CSC 370 - SUMMER 2020
DATABASE SYSTEMS
ASSIGNMENT 6
UNIVERSITY OF VICTORIA

Due: Wednesday, August 5th, 2020 at 11:55pm. **Late assignments will not be accepted.**

1 Overview

This assignment covers applied database design and the use of database interfaces in a high level programming language (Python). Your submission for this assignment will consist of a database definition (`CREATE` statements and accompanying constraint definitions) in SQL (submitted as a `.txt` file due to conneX limitations) and a set of short Python[1] programs to act as database front-ends. Some of the Python programs will read comma-separated input data and perform database insertion and update operations. The other programs will use `SELECT` statements to retrieve data from the DBMS and format it as a report for the user. This assignment is a small-scale example of the type of database interaction that you might normally encounter as a developer.

Although this assignment requires writing Python code, the core objective of the assignment is to **write as little Python as possible**: the Python programs you write should function as simple appliances to convert input data `INSERT` or `UPDATE` statements and convert query results to formatted text. For full marks, you will be expected to leave all of the processing logic (checking the validity of data, moderating conflicts which arise in the data entry, processing results, etc.) to the DBMS and perform no non-trivial data processing in your Python code.

Since you may not have used Python recently, some example programs have been posted which demonstrate Python's capabilities for reading comma-separated data and producing formatted output. You may also find the examples of the `psycopg2` interface from the lectures to be useful in designing your solution.

2 Computing Environment

Your Python programs will be marked based on their behavior when run in the Linux environment on `dblinux.csc.uvic.ca`. It is expected that you use your individual database on one of the CSC 370 database servers (either `studdb1.csc.uvic.ca` or `studdb2.csc.uvic.ca`) as the back-end DBMS for your assignment.

Since your programs will be marked on `dblinux.csc.uvic.ca`, you must ensure that all of the libraries and Python modules that you use are available and work correctly in that environment. Since your programs will contain very little non-trivial code, it should suffice to use the installed

1. All references to 'Python' in this document refer to the specific version of Python 3 installed as `/usr/bin/python3` on `dblinux.csc.uvic.ca`

psycopg2 module (for PostgreSQL connectivity) and the Python standard library, but you are free to use any other modules that may be available.

You are required to use Python 3 for this assignment (if you use Python 2, or any other language, your submission will not be marked).

3 The Task: Tracking Aircraft, Flights and Passengers

Your task for this assignment is to design a database schema and a set of front-end programs for a flight tracking database (similar in some ways to examples we have already seen over the semester). Your database will contain information on airports, aircraft, flights, passengers and ticket reservations. Since data input/output is managed by Python programs that you will write, you will have complete control over the database schema, unlike previous assignments where the schema was provided to you.

You will submit the files below. They **must** be named as shown.

- `create_schema.txt`: A schema creation file, consisting of an SQL script with `DROP` and `CREATE` statements for each table, along with `DROP` and `CREATE` statements for all required constraints. It must be possible to completely drop/recreate your database schema by running all of the commands in this file.
- `add_airports.py`
- `add_aircraft.py`
- `manage_flights.py`
- `manage_reservations.py`
- `report_all_flights.py`
- `report_itinerary.py`

4 The Database

For this assignment, your database will focus on airports, aircraft, flights (a single journey by a particular aircraft between two distinct airports), passengers and reservations (a booking by one passenger for one flight). Obviously, there are many other details that would be incorporated into a practical flight tracking database (and some of the constraints given below would be more realistic if more details were incorporated); you should implement only the specification given here (and omit any other details, even realistic ones, that are not described in this document). The subsections below detail the requirements and restrictions of each facet. You are not required to use any particular data model for your database, but you are encouraged to start by sketching an E/R diagram for the requirements. Part of the mark for your schema will be based on consistency and the elimination of redundancy. Requirements which are marked by an asterisk (*) in the sections below are **deliberately unrealistic** and are present to help simplify the task for this assignment.

For clarity, the set of constraints to enforce (which must be actively checked and enforced by your database) and constraints to assume (which you may assume will always hold in every input dataset, and therefore do not need to be checked) are given in point form at the end of each section.

When you are required to "enforce a constraint" in the sections below, your schema must contain a mechanism to prevent that constraint from being violated under any circumstances (that is, even if someone runs their own custom `INSERT`, `UPDATE` or `DELETE` statements on your database, the constraint must still be enforced). If an operation would violate a constraint, the operation must fail. Although you are free to use `CHECK` constraints and constraint triggers (and will likely have to use these for some constraints), remember that `FOREIGN KEY`, `PRIMARY KEY`, `NOT NULL` and `UNIQUE` are also constraints and can be very effective at preventing common violations.

