University of Toronto
CSC343, Fall 2023

Assignment 2

Warmup due: Wednesday, October 18, before 3pm
Full assignment due: Wednesday, November 1, before 3pm

This is the handout for the full Assignment 2. The Warmup assignment is just a subset of it: Query 4 (grader report) from Part 1 and function `num_groups` from Part 2.

Learning Goals

The purpose of this assignment is to give you practise writing complex stand-alone SQL queries, experience using psycopg2 to embed SQL queries in a Python program, and a sense of why a blend of SQL and a general-purpose language can be the best solution for some problems.

By the end of this assignment you will be able to:

- read and interpret a novel schema written in SQL
- write complex queries in SQL
- design datasets to test a SQL query thoroughly
- embed SQL in a high-level language using psycopg2 and Python
- quickly find and understand needed information in the postgreSQL documentation and the psycopg2 documentation
- recognize limits to the expressive power of standard SQL

Please read this assignment thoroughly before you proceed.

We will be testing your code in the CS Teaching Labs environment using PostgreSQL. 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 the CS Teaching Labs will not receive credit**.

Introduction

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/`). You have all used MarkUs, so you have experienced some of its features.

Getting to know the schema

Read the schema, which is available on Quercus.

The type of the `group_id` attribute in table `AssignmentGroup` is `SERIAL`. Read the demo, posted in file `serial-demo.txt`, to learn how SERIAL works. You will harness this feature in Part 2 when you write method `createGroups`, which needs to generate a series of group IDs.

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 members of an assignment group be given different grades?
- 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 the result be released before a grade was assigned?

Part 1: SQL Queries

General Requirements

To ensure that your query results match the form expected by the autotester (attribute types and order, for instance), We are providing a schema for the result of each query. These can be found in files `q1.sql`, `q2.sql`, ..., `q10.sql`. You must add your solution code for each query to the corresponding file.

At the beginning of the DDL file defining our database schema, we set the search path to `markus`, so that the full name of every table etc. that we define is actually `markus.`*whatever*. We have set the search path to `markus` at the top of the query files too, so that you do not have to use the `markus` prefix throughout. Do not change this, or any of the provided code.

You are encouraged to use views to make your queries more readable. However, you must make sure that each file is 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`.

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.

Your code must work on *any* database instance (including ones with empty tables) that satisfies the schema.

The queries

These queries are quite complex, and we have tried to specify them precisely. If behaviour is not specified in a particular case, we will not test that case.

Design your queries with the following definitions in mind:

- A *solo group* is a group with one member.
- We say that *a grader was declared for an assignment* if they have been assigned to grade at least one group on that assignment. They may not yet have assigned the group(s) any grades.
- We say *grading for a group is complete*, or *the group is graded*, if there is a row for that group in the `Result` table, regardless of the value of the `released` attribute.
- We say that *grading for an assignment is complete* if grading for every group declared for that assignment is complete.
- When calculating the average grade for an assignment, include only the grades of groups for which grading is complete.

Write SQL queries for each of the following:

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

   For each assignment, report the average grade across graded groups, and the number of groups with grades in each of several ranges (see below). Where there were no grades in a given range, report 0.

Attribute	
assignment_id	The assignment ID
average_mark_percent	The average grade across graded groups for this assignment, as a percent, or `NULL` if grading is not completed for any of the groups.
num_80_100	The number of groups with a grade in the range $[80 - 100]$
num_60_79	The number of groups with a grade in the range $[60 - 80)$
num_50_59	The number of groups with a grade in the range $[50 - 60)$
num_0_49	The number of groups with a grade in the range $[0 - 50)$
Everyone?	Every assignment ID should appear, even if there are no grades associated with it yet.
Duplicates?	No assignment ID 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.

   Find graders who meet all of these criteria:

   - They have been assigned to grade 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). 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.

   Report the grader's username and name, the average of their average grade on each assignment (e.g., if the assignments are Warmup, Essay1, and Essay2, report that average of the following averages: that grader's Warmup average, their Essay1 average, and their Essay2 average), 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.)

   For this query, You may assume that there are at least two assignments, and that no two assignments have the same due date; this means that there is unambiguously one first assignment and one last assignment.

Attribute	
grader_username	The username of a grader who meets the criteria
grader_name	The grader's name, in this form: first name then surname with a blank in between e.g., `'Diane Horton'`
average_mark_all_assignments	The average, across assignments, of this grader's average grade. (Each average grade should be taken as a percentage so that averaging them makes sense.)
mark_change_first_last	The increase between the grader's 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 that meet all of the following criteria:

   - Grading is complete.
   - There was at least one student who worked solo and there was at least one multi-member group. (This permits the following comparison to be made.)

- 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 as a percentage, the number of students (not groups) declared to be working in multi-member groups and the average grade across those groups as a percentage, (not students; so even if a group has three students, its grade will contribute only once to the average), and finally, the average number of students involved in each group.

