University of Toronto
CSC343, Winter 2017

Assignment 2

Due: Friday, March 17, at 8:00 pm sharp!

Learning Goals

By the end of this assignment you should have:

1. become fluent in SQL
2. learned your way around the postgreSQL documentation
3. learned how to embed SQL in a high-level language using JDBC
4. encountered limits of the expressive power of standard SQL

General Instructions

Please read this assignment thoroughly before you proceed. Failure to follow instructions will affect your grade.

We strongly encourage you to do all your development for this assignment on the CS Teaching Lab computers, either in the person or via a remote connection, and we urge you to use the self-tester to check for basic compatibility with our autotesting infrastructure. See the final section of the handout for further details.

You are allowed, and in fact encouraged, to work with a partner for this assignment. You must declare your team (whether it is a team of one or of two students) before the due date, even if using grace points to hand the work in a few hours later.

Once you have submitted your files, be sure to check that you have submitted the correct version; new or missing files will not be accepted after the due date, unless your group has grace tokens remaining.

Schema

In this assignment, we will work with a database that could support the online tool MarkUs. MarkUs is "an open-source tool which recreates the ease and flexibility of grading assignments with pen on paper, within a web application. It also allows students and instructors to form groups, and collaborate on assignments." (reference: `http://markusproject.org/`). By now, you have used MarkUs to hand in a few pieces of course work, so you have experienced some of its features.

Download these starter files from the course webpage:

- The database schema, `schema.ddl`
- A very small sample data set (with just enough data to show you how to format your own), `data.sql`

You can simply download the files from a browser and save them into a directory of your choice. You can also use the command line to download the files. For example:

```
> wget http://www.teach.cs.toronto.edu/~csc343h/winter/assignments/a2/schema.ddl
```

Your code for this assignment must work on *any* database instance (including ones with empty tables) that satisfies the schema, so make sure you read and understand it.

The schema definition uses several types of integrity constraints:

- Some attributes must be present (“NOT NULL”).
- Every table has a primary key (“PRIMARY KEY”). The attributes that are part of a primary key form a key in the sense we are used to (no repeats), and none of them may be NULL.
- Some tables define a set of attributes to be (“UNIQUE”). The attributes that are part of a set declared to be UNIQUE form a key in the sense we are used to (no repeats), but one or more of them can be NULL.
- Some tables use foreign key constraints (“REFERENCES”). By default, they will refer to the primary key of the other table.

Warmup: Getting to know the schema

To get familiar with the schema, ask yourself questions like these (but don’t hand in your answers):

- How would the database record that a student is working solo on an assignment?
- How would the database record that an assignment does not permit groups, that is, all students must work solo?
- Why doesn’t the Grader table have to record which assignment the grader is grading the group on?
- Can different graders mark the various members of a group in an assignment?
- Can different graders mark the various elements of the rubric for an assignment?
- In a rubric, what is the difference between “out of” and “weight”?
- How would one compute a group’s total grade for an assignment?
- How is the total grade for a group on an assignment recorded? Can it be released before the a grade was assigned?

Part 1: SQL Statements

In this section, you will write SQL statements to perform queries. For each query, you will insert the final results into a table; we will check the contents of that table to see if your query was correct.

We will provide files named `q1.sql`, `q2.sql`, . . . , `q10.sql` Each of these files defines the schema for the result table for that query, so that you will know exactly what attributes we are expecting, of what type, and in what order. Put each of your queries into the appropriate file, and make your final statement in each file be:

`INSERT INTO qX (SELECT` . . . *complete your SQL query here* . . . `)`

where X is the number of the query. This will populate the answer table with the correct tuples.

You are encouraged to use views to make your queries more readable. However, each file should be entirely self-contained, and not depend on any other files; each will be run separately on a fresh database instance, and so (for example) any views you create in `q1.sql` will not be accessible in `q5.sql`.

