University of California, Los Angeles
Department of Computer Science

Computer Science 143 **Prof. Ryan Rosario**

Homework 1
Solutions

Notes:

1. Some sample appropriate answers are given for queries. The rest of the red text is prose explaining tradeoffs, and how design decisions can be made.
2. It is important to *load* the data and look at it before beginning.
3. Optional attributes in SQL queries are written in `[]`. Do not copy paste the brackets as you will get a syntax error.

Part 1: Schemas and Architecture

Suppose you are working for the data team at *Bird Scooter*, a Santa Monica based startup that aims to revolutionize how people get around on wheels.

How the Bird Scooter service works: a user installs an app on their phone and enters their credit card information. The app shows a map of deployed scooters nearby. The user scans a QR code on the scooter using the app, which activates the scooter for use and begins the clock for per-minute and per-use billing. The user rides the scooter for a distance, for a certain number of minutes. When the user is done with the scooter, he/she leaves it somewhere, and marks the trip as complete in the app.

Each scooter has an identifier, and assume that since Bird is a startup, it only has 10,000 scooters deployed. Each scooter also has a flag that specifies a scooter as online (deployed), offline (not in use for whatever reason), or lost/stolen. Finally, each scooter is assigned to a home location that rarely changes. For example, some scooters may be assigned to UCLA, some may be assigned to Santa Monica, and others to Austin. Occasionally, a Santa Monica scooter may be reassigned to UCLA depending on demand, but we expect that such a change is very rare.

Assume that Bird currently only 500,000 registered users. A user is someone that has installed the app. Each user has an identifier, but not all users have a credit card number on file as many users install the app and then never ride a scooter. There is also an expiration date [present if a credit card number is present] and an email address.

Assume, for simplicity, that the app communicates over the Internet directly to a database server. Being a startup, this is probably how it is done, actually.

Exercises

(a) Develop a schema for the `scooter` and `user` table based on the requirements and use case described on the previous page. Write the schema as a `CREATE TABLE` statement. Specify a primary key, or composite primary key using the correct syntax. Mark (with a comment) which attributes, if any, are foreign keys (to determine this, you may have to answer the other parts first). Try to minimize storage space.

The `scooter` Table

We need a unique identifier for each scooter, and this identifier (for now) must accept values up to 10,000. Trivially, this ID is unsigned. The proper data type is `SMALLINT`. We could also make it `AUTO_INCREMENT`ing if we wish but it is not strictly necessary if we have our own format for assigning IDs. Of course, if we choose a format that is not a number, we will need to use another data type entirely. This will serve as the `PRIMARY KEY`.

To specify if a scooter is offline, online or lost/stolen, we use an ENUM as the flag. We could also use a BIT(2), one bit for online/offline, and one for stolen/lost or not stolen/lost, but this may cause data integrity concerns (if both bits are set, it means online and lost/stolen, which does not make sense). Later on, we can add more bits or more values to the ENUM.

Home location is a bit more vague. We could use an ENUM, but as Bird grows, we will need to add values to this ENUM frequently, and this may be cause for concern if we are not careful with how we add values to it. We could also use some kind of integer (start with TINYINT) as an identifier for a particular home location, and then join on this column with a fact table containing a mapping from the integer to the string with the name of the location, if we need it.

Thus, we can write the following schema:

```
CREATE TABLE scooter (
    scooter_id SMALLINT NOT NULL [AUTO_INCREMENT],
    -- AUTO_INCREMENT can be omitted if we have a way of creating IDs.
    status     ENUM('online', 'offline', 'lost/stolen') DEFAULT 'offline',
     -- or NOT NULL instead of DEFAULT
    home       TINYINT,
    PRIMARY KEY (scooter_id)
    -- no foreign keys expected
);
```

Note that if we have a single column as a primary key, we can just specify it as such after defining the column:

```
CREATE TABLE scooter (
    scooter_id SMALLINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    status     ENUM('online', 'offline', 'lost/stolen') DEFAULT 'offline',
    -- or NOT NULL instead of DEFAULT
    home       TINYINT
    -- no foreign keys expected
);
```

In the above schema, I explicitly mark scooter_id as NOT NULL. While this is a good practice to follow, in MySQL the PRIMARY KEY declaration handles it for us. I also added a DEFAULT value for status, but this is not truly necessary since it is not specified in the problem. Instead of using a DEFAULT value, we can force the user to set the flag by specifying status as NOT NULL.

