University of Toronto
CSC343, Winter 2025

Assignment 2

Due: February 26 at 4pm

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 airline travel, 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_airtravel_schema.ddl`. This schema is identical to the one from the A2 warmup.

Part 1: SQL Queries (50%)

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. **Route fullness**

   The percentage fullness of the plane is defined as $\frac{\text{The number of booked seats}}{\text{The number of available seats}} \times 100$. Create a table that is, essentially, a histogram of the percentages of how full planes have been on the flights that they have made (that is, only the flights that have actually departed, regardless of whether it arrived at the destination). The lower bounds in the ranges below are inclusive while the upper bounds are non-inclusive.

Attribute	
airline	The airline operating the route.
flight_num	The flight number identifying the route.
very_low	The number of flights for that route whose plane had between 0% and 20% capacity.
low	The number of flights for that route whose plane had between 20% and 40% capacity.
fair	The number of flights for that route whose plane had between 40% and 60% capacity.
normal	The number of flights for that route whose plane had between 60% and 80% capacity.
high	The number of flights for that route whose plane had 80% capacity or more.
Everyone?	Every route should be included once, even if no flights have departed for that route. Routes with no departed flights should have 0 in all categories.
Duplicates?	Each route should be included at most once. However, an airline might be included multiple times (once for each route they operate).

2. **Airport type**

   An airport's monthly average for inbound traffic is the average number of incoming flights scheduled to arrive to that airport in any month. Consider only the months when atleast one flight was scheduled to arrive at any airport in our database.
   Similarly, an airport's monthly average for outbound traffic is the average number of flights scheduled to depart from that airport in any month. Consider only the months when atleast one flight was scheduled to depart from any airport in our database.
   Note that for the above two definitions, we are using the scheduled times, regardless of whether the flight actually departed from its source or arrived at its destination. An airport is considered to have heavy inbound traffic if its monthly inbound traffic average is more than 75% of the inbound traffic average across all airports. Similarly, an airport is considered to have heavy outbound traffic if its monthly outbound traffic average is more than 75% of the outbound traffic average across all airports.

   Based on the above, we will define the following categories of airports:

   - Heavy-inbound: If the airport has heavy inbound traffic only.
   - Heavy-outbound: If the airport has heavy outbound traffic only.
   - Hub: If the airport has both heavy inbound and outbound traffic.

   For each airport, report their category, as well as the number of flights that constitute their inbound and outbound traffic.

Attribute	
code	The airport identifying IATA code.
name	The airport's name.
category	One of “heavy-inbound traffic”, “heavy-outbound traffic” or “hub” as described above. Use “N/A” for airports that don't fit in any of the above categories.
inbound_traffic	The number of distinct incoming flights to this airport over the entire duration (for arrivals) recorded in our database, or zero if none exists.
outbound_traffic	The number of distinct outgoing flights from this airport over the entire duration (for departures) recorded in our database, or zero if none exists.
Everyone?	Yes, include every airport in the database.
Duplicates?	No.

3. **Make Recommendations**

We will define a passenger's *travel log* as the list of unique cities a passenger has already visited i.e., the passenger must have *booked* a flight that *arrived* at that city i.e., you should only consider flights that have already arrived at their destination. For simplicity, this will include any city to which the passenger had a flight i.e., we will not make a distinction between layovers and final destinations.

Passenger P_1 is considered to be a *similar traveler* to another passenger P_2 if strictly more than 50% of P_2's *travel log* matches P_1's *travel log.* Note that this relationship is not necessarily symmetric.

Recommendations to a passenger are all the cities in a *similar traveler*'s *travel log*, but not in their *travel log.*

Attribute	
pid	the passenger ID.
name	The passenger's name in the form "⟨first_name⟩ ⟨last_name⟩".
city	The city name recommended to this passenger in the form "⟨city name⟩, ⟨country name⟩".
Everyone?	No, only passengers with at least one recommendation.
Duplicates?	A passenger might appear multiple times (once for each recommendation), but passenger, city pairs should only occur once in the result.

