Project 1 Part 2
ETL: Queries

Out: *October 14, 2020*
Due: *October 23rd, 2020, 11:59 P.M.*

1 Introduction: On to Part 2!

Assuming you have completed part 1, you now have extracted all of the information and transformed it into some coherent, consistent database! Yay!!!

This next part of the project involves writing a series of queries that will test your knowledge of SQL. As you progress through the assignment, you will build up your understanding of interesting/innovative ways to use SQL (and you will almost assuredly be a master of joins).

2 Goal

For this part of the project, your goal will be to write an application, `query`, which will make pre-defined queries against the SQLite database and print the results to the console.

3 Overview of the Data

The data should be pretty familiar to you at this point. Below is a summary of what the CSV's contained:

Note: This overview may not fully explain all of the nuances of the data: you are encouraged to look at the files yourselves (CSVs are human-readable) to better understand them. You should take all of this data and be able to enter it into database of your design, avoiding redundancies.

3.1 airlines.csv

This file contains basic informations on all of the airlines. There are two fields: the first is a code that is unique to the airlines (eg: YX) and the second is the name of the airline (eg: Republic Airlines). Note that not all airlines may have flight data associated with them.

3.2 airports.csv

This file contains information on all of the airports. There are two fields: the first is a code that corresponds uniquely to a particular airport and the second is the full, canonical name of the airport. Note that not all airports may have flight data associated with them.

3.3 flights.csv

It contains information on every single flight limited to a single month of data (note that your design should still be able to accommodate data from other months and/or years!).

3.4 Schema

Your database should be modeled after the following schema. We've only labeled primary keys, so make sure to include other constraints in your SQL table creation:

airlines ($\underline{airline\_id}, airline\_code, airline\_name$)
airports $\left(\underline{airport\_id}, airport\_code, airport\_name, city, state\right)$
flights($\underline{flight\_id}, airline\_id, flight\_num, origin\_airport\_id,$
$dest\_airport\_id, departure\_dt, depart\_diff, arrival\_dt,$
$arrival\_diff, cancelled, carrier\_delay, weather\_delay, air\_traffic\_delay, security\_delay$)

4 The Applications

4.1 `query`

This script should make pre-defined queries against a specified SQL database. The query to be executed will be specified via the command line. If the query requires input from the user (ie: a name, a start/end date, etc), that information will also be passed in via the command line.

The simplest call to the script looks like this:

```
./query \
data.db \
query0
```

Given those inputs, the application should execute Query #0 (queries are defined and numbered below) against the SQLite database at `data.db`.

A more complex call for the program might look like this:

```
./query \
data.db \
query7 \
Southwest\ Airlines\ Co. \
401 \
```

```
01/01/2012 \
02/20/2012
```

The expected input and output columns for each query are described in more detail below.

4.2 Testing your Application

Before you test your program, go to your ETLTester.java program, and change your PUB_PATH variable to whichever directory your "tests" directory is in (follow syntax of the example location to ensure no errors come up). To test your `query` application, `cd` to your ETL directory and run `/usr/bin/ant test`. Your application will be run using various inputs and compared to outputs from the TA solution code. Your application should pass all of the testcases using these tests: This is one of the major ways your handin will be evaluated.

If you believe that the script is returning incorrect results, please feel free to contact the TAs. Be sure to provide relevant lines of code so the TAs can evaluate your objection.

You can also test your SQL code outside of the Java application using the test database. See Appendix for details.

5 Queries

You will need to design SQL queries for your database that answer the following questions. Unless otherwise noted, all queries should be composed of a single SQL statement. The first three queries (queries A,B,C) should look familiar: They were the test queries for part 1! Make sure for TA testing purposes to name your queries "queryA", "query0", etc. **You can check the correct output for example input in the appropriate output file in the tests directory.**

- **queryA**: Count the number of airport codes.

  Input: N/A

  Output: One column. Number of airport codes.

- **queryB**: Count the number of airline codes.

  Input: N/A

  Output: One column. Number of airline codes..

- **queryC**: Count the number of total flights.

  Input: N/A

  Output: One column. Number of flights.

