University of Toronto
CSC343, Fall 2024

Assignment 2

Due: Wednesday, October 23, before 4:00 p.m.

Learning Goals

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

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

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

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**.

The Domain

In this assignment, we will work with a database to support a veterinary clinic, using a schema similar (but not exactly identical) to the one you used in Assignment 1. Keep in mind that your code for this assignment must work on *any* database instance (including ones with empty tables) that satisfies the schema.

All values of type time, timestamp etc. in the dataset provided are on a 24-hour clock.

Begin by getting familiar with the schema that we have provided `a2_vet_schema.ddl`. This schema is nearly identical to the one from the A2 warmup.

Part 1: SQL Queries

General requirements

In this section, you will write SQL statements to perform queries.

To ensure that your query results match the form expected by the auto-tester (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`, ..., `q5.sql`. You must add your solution code for each query to the corresponding file. Make sure that each file is entirely self-contained, and not dependent 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 queries

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

Write SQL queries for each of the following:

1. **Appointment Reminders**

   An "active" patient is one that has had any appointment in the last three calendar years, based on the current year (e.g, in 2024, consider appointments in 2022, 2023, and 2024). Find all active patients that have had a "diagnostic testing" procedure at least once per calendar year since their first appointment ever, but have not had one yet or have one scheduled this calendar year. Report the client ID, client name, email, and patient name.

Attribute	
c_id	The ID of a client
client_name	The name of the client
email	The email of the client
patient_name	The name of the patient who is overdue
Everyone?	Include only active patients who are overdue. If there are no such patients, the result will be empty.
Duplicates?	Yes, there can be duplicate rows, but each distinct patient should only be included once.

2. **Employee Activity**

   Generate a summary of employee activity over *all time* represented in the database. For every employee in the clinic, generate the information in the following tables. For every value below, report 0 or 0.0 (not NULL) for missing data, unless explicitly specified otherwise. Assume that an employee has "worked with" a client/patient/other employee if they were scheduled on the same appointment. You will likely find `COALESCE` useful here.

Attribute	
e_id	The employee ID
name	The employee name
hire_year	The year the employee was hired
num_appointments	The total number of appointments the employee has worked.
days_worked	The total number of days worked since the employee's start date.
avg_appointment_len	The average length of the appointments they have worked on. Report an interval of 0 hours if they haven't worked on any appointments.
clients_helped	The total number of distinct clients the employee has worked with.
patients_helped	The total number of distinct patients the employee has worked with.
num_coworkers	The total number of distinct other employees the employee has worked with.
total_supplies	The total quantity of units of supplies used by the employee.
Everyone?	Yes, include every employee in the database.
Duplicates?	No.

3. **Overworked vet techs**

   Assume that the number of hours an employee works in a day is defined by the start time and end times of the appointments they work. For example, if an employee works on three appointments in the same day, one from 9:00-10:30, one from 10:30-10:45, and one from 13:30-14:30, then they have worked a total of 2.75 hours that day, in two consecutive blocks of 1.75 and 1 hours respectively. You can assume that there are no overlapping appointments for an employee, and we do not consider appointments like 9:00-10:30 and 10:30-10:45 to be overlapping.

   We'll define an "exhausting day" as a day where the employee worked a total of 8 consecutive hours or more. Find all vet techs (RVTs) who had a least three weeks with at least two exhausting days per week. A week is defined as Monday to Friday, inclusive. Report the employee IDs and their total interval worked.

   Note:

- `date_trunc('week', '2024-10-9'::date)::date` would return the start of the week i.e., `2024-10-07`
- You can perform arithmetic on interval types.

Attribute	
e_id	the employee ID of this vet tech
time_worked	The total interval of time worked
Everyone?	No, only "overworked" vet techs
Duplicates?	No, each employee should appear at most once in the results.

4. **Mentorship**

   The clinic wants to establish a mentorship program where new employees who have worked there for less than 90 days, and whose start date is no later than the current date, ("mentees") are paired with more experienced employee who has worked there for at least 2 years ("mentor"). Mentees are paired with a mentor who has worked with all of the species the mentee has since they started. A mentor can be paired with multiple mentees, and vice versa, and it is possible there may not be mentors for every new employee. Report the IDs of the mentee and mentor. Dates are based on their start date relative to the current date. Use the `AGE` function to check for a 2 year interval. For simplicity, we use a time resolution of only years for mentors, i.e., an employee who has worked for 1 year and 364 days won't qualify for the mentor position.

Attribute	
mentee	the employee ID of the new employee
mentor	the employee ID of the experienced employee or NULL if no match exists.
Everyone?	Include every employee hired in the last 90 days who has had at least one appointment.
Duplicates?	No duplicate rows, but mentees and mentors can appear multiple times.

5. **Complex cases**

   An appointment is considered "complex" if it takes more than twice the average amount of time taken for appointments for patients of the same species. A patient is considered "complex" if they have at least one complex appointment.