While it may seem like a good idea to also specify a scooter's last known location,but this is redundant, and would require us to UPDATE many rows over and over again each time a scooter is used and parked. scooter is a simple fact table, and it should not have to be updated except to add new scooters, or move scooters from locale to locale.

I did not specify anything about NULL values in the home location. Perhaps when we add a scooter, we don't need to assign it to a location, then I can have NULLs, otherwise, I would specify it as NOT NULL.

Best practices of NOT NULL: It depends on the business case. *However*, it is much easier to start with a column being marked NOT NULL and then later removing the restriction than it is to allow NULL values and then later place the NOT NULL restriction, because what would happen to all of the existing NULLs? Something to keep in mind. There is actually a lot of history behind this.

The `user` Table

Each user has a unique identifier. For simplicity, we can assume this to be a `MEDIUMINT` since there are currently 500,000 users and this gives us plenty of room to grow. We could also specify that it `AUTO_INCREMENT`s and thus also be `NOT NULL`. Of course, we can use a different data type for the IDs entirely, such as a `CHAR` like in the `youtube` examples. Not all users have a credit card associated with their accounts, so credit card number can be `NULL`. We **should** secure the credit card number by using industry best practices using some kind of hash, but it is not necessary for CS 143 (unless ever explicitly stated). There are federal regulations in terms of how to protect credit card numbers, but we will just use `MD5` as an example for this solution. The length of the credit card number depends on which credit cards we accept. By using an integer type, we can avoid worrying about this nuance, or can we? What happens if a credit card number starts with 0? For our purposes, we will ignore this fact, but it is important that we *clearly* understand the formats of our attributes before we assign data types. We also need an expiration date. Since not all users have a credit card number, they also will not have an expiration date. Expiration dates are written as `mm/yy` but we do not have this exact type in MySQL, so we can improvise by using a `DATE` column type. There is an edge case here though: if a card expires 4/2019, and it is 4/30/18 at home, but you are in a different timezone where it is already 5/1/18, the card may not work. We could research how expiration dates work, or we can just be safe and use a `TIMESTAMP`. Once the local time hits 5/1/18, the card will no longer work. Finally, we have an email address field. You can make up something reasonable here, but RFC 5321 states a max of about 255 characters.

Point: When designing a schema, it is crucial to understand the format of the data that will be inserted into the table, and how it may change over time.

Putting all of this together, *one* good solution is

```
CREATE TABLE user (
    user_id   MEDIUMINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    ccnumber  BIGINT,
    -- if we want a hash, we could use CHAR(32/64) or BINARY(32/64) on MD5(ccnumber)
    expiration TIMESTAMP,
    -- our application would just have to ignore the day part, and the time part.
    -- DATE (or less preferably) DATETIME may also work, depends on the edge case.
    email     VARCHAR(255) NOT NULL
    -- It wasn't clear in the problem, but the email address should be required.
);
```

(b) Each interaction between a user and a Bird Scooter is called a *trip*, and we will create a schema for this `trip` table. Assume each trip has a unique identifier. The app will use this database table to determine where an available scooter is located. It will also (somehow) be used to determine the amount to charge the user. Additionally, data scientists would like to be able to use this table to determine daily and hourly trends in when users start and end scooter use and also identify scooter hotspots: areas where people frequently activate scooters and park them (just assume a location is a GPS coordinate: latitude and longitude, which can be represented numerically). Write the `CREATE TABLE` statement for this table. Specify the primary key. Mark (with a comment) which attributes, if any, are foreign keys (to determine this, you may have to answer the other parts first). Try to minimize storage space.