- **query0**: Write a query to get a list of airports in Alaska. Include the airport name, city and state in the result. List in **reverse alphabetical order of city.** List first 3 results.

  Input: N/A

  Output: Three Columns: Airport Name, City, State.

- **query1**: Write a query that, given an input airline name, gets the first flight of that airline from the flights relation based on LOWEST flight number. Please output the airline name, flight_number.

  Input: Airline Name

  Output: One Row, Two Columns: Airline Name, Flight Number.

- **query2**: Write a query that, given an input airline name, gets the total number of cancelled flights recorded. Please output just the count of flights cancelled.

  Input: Airline Name

  Output: One Row, One Columns: Count of cancelled flights.

- **query3**: Write a query that lists airports by the number of "high-traffic" airline flights that use the airport as an origin. We consider "High Traffic" airlines to be airlines that have more than 10,000 flights in the flights relation. Limit to top 5 results (**You are looking for top 5 origin airports in terms of count of flights from high traffic airlines**).

  Input: N/A

  Output: 5 Rows, 2 Columns: Origin Airport Name, Count of Flights from High Traffic Airlines using that airport as an origin.

  Note1: A **With** clause defining high traffic airlines might be useful.

- **query4**: Get all the reasons flights were delayed, along with their frequency, in order from highest frequency to lowest.

  Input: N/A

  Output: Two columns. The first column should be a string describing the type of delay. The four types of delays are Carrier Delay, Weather Delay, Air Traffic Delay, and Security Delay (Make sure to adhere these delay names). The second column should be the number of flights that experienced that type of delay. The results should be in order from largest number of flights to smallest.

  Note1: Try to think about what kind of SQL clause could be used for combining different records together into one table.

Note2: In SELECT clauses, unnamed fields are automatically given their order number as a name, for example, the first unnamed field is given the name '1' since it is the 1st field.

- **query5**: Get **all** airlines, along with the number of flights by that airline which were scheduled to depart on a particular day (whether or not they departed). Results should be ordered from highest frequency to lowest frequency, and then ordered alphabetically by airline name, A-Z.

  Input 1: A month (1 = January, 2 = February, ..., 12 = December)

  Input 2: A day (1, 2 ... 31)

  Input 3: A year (2010, 2011, 2012, etc)

  Output: Two columns. The first column should be the name of the airline. The second column should be the number of flights matching the criteria.

- **query6**: For a specified set of airports, return the number of departing and the number of arriving planes on a particular day (scheduled departures/arrivals). Results should be ordered alphabetically by airport name, A-Z.

  Input 1: A month (1 = January, 2 = February, ..., 12 = December)

  Input 2: A day (1, 2 ... 31)

  Input 3: A year (2010, 2011, 2012, etc)

  Input 4 .. n: The full, canonical name of an airport (ie: LaGuardia).

  Output: Three columns. The first column should be the name of the airport. The second column should be the number of flights that were scheduled to depart the airport on the specified day. The third column should be the number of flights that were scheduled to arrive at the airport on the specified day.

- **query7**: Calculate statistics for a specified flight (Airline / Flight Number) scheduled to depart during a specified range of dates (inclusive of both start and end).

  Input 1: An airline name (ie: American Airlines Inc.).

  Input 2: A flight number.

  Input 3: A start date, in MM/DD/YYYY format.

  Input 4: An end date, in MM/DD/YYYY format.

  Output: Six columns:

  1. The total number of times the flight was scheduled
  2. The number of times it was cancelled
  3. The number of times it departed early or on time and was not cancelled
  4. The number of times it departed late and was not cancelled
  5. The number of times it arrived early or on time and was not cancelled

6. The number of times it arrived late and was not cancelled

Note: Using WITH clauses in SQL can help us saving codes and processing resources. In this query, students are expected to use WITH clause to avoid using overly repetitive code. Otherwise, points will be taken off.

- **query8**: If I had wanted to get from one city to another on a specific day (flight must have taken off and landed on the specified day), what were my options if I limited myself to one hop (aka: a direct flight)? Results should be sorted by total flight duration, lowest to highest, and then sorted alphabetically by airline code, A-Z. Remember that we're looking at historical data: as such, we're interested in actual departure/arrival times, inclusive of delays.

