University of California, Los Angeles
Department of Computer Science

Computer Science 143 **Prof. Ryan Rosario**

Homework 3

Due Sunday, February 5, 2023 11:59pm ***via Gradescope***

Part 1: SQL

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. We have two relations in this dataset: `Station` and `RideCount`. `Station` has data about BART stations and `RideCount` contains counts of passengers that travelled between two stations per hour, per date. For example, it contains the number of passengers that rode between Millbrae and San Bruno during the 4pm hour on January 2, 2017.

```
-- The schema isn't exactly set up this way in the HW2 tarball since we are not modifying data,
-- we don't need to worry about referential integrity. The proper schema is provided for your reference.
CREATE TABLE station(
    Abbreviation char(4) PRIMARY KEY,
    Location varchar(23) NOT NULL,
    Name varchar(50) NOT NULL
);

CREATE TABLE ridecount(
    Origin char(4) REFERENCES station(Abbreviation),
    Destination char(4) REFERENCES station(Abbreviation),
    Throughput integer,
    TStamp timestamp,
    PRIMARY KEY(Origin, Destination, TStamp)
);
```

Exercises

(a) Write a query that computes the total number of trips that started at each BART station. Output the `Origin` and total Throughput as `total_throughput`.

```
 origin | total_throughput
--------+------------------
 12TH   |          1323517
 16TH   |          1313113
 19TH   |          1322154
 24TH   |          1336186
 ASHB   |           558777
 BALB   |          1160154
 BAYF   |           586765
 CAST   |           289594
 CIVC   |          2539919
 COLM   |           426739
(10 rows)
```

(b) Write a query that computes the total number of trips that started at each BART station. Only keep stations that had more than 1,000,000 boardings. Output the station `Name`, `Location` and total Throughput as `total_throughput`.

```
                Name                 |   Location    |   total_throughput
-------------------------------------+---------------+-------------------
 12th St. Oakland City Center (12TH) | Oakland       |           1323517
 16th St. Mission (16TH)             | San Francisco |           1313113
 19th St. Oakland (19TH)             | Oakland       |           1322154
 24th St. Mission (24TH)             | San Francisco |           1336186
 Balboa Park (BALB)                  | San Francisco |           1160154
 Civic Center/UN Plaza (CIVC)        | San Francisco |           2539919
 Downtown Berkeley (DBRK)            | Berkeley      |           1260580
 Embarcadero (EMBR)                  | San Francisco |           3895743
 Montgomery St. (MONT)               | San Francisco |           4029402
 Powell St. (POWL)                   | San Francisco |           3339266
(10 rows)
```

(c) Write a query that computes movement among cities. That is, compute the total number of trips (`Throughput`) between city A and city B and call it `total_rides`. For example, we want to know how many trips started in Oakland and ended in Fremont, and vice verse. Output the city (`Location`) corresponding to `Origin`, the city `Location` corresponding to `Destination` and `total_rides`. Do not output `Origin` or `Destination`, as that would not make sense. Sort from largest to smallest `total_rides`.

```
   location    |   location    | total_rides
---------------+---------------+-------------
 San Francisco | San Francisco |     6406085
 San Francisco | Oakland       |     3923381
 Oakland       | San Francisco |     3835810
 San Francisco | Berkeley      |     1113659
 Berkeley      | San Francisco |     1059120
 San Francisco | Walnut Creek  |      924245
 Walnut Creek  | San Francisco |      919046
 Oakland       | Oakland       |      869137
 San Francisco | El Cerrito    |      705177
 El Cerrito    | San Francisco |      650115
(10 rows)
```

(d) Write a query that finds the maximum `Throughput` ever recorded in this dataset, as well as the `Origin` and `Destination` for this trip, and the date/time (`Tstamp`). How can we do it **without** using `LIMIT` and `ORDER BY`?

```
 origin | destination |       tstamp        | throughput
--------+-------------+---------------------+------------
 24TH   | CIVC        | 2017-01-21 16:00:00 |       1826
(1 row)
```

Part 2: More SQL

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. In addition to what we discussed in lecture, you may find the functions `EXTRACT`, `EPOCH`, `CEILING`, `COALESCE` to be useful.

For these exercises we will write some queries to determine how much to charge Bird Scooter users for each ride they take. The tarball for HW3 contains some simulated (fake) data that you can use to test the syntax of your queries. You may notice that the the data types for these simulated relations are most `varchar` and there are no primary keys specified. This is so we do not give away the schema for those still completing HW2. We have the following relations:

```
customer(user_id, ccnum, expdate, email)
trip_start(trip_id, user_id, scooter_id, time, lat, lon)
trip_end(trip_id, user_id, scooter_id, time, lat, lon)
```