Also note that an empty string '' is distinct from the SQL value `NULL` (so any constraints which require a string value to be 'non-empty' require that the string be non-NULL, but also that the string not equal the empty string).

4.1 Airports

Every **airport** is identified uniquely among all airports on Earth by a three-character code issued by the International Air Transport Association (IATA). For example, the Victoria International Airport (in North Saanich) has the code 'YYJ'. The Vancouver International Airport (on Sea Island, west of Richmond) is 'YVR', while the Boundary Bay Airport in Delta (also part of Metro Vancouver) is 'YDT'.

In addition to their IATA code, airports have a name (e.g. 'Victoria International Airport'), although the name is not guaranteed to be unique. Some airports are 'international airports', which means they can handle flights which originate in or travel to a different country (*). In addition to tracking the IATA code, name and international airport status, you must also track the name of the country containing the airport (since this will be necessary to validate international flights). You may assume that country names are unique, so if the same country name is associated with two airports, those two airports are in the same country (*).

You may assume that airport names and country names are always at most 255 characters long, but you may not assume anything else about the structure of those names (including whether or not they can contain multiple words, punctuation, digits, etc.).

Constraints to enforce:

- The IATA code must be exactly three characters long and contain only uppercase letters.
- The airport name and country name are non-empty.
- IATA codes are unique (that is, no two different airports have the same IATA code).

Constraints to assume:

- The maximum length for airport names and country names will be obeyed.

4.2 Aircraft

An **aircraft** is identified by a unique identification code which may contain letters and/or digits (e.g. 'A101' or 'WJ3069132A'). Each aircraft is owned by one airline (*). Airlines are uniquely identifiable by their name (e.g. 'Air Canada' or 'WestJet'). Each aircraft has an associated model name (e.g. 'Boeing 737-300') and a fixed passenger capacity (*).

You may assume that identification codes are at most 64 characters long. You may also assume that airline names and aircraft model names are at most 255 characters long (but you may not assume anything about the structure of those identifiers otherwise).

The passenger capacity of an aircraft may be zero, but may not be negative. The ID code, airline name and model name must not be empty strings.

Constraints to enforce:

- Aircraft ID codes are unique
- Aircraft ID codes, airline names and model names are non-empty.
- Seating capacities are integers greater than or equal to zero.

Constraints to assume:

- The maximum lengths for aircraft ID codes, airline names and model names will be obeyed.
- Aircraft ID codes will only contain letters and digits.

4.3 Flights

A **flight** is a particular journey of an aircraft between two distinct airports. Every flight has an associated source airport, a destination airport and an aircraft. Every flight has a numerical Flight ID number, which uniquely identifies it among all other flights (since it is possible for there to be multiple flights of the same aircraft between the same pair of airports over time). Every flight also has an associated airline name (of the airline operating the flight), and this airline name must match the name of the airline which owns the aircraft used for the flight. As in Section 4.2, you may assume that airline names are at most 255 characters in length.

The source airport and destination airport of a particular flight must not be equal. If the source airport and destination airport are in different countries, then **both** airports must be 'international' airports.

Flights may be added, modified or deleted from the database with one of the front-end programs, so care must be taken to ensure that the constraints are all enforced during any of these operations.

The departure time of each flight must (obviously) be chronologically earlier than its arrival time. In addition, there are other constraints on which flights can exist (based on the realities of aircraft allocation) detailed in Section 4.3.1 below.

Actual flight tracking systems would need to account for local time differences (that is, the effect of time zones on long-distance flights). However, for this assignment we will assume that all date/time values provided to the database have already been reconciled into a common time zone, so there is no need to account for time zone differences in your submission.

4.3.1 Aircraft Consistency Constraints

For the addition, modification or removal of a flight F to be valid, all of the following constraints must be obeyed. Informally, these constraints are equivalent to 'Each aircraft must be on the ground for at least 60 minutes between flights, the same aircraft can't be used for two flights at the same time, and the aircraft must take off from the same airport where it landed'.

1. If there are any flights which use the same aircraft as F and have departure times before the arrival time of F:
   (a) All such flights must have an **arrival time** at least 60 minutes before the **departure time** of F.
   (b) The **most recent** flight before F (defined to be the flight whose arrival time is closest to the departure time of F) must have a destination airport equal to the source airport of F.
2. If there are any flights which use the same aircraft as F and have departure times after the departure time of F:
   (a) All such flights must have a **departure time** at least 60 minutes after the **arrival time** of F.
   (b) The next flight after F (defined to be the flight whose departure time is closest to the arrival time of F) must have a source airport equal to the destination airport of F.

