Problem Set 1: SQL

Release Date: September 11, 2023

Due Date: September 20, 2023 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 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 are sometimes not whole numbers in the table; if you aggregate over multiple lines on the same station most should sum to a whole number (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, 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* 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 and starter code from the Lecture Notes and Assignments page on the class website. To access the SQLite shell for the database, cd to the directory where you saved the database file 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 which are 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 name, and first and last station name. Sort the results by the line name, direction name, 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_ons` on weekdays between 4:00 PM and 6:29 PM (time period 6) per season for the "Kendall/MIT" Red Line station. Report the season, line ID, direction, and `total_ons`, sorted by the season and direction in ascending order. Display the output like '`Fall 2017|red|0|21390`'. **[10 points]**

Q4. Find the total length in miles and number of stations of each line's routes. Report the `route_id`, `direction`, `route_name`, number of stations, and length in miles for each route. Exclude the Green Line since the distance between stations is missing. Sort the results by total number of stations in descending order (break tie using total length in miles in descending). Display the output like '`0|0|Wonderland to Bowdoin|42|8.0`'. **[10 points]**

Q5. For each station in each season, 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 sum 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 by sorting by season and then station name, both in ascending order. Display the output like '`South Station|Fall 2017|10`'. **[10 points]**

Q6. Find the station(s) with the most gated entries over the summer of 2021 (June, July, August of 2021). Report the station name(s) and the number of gated entries. Display the output like '`Airport|42.0`'. The output should be one tuple. **[10 points]**

Q7. Find the station, time period, and season with the largest number of people who get off (the largest "`total_offs`"). A station may be associated with multiple directions; consider these directions to be distinct for the purposes of finding the largest `total_offs` (e.g., the `total_offs` for Kendall/MIT with a direction of 0 should be considered separately from the `total_offs` for Kendall/MIT with a direction of 1 when you are computing the largest `total_offs`). Report the `day_type`, `period_start_time`, `season`, `line_id`, `station_name`, and `total_offs` for this station. Display the output like '`saturday|9:00:00|Fall 2019|blue|Wonderland|21510`'. **[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 that was greater than average for all Orange Line stations at 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. Find the station with most number of routes passing through it. (E.g. North Station has six routes passing through it: orange line in both directions and two green lines in both directions) Report `station_name`, `route_id`, `line_id`, and total number of routes passing through the station. Sort the results by `line_id` in ascending order and then `route_id` in ascending order. Display the output like '`North Station|10|green|6`'. Note that the query should return n rows for n routes passing through the station. **[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.

Do not hard code answers into your SQL queries. We will execute your queries using different data with the same schema.

Answer each question in the corresponding file under the `pset1` directory in the handout. After filling in the queries, compress the folder by running the following command:

```
zip -j submission.zip pset1/*.sql
```

The `-j` flag lets you compress all the SQL queries in the zip file without path information. The grading scripts will not work correctly unless you do this.

Each submission will be graded based on whether the SQL queries fetch the expected sets of tuples from the database. Note that your SQL queries will be auto-graded by comparing their outputs (i.e. tuple sets) to the correct outputs. For your queries, the order of the output columns is important; their names are not. We will be comparing the output files using a function similar to diff. You can submit your answers as many times as you like.

We use the Autograder from Gradescope for grading in order to provide you with immediate feedback. The autograder will timeout after 10 minutes. If a query is taking too long, try changing it. All questions have solutions that run within seconds. Please be aware that we use a modified version of the database for grading purposes.

After completing the homework, you can submit your compressed folder `submission.zip` (only one file) to Gradescope: `https://www.gradescope.com/courses/583077`. See Piazza if you need the entry code.

6 Changes Since Release

1. Clarification of Q3: the question asks for between 4:00 PM and 6:30 PM, which is inclusive on both end, so the filter predicate should be period_start_time BETWEEN ‘16:00:00’ AND ‘18:30:00’.

2. Clarification of Q3: disgard the previous comment. the question asks for between 4:00 PM and 6:29 PM, which does not include 6:30PM, so the filter predicate should be period_start_time BETWEEN ‘16:00:00’ AND ‘18:29:59’.

3. Fixing some typo.