Again, each trip has a unique identifier which we can define as some kind of integer, without more information to the contrary. We do not know how many trips we expect and would have to monitor the situation carefully. We will just use an `INT` with the expectation that we will have to promote. Each trip is also associated with a user, otherwise we cannot bill them! Each trip has a start `TIMESTAMP` and an end `TIMESTAMP`. The data scientists can use these values to study time trends and finance can use it to bill the customer, thus both of these must be `NOT NULL`. Each trip also has a start location and an end location. Knowing the proper data type requires knowing the format of GPS coordinates. For our sanity, we can just use a `DOUBLE`. Most datum formats allow 5 or 6 digits after the decimal point and we should store as precise a location as possible, so let's use 6 digits after the decimal. The part before the decimal ranges from -90 to +90, so we need 8 total digits (the sign doesn't count), so it is preferable to use a `DECIMAL(8, 6)`. Let's assume that this data must be collected and thus must be `NOT NULL`. If we have no GPS signal from the user, we cannot locate scooters for them anyway. Finally, we may also want to know which scooter was used, say, to locate a stolen or missing scooter.

One appropriate solution:

```
CREATE TABLE trip (
    trip_id         INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    user_id         MEDIUMINT,
    -- user_id is a foreign key that references user table.
    scooter_id      SMALLINT NOT NULL,
    -- scooter_id is a foreign key that references scooter table.
    trip_start_time TIMESTAMP NOT NULL,
    trip_end_time   TIMESTAMP NOT NULL,
    trip_start_lat  DECIMAL(8, 6) NOT NULL,
    trip_start_lon  DECIMAL(8, 6) NOT NULL,
    trip_end_lat    DECIMAL(8, 6) NOT NULL,
    trip_end_lon    DECIMAL(8, 6) NOT NULL,
    FOREIGN KEY (user_id) REFERENCES user(user_id),
    FOREIGN KEY (scooter_id) REFERENCES scooter(scooter_id)
);
```

Note that it was only necessary to specify the `FOREIGN KEY` in a comment, but we learned on 4/18 that we use a syntax to specify it directly.

(c) For the `trip` table, there are two ways that we can write data to the table from the app. First, we could insert a row when the user activates the scooter, and then modify the row when the user parks the scooter and ends the session. Second, we can simply cache the ride data on the phone, and at the end of the session transmit this data to the database server to be inserted as a row. What are the advantages and disadvantages of both of these methods? Which would you (an employee at Bird) prefer and why? Is there an even better way?

In the first method, we write a row to the database with a `NULL` end time and location. First off, this violates the schema for the `trip` table. Even if it did not violate it though, this constant modification of rows in the table each time a scooter is activated and parked induces a lot of load on the database system because each row is written once, and then modified once. This table thus becomes more **read and write heavy** than it needs to be.

On the other hand, if instead we cache the identifiers, start time and start location of a trip and then only write a row to the table once we park the scooter, we will have a data integrity nightmare. What happens if the phone loses network connectivity? What happens if the user switches off the phone? We lose the entire trip and the user does not pay. One way around this problem is to simply charge the user the maximum daily rate (like when you lose a parking voucher in a paid lot), but we *still* do not have a record that a ride even started!

Between these two cases, the first method would be preferable to the business; however, there are at least two better ways to handle this situation.

In the first, we split the `trips` table into two: `trip_starts` and `trip_ends` and we write a row once when the scooter is activated and we do not let the user scoot away until the record is written. Then when the scooter is parked, we write a row to the `trip_ends` table. To charge the user at the end of the month, we would then `LEFT JOIN` these two tables together on `trip_id` and compute the amount of time and a charge. The data scientists would also use this joined table. The start table would become **write heavy** and the end table would still be **read and write heavy** (the map of scooters makes this table read heavy), but the situation is not as severe as it was before.