Constraints to enforce:

- The associated source airport, destination airport and aircraft ID are valid (and exist in the database already).
- The source airport is distinct from the destination airport.
- If the source airport and destination airport are in different countries, **both** airports must be 'international' airports.
- The name of the airline associated with this flight is the same as the name of the airline which owns the aircraft used.
- The departure time is chronologically earlier than the arrival time.
- The conditions described in Section 4.3.1 are met (for all operations: insertion, deletion and modification).
- For modifications: The aircraft used must have a seating capacity greater than or equal to the number of existing reservations for the flight (see Section 4.5 below for details).
- For deletion: A flight F may not be deleted if any reservations exist for F.

Constraints to assume:

- The maximum lengths for airline names will be obeyed.

4.4 Passengers

A **passenger** is an individual with a name and a unique integer passenger ID. You may assume that passenger names are between 1 and 1000 characters in length (inclusive), but you may not assume anything else about a passenger's name.

Constraints to enforce:

- Passenger names are non-empty.
- Passenger ID numbers are unique.

Constraints to assume:

- The maximum lengths for names will be obeyed.

4.5 Reservations

A **reservation** is an association between a particular passenger and a particular flight. A particular passenger may have at most one reservation on a particular flight. There is no requirement that a passenger's set of reservations makes any logistical sense (that is, it is okay for one passenger to reserve seats on 10 different flights which take off from different airports at the same time), since from an accounting perspective it is still profitable for the air travel industry if passengers pay for seats they never actually use.

The total number of reservations for a particular flight must be less than or equal to the seating capacity of the aircraft used for that flight (*). This constraint must be enforced both when modifying reservations but also when modifying flights (since there are cases where the aircraft used for a flight may change; this modification must fail if there are more reservations for the flight than there are seats on the new aircraft).

Constraints to enforce:

- The passenger and flight associated with the reservation are valid (and already exist in the database).
- The total number of reservations for a particular flight must be less than or equal to the seating capacity of the associated aircraft.

Constraints to assume:

- N/A

4.6 Deliverable: Schema creation script

The submitted `create_schema.txt` file will be an SQL script (which can be run, in its entirety, on the DBMS) containing `CREATE` statements (and any other necessary statements, including `INSERT`, `UPDATE` and `ALTER` if required) for all tables an constraints to allow the programs described below to function correctly. To be clear, your Python programs should **not** contain the SQL `CREATE` statements.

You **must** also include, at the top of your file, `DROP` statements for each database object (table, function, etc.) created in your script. You should use the 'if exists' notation to ensure that the `DROP` statements continue to work even if the tables/functions being dropped do not already exists. It must be possible to run the schema creation statements repeatedly without errors, or it will not be possible to fully evaluate your submission.

The goal of this format is to allow the `create_schema.txt` script to be run as a way of clearing out the database and preparing it for a fresh testing sequence. A similar format has been followed in the posted SQL scripts from the modifications, constraints and transactions lectures.

Include a comment with your name and student number at the beginning of the file.

In addition to implementing the requirements described in the previous sections, your database design must also obey the following style constraints.

- All tables have a primary key.
- For any fields which are required to be present, `NOT NULL` directives must be used to prevent `NULL` values from being added. Recall that there is no need for a `NOT NULL` directive on columns which are part of the primary key.

- String fields must use the `varchar` type, not a Postgres-specific string type like `text`.
- Every definition of a foreign key constraint **must** include `ON DELETE` and `ON UPDATE` policies (you can set the policies to `RESTRICT` if needed, but you must explicitly include the policy directives instead of allowing the default to be used).

Advice: If you want to run your entire `create_schema.txt` script (to refresh your database), you can try one of the three options below.

- Open a `psql` session and use the `\i` command to run the script.
- Pipe the contents of `create_schema.txt` into `psql` on the command line. For example,
  ```
  psql -h studdb1.csc.uvic.ca dbname username < create_schema.txt
  ```
- Open the script in DBeaver and use Alt-X to run the entire file.

