Table of Contents

Part 1: Database

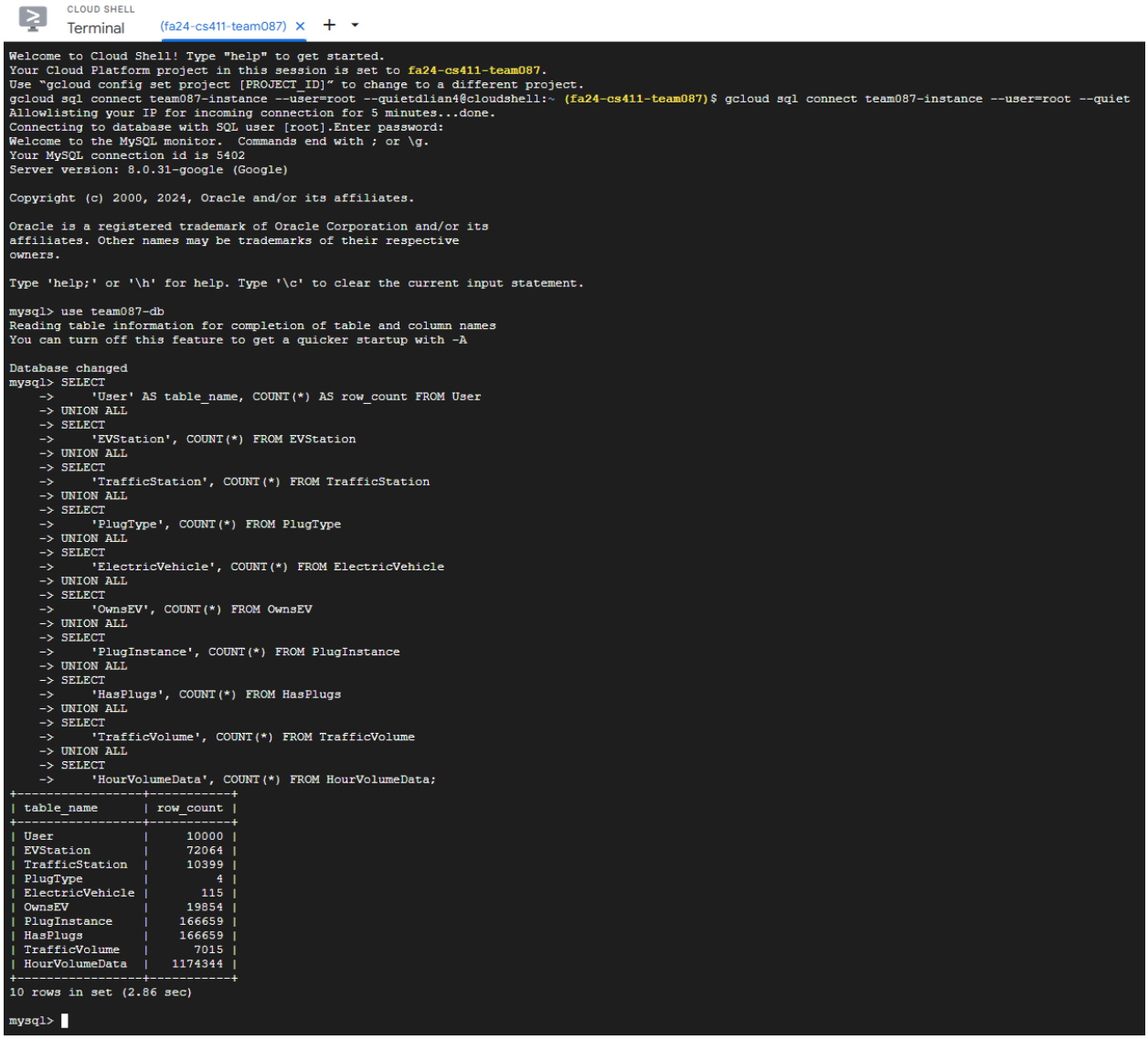
DDL Statements

```
CREATE TABLE User(
    user_id INT AUTO_INCREMENT,
    username VARCHAR(100) NOT NULL,
    email VARCHAR(100) NOT NULL,
    password CHAR(32) NOT NULL,
    ssn VARCHAR(11),
    address VARCHAR(100),
    state VARCHAR(2),
    city VARCHAR(100),
    zip VARCHAR(10),
    first_name VARCHAR(100),
    last_name VARCHAR(100),
    middle_initial VARCHAR(1),
    creation_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (user_id)
);

CREATE TABLE EVStation(
    station_id INT AUTO_INCREMENT,
    name VARCHAR(100),
    latitude DECIMAL(8, 6),
    longitude DECIMAL(9, 6),
    state VARCHAR(2),
    zip VARCHAR(10),
    city VARCHAR(100),
    address VARCHAR(100),
    phone VARCHAR(100),
    PRIMARY KEY (station_id)
);

CREATE TABLE TrafficStation(
    state_code VARCHAR(2),
    station_id VARCHAR(6),
    latitude DECIMAL(8, 6),
    longitude DECIMAL(9, 6),
```

```
    PRIMARY KEY (state_code, station_id)
);

CREATE TABLE PlugType(
    type_id INT AUTO_INCREMENT,
    type_name VARCHAR(100),
    PRIMARY KEY (type_id)
);

CREATE TABLE ElectricVehicle(
    ev_id INT AUTO_INCREMENT,
    make VARCHAR(100),
    model VARCHAR(100),
    plug_type INT,
    range_km REAL,
    battery_capacity REAL,
    PRIMARY KEY (ev_id),
    FOREIGN KEY (plug_type) REFERENCES PlugType(type_id) ON DELETE SET
NULL
);

CREATE TABLE OwnsEV(
    user_id INT,
