index.html

- System catalogs that give you information about tables:
http://www.postgresql.org/docs/8.4/static/catalogs.html

- Statistics used by the query planner
http://www.postgresql.org/docs/8.4/static/planner-stats.html

- How to manipulate the query planner (such as disabling the use of a join algorithm)
http://www.postgresql.org/docs/8.4/static/runtime-config-query.html

- Syntax of EXPLAIN command
http://www.postgresql.org/docs/8.4/static/sql-explain.html

- How to use EXPLAIN command and interpret its output
http://www.postgresql.org/docs/8.4/static/using-explain.html

- Creating an index
http://www.postgresql.org/docs/8.4/static/sql-createindex.html

- Creating a clustered index
http://www.postgresql.org/docs/8.4/interactive/sql-cluster.html

---

MongoDB (80 points)

Introduction

In this project, you will learn how to transfer data from SQL to MongoDB and learn to perform MongoDB queries. There are two parts in this project. In part 1, you will need to write a Java program to export a small portion of Fakebook data stored in Project 2 tables into one JSON file which serves as the input to part 2. In part 2, you will need to import this JSON file into MongoDB and perform a couple of queries in the form of JavaScript functions.

(**Note**: we have provided you the JSON file that should result from your part 1, to allow you to work on part 1 and part 2 in either order, and to help you check the correctness of part 1. See more on this later in this document.)

JSON objects and arrays

MongoDB uses a format called JSON (Javascript Object Notation) extensively. JSON is a key-value representation, in which values can also be JSON objects. Here are some examples of objects in JSON notation:

Example 1 (JSON object):

```
{"firstName":"John", "lastName":"Doe"}
```

Here, "firstName" is a key, and the corresponding value is "John". Similarly, "lastName" is a key and "Doe" is a value. Think of it like a map. Here is some Javascript code that uses the above:

```
var employee = {"firstName":"John", "lastName":"Doe"};
employee["firstName"];   // displays John

One can also have a JSON array, which are an array of JSON objects.
For example, variable employees is an array of JSON objects.

var employees = [
     {"firstName":"John", "lastName":"Doe"},
     {"firstName":"Atul", "lastName":"Prakash"},
     {"firstName":"Barzan","lastName": "Mozafari"}
];

employees[1]["firstname"];  // prints Atul
```

Nesting is possible. For example, in a key-value pair, the value can be a JSON array, another JSON object, or just a simple string or number.

MongoDB does not use SQL. But there are some similarities. Here are a few simple examples. See the following more examples: https://docs.mongodb.org/manual/reference/sql-comparison/

SQL	MongoDB
`Table`	`Collection. Initialized using a JSON array.`
`Tuple or row of a table`	`Document. Corresponds to a JSON object`
`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});`

Install mongodb on your local machine:

We encourage you to install mongodb locally in your laptop to have a more pleasant development environment and also to explore mongdb's functionality by yourself.
You can follow instructions in the following links. *Post on piazza if you get stuck and feel free to help other students get mongo installed. A great way is to reply to any questions related to installation on Piazza, based on your experience. Google search on installation error messages can also help.*

Install mongodb on Mac:
https://docs.mongodb.com/v3.2/tutorial/install-mongodb-on-os-x/
(Note: You may need to use sudo command at times to temporarily become root if you get Permission Denied errors).

Install mongodb on Windows:
https://docs.mongodb.com/v3.2/tutorial/install-mongodb-on-windows/

Install mongodb on Linux
https://docs.mongodb.com/v3.2/administration/install-on-linux/
(Note: You may need to use sudo command at times to temporarily become root if you get Permission Denied errors.)

When you have it successfully installed, the following commands should work from a Terminal:

% `mongo`

% `mongoimport`

If you have trouble installing locally, you can also use the mongo server on eecs484.eecs.umich.edu directly by supplying a userid and password. See Part 2 below.

Part 1 - Export SQL data to JSON using JDBC

This part does not really make use of MongoDB. It instead relies on your knowledge from Project 2 (SQL!). You will be retrieving data from the public tables of Project 2 and outputting a subset of the information as a JSON array. We have given you the output file as well that we are expecting as output so that you can check your answer.