5 Front-end Programs: Data Entry

You are required to write four Python 3 programs which read data from a text file (in a simple comma-separated spreadsheet format) and make insertions or modifications to the database. You are permitted to use any features of Python 3 available on `dblinux.csc.uvic.ca`, but, in general, you should avoid as much processing as possible in the front-end programs. Even simple processing like input validation (for example, checking if a grade is within the required range) should be implemented with constraints in the DBMS, not on the front-end. For full marks, your front-end programs must be as simple as possible and leave all non-trivial data processing to the DBMS: a perfect solution will consist of a simple read loop to read each line of the input file and a few statements to run SQL on the server (and possibly handle any exceptions that may occur). You will be deducted marks if your solutions for these programs contain `SELECT` statements in any form: either stand-alone (that is, a query that is retrieved by your program, inspected, and used to determine what to insert/delete) or nested within an `INSERT` statement (although nested `SELECT` statements inside `INSERT` statements will be subject to a smaller deduction).

Several example programs have been posted which detail how to read comma-separated data and use the `psycopg2` module to interact with the DBMS. The total number of lines of Python needed to implement each part should be relatively small, and it should be possible to reuse most of the code for reading input between the programs.

The following requirements apply for all of the programs below.

- Each program may be run repeatedly on the same database (before or after other modifications have been performed by other programs).
- Each program should perform operations on the database in the same order they appear in the input file.
- In the case where the data entry is successful, the program should not generate any output to standard output or standard error. In cases where an error of any kind occurs, the program should print an error message to indicate that the data entry failed. If the program crashes due to an unhandled exception, you will lose marks.

5.1 Creating airports: `add_airports.py`

The `add_airports.py` script is invoked from the command line as follows.

`python3 add_airports.py <input file>`

Where '`<input file>`' is a file name (e.g. '`airports_to_add.txt`').

Each line of the provided input file will either be blank (or consist entirely of spaces), in which case it will be ignored, or will contain 4 values, with a single comma character between each field, in the form

`<airport code>,<airport name>,<country name>,<international>`

The field `<international>` will be either the string 'true' or the string 'false'.

For example, the input sample below contains records for 15 airports.

```
YVR,Vancouver International Airport,Canada,true
YYJ,Victoria International Airport,Canada,true
YCD,Nanaimo Airport,Canada,false
YDT,Boundary Bay Airport,Canada,false
YXX,Abbotsford International Airport,Canada,true
YYC,Calgary International Airport,Canada,true
YYZ,Lester B. Pearson International Airport,Canada,true
YHU,Montreal/Saint-Hubert Airport,Canada,false
YUL,Montreal-Trudeau International Airport,Canada,true
LAX,Los Angeles International Airport,United States,true
LGA,LaGuardia Airport,United States,false
JFK,John F. Kennedy International Airport,United States,true
EWR,Newark Liberty International Airport,United States,true
HND,Tokyo International Airport,Japan,true
NRT,Narita International Airport,Japan,true
```

Some starter code has been posted which uses the `csv` module to parse the input data. If you use the provided code as the basis for your submission, you should not need to worry about reading the input format.

The `add_airports.py` program will read each line from the input file and insert records into the database for each airport. The set of all modifications made by the program must be atomic with respect to the database: If a database error occurs at any point during execution, the set of all modifications must be rolled back. As a result, a particular run of `add_airports.py` will either successfully read an input file and modify the database, or fail and make no modifications whatsoever.

The `add_airports.py` program must catch any exceptions that occur as a result of database errors and print an error message. As stated above, an error should result in the database transaction being rolled back such that no modification occurs, but it is important that your Python program does not crash: instead, it should catch the exception, display some kind of error message (ideally a descriptive one) and exit gracefully.

Records should be added to the database in the same order as they appear in the file.

5.2 Creating aircraft: `add_aircraft.py`

The `add_aircraft.py` script is invoked from the command line as follows.

`python3 add_aircraft.py <input file>`

Where '`<input file>`' is a file name (e.g. '`aircraft_to_add.txt`').

Each line of the provided input file will either be blank (or consist entirely of spaces), in which case it will be ignored, or will contain 4 values, with a single comma character between each field, in the form

`<aircraft ID>,<airline name>,<model name>,<seating capacity>`

The `<seating capacity>` field will be an integer.

For example, the input sample below contains records for eight aircraft.

```
CFGDT,Air Canada,Boeing 787-9,270
CFGKP,Air Canada,Airbus A321-211,180
CFIVW,Air Canada,Boeing 777-333ER,320
CGAUN,Air Canada,Boeing 767-233,200
CFENJ,WestJet,DeHaviland DHC-8-402,75
CFNWD,WestJet,Boeing 737-8,170
CGXPE,Island Express Air,Piper PA-31,6
CGIEA,Island Express Air,Beech B100,12
```

Some starter code has been posted which uses the `csv` module to parse the input data. If you use the provided code as the basis for your submission, you should not need to worry about reading the input format.