    ev_id INT,
    PRIMARY KEY (user_id, ev_id),
    FOREIGN KEY (user_id) REFERENCES User(user_id) ON DELETE CASCADE,
    FOREIGN KEY (ev_id) REFERENCES ElectricVehicle(ev_id) ON DELETE
CASCADE
);

CREATE TABLE PlugInstance(
    instance_id INT AUTO_INCREMENT,
    type_id INT,
    power_output REAL,
    in_use BOOLEAN,
    base_price DECIMAL(4, 2),
    usage_price DECIMAL(4, 2),
    PRIMARY KEY (instance_id),
    FOREIGN KEY (type_id) REFERENCES PlugType(type_id) ON DELETE SET NULL
);
```

```sql
CREATE TABLE HasPlugs(
    station_id INT,
    instance_id INT,
    PRIMARY KEY (station_id, instance_id),
    FOREIGN KEY (station_id) REFERENCES EVStation(station_id) ON DELETE
CASCADE,
    FOREIGN KEY (instance_id) REFERENCES PlugInstance(instance_id) ON
DELETE CASCADE
);

CREATE TABLE TrafficVolume(
    volume_id INT AUTO_INCREMENT,
    state_code VARCHAR(2),
    station_id VARCHAR(6),
    PRIMARY KEY (volume_id),
    FOREIGN KEY (state_code, station_id) REFERENCES
TrafficStation(state_code, station_id)
);

-- This table must be joined with TrafficVolume on volume_id
CREATE TABLE HourVolumeData(
    volume_id INT,
    month_record INT NOT NULL CHECK(month_record BETWEEN 1 AND 12),
    year_record INT NOT NULL,
    day_of_week INT NOT NULL CHECK(day_of_week BETWEEN 1 AND 7),
    hour INT NOT NULL CHECK(hour BETWEEN 0 AND 23),
    volume REAL,
    FOREIGN KEY (volume_id) REFERENCES TrafficVolume(volume_id)
);
```