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 percentage grade among students who worked alone.
num_collaborators	The number of students (not groups) declared to be working in groups
average_collaborators	The average percentage grade across those groups (not students).
average_students_per_group	The average number of students involved in each group (include in this calculation both solo groups and multi-member groups).
Everyone?	Include only assignments that meet the criteria.
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 for that assignment
num_not_marked	The number they have been assigned for that assignment but have not yet graded
min_mark	The minimum percentage grade they have given for that assignment, or null if they have given no grades for it.
max_mark	The maximum percentage grade they have given for that assignment, or null if they have given no grades for it.
Everyone?	Every assignment that has grader(s) declared should appear once for every grader who has been assigned to it.
Duplicates?	No assignment-grader pair should appear more than once.

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 once for every grader who has been assigned to it.
Duplicates?	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 You can use subtraction on values of type TIMESTAMP to compute this time-between value; do not round it, for instance, to a number of days.

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 or more files could be submitted at the same time. In that case, report a row for every first-last combination for the group. For instance, if a group has three first files and two last files, there will be six rows for that group.

Similarly, if a group has N files submitted, but all were submitted at the same time, and hence are their first and last submissions, the group will have N^2 rows.

Attribute		
group_id	The ID of an A1 group	
first_file	The name of the first file submitted by anyone in the group	or null if no file was submitted
first_time	The timestamp for its submission	
first_submitter	The username of the group member who submitted it	
last_file	The name of the last file submitted by anyone in the group	
last_time	The timestamp for its submission	
last_submitter	The username of the group member who submitted it	
elapsed_time	The time between the first and last submission for this group (as type `INTERVAL`)	
Everyone?	Every group defined for A1 should appear.	
Duplicates?	A group may occur more than once if there is a tie for its first and/or last submission.	

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

   Report the username of all graders who meet all of the following criteria:

   - For every assignment, the grader has been assigned at least one group (The group could be solo or larger, and they needn't have assigned a grade to any of the groups.)
   - For all students, there exists at least one assignment such that the grader has been assigned to grade that student. The student may have worked solo or as part of a group, and the grader need not have completed grading them.

Attribute	
grader	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.

   Find students who have submitted at least 1 file for every assignment and who have been part of a multi-member group for every assignment that allowed it. Report their username, their average grade on the assignments that allowed groups, and their average grade on the assignments that did not allow groups.

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

Attribute	
username	The username of a student who meets the criteria
group_average	Their average percentage grade on the assignments that allowed groups. (Assume that at least one assignment allows groups.)
solo_average	Their average percentage 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 were part of a multi-member group whenever the assignment permitted it, and always worked together (possibly with other students in a larger group).

   Note: The result will also be empty if no assignment allows multi-member groups or if one or more assignments that allow multi-member groups has no groups declared.

HINT: SQL allows you to compare strings lexicographically (in the order they would appear in the dictionary). Consequently, you can use comparison operators ($>$, $<$, $=$, $\geq$, $\leq$) or `ORDER BY` on string attributes in the same way you use them on integer attributes. For example, the result of `select 'apple' > 'banana';` would be False, since `'apple'` would appear first in the dictionary.

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.

For each group for whom grading is complete on assignment A1, compute their grade as a percentage, the difference between their grade and the average A1 grade across groups (not users), and either "above", "at", or "below" to indicate whether they are above, at or below this average.

Do not round any values. This means that there may not be many groups whose grade is at the 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 as percentage points (negative if they are below average; positive if they are above average). 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.

For this query, you may assume that the grading for at least one group is complete for A1 (so there is an average to compare to).

SQL Tips

- There are many details of the SQL library functions that we are not covering. It's just too vast! Expect to use the postgreSQL documentation to find the things you need. Chapter 9 on functions and operators is particularly useful. Google search is great too, but the best answers tend to come from the official postgreSQL documentation.
- When subtracting timestamp values, you get a value of type INTERVAL. If you want to compare to a constant time interval, you can do this, for example:

```
WHERE (a - b) > INTERVAL '0'
```

- You might find the following helpful to make your solutions more succinct.:
  - `COALESCE(value [, ...])` to return the first of its arguments that is not null.
  - A `CASE` expression to generate a different value based on a condition:

```
CASE WHEN condition THEN result
[WHEN ...]
[ELSE result]
END
```

  The PostgreSQL documentation on functions and conditionals covers these in more details.
- Please use line breaks so that your queries do not exceed an 80-character line length.

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. Operations in MarkUs can be implemented by Python 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 psycopg2 and to demonstrate the need to get Python involved, not only because it can provide a nicer user-interface than postgreSQL, but because of the expressive power of Python.