The `add_aircraft.py` program will read each line from the input file and insert records into the database for each aircraft. The set of all modifications made by the program must be atomic with respect to the database: If a database error occurs at any point during execution, the set of all modifications must be rolled back. As a result, a particular run of `add_aircraft.py` will either successfully read an input file and modify the database, or fail and make no modifications whatsoever.

The `add_aircraft.py` program must catch any exceptions that occur as a result of database errors and print an error message. As stated above, an error should result in the database transaction being rolled back such that no modification occurs, but it is important that your Python program does not crash: instead, it should catch the exception, display some kind of error message (ideally a descriptive one) and exit gracefully.

Records should be added to the database in the same order as they appear in the file.

5.3 Flight management: `manage_flights.py`

The `manage_flights.py` script is invoked from the command line as follows.

`python3 manage_flights.py <input file>`

Where '`<input file>`' is a file name (e.g. '`flights.txt`').

Each line of the provided input file will either be blank (or consist entirely of spaces), in which case it will be ignored, or will be either

`DELETE,<flight ID>`

or

`<CREATE_or_UPDATE>,<flight ID>,<airline name>,<source airport>,<dest airport>,<departure time>,<arrival time>,<aircraft id>`

The first field in the line will be either "CREATE", "UPDATE" or "DELETE", indicating which operation to perform.

If the operation is "DELETE", the flight with the provided ID will be removed from the database. If that ID was not already present, the program must generate an error (this condition can be detected by checking the number of rows deleted by the SQL `DELETE` statement).

If the operation is "CREATE", a new flight with the provided details will be added. If a flight already exists with the provided ID, the program must generate an error. If the operation is "UPDATE", the flight with the provided ID will be updated with the other information; if no such flight exists, the program must generate an error (like deletion, it is possible to detect this by checking the number of rows modified by the SQL `UPDATE` statement).

The `<flight ID>` field will be an integer. The other fields will be strings, with `<departure time>` and `<arrival time>` being timestamp values in the format 'YYYY-MM-DD HH:mm' (which can be directly inserted into Postgres `TIMESTAMP` fields, so no additional processing should be needed). As mentioned in section 4.3, we will assume that all date/time values are already normalized to some common time zone (so there is no need to reconcile time zone differences). The `<source airport>` and `<dest airport>` fields will be three-character airport codes (like 'YYJ').

For example, the input sample below contains several flight management operations, using the set of airports and aircraft shown in previous examples. Notice that the order of data entry does not match the chronological ordering of flights, and that, until the entire dataset is read, the aircraft consistency constraints (see Section 4.3.1) are not necessarily met.

```
CREATE,1000,WestJet,YYJ,YYC,2020-07-01 06:10,2020-07-01 07:50,CFNWD
CREATE,1001,Air Canada,YVR,HND,2020-06-10 10:00,2020-06-10 20:35,CFGDT
CREATE,1002,Air Canada,YVR,YYC,2020-03-13 18:07,2020-03-13 18:42,CFGKP
CREATE,1003,Air Canada,YYC,YYJ,2020-03-12 22:19,2020-03-13 00:10,CFGKP
CREATE,1004,Air Canada,YYJ,YVR,2020-03-13 05:51,2020-03-13 06:19,CFGKP
CREATE,1005,WestJet,YYC,YYJ,2020-07-01 18:10,2020-07-01 19:50,CFNWD
CREATE,1006,Island Express Air,YYJ,YXX,2020-07-04 06:50,2020-07-04 07:25,CGXPE
CREATE,1007,Island Express Air,YXX,YCD,2020-07-04 08:45,2020-07-04 09:20,CGXPE
CREATE,1008,Island Express Air,YCD,YVR,2020-07-04 10:30,2020-07-04 10:50,CGXPE
DELETE,1005
UPDATE,1008,Island Express Air,YCD,YYJ,2020-07-04 10:30,2020-07-04 10:55,CGXPE
```

Some starter code has been posted which uses the `csv` module to parse the input data. If you use the provided code as the basis for your submission, you should not need to worry about reading the input format.

The `manage_flights.py` program will read each line from the input file and make the appropriate insertions/updates/removals for each operation. The set of all modifications made by the program must be atomic with respect to the database: If a database error occurs at any point during execution, the set of all modifications must be rolled back. As a result, a particular run of `manage_flights.py` will either successfully read an input file and modify the database, or fail and make no modifications whatsoever.