At the beginning of the DDL file, we set the search path to `markus`, so that the full name of every table etc. that we define is actually `markus.`*whatever*. You may set the search path to `markus` at the top of your query files too, so that you do not have to use the `markus` prefix throughout. But do not use the schema called `public`.

The output from your queries must exactly match the specifications in the question, including attribute names, attribute types, and attribute order. Row order does not matter.

Write SQL queries for each of the following. Note that every query can be written in SQL, and that the questions are not in order of difficulty.

Throughout, when we ask for an assignment grade, report it as a percentage, such as 82.5.

1. **Distributions.** We'd like to compare the grade distributions across assignments.

   For each assignment, report the average grade, the number of grades between 80 and 100 percent inclusive, the number of grades between 60 and 79 percent inclusive, the number of grades between 50 and 59 percent inclusive, and the number of grades below 50 percent. Where there were no grades in a given range, report 0.

Attribute	
assignment_id	The assignment ID
average_mark_percent	The average grade for this assignment, as a percent
num_80_100	The number of grades between 80 and 100 percent inclusive
num_60_79	The number of grades between 60 and 79 percent inclusive
num_50_59	The number of grades between 50 and 59 percent inclusive
num_0_49	The number of grades below 50 percent
Everyone?	Every assignment should appear, even if there are no grades associated with it yet.
Duplicates?	No assignment should appear twice.

2. **Getting soft?** We'd like to know if graders give higher and higher grades over time, perhaps due to fatigue. To investigate this, we only want to examine graders who've had lots of grading experience throughout the course.

   For this question when we consider a grader's average on an assignment, we will mean their average across individual students, not groups. For instance, the grade earned by a group of three students will contribute three times to the average.

   Find graders who meet all of these criteria:

   - They have graded (that is, they have been assigned to at least one group) on every assignment.
   - They have completed grading (that is, there is a grade recorded in the Result table) for at least 10 groups on each assignment.
   - The average grade they have given has gone up consistently from assignment to assignment over time (based on the assignment due date).

   Report their name, the average across assignments of their average grade, and the increase between their average for the first assignment and their average for the last assignment. (For example, if the former is 72.5 percent and the latter is 82.9 percent, the increase is 10.4 percentage points.) You may assume that no two assignments have the same due date; this means that there is unambiguously one first and one last assignment.

Attribute	
ta_name	The name of a grader who meets the criteria, in this form: first name plus surname with a blank in between
average_mark_all_assignments	The average across assignments of their average grade
mark_change_first_last	The increase between their average for the first assignment and their average for the last assignment, as a number of percentage points.
Everyone?	Include only graders who meet the criteria.
Duplicates?	No grader can appear twice.

3. **Solo superior.** We are interested in the performance of students who work alone vs in groups.

   Find assignments where the average grade of those who worked alone is greater than the average grade earned by groups. For each, report the assignment ID and description, the number of students declared to be working alone and their average grade, the number of students (not groups) declared to be working in groups and the average grade across those groups (not students), and finally, the average number of students involved in each group, (include in this calculation those who worked solo).

   If groups have been declared for an assignment but no grades have been recorded, you will be able to report counts but not average grades — the averages will have to be null.

Attribute	
assignment_id	ID of the assignment
description	Description of the assignment
num_solo	The number of students declared to be working alone
average_solo	The average grade among students who worked alone, or null if there were none who worked alone and have a grade recorded in Result.
num_collaborators	The number of students (not groups) declared to be working in groups
average_collaborators	The average grade across those groups (not students), or null if there were no groups that have a grade recorded in Result.
average_students_per_group	The average number of students involved in each group, (include those who worked solo in this calculation)
Everyone?	Every assignment should appear, even if no grades are available for it yet.
Duplicates?	No assignment should appear twice.

4. **Grader report** We want to make sure that graders are giving consistent grades.

   For each assignment that has any graders declared, and each grader of that assignment, report the number of groups they have already completed grading (that is, there is a grade recorded in the Result table), the number they have been assigned but have not yet graded, and the minimum and maximum grade they have given.

