University of Toronto
CSC343, Winter 2022

Assignment 2

Due: Sunday, March 13, before 4pm

Before you start: Academic Integrity

Academic integrity is taken very seriously by the University. As this assignment is worth more than 10% of your course grade, any allegations of academic dishonesty go immediately to the Dean, and may take a long time to resolve. This is true whether you copied the work or ideas of someone else, or allowed someone to look at and copy your work.
Only show your work to your assignment partner or the course teaching team, and do not look at the work of anyone else.
If you have questions or need help, use the resources provided in the course.

Learning Goals

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
- quickly find and understand needed information in the postgreSQL and psycopg2 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 of **international airports**. Keep in mind that your code for this assignment must work on *any* database instance (including ones with empty tables) that satisfies the schema.

We have provided a small set of example data, will be up to you to make your own additions to the dataset to test your queries thoroughly (which you can do by adding rows to the given CSV files, or using INSERT statements in Postgres).

You may download the code from Quercus in `data.zip`, or copy the files on the Teaching Labs using the following command:

```
> cp -r ~csc343h/winter/pub/assignments/a2/* .
```

Part 1: SQL Statements

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 autotester (attribute types and order, for instance), we are providing a schema template 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 does 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 queries

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

Design your queries with the following in mind:

- When we say that a passenger *took a flight*, we mean that the flight was completed, that is, it has gone from departure to arrival. You can assume that if a departure was made, an arrival also occured.
- Each flight has *scheduled* departure and arrival times, which are the times presented to the passenger upon booking a flight. The *actual* departure and arrival times are specified in the Departure and Arrival tables.
- The date that a passenger took the flight is the date of the actual departure (not the scheduled departure time).
- All timestamps are in one time zone (UTC).

Write SQL queries for each of the following:

1. **Airlines.** For each passenger, report their passenger ID, full name, and the number of different airlines on which they took a flight.

Attribute	
pass_id	id of a passenger.
name	full name of the passenger (for example, Kelly Watts).
airlines	the number of different airlines on which they took a flight.
Everyone?	Every passenger should be included, even if they have never taken a flight.
Duplicates?	No passenger can be included more than once.

2. **Refunds!** Airlines offer refunds on the price paid for a flight booking for late departures as follows:
For domestic flights (flights within the same country), a 35% refund is given for a departure delay of 5 hours or more, or a 50% refund for 10 hours or more.
For international flights (flights between different countries), a 35% refund is given for a departure delay of 8 hours or more, or a 50% refund for 12 hours or more.
(the refunds are not additive, you cannot get both a 35% refund and a 50% refund).

   However, if the pilots manage to make up time during the flight and have an arrival delay that is at most *half* of the departure delay, then no refund is provided, regardless of how many hours the delay was.

   For every airline and year that the airline had flights which required refunds, return the **total** refund money given in that particular year in each seat class. This means there should be at most three records per airline-year combination (you should not include a seat class if no one booked it for any refunded flight). The year of a flight comes from its scheduled departure date.

Attribute	
airline	the airline code.
name	the airline name.
year	a specific year.
seat_class	one of the three seat classes (economy, business, first).
refund	the total money refunded to that seat class that year.
Everyone?	Every airline-year-seat_class combination should be included in each year the airline had a flight which required refunds, unless a seat class had no bookings that year for a refunded flight, then you should not include that seat class.
Duplicates?	No duplicates.

3. **North and South Connections.** Corporate passengers that travel between the US and Canada sometimes want to get to their destination on the same day they left, and as early as possible. Because of limited same-day flight options, they may also have to connect through one or two different airports before reaching their destination.
   Consider the day April 30, 2022 (2022-04-30). For each pair of outbound/inbound **cities** (not airports - a city can have multiple airports) between Canada and the USA in **both** directions, list the number of possible different routes that a passenger can take on that day to get from the outbound city to the inbound city. You should list the numbers in three different columns: one for the number of direct flight routes (no connections), one for the number of one-connection routes, and one for the number of two-connection routes. In addition, you should include the **earliest possible** arrival time to the inbound city from all of the routes you found.