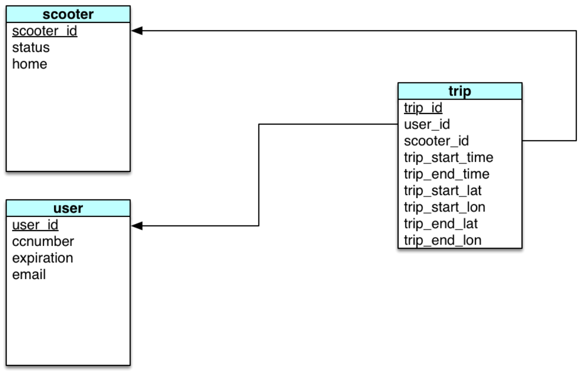
```
CREATE TABLE trip_starts (
    trip_id INT NOT NULL [AUTO_INCREMENT] PRIMARY KEY,
    -- AUTO_INCREMENT is optional if we have our own ID mechanism.
    user_id SMALLINT NOT NULL,
    scooter_id SMALLINT NOT NULL,
    start_time TIMESTAMP NOT NULL,
    start_lat DECIMAL(8, 6) NOT NULL,
    start_lon DECIMAL(8, 6) NOT NULL,
    FOREIGN KEY (user_id) REFERENCES user(user_id),
    FOREIGN KEY (scooter_id) REFERENCES scooter(scooter_id)
);
```

Again, we can specify the foreign key in the syntax, or we can just mention it in a comment. **On the exam**, use the proper notation, but this homework was released before we discussed this.

The second method uses one table called `trips` still, but adds an additional `action` flag as a column (`BIT` or `ENUM`) to mark the row as a `TRIP_START` or `TRIP_END`. The primary key then becomes composite: `trip_id` *and* `action`. This table is still **read write heavy**.

```
CREATE TABLE trips (
    trip_id INT NOT NULL,
    user_id SMALLINT NOT NULL,
    scooter_id SMALLINT NOT NULL,
    action_time TIMESTAMP NOT NULL,
    action_lat DECIMAL(8, 6),
    action_lon DECIMAL(8, 6),
    action     ENUM('trip_start', 'trip_end')
    -- or BIT(1)
    PRIMARY KEY(trip_id, action),
    FOREIGN KEY (user_id) REFERENCES user(user_id),
    FOREIGN KEY (scooter_id) REFERENCES scooter(scooter_id)
);
```

(d) Using the tables you just developed in all of Part 1, draw the schema diagram. For an example of a schema diagram, see Figure 2.8 in the text (page 47), or page 23 in the lecture slides for Lecture 2.



Some other things to think about:

- Would you include the number of minutes the trip lasted in the `trip` table? Why or why not?

  No, because this is redundant information and can cause data integrity issues if for some reason the difference in time does not match the trip length entered into the row, though we can add this as a constraint.

- What are the advantages and disadvantages of including the `charge` to the user in the `trips` table?

  The advantage is that it would allow fast querying of charges, and may not be a bad idea since a user is only charged according to the current fee structure. In other words, we would never be able to recalculate the charge. The disadvantage is redundancy. The charge is a function of the start time, end time and current fee, so it is not necessary to store this in the table.

Part 2: SQL and Relational Algebra

If you wish to check your syntax, you can load this dataset into MySQL by following the directions on CCLE under Homework 1.

Hints: For some of these queries, you will need to use functions on attributes. Check out the list of date and time functions here, as it should be very useful: `https://dev.mysql.com/doc/refman/5.7/en/date-and-time-functions.html`. **You do not need to memorize them!** Note that these are *not* aggregation functions because aggregation functions may take multiple inputs and produce one output per group. These functions take one input and return one output without using any grouping variables. This does not mean that your queries will not use the aggregation functions we discussed in class though.

Bay Area Rapid Transit (BART) is a subway system that stops at various places on the Bay Area Peninsula, City of San Francisco, underwater to Oakland and the East Bay. When a user inserts a ticket, or scans a pass card on a turnstile, BART records that a user entered the subway system. On exit, the passenger does the same thing to exit the station (inserts a ticket or scans a fare card) and BART records that the user's journey is complete. Throughput can be defined as the number of passengers that entered at origin A, and exited at destination B.

In this exercise, we will do various queries on this data to answer several interesting questions.