(a) Write a query that computes two new columns: (1) the elapsed time (in minutes) of each trip, and (2) the total cost of the trip. The trip charge is computed as follows: $1 flat rate for each trip plus $0.15 per minute. Fractional minutes should be rounded up (ceiling), so 4.02 minutes becomes 5 minutes. If the trip does not have an end time (scooter was stolen etc.), the length of the trip shall be 24 hours (1440 minutes) and the user should be charged based on 24 hours of use. Your results should include the `trip_id`, `user_id`, `trip_length` and `trip_cost`. There are at least two ways to do this problem. We prefer the method with the subquery and/or join. Compute the length of each trip first, and then compute the cost. Order the results by `trip_id` in ascending order. **Also, submit the top 10 rows of your output, without any special ordering.**

You should get the following output. Pay attention to the `trip_cost` column.

```
 user_id | trip_id | trip_length | trip_charge
---------+---------+-------------+-------------
   20685 |       0 |           2 |        1.30
   34808 |       2 |           3 |        1.45
   25463 |       3 |        1440 |      217.00
   26965 |       4 |           2 |        1.30
     836 |       5 |           1 |        1.15
    3260 |       6 |           5 |        1.75
   13850 |       7 |           3 |        1.45
   23528 |       9 |           4 |        1.60
   35829 |      11 |        1440 |      217.00
   18494 |      12 |        1440 |      217.00
(10 rows)
```

(b) Modify your query so that it computes the *total* amount that each user has spent on Bird Scooter. Again, use your previous response and assume that we did not store the intermediate result from the previous part. Report the `user_id` and the total amount spent as `total_spent`. Sort by `user_id` in ascending order. Be careful here.

You should get the following output:

```
 user_id | total_spent
---------+-------------
       0 |      662.00
       1 |        8.25
       2 |      674.90
       3 |       14.80
       4 |      885.10
       5 |      445.30
       6 |       17.70
       7 |       11.15
       8 |      233.40
       9 |      666.65
(10 rows)
```

(c) In class, we learned several ways to classify joins. List all of the adjectives we can use to describe the join in part (a). Your choices are:

Inner	Outer	
Left	Right	Full
Equi	Non-equi	
Natural	Self	

(d) In our schema, we have specified that `user_id` is the primary key for `customer`. If we want to specify that no two users can share the same credit card number, and no two users can share the same email address, what constraint should we put on those columns instead? **Hint:** You may need to look at the PostgreSQL documentation, particularly the Constraints subchapter.

(e) **Application Programming Exercise.** (Please submit this as a separate file `hw3.py`): Write a Python program that connects to the database, executes the query you wrote in the previous part, and produces a report that looks like the following. The format does not need to be identical, just close enough to show that you can use the output of the database and do something with it.

```
BIRD SCOOTER
User Charges for 2021

User ID        Charge
-----------    -----------
0               $ 662.00
1               $ 8.25
etc...
```

Hint: Practically all of the code is already written for you in the Lecture 7 slides. All you need to do is substitute in your query, eliminate the query parameter from my example, and then write some code to make the final computation.

Solutions:

1a)
```
SELECT
    origin,
    SUM(throughput) AS total_throughput
FROM ridecount
GROUP BY origin
ORDER BY origin;
```

1b)
```
SELECT
    name,
    location,
    SUM(throughput) AS total_throughput
FROM station
INNER JOIN ridecount
ON station.abbreviation = ridecount.origin
GROUP BY
    name,
    location
HAVING SUM(throughput) > 1000000
ORDER BY name;
```

1c)
```
SELECT
    orig.location,
    s.location,
    SUM(throughput) as total_rides
FROM(
    SELECT *
    FROM ridecount r
    JOIN station s
    ON r.origin=s.abbreviation ) orig
JOIN station s
ON s.abbreviation = orig.destination
GROUP BY
    s.location,
    orig.location
ORDER BY total_rides DESC;
```

1d)

```
SELECT
    origin,
    destination,
    tstamp,
    throughput
FROM ridecount
WHERE throughput = (SELECT MAX(throughput) FROM ridecount);
```

2a)

```
SELECT
    sub.user_id,
    sub.trip_id,
    trip_length,
    (1+.15*trip_length) AS trip_charge
FROM (
    SELECT
        s.user_id,
        s.trip_id,
        coalesce(ceiling(extract(epoch FROM (e.time-s.time)/60)),1440)
    AS trip_length
    FROM trip_start s
    LEFT OUTER JOIN trip_end e
    ON s.trip_id=e.trip_id ) sub
ORDER BY trip_id
LIMIT 10;
```

2b)

```
SELECT
    sub2.user_id,
    SUM(trip_charge) AS total_spent
FROM (
    SELECT
        sub1.user_id,
        (1+.15*trip_length) AS trip_charge
    FROM (
        SELECT
            s.user_id,
            s.trip_id,
            coalesce(ceiling(extract(epoch FROM (e.time-
            s.time)/60)),1440) AS trip_length
        FROM trip_start s
        LEFT OUTER JOIN trip_end e
        ON s.trip_id=e.trip_id ) sub1
    ) sub2
GROUP BY user_id
ORDER BY user_id;
```