Screenshot of Connection



Rows in Each Table

RESULTS

table_name	row_count
User	10000
EVStation	72064
TrafficStation	10399
PlugType	4
ElectricVehicle	115
OwnsEV	19854
PlugInstance	166659
HasPlugs	166659
TrafficVolume	7015
HourVolumeData	1174344

Part 1: Queries

NOTE:

We have included a variety of queries for this section in case some of them do not meet the variety or complexity requirements. Don't feel the need to go over all of them if the first 4 are good enough to meet the base query requirements <3.

Q0: Compute Haversine Distance Between 2 Latitudes And Longitudes (Utility Query)

```
-- User's current latitude and longitude
SET @my_latitude = 34.040539;
SET @my_longitude = -118.271387;
-- Distance pruning threshold, can be used in indexing to efficiently
exclude other records that aren't within a valid range.
SET @distance_threshold = 40;

SELECT
      -- This will be the set of distances between each Station and our
current position within the @distance_threshold
      Distances.distance_km
FROM (
      SELECT
      latitude,
      longitude,
      -- Haversine distance calculation
      (
            6371 * 2 * ASIN(
                  SQRT(
                  POWER(SIN(RADIANS(latitude - (@my_latitude)) / 2), 2) +
                  COS(RADIANS(@my_latitude)) *
                  COS(RADIANS(latitude)) *
                  POWER(SIN(RADIANS(longitude - (@my_longitude)) / 2), 2)
                  )
            )
      ) AS distance_km
      FROM <INSERT TABLE HERE> -- EVStation || TrafficStation
      WHERE
      -- latitude BETWEEN ensures we only include EVStations or
TrafficStations that can possibly be within @distance_threshold of our
position
```

```
      -- @distance_threshold/111.0 is the number of degrees of latitude 40
km is
      -- We can use this to exclude stations outside this range which can
be especially fast if we use a B+ Tree for indexing.
      latitude
      BETWEEN @my_latitude - (@distance_threshold/111.0)
            AND @my_latitude + (@distance_threshold/111.0)
      AND
      -- Longitude distance actually depends on current latitude, so we
have to calculate this a bit differently, but it is still the same overall
      longitude
      BETWEEN @my_longitude - (@distance_threshold/(111.0 *
COS(RADIANS(@my_latitude))))
            AND @my_longitude + (@distance_threshold/(111.0 *
COS(RADIANS(@my_latitude))))
) AS Distances
```

Throughout our project, we will utilize this subquery as a common building block. Many of our application functions revolving around an EV trip planner require us to compute distances between points on the planet. While we could accomplish this with an API call, it is much more interesting (and easier to satisfy project requirements) to try this with nothing but SQL. To do this, we implemented the Haversine distance equation which computes the point-to-point distance between any latitude and longitude pair in km accounting for the curvature of the Earth.

While this query is a common building block in many of our other queries, we believe each query is distinct enough that they can be counted separately even though many share this same calculation. To justify this, we have included justifications about how each query is different in function and in its intent for usage.

Additionally, we use parameters in our queries that will need to be set at runtime depending on the application state.Things like current location can only be determined at runtime, so location and EV arguments may be added at runtime and can vary.