This dataset consists of BART ridership data from 2017. The schema for the two tables in this database are provided below in case you do not wish to use the data.

```
-- This table is for your own information to see which station is which.
CREATE TABLE station(
    Abbreviation CHAR(4),
    Description VARCHAR(1000),
    Location VARCHAR(23),
    Name VARCHAR(50)
);

CREATE TABLE rides2017(
    Origin CHAR(4),
    Destination CHAR(4),
    Throughput INT,
    DateTime DATETIME
);
```

Exercises

(a) Write a query to compute the total throughput (passengers, or number of trips) by *time of day* for the year of 2017. The result should contain only the hour of day, named `hour`, and the number of trips named `trips`.

```
SELECT
    HOUR(DateTime) AS hour,
    -- TIME(DateTime) is also fine since each row is an hour.
    SUM(Throughput) AS trips
FROM rides2017
GROUP BY hour;
```

If you run the query, you should get the following results.

```
+----------+---------+
| hour     | trips   |
+----------+---------+
| 00:00:00 |  319034 |
| 01:00:00 |   67497 |
| 02:00:00 |   10274 |
| 03:00:00 |    3989 |
| 04:00:00 |   60098 |
| 05:00:00 |  448126 |
| 06:00:00 | 1269004 |
| 07:00:00 | 2945005 |
| 08:00:00 | 4411567 |
| 09:00:00 | 3494007 |
| 10:00:00 | 1900128 |
| 11:00:00 | 1447244 |
| 12:00:00 | 1422823 |
| 13:00:00 | 1474797 |
| 14:00:00 | 1632535 |
| 15:00:00 | 2071627 |
| 16:00:00 | 3062411 |
| 17:00:00 | 4620962 |
| 18:00:00 | 4362644 |
| 19:00:00 | 2472355 |
| 20:00:00 | 1377196 |
| 21:00:00 | 1062459 |
| 22:00:00 |  920195 |
| 23:00:00 |  685018 |
+----------+---------+
```

(b) Write a query that lists the one pair of station codes that had the largest throughput on the weekdays in 2017.

```
SELECT
    Origin,
    Destination,
    SUM(Throughput) AS total
FROM rides2017
WHERE DAY(DateTime) >= 2 AND DAY(DateTime) <= 6
-- BETWEEN 2 AND 6 is equivalent *in this case*
GROUP BY Origin, Destination
ORDER BY total DESC
LIMIT 1;
```

You should get one row, the Powell to Balboa trip (San Francisco).

```
+--------+------------+-------+
| Origin | Destination | total |
+--------+------------+-------+
| POWL   | BALB        | 47161 |
+--------+------------+-------+
```

(c) Write a query that returns the 5 destinations that saw the highest average throughput on Mondays between 7am and 10am, and rank them from highest to lowest. Return the destinations and their averages.

```
SELECT
    Destination,
    AVG(Throughput) AS avg_trips
FROM rides2017
WHERE DAYOFWEEK(DateTime) = 2 AND HOUR(DateTime) >= 7 AND HOUR(DateTime) < 10
-- BETWEEN 7 AND 10 would not be accepted on the exam because the HOUR() function
-- truncates times. 10:59:59am is evaluated to 10, yet it's after 10am. Careful!
GROUP BY Destination
ORDER BY avg_trips DESC
LIMIT 5;
```

You should get the following 5 destination stations:

```
+-------------+-------------+
| Destination | avg_trips   |
+-------------+-------------+
| EMBR        |    188.0382 |  -- Embarcadero Station (298 Market St, SF)
| MONT        |    175.1023 |  -- Montgomery Station (598 Market St, SF)
| CIVC        |     71.5688 |  -- SF Civic Center (1150 Market St)
| POWL        |     62.9843 |  -- Powell Station (899 Market St, SF)
| 12TH        |     39.9817 |  -- Oakland City Center (1245 Broadway)
+-------------+-------------+
```

If instead you used the `BETWEEN` keyword, your results are as follows. Notice the difference?! This is why we won't be able to accept it on the exam.

```
+-------------+-------------+
| Destination | avg_trips   |
+-------------+-------------+
| EMBR        |    158.7725 |  -- Embarcadero Station (298 Market St, SF)
| MONT        |    150.2950 |  -- Montgomery Station (598 Market St, SF)
| CIVC        |     60.9626 |  -- SF Civic Center (1150 Market St)
| POWL        |     57.2327 |  -- Powell Station (899 Market St, SF)
| 12TH        |     33.3756 |  -- Oakland City Center (1245 Broadway)
+-------------+-------------+
```