2e)

```
import psycopg2
connection = psycopg2.connect(user="root", password="2023winter",
host="localhost", port="5432", database="cs143")
connection.autocommit = True
with connection.cursor() as cur:
    cur.execute("""
        SELECT sub2.user_id, SUM(trip_charge) AS total_spent
        FROM (
            SELECT sub1.user_id, (1+.15*trip_length) AS trip_charge
            FROM (
                SELECT s.user_id, s.trip_id,
                coalesce(ceiling(extract(epoch FROM (e.time-
s.time)/60)),1440) AS trip_length
                FROM trip_start s
                LEFT OUTER JOIN trip_end e
                ON s.trip_id=e.trip_id
                ) sub1
            ) sub2
        GROUP BY user_id
        ORDER BY user_id
        LIMIT 10;
        """)
    rows = cur.fetchall()
    print("BIRD SCOOTER")
    print("User Charges for 2021"+"\n")
    print("User ID"+"\t"+"Charge")
    print("------- ------")
    for result in rows:
        uid, total_spent = result
        print("{} \t\t{}".format(uid, total_spent))
```

Part 2: SQL Schemas

Use the Bird Scooter scenario from Homework 1.

```
Scooter(scooter_id, brand, is_online)
# scooter_id is some value that identifies a scooter.
# brand is a code that represents what type of scooter it is (e.g. Segway, Xiaomi)
# is_online just indicates whether or not the scooter is able to be used.

User(user_id, ccnum, expdate, email, name)
# user_id is some value that identifies the user
# ccnum is the user's credit card number (hashed, hopefully!) and may be null, and may change.
# expdate is the credit card expiration date and may also be null, and may change.
# email is the user's email address.
# name is the, well, user's name

Trip(trip_id, user, scooter, start_time, end_time)
# trip_id is some value that identifies the trip.
# user uniquely identifies the user that took the trip.
# scooter uniquely identifies the scooter that the user rode for this trip.
# start_time and end_time represent the start and end times of the trip.
```

In Homework 1, we created a relational schema and diagram for the Bird Scooter example. In this problem, we will create a SQL schema using the `CREATE TABLE` syntax. This means we also need to pick the proper data types for each column. For a description of how Bird Scooter works, see Homework 1.

We need a table to represent a `scooter`. Each of the following statements is designed to give you a hint as to the proper data type.

1. Each scooter has an identifier `scooter_id`, a number. Since Bird is a startup, we assume that there are no more than 10,000 scooters.
2. Each scooter has a flag `status` that marks it as online, offline (broken etc.), and lost/stolen. Each scooter can have only one of these states at a time, and must have a state.

We need a table to represent a `customer` (`user` is a system keyword so I will not use it):

1. Each user has an identifier `user_id`, a number, and we assume that Bird has at most 500,000 users for now.
2. A user is just someone that installed the app, not necessarily someone that will use a scooter. Thus, they may, or may not have a credit card number `ccnum` (16 digits) and expiration date `expdate`. Expiration dates usually look like MM/YY, but to make this simpler so you can use a more apparent data type, it is safe to assume that the card expires at midnight (00:00) on the 1st of the month.
3. Each user must have an `email` address. Assume an email address length is at most 100.

We need a table to represent a `trip`. To keep it simpler, we will include start and end information in this table, but the end of trip information may be missing. Each trip is associated with:

1. a unique identifier `trip_id`, a number. Assume that the total number of rides is not small.
2. exactly one user `user_id` and exactly one scooter `scooter_id`.
3. a `start_time` and `end_time`, which includes the date.
4. a `pickup` and `dropoff` location as a GPS coordinate (a latitude/longitude pair). *Hint:* See the documentation here. Note that latitude and longitude together form a point on a Cartesian plane (actually a sphere, but we will assume Cartesian plane for this problem).

Exercise. Write the SQL schema for the tables discussed above using `CREATE TABLE`. Specify a primary key, or composite primary key using the correct syntax. Specify the proper foreign key relationship on each table (if one exists) using the proper syntax. Try to minimize storage space because we can always promote later.

Solutions:

```
CREATE TYPE status_setting AS ENUM (‘online’, ‘offline’, ‘lost/stolen’);
CREATE TABLE scooter(
	scooter_id smallint,
	status status_setting NOT NULL,
	PRIMARY KEY (scooter_id) );

CREATE TABLE customer(
	user_id integer,
	ccum character(16),
	expdate date,
	email varchar(100)
	PRIMARY KEY (user_id) );

CREATE TABLE trip(
	trip_id integer,
	user_id integer,
	scooter_id smallint,
	start_time timestampz,
	end_time timestampz,
	pickup point,
	dropoff point,
	PRIMARY KEY (trip_id),
	FOREIGN KEY (user_id) REFERENCES customer(user_id),
	FOREIGN KEY (scooter_id) REFERENCES scooter(scooter_ID) );
```