  Input 1: A departure city name (ie: Providence, Newark, etc).

  Input 2: A departure state name (ie: Rhode Island, New York, etc).

  Input 3: An arrival city name (ie: Providence, Newark, etc).

  Input 4: An arrival state name (ie: Rhode Island, New York, etc).

  Input 5: A date, in MM/DD/YYYY format.

  Output: Seven columns, each row representing a flight:

  1. The airline code
  2. The flight number
  3. The departure airport code
  4. The actual departure time (HH:MM)
  5. The arrival airport code
  6. The actual arrival time (HH:MM)
  7. The total duration of the flight in minutes.

- **query9**: Same as above, but for two hops. Results should be sorted by total duration, then sorted alphabetically by airline code for each hop, and then sorted by the actual depart time of the first hop, from the earliest to the latest.

  Input 1: A departure city name (ie: Providence, Newark, etc).

  Input 2: A departure state name (ie: Rhode Island, New York, etc).

  Input 3: An arrival city name (ie: Providence, Newark, etc).

  Input 4: An arrival state name (ie: Rhode Island, New York, etc).

  Input 5: A date, in MM/DD/YYYY format.

  Output: Thirteen columns, each row representing a series of flights. For each hop, you should have:

  1. The airline code
  2. The flight number

3. The departure airport code
4. The actual departure time (HH:MM)
5. The arrival airport code
6. The actual arrival time (HH:MM)

The final column should indicate the total travel time in minutes, from departure of the first flight to arrival of the last.

Note: You cannot visit an aiport in the same city and state as the origin or the destination on your way from the origin to the destination. For example, if the origin is New York, New York, and the destination is Providence, Rhode Island, then JFK → LGA, LGA → PVD is invalid because LGA is in the same city as JFK.

- **query10**: Same as above, but for three hops. Results should be sorted by total duration, then sorted alphabetically by airline code for each hop, and then sorted by the actual depart time of the first hop, from the earliest to the latest.

  Input 1: A departure city name (ie: Providence, Newark, etc).

  Input 2: A departure state name (ie: Rhode Island, New York, etc).

  Input 3: An arrival city name (ie: Providence, Newark, etc).

  Input 4: An arrival state name (ie: Rhode Island, New York, etc).

  Input 5: A date, in MM/DD/YYYY format.

  Output: Nineteen columns, each row representing a series of flights. For each hop, you should have:

  1. The airline code
  2. The flight number
  3. The departure airport code
  4. The actual departure time (HH:MM)
  5. The arrival airport code
  6. The actual arrival time (HH:MM)

  The final column should indicate the total travel time in minutes, from departure of the first flight to arrival of the last.

  Note: The city, state restriction from Query 10 still holds.

6 Working on the Project

6.1 Getting Started

To get started with Part 2, you should have to make the following changes to your ETL Part 1 directory:

1. Overwrite build.xml in the **.../etl** directory with the new one you can download from Canvas (In the ETL Part 2 assignment).
2. Overwrite the existing ETLTester.java in **.../etl/src/edu/brown/cs/cs127/etl/tester/** with the new from Canvas (In the ETL Part 2 assignment).
3. Change the **PUB_PATH** variable in ETLTester.java to be the location of your **tests** directory, which you can download from Canvas (In the ETL Part 2 assignment).

The directory contains the build file `build.xml`. This enables automation in compiling your project. To compile, while in that directory type `/usr/bin/ant`. This automatically includes the support code in your classpath when compiling.

Libraries are included as JARs in the `lib/` directory. Your code should go in `src/`.

6.2 Importing into Eclipse

1. Expand the stencil code inside your course directory. That should create a directory named "etl"
2. Open Eclipse. From the top menu bar, navigate to File → Import.
3. From there, expand the "General" tab, and select "Existing Projects into Workspace."
4. Click the "Browse" button next to "Select root directory" and browse to the etl directory inside your course directory. Click OK.
5. Check the box next to the project (if it isn't already checked) and click Finish.