   Note: A minimum connection time of at least **30 minutes** is required so passengers have a chance to get from one plane to the next. Connections must be made from the **same** airport.

Attribute	
outbound	the outbound city.
inbound	the inbound city.
direct	the total number of possible direct routes that start and end on April 30, 2022.
one_con	the total number of possible one-connection routes that start and end on April 30, 2022.
two_con	the total number of possible two-connection routes that start and end on April 30, 2022.
earliest	the earliest possible arrival time (full timestamp) from all of the direct, one-connection, and two-connection routes. NULL if you did not find any routes.
Everyone?	Include **all** pairs of cities between Canada and the USA in **both** directions (e.g. Toronto to New York **and** New York to Toronto).
Duplicates?	No duplicate outbound/inbound city pairs.

4. **Plane capacity histogram.** 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). The capacity of a plane includes the seats from all of its seat classes. The upper bounds in the ranges below are non-inclusive.

Attribute	
airline	the airline of the plane.
tail_number	the tail number of the plane.
very_low	The number of flights that the plane had between 0% and 20% capacity.
low	The number of flights that the plane had between 20% and 40% capacity.
fair	The number of flights that the plane had between 40% and 60% capacity.
normal	The number of flights that the plane had between 60% and 80% capacity.
high	The number of flights that the plane had more than 80% capacity.
Everyone?	Every plane should be included, even if they haven't flown at all.
Duplicates?	No plane can be included more than once.

5. **Flight hopping.** For this query, you are going to learn about and use two interesting features of PostgreSQL: Common Table Expressions and the RECURSIVE clause.

   A **Common Table Expression** (CTE) is very similar to a VIEW, except instead of saving the view in your schema, you simply use the expression immediately.
   You can define a CTE using the WITH clause, and then use it immediately, all within **one** query execution:

```
WITH air_canada_flights AS           -- air_canada_flights is the CTE
(SELECT * FROM Flight f
 WHERE f.airline = ‘AC’ )

SELECT * from air_canada_flights;
```

   Notice how the semicolon is at the end of the entire query, showing that the CTE declaration (WITH) and the subsequent SELECT statement all happen together, as opposed to separately as in the case of a VIEW.

   The RECURSIVE clause allows you to start with a base CTE, and use it to write recursive queries. For example, the query below counts the numbers from 1 to 10:

```
WITH RECURSIVE numbers AS (
   (SELECT 1 AS n)   -- the base query
   UNION ALL
   (SELECT n + 1 FROM numbers WHERE n < 10)  -- the recursive query
)
SELECT * FROM numbers;
```

   The base query starts the count by setting the numbers CTE to have only the number 1. Then that CTE is fed into the recursive query, and a table with only the number 2 is produced and assigned to numbers. Then that table is fed into the recursive query and we get 3, and so on until the terminating condition, WHERE n < 10, stops the execution of the recursive queries. We then do a UNION ALL to include all the records from every execution of the recursive query.

   You can read more about the RECURSIVE clause here and find examples on how to use it:
   https://www.postgresqltutorial.com/postgresql-recursive-query/

   “Flight hopping” is defined as getting on a plane, landing at a destination, and then within 24 hours getting on another plane from that destination. You can keep doing this, hopping from flight to flight, as long as the time interval between the arrival of one flight and the departure of the next one is less than 24 hours.

   Suppose we start at Toronto’s Pearson Airport (YYZ). For a given date and maximum number of flights, list **all** possible airports (not cities) you can get to by flight-hopping, and the number of flights it would take to get to each airport. You should include all airports you can get to from YYZ, up to and including the maximum number of flights given. Do not include the ‘0 flights’ case.
   Keep all airports, even if you could get to them by a different number of flights (up to the max).
   Use only scheduled departure and arrival times. We will not enforce a minimum connection time like query 3.