Modifications should be made to the database in the same order as they appear in the file. However, since some constraints (specifically the aircraft consistency constraints in Section 4.3.1) cannot be

satisfied immediately, all such constraints should be deferred until the entire file is read (but, obviously, the constraints must be enforced before any modification is committed permanently to the database).

5.4 Reservation management: `manage_reservations.py`

The `manage_reservations.py` script is invoked from the command line as follows.

```
python3 manage_reservations.py <input file>
```

Where '`<input file>`' is a file name (e.g. '`reservations.txt`').

Each line of the provided input file will either be blank (or consist entirely of spaces), in which case it will be ignored, or will contain exactly 4 text fields, with a single comma character between each field, in the form

```
<CREATE_OR_DELETE>,<flight ID>,<passenger ID>,<passenger name>
```

The first field, `<CREATE_OR_DELETE>`, will either be the word "CREATE" or the word "DELETE", indicating which operation to perform. Notice that the passenger's name will appear on every line that affects them; see below for details on this aspect.

For example, the input sample below shows several reservation creations and deletions.

```
CREATE,1000,12345,Alissa Aubergine
CREATE,1000,12346,Hannah Hindbaer
CREATE,1000,12347,Behrouz Boruvka
CREATE,1006,12348,Leona Loganberry
CREATE,1006,12349,Kaito Kiichigo
CREATE,1008,12350,Alistair Avocado
DELETE,1000,12346,Hannah Hindbaer
CREATE,1007,12348,Leona Loganberry
CREATE,1008,12348,Leona Loganberry
```

Some starter code has been posted which uses the `csv` module to parse the input data. If you use the provided code as the basis for your submission, you should not need to worry about reading the input format.

The `manage_reservations.py` program will read each line from the input file and make the appropriate insertions/updates/removals for each operation. The set of all modifications made by the program must be atomic with respect to the database: If a database error occurs at any point during execution, the set of all modifications must be rolled back. As a result, a particular run of `manage_reservations.py` will either successfully read an input file and modify the database, or fail and make no modifications whatsoever.

An `CREATE` operation must not succeed if the passenger already has a reservation on the same flight (since each passenger can reserve at most one seat on each flight).

There is no direct mechanism to 'add passengers' to the database. Instead, a passenger is created the first time they are added to a reservation. The name used for the first reservation will become the passenger's permanent name. All future reservation CREATE/DELETE operations involving that passenger must use **exactly** the same name as the first reservation or the operation must fail. For deletions, this can be achieved by attempting to delete rows containing both the provided ID

and name (and producing an error if no such rows are found). For the create operations, one way to achieve this is to add an `INSERT` trigger that allows the same passenger to be inserted multiple times as long as the name and passenger ID match the existing record (with the understanding that all such duplicate insertions are silently ignored). If you can't figure out a way to implement the constraint directly into the database, you can use Python logic to enforce it (but you will not receive full marks for doing so).

A passenger will not exist until the first time a reservation is made in their name. However, once added, passenger will never be deleted from the database (even if all associated reservations are removed).

Modifications should be made to the database in the same order as they appear in the file. Constraint enforcement should be deferred until all entries in the file are processed, since it is possible that the relevant constraints may not hold at some intermediate point in the file. For example, a file might contain CREATE lines which create more bookings than a flight can allow, but then balance the number out with DELETE lines before the end of the file.

6 Front-end Programs: Reports

You are required to write three Python 3 programs which query the database (using whatever sequence of `SELECT` statements you find appropriate) and generate data 'reports' to standard output.

Sample Python scripts with output commands to produce mockups of the expected format have been posted to conneX, so you do not need to worry about the tedious process of designing the output formatting commands. However, since code has been provided to format your output, you are expected to produce output identical to the format described in the mockup files (whether or not you actually use the mockup files as starter code).

6.1 Flight Lists: `report_all_flights.py`

The `report_all_flights.py` script is invoked from the command line as follows.

```
python3 report_all_flights.py
```

The program takes no arguments. The output of the program (on standard output, not to a file) will be a list of all flights in the database, ordered by departure time (with any flights whose departure time is the same ordered in ascending order by flight ID). For each flight, the flight ID, airline, departure/arrival times, total duration (in minutes), source/destination airports, aircraft ID, aircraft model name, number of reservations and total seat capacity will be shown. The exact format should follow the formatting given in the mockup `report_all_flights.py` program posted to conneX (and in the example below).