Attribute	
assignment_id	The ID of an assignment that has one or more graders declared
username	A grader who is declared for that assignment
num_marked	The number of groups they have already graded
num_not_marked	The the number they have been assigned but have not yet graded
min_mark	The minimum grade they have given for that assignment, or null if they have given no grades
max_mark	The maximum grade they have given for that assignment, or null if they have given no grades
Everyone?	Every assignment that has graders declared should appear.
Duplicates?	No assignment should appear twice.

5. **Uneven workloads** We want to make sure that graders have had fairly even workloads.

   Find assignments where the number of groups assigned to each grader has a range greater than 10. For instance, if grader 1 was assigned 45 groups, grader 2 was assigned 58, and grader 3 was assigned 47, the range was 13 and this assignment should be reported. For each grader of these assignments, report the assignment, the grader, and the number of groups they are assigned to grade.

Attribute	
assignment_id	The ID of an assignment with a range (as described above) greater than 10
username	A grader for that assignment
num_assigned	The number of groups this grader has been assigned
Everyone?	Every assignment with a range over 10 should appear.
Duplicates?	No assignment should appear more than once. No assignment-grader pair should appear more than once.

6. **Steady work.** We'd like groups to submit work early and often, replacing early submissions that are flawed or incomplete with improved ones as they work steadily on the assignment.

   For each group on assignment A1 (the assignment whose description is 'A1'), report the group ID, the name of the first file submitted by anyone in the group, when it was submitted, and the username of the group member who submitted it, the name of the last file submitted, when it was submitted, and the username of the group member who submitted it, and the time between submission of the first and last file.

   It is possible that a group submitted only 1 file. In that case the first file and the last file submitted are the same. It is also possible that two files could be submitted at the same time. In that case, report a row for every first-last combination for the group.

<table>
<tr><th>Attribute</th><th></th><th></th></tr>
<tr><td>group_id</td><td>The ID of an A1 group</td><td></td></tr>
<tr><td>first_file</td><td>The name of the first file submitted by anyone in the group</td><td rowspan="7">or null if no file was submitted</td></tr>
<tr><td>first_time</td><td>The timestamp for its submission</td></tr>
<tr><td>first_submitter</td><td>The username of the group member who submitted it</td></tr>
<tr><td>last_file</td><td>The name of the last file submitted by anyone in the group</td></tr>
<tr><td>last_time</td><td>The timestamp for its submission</td></tr>
<tr><td>last_submitter</td><td>The username of the group member who submitted it</td></tr>
<tr><td>elapsed_time</td><td>The time between the first and last submission for this group</td></tr>
<tr><td>Everyone?</td><td>Every group defined for A1 should appear.</td><td></td></tr>
<tr><td>Duplicates?</td><td>A group may occur more than once if there is a tie for its first and/or last submission.</td><td></td></tr>
</table>

7. **High coverage** We are interested in identifying graders who have broad experience in this course.

   Report the username of all graders who have been assigned at least one group (the group could be solo or larger) for every assignment and have been assigned to grade every student (whether in a solo or larger group) on at least one assignment.

Attribute	
ta	The username of a grader who meets the criteria.
Everyone?	Only graders who meet the criteria should appear.
Duplicates?	No grader should appear more than once.

8. **Never solo by choice** We are interested in the performance of students who chose to work in multi-student groups wherever possible.

   For this question, assume that at least one assignment allows groups.

   Find students who never worked solo on an assignment that allows groups, and who submitted at least one file for every assignment (indicating that they did contribute to the group). Report their username, their average grade on the assignments that allowed groups, and their average grade on the assignments that did not allow groups.

Attribute	
username	The username of a student who meets the criteria
group_average	Their average grade on the assignments that allowed groups. (Assume that at least one assignment allows groups.)
solo_average	Their average grade on the assignments that did not allow groups, or null if there were none.
Everyone?	Every student who meets the criteria should appear.
Duplicates?	No student should appear twice.