You are provided with 3 files: GetData.java, Main.java, and Makefile for this part. You will write code inside GetData.java, resulting a file `output.json`.

Also, you are provided sample.json, which is one of the possible correct outputs. You should get an equivalent JSON array in output.json as in sample.json.

1) Main.java
This file contains the main function to run the program. The only modification you need to make to this file is to provide your SqlPlus username and password that you used in Project 2 as well. Please refer to your project 2 files in case you forgot what it is, since it is probably embedded in one of the files there.

```
public class Main {

    static String dataType = "PUBLIC";
    static String oracleUserName = "username"; //replace with your Oracle account name
    static String password = "password"; //replace with your Oracle password
...
```

2) GetData.java

This file contains the function you need to write to export data from SqlPlus to a JSON file. The function will output a JSON file called “output.json” which you will need to submit.

```
public JSONArray toJSON() throws SQLException{

    // Your implementation goes here....



    // This is an example usage of JSONArray and JSONObject
    // The array contains a list of objects
    // All user information should be stored in the JSONArray object: users_info
    // You will need to DELETE this stuff. This is just an example.

    // A JSONObject is an unordered collection of name/value pairs. Add a few name/value pairs.
    JSONObject test = new JSONObject();     // declare a new JSONObject
    // A JSONArray consists of multiple JSONObjects.
    JSONArray users_info = new JSONArray();

    test.put("user_id", "testid");          // populate the JSONObject
    test.put("first_name", "testname");

    JSONObject test2 = new JSONObject();
    test2.put("user_id", "test2id");
    test2.put("first_name", "test2name");

    users_info.add(test);                   // add the JSONObject to JSONArray
    users_info.add(test2);                  // add the JSONObject to JSONArray
    return users_info;
}
```

You need to use JDBC to query relevant project 2 tables to get information about users and store the information in a JSONArray called `users_info`. It is OK to use multiple queries -- in fact, that may be more convenient to do.

The `users_info` object is a JSONArray and contains an array of JSONObjects. Each JSONObject should contain information about one user. For each user (stored as a JSONObject), you will need to retrieve the following information:

- user_id
- first_name

- last_name
- gender
- YOB
- MOB
- DOB
- hometown (JSONObject). Hometown contains city, state and country.
- friends (JSONArray). Friends contains an array of user_ids which are greater than the user_id of that user.

Following is an example of one user JSONObject (order of key/value pairs in JSONObject doesn't matter).


```
{
  "first_name": "Frodo",
  "friends": [
    184,
    214,
    240,
    242,
    244,
    255,
  ],
  "hometown": {
    "state": "Gondor",
    "country": "Middle Earth",
    "city": "Edhellond"
  },
  "DOB": 28,
  "MOB": 12,
  "last_name": "GARCIA",
  "gender": "male",
  "YOB": 547,
  "user_id": 163
},
```


See the file `sample.json` for an example valid output.

3) Makefile

We provide a simple Makefile to compile the Java files and run them. You may make changes to this file if necessary. To compile the code, do:

$ make

To run the code, do:

$ make run

An output file `output.json` should result.

4) output.json

Since the order of attributes inside a jsonObject is not fixed, there are a lot of correct answers for `output.json`. However, when you import them into database, they should be all identical for queries. For your convenience, we have provided a sample.json, which is one of the correct answers.

To test whether your `output.json` is correct, you could do Part 2 with your `output.json` as the input instead of `sample.json`. If you get the same answers, that is a good sign (though not a proof). If you get different answers, then something is definitely wrong your `output.json`.

Part 2 - Query MongoDB

The first step is to import `output.json` file from part 1 to MongoDB as a collection. Alternatively, you can use `sample.json` as your input file.

