Problem Set 1: SQL

Release Date: September 9, 2024

Due Date: September 18, 2024 by 11:59 pm ET

1 Introduction

The purpose of this assignment is to give you hands-on experience with the SQL programming language. SQL is a declarative language in which you specify the data you want in terms of its properties. This assignment focuses on the `SELECT` subset of SQL, which is all about querying data rather than modifying it.

We will be using SQLite, which provides a standards-compliant SQL implementation. In reality, there are slight variations between the SQL dialects of different vendors (PostgreSQL, MySQL, SQLite, Oracle, Microsoft, etc.)—especially with respect to built-in functions. The SQL tutorial at `http://sqlzoo.net/`, provides a good introduction to the basic features of SQL. After following this tutorial you should be able to answer most of the problems in this problem set. A more detailed tutorial is available at `https://www.sqlitetutorial.net/`.

To install SQLite, you can simply use the command `apt install sqlite3` on Debian-based Linux distributions like Ubuntu, or `brew install sqlite3` on Mac. Downloads for the pre-compiled binaries can be found at `https://sqlite.org/download.html` for Windows (as well as Linux and Mac, if you'd prefer to install with the binaries). If you use a pre-compiled binary, you might have to make sure that the path to your installed directory is in the `PATH` environment variable.

The SQLite SELECT documentation will be helpful to you, and you can access all the other SQLite documentation on that site as well. You may also wish to refer to Chapter 5 of "Database Management Systems."

Note that we are using SQLite 3.37.2 in the autograder.

2 Dataset

For this assignment, we use a dataset derived using data from the Massachusetts Bay Transportation Authority's (MBTA) Open Data Portal. This dataset includes information about the MBTA's subway lines (e.g., lines, stations, ridership, etc.). Everything you need to understand the dataset is contained in your SQLite database and this document.

The database tables include:

`lines`: contains the ID and name of each T rail line in the dataset. Note that the Silver Line is not included; even though its name follows the same pattern, it does not operate on rails and is categorized as a Bus by the MBTA.

`routes`: details the different rail routes that operate on the lines; for example, the Red Line has one route which services Braintree to Alewife and another distinct route which services Ashmont to Alewife. The table contains a unique route ID, the route name, the ID of the line it belongs to, the IDs of the first and last stations of that route, and the direction, given in a binary field and a string description.

`stations`: contains the ID and name of each T station in the dataset. Note that this stations list is a snapshot of the past (to be specific, June 2020); newly opened stations, such as the Union Square station, are not included, and vintage stations, such as the BU West station, are included.

`station_orders`: describes the order of stations along each route. The table contains a route ID, station ID, number of the station in the order of that route, and distance (in miles) from the previous station to the current station. Note that all initial stations of each route have a set distance of 0 since there is no previous station. In addition, all Green Line distances are set to `NULL` since the MBTA did not provide a complete dataset in this case.

`gated_station_entries`: contains the number of people entering the gates of each station in half-hour increments. The table contains the service date and time, station and line IDs (some stations have gates for multiple lines; i.e. Downtown Crossing is

a station on both the Red and Orange lines), and number of entries. Note that the number of gated entries in the database table are sometimes not integers; if you aggregate over multiple lines on the same station most should sum to an integer (except for stations which exist on the Silver Line, like South Station).

`rail_ridership`: includes ridership trends for Fall 2017, 2018, and 2019 over various time slices of the week. The table contains the season (i.e. "Fall 2017"), line ID, direction in binary, time period ID, and station ID as the primary key; further, we have the total number of people who got *on* the train, the total number of people who got *off* the train, the number of non-holiday days in operation during that portion of the season, the average number of people who got *on* the train per operating day, the average number of people who got *off* the train per operating day, and the average *flow*, or number of people who were in the train but did not board or disembark at that station.

`time_periods`: identifies time slices of the week used to interpret rail ridership patterns. The table contains an ID for each time period (e.g., `time_period_01`), the type of day (e.g., `weekday`), a textual description of the time period (e.g., `AM_PEAK`), and the start and end times for the time slice in 24 hour notation (e.g., `03:00:00`).

3 Using the Database

Download the database file from the Lecture Notes and Assignments page on the class website. You can access detailed information about each table or test your solutions interactively in the shell. To access the SQLite shell for the database, `cd` to the directory where you saved the database file (`.sqlite`) and run:

```
sqlite3 mbta.sqlite
```

Once in the SQLite shell, there are two kinds of commands useful to a database user: client meta-commands and SQL commands.

3.1 Client Meta-Commands

The most important one, of course, is `.help`, which gives you help on meta-commands:

```
sqlite> .help
.auth ON|OFF             Show authorizer callbacks
.backup ?DB? FILE        Backup DB (default "main") to FILE
.bail on|off             Stop after hitting an error.  Default OFF
.binary on|off           Turn binary output on or off.  Default OFF
.cd DIRECTORY            Change the working directory to DIRECTORY
.changes on|off          Show number of rows changed by SQL
...
```

We can list all the table schemas in the database with `.tables`:

```
sqlite> .tables
gated_station_entries  routes                 time_periods
lines                  station_orders
rail_ridership         stations
```

We can view the schema (recall, that the "schema" of a database is like a class definition in an object oriented language) of a given table using `.schema <table_name>`:

```
sqlite> .schema routes
CREATE TABLE routes (
  route_id INTEGER,
  line_id TEXT,
  first_station_id TEXT,
```

```
  last_station_id TEXT,
  direction INTEGER,
  direction_desc TEXT,
  route_name TEXT,
  PRIMARY KEY (route_id)
);
```

You can change the way the SQLite shell displays the result sets to suit you better. In particular, you may find the commands `.header on` and `.mode column` useful.