9. **Inseparable** Report pairs of students who each did group work whenever the assignment permitted it, and always worked together (possibly with other students in a larger group).

Attribute	
student1	The username of the student in the pair that comes first alphabetically
student2	The username of the student in the pair that comes second alphabetically
Everyone?	Only pairs that meet the criteria should appear.
Duplicates?	No pair should appear twice.

10. **A1 report** We'd like a full report on the A1 grades per group, including a categorization.

    Compute the grade out of 100 for each group on assignment A1, the difference between their grade and the average A1 grade across groups (negative if they are below average; positive if they are above average), and either "above", "at", or "below" to indicate whether they are above, at or below this average.

Attribute	
group_id	The ID of a group for the assignment whose description is 'A1'
mark	The group's grade on A1, as a percentage, or null if they have no grade
compared_to_average	The difference between the group's grade and the average grade or null if they have no grade
status	Either "above", "at", or "below" to indicate whether they are above, at or below this average, or null if they have no grade
Everyone?	Every group declared for A1 should appear.
Duplicates?	No group should appear more than once.

Part 2: Embedded SQL

You've written some queries that run on our version of a MarkUs database, but think about how you use MarkUs. You use a graphical-user interface that lets you perform operations like inviting a student to join a group with you, and viewing your results for an assignment or test. Similarly, instructors and TAs perform operations (that are not available to you), like giving a group a grade on a rubric element, and releasing the grades for an assignment so that students can see them. All of this is triggered through actions in the user-interface (UI) like clicking buttons and completing text boxes, then results are reported back via the UI. But the UI ultimately has to connect to the database where the core data is stored. Some of the operations in MarkUs can be implemented by Java methods that are merely a wrapper around a SQL query. Other features include computation that can't be done, or can't be done conveniently, in SQL.

For Part 2 of this assignment, you will not build a user-interface, but will write several methods that the app would need. It would need many more, but we'll restrict ourselves to just enough to give you practise with JDBC and to demonstrate the need to get Java involved, not only because it can provide a nicer user-interface than postgreSQL, but because of the expressive power of Java.

General requirements

- You may not use standard input, standard output or standard error. Doing so even once may result in the autotester terminating, causing you to receive a **zero** for this part.
- You will be writing a method called `connectDb()` to connect to the database. When it calls the `getConnection()` method, it must use the database URL, username, and password that were passed as parameters to the `connectDb()` method; these values not be "hard-coded" in the method. Our autotester will use the `connectDb()` and `disconnectDB()` methods to connect to the database with our own credentials.
- You should **not** call `connectDb()` and `disconnectDB()` in the other methods we ask you to implement; you can assume that they will be called before and after, respectively, any other method calls.
- All of your code must be written in `Assignment2.java`. This is the only file you may submit for this part.
- You may not change the method signature of any of the methods we've asked you to implement. However, you are welcome to write helper methods to maintain good code quality.
- As you saw in lecture, to run your code, you will need to include the JDBC driver in your class path. You may wish to review the related JDBC Exercise posted on the course website.

Your task

Open the starter code in `Assignment2.java`, and complete the following methods.

1. `connectDB`: Connects to the database and sets the search path.
2. `disconnectDB`: Closes the database connection.
3. `assignGrader`: Assigns a grader for a group for an assignment.
4. `recordMember`: Adds a member to a group for an assignment.
5. `createGroups`: Creates student groups for an assignment.

The full specification for each method is provided in its Javadoc comments. The first two are very simple, and `createGroups` is the most complex of the methods.

You will have to decide what the database will do and what you'll do in Java. At one extreme, you could use the database for very little other than storage: for each table, you could write a simple query to dump its contents into a data structure in Java and then do all the real work in Java. This is a bad idea. The DBMS was designed to be extremely good at operating on tables! You should use SQL to do as much as it can do for you, and part of your mark for Part 2 will be based on whether or not you do so.