If you are working on CAEN machine, use the following command to input `sample.json` into the database, for example:

```
$ module load mongodb
$ mongoimport --host eecs484.eecs.umich.edu --username <uniquename>
--password <password>  --collection users --db <uniquename> --file
sample.json --jsonArray
```

You can also do this by modifying the Makefile and then doing:

```
$ make setupsampledb
```

Alternatively, to use your output.json, you can do:

```
$ make setupmydb
```

On eecs484 server, we have set up Mongo databases for each student. The database name is your uniquename, and the username is also your uniquename. Password is eecs484class for all student, you can change your password for your database, see(https://docs.mongodb.org/manual/tutorial/manage-users-and-roles/).

What you need to do is

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

You can do it either in mongo shell or run it as script

(If you are using a private, local copy of mongodb on your personal machine, you can instead just omit the --host, --username, and --password flags).
When importing a json file, **please use "users" as your collection name, and do not modify this collection.** The above command does that.

You can create additional collections besides <users> if you want as helper collections to answer the queries.

For the second step of this part, you will need to write 5 queries in the form of JavaScript functions to query MongoDB. MongoDB can load JavaScript files. You can find more information via the following link.

https://docs.mongodb.org/manual/tutorial/write-scripts-for-the-mongo-shell/

If a collection is created in a query, you may reuse that collection in subsequent queries to save time.

Note: Since only hometown information is retrieved in part 1. We assume that the **city in queries means hometown city.**

Query 1: Find users who live in a certain city

This query should return a javascript array of user_ids who live in the specified city. City is passed as a parameter of function find_user.

Hint: A possible answer would start out like:
var result = db.users.find(....); // Read MongoDB tutorials on the find command.

In addition, you may find the following useful.

https://docs.mongodb.org/v3.0/reference/method/cursor.forEach/

Instead of using forEach, you can also iterate over the cursor result that is returned by result, and push the user_id from the result into a Javascript array variable.

```
function find_user(city, dbname){
   db = db.getSiblingDB(dbname)
   //implementation goes here
```

```
    // returns a Javascript array. See test.js for a partial correctness check.  This will be
   // an array of integers.    The order does not matter.
}
```

Note: Query 2-5 assume that the variable `db` has been initialized from Query 1 above. Do not drop the db database.

Query 2: Unwind friends

Each document in the <users> collection represents one user's information, including a list of friends' id.

In this query, you need to unwind the friends list such that the resulting collection contains document which represents a friend pair. You don't need to return anything. **The new collection must be named flat_users.**
You may find this link on MongoDB unwind helpful.
https://docs.mongodb.org/manual/reference/operator/aggregation/unwind/

```
function unwind_friends(dbname){
   db = db.getSiblingDB(dbname)
   //implementation goes here


   // returns nothing. It creates a collection instead as specified above.
}
```

You may also find the following useful:

https://docs.mongodb.org/manual/reference/operator/aggregation/

In particular, besides $unwind, $project and $out can also be useful. $out can create a collection directly. Instead of $out, you can also iterate over the resulting cursor from the query and use insert operator to insert the tuples into flat_users. See the documentation link in the query below as well.

Query 3: Create a city to user_id mapping

Create a new collection. Documents in the collection should contain two fields: _id field holds the city name, users field holds an array of user_id who live in that city. You don't need to return anything. **The new collection must be named cities.**

You may find the following link helpful.
https://docs.mongodb.org/manual/reference/operator/aggregation/out/

```
function cities_table(dbname) {
    db = db.getSiblingDB(dbname)
    //implementation goes here

    // Returns nothing. Instead, it creates a collection inside the database.
}
```

Query 4: Recommend friends

Find user pairs such that, one is male, the other is female, their year difference is less than year_diff, and they live in same city and they are not friends with each other.
Store each friend pair as an array (**male first, female second**), and store all pairs as an array of arrays, and return the array at the end of the function.

```
function suggest_friends(year_diff, dbname) {
    db = db.getSiblingDB(dbname)
    //implementation goes here

    // Return an array of arrays.
}
```

Query 5: Find the oldest friend

Find the oldest friend for each user who has friends. For simplicity, use only year of birth to determine age. If there is a tie, use the one with smallest user_id.

Notice that in the <users> collection, each user only has the information of friends whose user_id is greater than that user, due to the requirement in Fakebook database. You need to find the information of friends who have smaller user_ids than the user. You should find the idea of query 2 and 3 useful.