When run on the database produced by the all of the input files in the examples of Section 5, the following result will be produced.

```
Flight 1003 (Air Canada):
    [2020-03-12 22:19:00] - [2020-03-13 00:10:00] (111 minutes)
    Calgary International Airport -> Victoria International Airport
    CFGKP (Airbus A321-211): 0/180 seats booked
```

```
Flight 1004 (Air Canada):
    [2020-03-13 05:51:00] - [2020-03-13 06:19:00] (28 minutes)
    Victoria International Airport -> Vancouver International Airport
    CFGKP (Airbus A321-211): 0/180 seats booked
Flight 1002 (Air Canada):
    [2020-03-13 18:07:00] - [2020-03-13 18:42:00] (35 minutes)
    Vancouver International Airport -> Calgary International Airport
    CFGKP (Airbus A321-211): 0/180 seats booked
Flight 1001 (Air Canada):
    [2020-06-10 10:00:00] - [2020-06-10 20:35:00] (635 minutes)
    Vancouver International Airport -> Tokyo International Airport
    CFGDT (Boeing 787-9): 0/270 seats booked
Flight 1000 (WestJet):
    [2020-07-01 06:10:00] - [2020-07-01 07:50:00] (100 minutes)
    Victoria International Airport -> Calgary International Airport
    CFNWD (Boeing 737-8): 2/170 seats booked
Flight 1006 (Island Express Air):
    [2020-07-04 06:50:00] - [2020-07-04 07:25:00] (35 minutes)
    Victoria International Airport -> Abbotsford International Airport
    CGXPE (Piper PA-31): 2/6 seats booked
Flight 1007 (Island Express Air):
    [2020-07-04 08:45:00] - [2020-07-04 09:20:00] (35 minutes)
    Abbotsford International Airport -> Nanaimo Airport
    CGXPE (Piper PA-31): 1/6 seats booked
Flight 1008 (Island Express Air):
    [2020-07-04 10:30:00] - [2020-07-04 10:55:00] (25 minutes)
    Nanaimo Airport -> Victoria International Airport
    CGXPE (Piper PA-31): 2/6 seats booked
```

6.2 Aircraft Statistics: `report_aircraft.py`

The `report_aircraft.py` script is invoked from the command line as follows.

```
python3 report_aircraft.py
```

The program takes no arguments. The output of the program (on standard output, not to a file) will be a list of all aircraft in the database, ordered by aircraft ID. For each aircraft, the ID, owning airline and model name will be shown, along with the number of flights, the total number of flight hours (rounded to an integer), average passenger reservations per flight (rounded to two places of decimal precision) and seating capacity. The total flight hours will be computed as the sum, for all flights involving the selected aircraft, of the difference between each departure and arrival time of that flight, with the final sum rounded to the nearest hour. The average number of passenger reservations per flight is equal to the total number of reservations across all flights involving the selected aircraft divided by the total number of such flights. If the aircraft has not been involved in any flights, the average number of passengers is defined to be zero. The exact format should follow the formatting given in the mockup `report_aircraft.py` program posted to conneX (and in the example below).

When run on the database produced by the all of the input files in the examples of Section 5, the following result will be produced.

```
CFENJ (DeHaviland DHC-8-402): WestJet
    Number of flights : 0
    Total flight hours: 0
    Average passengers: (0.00/75)
CFGDT (Boeing 787-9): Air Canada
    Number of flights : 1
    Total flight hours: 11
    Average passengers: (0.00/270)
CFGKP (Airbus A321-211): Air Canada
    Number of flights : 3
    Total flight hours: 3
    Average passengers: (0.00/180)
CFIVW (Boeing 777-333ER): Air Canada
    Number of flights : 0
    Total flight hours: 0
    Average passengers: (0.00/320)
CFNWD (Boeing 737-8): WestJet
    Number of flights : 1
    Total flight hours: 2
    Average passengers: (2.00/170)
CGAUN (Boeing 767-233): Air Canada
    Number of flights : 0
    Total flight hours: 0
    Average passengers: (0.00/200)
CGIEA (Beech B100): Island Express Air
    Number of flights : 0
    Total flight hours: 0
    Average passengers: (0.00/12)
CGXPE (Piper PA-31): Island Express Air
    Number of flights : 3
    Total flight hours: 2
    Average passengers: (1.67/6)
```

6.3 Passenger Itineraries: `report_itinerary.py`

The `report_itinerary.py` script is invoked from the command line as follows.