Finally, to exit the SQLite shell, you can use `.exit`

3.2 SQL Commands

All SQL queries in SQLite must be terminated with a semi-colon. For example, to get a list of all records in the `stations` table, you would type:

```
SELECT * FROM stations LIMIT 10;
```

This query requests a maximum of 10 rows from the table. Using `LIMIT` in this manner one can explore the data small bits at a time. If you really wanted to produce all the records, though, the query is:

```
SELECT * FROM stations;
```

You can use `Ctrl+C` to end a query that is taking too long. Note that using the `LIMIT` keyword, when used by itself, offers no guarantee on which 10 rows from the result are returned, so do not assume an ordering.

4 Questions

Q1. Find all stations at least 1 mile away from the previous station. Report the station ID, route ID, and distance (in miles) to the previous station, sorted by decreasing distance. Break ties in distance by sorting by route ID and then station ID, both in ascending order. Display the output like '`place-abcde|0|10.0`'. **[5 points]**

Q2. Find the first and last station for each route on each line. Report the line name, route direction description, and first and last station names. Sort the results by line name, direction description, first station name, and then last station name—all in ascending order. Display the output like '`Green Line|East|Some Station Name|Another Station Name`'.

[5 points]

Q3. Report the historical total `total_ons` on weekdays between 4:00 PM and 6:30 PM of each season for the "Kendall/MIT" Red Line station. Report the season, line ID, direction binary, and `total_ons`, sorted by season and direction binary in ascending order. Display the output like '`Fall 2017|red|0|21390`'. **[10 points]**

Q4. For route, direction and station in the route, find the distance of the station from the start of the route. Report the `route_id`, `direction`, `route_name`, `station_id`, distance in miles from start. Exclude the Green Line since the distance between stations is missing. Sort the results by distance from the start in miles, break ties by `route_id` and `direction` (all ascending). '`0|0|Wonderland to Bowdoin|place-wondl|0.0`'. **[10 points]**

Q5. For each season, for each station, find the average number of line service days. (That is, find the average of the number of `number_service_days` over different lines, directions and time periods, but do not average over different values for `season`.) Report the station name, season, and averaged `number_service_days` value, sorted by that average value in descending order. Break ties with season and then station name, both in ascending order. Display the output like '`South Station|Fall 2017|10`'. **[10 points]**

Q6. Find the station with the most total gated entries over the summer of 2021 (June, July, August of 2021). Report the station name and the number of gated entries. Display the output like '`Airport|42.0`'. The output should be one tuple. You do not need to cast the number of gated entries into an integer. **[10 points]**

Q7. For each line and time period, find all stations with the maximum number of people getting off the train, adding 'total_offs' for all seasons and directions. Report your answer as `line_id`, `time_period`, `station_name`, sum of 'total_offs'. Sort Results by sum of 'total_offs' (descending). Break ties by `line_id` and `time_period`, `station_name` (ascending). Display the output like '`red|AM_PEAK|South Station|1674914`'. **[10 points]**

Q8. Find every Orange Line station in Fall 2018 that, during `time_period_01` and the direction of 0, had a `total_ons` passenger count greater than average for all Orange Line stations during that same time period, same season, and in the same direction. Report the station name and the `total_ons` value. Sort the results by `total_ons` in descending order and then station name in ascending order. Display the output like '`Malden Center|21400`'. **[10 points]**

Q9. For every pair of distinct routes that share at least one stop, find the number of station they share. Two routes that differ only by direction are considered distinct for this question. Report `route1_id` $<$ `route2_id`, `route1_name`, `route2_name`, the smallest `station_id` among the stations these routes share twice, and the number of shared stations. Order by the number of shared stations (descending), and resolve ties by `route1_id`, `route2_id` (ascending). Display the output like '`2|6|(B) Government Center to Boston College|(B) Boston College to Government Center|place-alsgr|place-alsgr|25`'. The number of rows should be the number of pairs of distinct routes that share at least one stop.

[15 points]

Q10. For each line, in the Fall 2019 season, find the station with "maximally bypassed ratio". That is, the station "s" that has the largest ratio "(a - b)/a", where "a" is the the sum of `average_flow` values for all time periods and all directions of "s" and "b" is the total sum of: the sum of its `average_ons` and sum of its `average_offs` values. Therefore, the ratio "(a - b)/a" represents the proportion of people who bypassed one station. Report the station name, its line name, and its bypassed ratio. Sort the results by line name in ascending order. Display the output like '`Some Station|Green Line|0.912111111`'. HINT: You may need to use function CAST(total_flow AS REAL) to cast the summation of flows (i.e. "a" above) to real number. **[15 points]**

5 Submission

You may work in pairs on this problem set. Only one of you needs to submit on Gradescope, but the other member must be added as a group member on the submission.

Answer each question in the corresponding file under the `submission` directory in the handout. Don't change the file names of your answers, i.e., your query for question 1 must be in a file named `q1.sql` etc.

After completing the homework, you can submit your queries by uploading the `.sql` files to the PSET 1 assignment on Gradescope: `https://www.gradescope.com/courses/846234`. See Piazza if you need the entry code. You can submit your answers as many times as you like.

Each submission will be graded based on whether the SQL queries fetch the expected sets of tuples in the correct order from the database. Note that your SQL queries will be auto-graded by comparing their outputs (i.e. tuple sets) to the correct outputs. Do not hard code answers or any intermediate values into your SQL queries. The only constants and string literals that appear in your solution should be the ones that appear in the problem statements (not including the examples). We will execute your queries using **different data** with the same schema. For your queries, the order of the output columns is important; their names (i.e. the SQL alias after the `AS` keywords) are not important.

The autograder from Gradescope should provide you with immediate feedback. The autograder will timeout after 5 minutes. If a query is taking too long, try changing it. All questions have solutions that run within a few seconds.