Return a javascript object with keys as user_ids and the value of keys is the oldest friend's id. The number of keys of the object should equal to the number of users who has friends.

```
function oldest_friend(dbname){
  db = db.getSiblingDB(dbname)

  //implementation goes here
  //return an javascript object described above
```

```
}
```

Query 6: Find the Average friend count for users

We define the `friend count` as the number of friends of a user. The average friend count is the average `friend count` towards a collection of users. In this function we ask you to find the `average friend count` for the users collection.

```
function find_avg_friend_count(dbname){
  db = db.getSiblingDB(dbname)

 //implementation goes here
 //return an javascript object described above
}
```

Return a decimal as the average user friend count of all users in the “users” collection.

Query 7: Find the city average friend count using MapReduce

MapReduce is a very powerful yet simple parallel data processing paradigm. Please refer to the discussion 8 (will be released on 11/30) for more information. In the question we are asking you to use MapReduce to find the “average friend count” at the city level (i.e., average friend count per user where the users belong to the same city). We have set up the MapReduce calling point in the test.js and we are asking to write the mapper, reducer and finalizer (if needed) to find the average friend count per user for each city.

You may find the following link helpful.
https://docs.mongodb.com/v3.2/core/map-reduce/

```
var city_average_friendcount_mapper = function() {
  // implem ction of average friend count
};

var city_average_friendcount_reducer = function(key, values) {
  // implement the reduce function of average friend count
};

var city_average_friendcount_finalizer = function(key, reduceValue) {
  // We've implemented a simple forwarding finalize function. This implementation  is naive: it just forwards the reduceVal to the output collection.
  // Feel free to change it if needed. However, please keep this unchanged: the var ret should be the average friend count per user of each city.
  var ret = reduceVal;
  return ret;
```

};

Sanity check: Note that after running the test.js, running db.friend_city_population.find() in mongo shell, you should find the documentation friend_city_population have the records in the similar form as following:
{ "_id" : "some_city", "value" : 15.23}
Where the _id is the name of the city, and the value is the average friend count per user.

Sample test

We offer a test.js which will call all 7 query javascript files and will print “Query x is correct!” if you query passes the simplistic test in test.js. In test.js, you **need to put your database name in the dbname variable**. Please make sure that all 7 query javascript files are in the same directory as test.js.

Also, please note that the autograder will use a similar program as test.js but is more exhaustive. In particular, we compare the output of your queries against a reference output in more depth. You are free to make changes on test.js for more exhaustive testing.

To run the tests from command-line, you can do:

```
$ module load mongodb     # This is only required one-time per login.

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

This will access to your database(created as your uniquename) on eecs484 mongodb server and run the script test.js.

Alternatively, you can use the Makefile as well.

```
$ make mongoquerytest
```

will basically do the above for you.

If you want to open the mongodb shell, you will do:

```
$mongo <uniquename> -u <uniquename> -p <password> --host eecs484.eecs.umich.edu
```

Again, if you are using a local mongodb on your personal computer, just do the following instead:

$ mongo dname < test.js

Note: it may take some time to run query 4 and query 5. However, query 4 should take less than 3 minutes and query 5 should take less than 6 minutes. You will receive 0 on that query if it exceeds this time limit.

What to submit

You should submit a zip file named p4.zip. The zip file should include:

GetData.java -- Your java program to create the json file. **Do not change the code for writing to output.json.**

query1.js -- return a javascript array of user_ids.

query2.js -- create a new collection called flat_users, nothing to return.

query3.js -- create a new collection called cities, nothing to return.

query4.js -- return a javascript array of user pairs.

query5.js -- return an javascript object, keys are user_ids, value for each key is the oldest friend id for that user_id

query6.js -- return a decimal which is the “average friend count” for all users

query7.js finish the mapper, reducer and finalizer

Where to submit

For Mongodb part of the project, we have made an autograder available at grader484.eecs.umich.edu. You will submit the mongodb part of the project online.

(Postgres portion is to be submitted via Google forms on the link given earlier in the specs.)