4. **Airline information**

For each airline, report the following.

Attribute	
airline	the two-character airline IATA code.
num_planes	The number of planes owned by the airline. If the airline doesn't own any planes, report 0, not NULL.
num_routes	The number of routes operated by the airline. If the airline doesn't operate any routes, report 0, not NULL.
avg_flights	The average number of flights per route scheduled by the airline (regardless of whether the flight departed or not). If the airline doesn't operate any routes, or no flights are recorded for this airline, report NULL.
avg_delay	The average delay across all flights operated by the airline for departed flights, regardless of arrival status. If the airline has no flights or has no departed flights recorded, report NULL.
avg_first_price	The average price *paid* across all bookings from all flights operated by the airline for first-class seats. Consider all bookings, regardless of the flights' departure status. If the airline has no flights recorded, or no bookings for first-class seats, report NULL.
avg_business_price	The average price *paid* across all bookings from all flights operated by the airline for business-class seats. Consider all bookings, regardless of the flights' departure status. If the airline has no flights recorded, or no bookings for business-class seats, report NULL.
avg_economy_price	The average price *paid* across all bookings from all flights operated by the airline for economy-class seats. Consider all bookings, regardless of the flights' departure status. If the airline has no flights recorded, or no bookings for economy-class seats, report NULL.
Everyone?	Include every airline recorded in the database.
Duplicates?	No.

5. **Frequent flyers**

We aim to identify frequent flyers, who haven't booked any flights in 2025.

Find flyers who have *visited* at least 5 distinct countries (you may assume country names are unique), have *booked* at least 10 distinct flights in 2023 (the booking date is in 2023), have *booked* more distinct flights in 2024 (the booking date is in 2024) than in 2023, and have *booked* no flights for 2025.

By *visited*, we mean that they have booked a flight, that arrived at its destination. By *booked*, we mean that they have made a booking, regardless of whether the plane departed from the source or arrived at the destination.

Note that the sample data provided to you will not produce any rows in the result for this query. It is your responsibility to do thorough testing of your queries.

Attribute	
pid	The passenger ID.
email	The passenger's email.
Everyone?	Include only passengers that satisfy the above criteria.
Duplicates?	No.

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 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 '1 hour'
  ```

- When dealing with a value of type `TIMESTAMP`, `DATE`, or `TIME`, the `EXTRACT` function is handy for pulling out pieces.
- You might also find `date_trunc` useful.
- The `CASE` statement and `COALESCE` are also 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 (20%)

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` and `u2.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 prices of all flights operated by "Air Canada" (IATA code: "AC"), scheduled to depart between June 1, 2025 and August 31, 2025 (inclusive) to destinations in the United States and Canada as follows:

- First-class seats are reduced by 20% (new price is 0.8 times the old price).
- Business-class seats are reduced by 30% (new price is 0.7 times the old price).
- Economy-class seats are reduced by 40% (new price is 0.6 times the old price).

Only change the prices listed in the `FlightPrice` table i.e., don't change existing bookings for these flights.

2. **Decommissioned planes** (`u2.sql`) Delete all information related to planes that have not been used since December 31, 2021, based on the actual arrival time i.e., planes, whose most recent actual arrival time was on Dec 31, 2021 or earlier. A plane on a flight scheduled to arrive on Dec 31, 2021 that actually arrived on January 1, 2022 (or later) should not be deleted.

Part 3: Embedded SQL (30%)

Imagine creating software for airline travel that provides passengers and staff with different features, such as scheduling and booking flights. 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 air travel 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 `AirTravel` 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_embedded.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_embedded.py`.

1. `make_booking`: A method that would be called to make a booking for a passenger.

2. `find_unreachable_from`: A method that would be called to find all airports unreachable from a target airport.
3. `reassign_plane`: A method that would be called to find alternative planes for a subset of flights.

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 route
    FROM Booking B JOIN Flight F ON B.flight = F.fid
    WHERE price < 500
"""
```

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-empty-flight`. 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 February 24 (2 days earlier than deadline).

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.