I did not specify how to compute the average. Some students used a subquery to collapse out the `Origin` column, sum the throughputs in a subquery and then compute the average in the outer query. This is more complex than I had intended when I wrote my original solution, but works perfectly fine, and the result is more realistic.

```
SELECT
    Destination,
    AVG(total) AS average
FROM (
    SELECT
        Destination,
        SUM(Throughput) AS total
    FROM rides2017
    WHERE DAYOFWEEK(DateTime) = 2 AND HOUR(DateTime) >= 7 AND HOUR(DateTime) < 10
    GROUP BY Destination, DateTime
) sums
GROUP BY Destination
ORDER BY average DESC
LIMIT 5;

+-------------+-----------+
| Destination | average   |
+-------------+-----------+
| EMBR        | 8475.6481 |
| MONT        | 7892.5741 |
| CIVC        | 3208.6667 |
| POWL        | 2831.9630 |
| 12TH        | 1779.1852 |
+-------------+-----------+
```

Note that what was important, the order, remains the same!

We will leave the required problem at that, but a more interesting problem would have been to find the top 5 destinations for morning rush hour, and and also the top 5 for *evening* rush hour. To get the top 5 destinations of evening rush hour, we use the following query and get the following results:

```
SELECT
    Destination,
    AVG(Throughput) AS avg_trips
FROM rides2017
WHERE DAYOFWEEK(DateTime) = 2 AND HOUR(DateTime) >= 17 AND HOUR(DateTime) < 19
GROUP BY Destination
ORDER BY avg_trips DESC
LIMIT 5;
```

```
+-------------+------------+
| Destination |  avg_trips |
+-------------+------------+
| PHIL        |    39.1345 |  -- Pleasant Hill/Contra Costa/Walnut Creek (East Bay)
| 24TH        |    38.0749 |  -- Mission District (2800 Market St, SF)
| DUBL        |    38.9378 |  -- Dublin/Pleasanton
| FRMT        |    37.2117 |  -- Fremont (East Bay, Last Stop)
| BALB        |    37.0520 |  -- Balboa (SF)
+-------------+------------+
```

If instead you use `BETWEEN`, you would get the following:

```
+-------------+------------+
| Destination |  avg_trips |
+-------------+------------+
| DUBL        |    34.3033 |  -- Dublin/Pleasanton
| PHIL        |    33.4651 |  -- Pleasant Hill/Contra Costa/Walnut Creek (East Bay)
| 24TH        |    33.1783 |  -- Mission District (2800 Market St, SF)
| FRMT        |    32.5102 |  -- Fremont (East Bay, Last Stop)
| BALB        |    31.1904 |  -- Balboa (SF)
+-------------+------------+
```

And with the subquery:

```
SELECT
    Destination,
    AVG(total) AS average
FROM (
    SELECT
        Destination,
        SUM(Throughput) AS total
    FROM rides2017
    WHERE DAYOFWEEK(DateTime) = 2 AND HOUR(DateTime) >= 17 AND HOUR(DateTime) < 19
    GROUP BY Destination, DateTime
) sums
GROUP BY Destination
ORDER BY average DESC
LIMIT 5;
```

```
+-------------+-----------+
| Destination | average   |
+-------------+-----------+
| 24TH        | 1610.7778 |
| PHIL        | 1494.7222 |
| BALB        | 1484.1389 |
| DELN        | 1422.6667 |
| DUBL        | 1409.3333 |
+-------------+-----------+
```

(d) It is 2018. Suppose you are working with BART to expand stations and services based on 2017 ridership data. You want to know which stations are *overcrowded*. Suppose we consider an originating station as overcrowded if the maximum hourly throughput across all one-hour periods in the year is greater than 100 times the average hourly throughput across all one-hour periods in the year. Write a query that retrieves all originating stations that were overcrowded in 2017. (Passengers spend more time in originating stations because they must wait for the train there.)