Attribute	
destination	an airport code for the final destination of a flight-hopping session.
num_flights	the number of flights it took to get to that airport through 24-hour flight hopping.
Everyone?	Include all airport destinations you can flight hop to (including the origin airport, YYZ, which would require at least two flights). Include only those from the given start date and **all** number of flights up to the maximum. **The date and maximum number of flights will be provided from the q5_parameters relation.**
Duplicates?	Duplicates should be included if there are multiple routes that can get you to the same destination in the same number of flights.

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.

- For our autotester, make sure you are using the correct encoding for quotation characters (and any other characters) in your queries (if you copy some text from this handout rather than writing it yourself this might be an issue, so to avoid this type your answers directly into your .sql file).
- 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’
  ```

  Or, you can write an explicit time interval:

  ```
  WHERE (a - b) > ‘01:30:00’  -- greater than 1 hour and 30 minutes.
  ```
- Please use line breaks so that your queries do not exceed an 80-character line length.

Continued on next page...

Part 2: Embedded SQL

For Part 2, you will complete methods in **a2.py**. You can find the documentation for psycopg2 here: `https://www.psycopg.org/docs/`

Imagine an integrated system for an airline that allows passengers and airline workers to perform tasks. This could be something like booking flights for passengers, or seeing an overview of the plane's current seating arrangements for the check-in counter at the airport. The app could have a graphical user-interface, written in Python, 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 on the app, like button clicks, and output to go to the screen via the graphical user-interface. Other app features will 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.

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.)
- 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.
- You should **not** call `connect_db()` and `disconnect_db()` 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.
- 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.
- Carefully read the section on the next page entitled 'How seats are booked' to understand the appropriate way to write your methods.

Your task

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

1. `book_seat`: A method that would be called when a passenger wants to book a seat on a plane.
2. `upgrade`: A method that would be called to see if economy class passengers without a seat can be upgraded to business or first class when economy class is overbooked (more economy bookings than economy seats).

You should not call `upgrade` from `book_seat` or vice-versa.

How seats are booked

Seats are assigned automatically by the system when a passenger books a flight. To keep things consistent, we will constrain this process like so:

- In every plane, there are up to 6 seats per row, designated by the letters `A,B,C,D,E,` and `F`.
  For example, the seat `1E` represents the fifth seat in row `1`.
- There are three seat classes, `first` class, which starts at row `1`, followed by `business` class, and then `economy`.
- When a passenger wants to book a seat in a particular seat class, seats are assigned row by row, from A to F across the row, until the capacity of a seat class is reached. At that point, a new row starts for the next seat class.
  For example, if there are 8 seats in first class, the seats are booked in this order: `1A, 1B, 1C, 1D, 1E, 1F, 2A, 2B`. For the next class, business class, we start on a new row, so the first seat to be booked will be `3A`. If there are 15 seats in business class, the last seat in business class will be `5C`, and the first seat in economy will be `6A`.
- Economy class can be overbooked. Up to 10 seats after filling up all economy seats, passengers will not get a seat booked, and instead will have `NULL` values for the seat number and letter. After 10 NULL seats are booked, no more can be booked.
  After bookings are done, airline workers can attempt to upgrade `NULL` seats from economy to a higher seat class, starting with business, followed by first class (with seats assigned in the same order as normal booking). Upgrades are done in order of earliest booking. If there are no seats remaining in the higher classes to upgrade the overbooked seats, then those passengers will unfortunately not be able to take the flight.

You should use these constraints and assumptions when writing your code.
Read the **docstrings** in the code for more information on what you must do.

SQL vs. Python

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! Imagine if there were millions of records - they can be much better processed in SQL than in your computer's local memory. 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 psycopg2 tips

- 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. You can break the string into pieces using linebreaks and putting parentheses around the string. 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:

  ```
  cur.execute(("select * "
               "from Booking b "
               "where b.pass_id = %s;"),
            ...)
  ```

- Remember that if you are making any database modifications, you must **commit** those changes using the connection variable. for example: `self.db_conn.commit()`
- Please refer to the psycopg2 exercise from class for some common mistakes and the error messages they generate.

Submission instructions

You must declare your team on MarkUs even if you are working solo. 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.