   Find the complex patient(s) that has had the most complex appointments and how many complex appointments they had. If no patients are complex, then all patients should be reported with 0 (not NULL) as the number of complex appointments.

Attribute	
p_id	the ID of the patient who meets the criteria of this question.
num_complex	The total number of complex appointments that patient has had.
Everyone?	Include only patients who meet the criteria of this question.
Duplicates?	No patient can be included more than once.

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 postgreSQL documentation.
- When dealing with a value of type TIMESTAMP, DATE, or TIME, the EXTRACT function is handy for pulling out pieces.
- The CASE statement and COALESCE turns out to be quite useful.
- You may use any features defined in our version of PostgreSQL on the Teaching Labs. This is not a pointed hint; we have not deliberately left features for you to discover that will dramatically simplify your work. However, we do expect that you will need to look up details in the documentation, and in fact being able to do that quickly and effectively is one of the learning outcomes of the assignment.
- Please use line breaks so that your queries do not exceed an 80-character line length.

Part 2: SQL Updates

General requirements

In this section, you will write SQL statements to perform updates. Some questions can be accomplished with a single DELETE, UPDATE, or INSERT statement, but others will require several steps.

Since you are modifying tables from the schema we have provided you, rather than finding results that are then added to a table for the autotester, we have not provided you with starter code files for this section.

You should name your files for the updates below `u1.sql`, `u2.sql`, and `u3.sql`.

Make sure that each file is entirely self-contained, and not dependent on any other files; each will be run separately on a fresh database instance.

The updates

Write SQL statements for each of the following:

1. **Big Sale!** (`u1.sql`) Reduce the price of every "food" supply by 50%. Do not round the resulting price.
2. **Cats Don't Like Needles** (`u2.sql`) The staff at the clinic have found that cats become especially uncooperative after they have bloodwork done, making it hard to complete any following procedures.

   For any appointments on or after Nov 1, 2024 for cats, reorder the procedures so that all "blood work" procedures happen as the last procedure(s). Maintain the relative order of the rest of the procedures.

   For this problem, it is OK to temporarily violate the assumptions noted as comments in the DDL file, although all those assumptions must hold in your final result. You may find `generate_series` helpful here, but other solutions exist as well.
3. **Client Cleanup** (`u3.sql`) Delete all information related to any client that has not had any appointments since January 1, 2019. If a client has no pets recorded in the clinic, they are considered to have not had any appointments since January 1, 2019.

Part 3: Embedded SQL

Imagine creating software for a veterinary clinic that provides clients and staff with different features, such as booking appointments and scheduling staff. The users will probably issue these requests through a graphical user-interface, but ultimately it has to connect to the database where the core data is stored. Some of the features will be implemented by Python methods that are merely a wrapper around a SQL query, allowing input to come from gestures the user makes, like button clicks, and output to go to the screen via the graphical user-interface. Other features will include computation that can't be done, or can't be done conveniently, in SQL.

For Part 3 of this assignment, you will not build a user-interface, but will write several methods that such a vet clinic website/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

- You may not use standard input in the methods that you are completing. Doing so will result in the autotester timing out, causing you to receive a **zero** on that method. (You can use standard input in any testing code that you write outside of these methods, however.)
- You may not change the header of any of the methods we've asked you to implement, not even to declare that a method may throw an exception. Each method must have a try-except clause so that it cannot possibly throw an exception.

- You should not hardcode your connection information anywhere in the `VetClinic` class. Our autotester will use the `connect()` and `disconnect()` methods to connect to the database with our own credentials.
- You should **not** call `connect()` and `disconnect()` in the 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.
- Do only what the method docstring says 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.
- If behaviour is not specified in a particular case, we will not test that case.
- Do not write any code outside of a function/method or the main block.

Your task

Complete the methods that we have documented in the starter code in `a2.py`.

1. `calculate_vacation_credit`: A method that would be called to compute the vacation that each employee has accumulated over the course of their employment at the clinic.
2. `record_employee`: A method that would be called to add a new employee and information about them to the database.
3. `reschedule_appointments`: A method that would be called to reschedule a particular employee's appointments from one date to another.

You will have to decide how much to do in SQL and how much to do in Python. 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 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.

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.

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.

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. 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 a checker that you can run through MarkUs. If the checker passes, this tells you only that your code runs and that it passes a basic set of tests. 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. Your work will be marked only for correctness.

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.

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-single-patient`. 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 Embedded exercise posted on our Lectures page. We 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 no effect on our auto-testing. Do not write any code outside of a function or the main block!

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. If you need to dissolve your group, you need to contact us. The deadline for resolving any issues with groups is October 21.

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.

This assignment will be autotested. It is your responsibility to make sure your filenames and contents are what you expect. There are no remarks on autotested assignments.