This problem was, well, quite problematic. My main interest here was the `HAVING` clause vs. the `WHERE` clause. Since we had not learned subqueries yet, there was only one reasonable interpretation of the problem: group by the origin, and select the maximum and average throughputs and use them in a `HAVING`.

```
SELECT
    Origin,
    MAX(Throughput) AS max_throughput,
    AVG(Throughput) AS avg_throughput
FROM rides2017
GROUP BY Origin HAVING max_throughput > 100 * avg_throughput;
```

or more succintly

```
SELECT
    Origin
FROM rides2017
GROUP BY Origin
HAVING MAX(Throughput) > 100 * AVG(Throughput);
```

We get the following results.

```
+--------+
| Origin |
+--------+
| 24TH   |  -- Mission District (2800 Market St, SF)
| FTVL   |  -- Fruit Vale (Oakland Neighborhood)
| GLEN   |  -- Glen Park (2901 Diamond St, SF)
| MLBR   |  -- Millbrae (End of Line, connect with Caltrain)
| NBRK   |  -- North Berkeley
| PLZA   |  -- El Cerrito Plaza (and Albany, Kensington)
| ROCK   |  -- Rockridge (Oakland neighborhood on Hayward Fault)
+--------+
```

But, there is another interpretation of the problem that requires a more sophisticated query involving a subquery. Some found the more natural interpretation was to first group by the `Origin` and sum the `Throughput` across all `Destination` and `DateTime`. **This is performed in a subquery.** We can then compute the average by dividing the sum by the number of rows that were included in the sum, outside of the subquery and place the definition of *overloaded* in the `HAVING` clause. Note that there are several ways to compute an average, and if it were critical, I would be explicit.

```
SELECT
    Origin
FROM (
    SELECT
        Origin,
        SUM(Throughput) AS rides_by_origin_time
    FROM rides2017
    GROUP BY Origin, DateTime
) sums
GROUP BY Origin
HAVING MAX(rides_by_origin_time) > 100 * AVG(rides_by_origin_time);
```

This does yield the empty set, but if we reduce the 100 to something smaller, we get results. So, while I hoped the solutions would be simpler, good job to anyone that identified this interpretation!

(e) Suppose we take the **result** from part (a) and call it `hourly_ridership`. Given the following query, write the equivalent expression in the relational algebra.

```
SELECT
    hour,
    trips / 100
FROM hourly_ridership
WHERE (hour >= 7 AND hour < 10) OR (hour >= 17 OR hour < 19);
```

$$\Pi_{hour,trips/100}\left(\sigma_{(hour \geq 7 \wedge hour < 10) \vee (hour \geq 17 \vee hour < 19)}\left(\texttt{hourly\_ridership}\right)\right)$$

(f) Suppose we want to study how the weather affects how busy particular stations are. In the `Occupancy` relation, we have the name of a station called `Station`, the `DateTime`, and the number of people passing through the station called `Riders`, as attributes. In the `Weather` relation, we have the `Station`, `DateTime`, `Temperature`, `Condition` (for simplicity assume a string, like "cloudy") attributes. We are *only* interested in comparing occupancy during "sunny" hours and "rainy" hours, and we only care about `Station`, `DateTime`, `Riders` and `Condition`. Write an expression in the relational algebra that represents this context.

$$\Pi_{Station,DateTime,Riders,Condition}\left(\sigma_{condition=\text{“}cloudy\text{”} \vee condition=\text{“}sunny\text{”}}\left(\texttt{Occupancy} \bowtie \texttt{Weather}\right)\right)$$

The following is also valid, though more complex.

$$\Pi_{Station,DateTime,Condition}\Big(\\ \left(\texttt{Occupancy} \bowtie \rho_{\texttt{Filtered}}\left(\Pi_{Station,DateTime,Condition}\left(\sigma_{condition=\text{“}sunny\text{”} \vee condition=\text{“}rainy\text{”}}\left(\texttt{weather}\right)\right)\right)\right) \\ \Big)$$