`python3 report_itinerary.py <passenger ID>`

The output of the program (on standard output, not to a file) will begin with a header containing the passenger name and ID, followed by a list of all flights for which the selected passenger has a reservation, sorted into ascending order by departure time (and then flight ID if there are any ties). For each flight, the flight ID, airline, source/destination airport names, departure/arrival times, total duration (in minutes), aircraft ID and aircraft model name will be displayed. The exact format should follow the formatting given in the mockup `report_itinerary.py` program posted to conneX (and in the example below).

If no passenger with the provided ID exists, or if such a passenger exists but has no reservations in the system, the program will print an error message and exit with no further output.

When run on the database produced by the all of the input files in the examples of Section 5, the following result will be produced.

```
Itinerary for 12348 (Leona Loganberry)
Flight 1006 (Island Express Air):
    [2020-07-04 06:50:00] - [2020-07-04 07:25:00] (35 minutes)
    Victoria International Airport -> Abbotsford International Airport (CGXPE: Piper PA-31)
Flight 1007 (Island Express Air):
    [2020-07-04 08:45:00] - [2020-07-04 09:20:00] (35 minutes)
    Abbotsford International Airport -> Nanaimo Airport (CGXPE: Piper PA-31)
Flight 1008 (Island Express Air):
    [2020-07-04 10:30:00] - [2020-07-04 10:55:00] (25 minutes)
    Nanaimo Airport -> Victoria International Airport (CGXPE: Piper PA-31)
```

7 Evaluation

Your submitted schema creation script must run in a single pass (that is, as a script without the need for human intervention and without generating any errors) on your individual PostgreSQL database on the `studdb1.csc.uvic.ca` or `studdb2.csc.uvic.ca` server.

Your Python code must run correctly using the built-in installation of Python 3 (specifically, the `python3` command which resolves to `/usr/bin/python3`) on the `dblinux.csc.uvic.ca` login server. If your code does not meet that requirement, it will not be marked. All of your Python 3 programs must use the `studdb1.csc.uvic.ca` or `studdb2.csc.uvic.ca` database servers exclusively for all data storage; if you use local files to store data (or somehow connect to another database server), your code will not be marked.

When you hand in your code, it must be written to connect directly to the database server without prompting the user for a password. This means that you will have to hard-code your database account password into your scripts as text (so the members of the CSC 370 teaching team will be able to see the password). Therefore, **you must change your database password to something you feel comfortable sharing with us** before handing in the assignment. In particular, **do not use your Netlink, CSC account or conneX password** as your database password, and never divulge information about your Netlink password, CSC account password or conneX password to anyone, including your instructor or any other department personnel.

The 40 marks are distributed among the components of the assignment as follows.

Marks	Component
11	The database schema is well designed (using a normalized data model and obeying the style requirements/best practices covered by CSC 370) and consistent. In particular, primary keys are specified for every table and foreign keys are defined where applicable.
7	The suite of data entry and report programs functions correctly on test sequences containing strictly valid data (with no violations of any constraints). Note that you may still receive these marks even if some of the constraints are enforced on the client side.
7	The suite of data entry and report programs functions correctly on test sequences containing invalid data. Errors with input data must be properly handled, and inconsistent data must not be added to the database. Note that you may still receive these marks even if some of the constraints are enforced on the client side.
15	All constraints and data validation logic are integrated into the database schema instead of being enforced by the client-side programs. When this has been done correctly, the Python programs will perform the **bare minimum** amount of processing (all data validation, data processing and constraint enforcement is handled on the server-side), and it will be impossible to add invalid data to the database, even if manual `INSERT/UPDATE/DELETE` statements are used instead of the client-side data entry programs.

Acceptable Collaboration

You are encouraged to discuss this assignment with your colleagues (and even to discuss conceptual aspects and collaborate on solution strategies), but under no circumstances can **any** code (SQL or Python) be shared by **any means** (including sending it electronically, displaying it via screenshare, printing it out and sending it in the mail, reading it word-for-word, etc.). Any violation of this will be considered plagiarism; in particular, note that providing your code to others for any reason (even if you believe that it will not be used by the other party for this assignment) is considered plagiarism.

Submission Instructions

All submissions for this assignment will be accepted electronically. Submit all of your files through the Assignments tab on conneX. Do not use the `.sql` extension for any files, since conneX does not properly handle files with the `.sql` extension. Name each file as specified in the sections above. You are permitted to delete and resubmit your assignment as many times as you want before the due date, but no submissions or resubmissions will be accepted after the due date has passed.

Ensure that each file contains a comment with your name and student number.

After submitting your assignment, conneX will automatically send you a confirmation email to your `@uvic.ca` email address If you do not receive such an email, your submission was not received. If you have problems with the submission process, send an email to the instructor **before** the due date.