General requirements

- The methods we have asked you to write (`clock_in, pick_up, and dispatch`), and any helper methods they call, must not take input from the user or from a file. Doing so will result in the autotester timing out, causing you to receive a **zero** on that method. You can take input, however, in your main block and testing functions.
- Do not change any of the code provided. In particular, you may not change the header of any of the methods we've asked you to implement. Each method must have a try-except clause so that it cannot possibly throw an exception.
- You have been provided with methods called `connect()` and `disconnect()` that allow you to respectively connect to and disconnect from the database. You must **NOT** make any modifications to either method. You should also not modify the private method `_register_geo_loc()`.
- You should **NOT** call `connect()` and `disconnect()` 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 `a2.py`. This is the only file you may submit for this part.
- You are welcome to write helper methods to maintain good code quality.
- Within any of your methods, you are welcome to define views to break your task into steps. Drop those views before the method returns, or otherwise a subsequent call to the method will raise an error when it tries to define a view that already exists. Alternatively, you can declare your view as temporary so that it is dropped automatically once the connection is closed. The syntax for this is `CREATE TEMPORARY VIEW name AS ...`
- Your methods should do only what the docstring comments say to do. In some cases there are other things that might have made sense to do but that we did not specify (in order to simplify your work). Don't do those extra things.
- If behaviour is not specified in a particular case, we will not test that case.

Your task

Complete the following methods in the starter code in `a2.py`:

1. `num_groups`: Returns the number of groups defined for an assignment.
2. `assign_grader`: Assigns a grader to a group for an assignment.
3. `remove_student`: Removes a student from a group for an assignment.

4. `create_groups`: Creates student groups for an assignment.

You will have to decide how much to do in SQL and how much to do in Python. 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 Python and then do all the real work in Python. 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. In particular, there is no need to use Python data structure such as dictionaries, sets, or even simple lists for temporary storage.

We don't want you to spend a lot of time learning Python for this assignment, so feel free to ask lots of Python-specific questions as they come up.

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 colleagues 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 Python tips

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

Some of your SQL queries may be very long strings. You should write them on multiple lines, both for readability and to keep your code within an 80-character line length. You can achieve that by breaking the string into pieces and using `+` to concatenate them together. You would need to put a blank at the end of each piece so that when they are concatenated you will have valid SQL. Example:

```
sql_query =
   "SELECT client_id " +
   "FROM Request r JOIN Billed b ON r.request_id = b.request_id " +
   "WHERE amount > 50";
```

It is much easier to do this using Multi-line strings in Python. Multi-line strings are declared using triple quotes, and are allowed to span multiple lines.

```
sql_query = """
        SELECT client_id
        FROM Request r JOIN Billed b ON r.request_id = b.request_id
        WHERE amount > 50
   """
```

How your work will be marked

This assignment will be entirely marked via auto-testing. Your mark for each part will be determined by the number of test cases that you pass. To help you perform a basic test of your code (and to make sure that your code connects properly with our testing infrastructure), we will provide some self-tests that you can run through MarkUs. We will of course test your code on a more thorough set of test cases when we grade it, and you should do the same.

The exact division of grades for the full A2 has not yet been decided, but this was the division last year:

Part 1 (SQL queries)	self-tests on MarkUs	1
	full test suite	22
Part 2 (Embedded SQL code in Python)	self-tests on MarkUs	1
	full test suite	8
TOTAL		32

Some advice on testing your code

Testing is a significant task, and is part of your work for A2. You'll need a dataset for each condition / scenario you want to test. These can be small, and they can be minor variations on each other. Suggestion: Use names that indicate something about the scenario, like "Tough Grader" for a TA who gives a lot of low marks.

We suggest you start your testing for a given query by making a list of scenarios and giving each of them a memorable name. Then create a dataset for each, and use systematic naming for each file, such as `q1-all-high-grades`. Then to test a single query, you can:

1. Import the schema into psql (to empty out the database and start fresh).
2. Import the dataset (to create the condition you are testing).
3. Then import the query and review the results to see if they are as you expect.

Repeat for the other datasets representing other conditions of interest for that query.

Testing your embedded SQL code can be done in a similar way to the "Coordinating" part of the Embedded exercises posted on our Lectures page. I recommend being very organized, as described above. In this case, to test a method on a particular dataset:

1. Have two windows logged in to dbsrv1.
2. In window 1, start psql and import the schema (to empty out the database and start fresh) and then the dataset you are going to test with.
3. In window 2 (remember, this is on dbsrv1), modify the main block of your a2 program so that it has an appropriate call to the method you are about to test. Then run the Python code.
4. Back in psql in window 1, check that the state of your tables is as you expect.

Don't forget to check the method's return value too.

You may find it helpful to define one or more functions for testing. Each would include the necessary setup and call(s) to your method(s). Then in the main block, you can include a call to each of your testing functions, comment them all out, and uncomment-out the one you want to run at any given time.

Your main block and testing functions, as well as any helper methods you choose to write, will have to effect on our auto-testing.

Submission instructions

You must declare your team on MarkUs even if you are working solo, and must do so before the due date. If you plan to work with a partner, declare this as soon as you begin working together.

For this assignment, you will hand in numerous files. MarkUs shows you if an expected file has not been submitted; check that feedback so you don't accidentally overlook a file. Also check that you have submitted the correct version of your file by downloading it from MarkUs. New files will not be accepted after the due date.