7 Working with SQLite

7.1 From the command-line

SQLite is installed on all Sunlab machines. It can be accessed from the command line using sqlite3. For more information on using SQLite from the command line, see `http://www.sqlite.org/sqlite.html`

7.2 From Java

SQLite can be accessed via JDBC (Java's main database connectivity interface). There will not be an official help session on how to use JDBC, but TAs will be happy to answer questions on hours or via email. Students are highly encouraged to check out `https://www.tutorialspoint.com/jdbc/index.htm`, which has a wonderful tutorial on working with JDBC and SQLite.

8 Tips

8.1 Type System in SQLite

Students can refer to the following link `http://www.sqlite.org/datatype3.html` as a reference. Note that `DATETIME` is a valid type.

Note: Think about what datatype is most appropriate for the given field.
e.g. the pros and cons of using `TEXT` as opposed to `CHAR(n)` or `VARCHAR(n)`

8.2 Date/Time Functions in SQLite

In the raw data, date time is stored in string format. So students might want to use the date/time function in SQLite to convert the string into corresponding date format. Function **strftime(format, timestring, modifier, modifier, ...)** could be useful. It returns the date formatted according to the format string specified in argument first. The second parameter is used to mention the time string and followed by one or more modifiers can be used to get a different result.
For example, `SELECT strftime(%Y-%m-%d %H:%M:%S','now')` returns the formatted text string of current date. And here is a complete list of valid strftime() substitutions:

%d day of month: 00
%f fractional seconds: SS.SSS
%H hour: 00-24
%j day of year: 001-366
%J Julian day number
%m month: 01-12
%M minute: 00-59
%s seconds since 1970-01-01
%S seconds: 00-59
%w day of week 0-6 with Sunday==0
%W week of year: 00-53
%Y year: 0000-9999
%% %

Students can refer to the following link `http://www.sqlite.org/lang_datefunc.html` for more details, which might prove useful.

Also, you might need to do string concatenation, and "||" is the string concat operator in SQLite instead of "+", which is more commonly seen in other languages.

9 Handin

We expect the following components to be included in your handin (this is a reiteration of the Goal section of the handout):

- An E-R Diagram of your design (From Part 1).
- Your `import` application.
- Your `query` application.
- A README file, describing any bugs in your code (or asserting their absence)

10 Q&A

Here are some FAQs. If you have any question, check this section first to see whether there is an answer here.

- **Are we being graded on coding style, efficiency and commenting for our importer code for the ETL**?

  No, but if your importer is incorrect then having comments and neat code would help your grader in allotting partial credit where it is due.

- **For some airports, we cannot find their cities and states. What's their expected value in the table?**

  NULL or empty string.

- **For query 7, if the airport name doesn't exist, such as 'ABCD', should we output a line like ('ABCD', 0, 0) for it, or just omit it?**

  Omit it.

- For Query 6, you are expected to get **ALL** airlines, along with the number of flights for that airline for the given day. The question is not "get the airlines that have a flight on the given day". Therefore, an airline could have 0 flight for a day and you would want to have that in your output.

- **Are we allowed to use WITH clauses for queries 9 and 10?**

  Yes.

- **Should we check if the user inputs the correct number of args for our queries?**

  Not necessary. You can assume that the correct number of args is always given.

- **For the middle hop in query 10, would the hop for XXX → JFK then LGA → YYY be a valid path from XXX to YYY since JFK and LGA are both in New York, NY?**

  No. You always have to depart from the airport you just arrived at.

11 Appendix

11.1 Testing

You can always test your queries by running SQL commands in SQLite. Moreover, in the tests directory, there are a series of files that contain example input (the input file), the expected output (the output file), and number of rows/columns of that output (the count file). Use these to see how your queries are performing!

11.2 Storing and running queries in a file

A query may be long and difficult to copy-paste in sqlite3 over and over to test minor changes. To alleviate this, we suggest that you save your query in a file, say 'query.sql', then to run it use the following terminal command:

```
sqlite3 path/to/db < query.sql
```

12 Final Words

Good luck, and as always, feel free to ask TA's any questions!