I don't want you to spend a lot of time learning Java for this assignment, so feel free to ask lots of Java-specific questions as they come up. Also, keep in mind that to complete the assignment, you shouldn't need any fancy data structures that you build either "by hand" or using built-in Java classes. The most you should use is an array or an `ArrayList` — and you rarely need it. You can find documentation on the latter here: `http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html`.

A small schema change: SERIAL

In order to complete method `createGroups`, you will need to generate a series of group IDs. Not surprisingly, SQL has a feature that can do this for us. (Read the demo, posted in file `serial-demo.txt`). In order to harness this feature, we have changed the type of the `group_id` attribute in table `AssignmentGroup` to `SERIAL`. Please download a copy of the revised schema and use it when writing and testing your code. The only change is to this one attribute in this one table, and it should have no impact on any queries you have written for Part 1.

About our group formation algorithm

The algorithm for createGroups puts students with similar past performance together. Whether homogeneity or heterogeneity (or some other property) is desirable on this "attribute" of students, or on other attributes, is an interesting question. There is quite a bit of research on this. (In a project that I worked on with colleages a few years ago, we developed software that allows the instructor to set the priorities and then optimizes group formation according to those priorities.) Our purpose here is not to propose that the algorithm is a good one, but to show you how an arbitrarily complex algorithm could be integrated with an underlying SQL database. Thought question: could our relatively simple group-formation algorithm be implemented in SQL? That is, could you write an `INSERT INTO` that would generate the appropriate rows for table `AssignmentGroup`?

Additional tips

You will need to look up details about built in types (such as timestamps) and how to work with them. You may find the `coalesce` feature helpful. Become familiar with the documentation for postgreSQL. (But don't waste time searching without purpose for features that you hope will save you.)

In your JDBC code for Part 2, some of your SQL queries may be very long strings. You should write them on multiple lines for readability, and to keep your code within an 80-character line length. But you can't split a Java string over multiple lines. You'll need to break the string into pieces and use `+` to concatenate them together. Don't forget to put a blank at the end of each piece so that when they are concatenated you will have valid SQL. Example:

```
String sqlText =
   "select client_id " +
   "from Request r join Billed b on r.request_id = b.request_id " +
   "where amount > 50";
```

This makes the string read like a query at the postgreSQL shell, except with some extra quotes and concatenation characters.

Here are some common mistakes and the error messages they generate (you may remember these from the exercise we did in class):

- You forget the colon:

```
cdf> java -cp /local/packages/jdbc-postgresql/postgresql-9.4.1212.jar Example
Error: Could not find or load main class Example
```

- You ran it on a machine other than dbsrv1

```
cdf> java -cp /local/packages/jdbc-postgresql/postgresql-9.4.1212.jar: Example
SQL Exception.<Message>: Connection to localhost:5432 refused. Check that the
hostname and port are correct and that the postmaster is accepting TCP/IP connections.
```

Aside: Your prompt on cdf is probably different from mine. I set mine to "`cdf>`".

Important: How we will assess the correctness of your code

We will be testing your code in the CS Teaching Labs environment using PostgreSQL. More specifically, we will test your queries and your JDBC code using an autotester integrated into MarkUs. You will be able to run a self-test on MarkUs, using the same autotesting infrastructure. It is your responsibility to make sure your code runs in this environment before the deadline! **Code which works on your machine but not on our autotester will earn a grade of zero for correctness, and will not be eligible for a remark request**.

The self-test will confirm that your code connects to ours as expected, for instance, that your attribute types and attribute order are as specified (for the SQL query part) and that your method names and parameters are consistent with ours (for the JDBC part). It will also check that your code produces the correct results, but only for one very simple test case. The self-test will not run your code through a thorough test suite. It's part of your job to plan and implement a thorough set of tests for your own code. If you do, it will sail through the thorough autotesting that we will ultimately use when grading.