Q1: Get All Compatible Stations With My Car Within Some Range To A Coordinate

```
SET @my_latitude = 34.040539;
SET @my_longitude = -118.271387;
SET @distance_threshold = 40; -- (in km)
SET @ev_id = 34; -- (CyberTruck)

SELECT
      Distances.ev_station_id,
      Distances.distance_km,
      PlugInstance.type_id,
      PlugInstance.power_output,
      PlugInstance.in_use,
      PlugInstance.base_price,
      PlugInstance.usage_price,
      EVStation.address,
      EVStation.state,
      EVStation.zip,
      ROUND((SELECT battery_capacity FROM ElectricVehicle WHERE
ev_id=@ev_id) / PlugInstance.power_output, 2) AS time_to_charge_hr,
      -- Total charge price is base price + price to charge to full
      -- Full charge price is usage_price ($/kwh)
      -- usage_price = #/kwh, battery_capacity = kwh,
      ROUND(base_price +
      ROUND((SELECT battery_capacity FROM ElectricVehicle WHERE
ev_id=@ev_id), 2) * usage_price
      ,2) AS price_to_charge
FROM (
      SELECT
      latitude,
      longitude,
      station_id AS ev_station_id,
      -- Haversine formula implementation
      -- R = Earth's radius in kilometers
      ROUND(
            (
            6371 * 2 * ASIN(
                  SQRT(
                  POWER(SIN(RADIANS(latitude - (@my_latitude)) / 2), 2) +
                  COS(RADIANS(@my_latitude)) *
                  COS(RADIANS(latitude)) *
```

```
                  POWER(SIN(RADIANS(longitude - (@my_longitude)) / 2), 2)
                  )
            )
      ), 3) AS distance_km
      FROM EVStation
      WHERE
      -- Pre-filtering using bounding box
      -- 1 degree of latitude \approx 111 km at the equator
      -- Variance between latitude is mostly constant, so we can just
      -- threshold based on the difference.
      -- 1 degree of longitude varies with latitude
      latitude
      BETWEEN @my_latitude - (@distance_threshold/111.0)
            AND @my_latitude + (@distance_threshold/111.0)
      AND
      longitude
      BETWEEN @my_longitude - (@distance_threshold/(111.0 *
COS(RADIANS(@my_latitude))))
            AND @my_longitude + (@distance_threshold/(111.0 *
COS(RADIANS(@my_latitude))))
) AS Distances
JOIN
EVStation
      ON Distances.ev_station_id = EVStation.station_id
JOIN HasPlugs
      ON EVStation.station_id = HasPlugs.station_id
JOIN PlugInstance
      ON PlugInstance.instance_id = HasPlugs.instance_id
WHERE
      distance_km <= @distance_threshold
      AND PlugInstance.type_id = (SELECT plug_type FROM ElectricVehicle
WHERE ev_id = @ev_id)
ORDER BY
      Distances.distance_km ASC
LIMIT 15;
```

```
-- Set parameters
SET @my_latitude = 34.040539;
SET @my_longitude = -118.271387;
SET @distance_threshold = 40; -- in kilometers
SET @ev_id = 34; -- ElectricVehicle.ev_id for CyberTruck
```

```sql
-- Get the plug type of the user's EV
SET @plug_type = (SELECT plug_type FROM ElectricVehicle WHERE ev_id = @ev_id);

-- Main query
SELECT
    FinalResults.ev_station_id,
    FinalResults.distance_km,
    FinalResults.type_id,
    FinalResults.avg_power_output,
    FinalResults.available_plugs,
    FinalResults.avg_base_price,
    FinalResults.avg_usage_price,
    EVStation.address,
    EVStation.state,
    EVStation.zip,
    ROUND(EV.battery_capacity / FinalResults.avg_power_output, 2) AS
avg_time_to_charge_hr,
    ROUND(FinalResults.avg_base_price + EV.battery_capacity *
FinalResults.avg_usage_price, 2) AS avg_price_to_charge
FROM
(
    -- Subquery to calculate distances and filter by plug type
    SELECT
        Distances.ev_station_id,
        Distances.distance_km,
        PI.type_id,
        AVG(PI.power_output) AS avg_power_output,
        COUNT(PI.instance_id) AS available_plugs,
        AVG(PI.base_price) AS avg_base_price,
        AVG(PI.usage_price) AS avg_usage_price
    FROM
    (
        -- Subquery to calculate distances
        SELECT
            EVS.station_id AS ev_station_id,
            ROUND(
                6371 * 2 * ASIN(
                    SQRT(
                        POWER(SIN(RADIANS(EVS.latitude - @my_latitude) / 2), 2) +
```

```
                    COS(RADIANS(@my_latitude)) * COS(RADIANS(EVS.latitude)) *
                    POWER(SIN(RADIANS(EVS.longitude - @my_longitude) / 2), 2)
                )
            ), 3
        ) AS distance_km
    FROM EVStation EVS
    WHERE
        EVS.latitude BETWEEN @my_latitude - (@distance_threshold / 111.0)
            AND @my_latitude + (@distance_threshold / 111.0)
        AND EVS.longitude BETWEEN @my_longitude - (@distance_threshold /
(111.0 * COS(RADIANS(@my_latitude))))
            AND @my_longitude + (@distance_threshold / (111.0 *
COS(RADIANS(@my_latitude))))
    ) AS Distances
    JOIN HasPlugs HP ON Distances.ev_station_id = HP.station_id
    JOIN PlugInstance PI ON HP.instance_id = PI.instance_id
    WHERE
        Distances.distance_km <= @distance_threshold
        AND PI.type_id = @plug_type
    GROUP BY
        Distances.ev_station_id,
        Distances.distance_km,
        PI.type_id
) AS FinalResults
JOIN EVStation ON FinalResults.ev_station_id = EVStation.station_id
JOIN ElectricVehicle EV ON EV.ev_id = @ev_id
ORDER BY
    FinalResults.distance_km ASC
LIMIT 15;
```

Arguments

- @my_latitude: Current latitude
- @my_longitude: Current longitude
- @distance_threshold: Only get information for stations below this number.
- @ev_id: The `ElectricVehicle.ev_id` that we want to find plugs for

Functionality

This query pair determines the closest EV PlugInstances that are compatible with a given ElectricVehicle given the user's location. This location will be determined at runtime outside the scope of the database. We intend to use this query as a general search function so that user's can immediately identify the best plugs that are near them, as well as the expected cost that they will pay for charging, and the time it takes to charge based on their vehicle.

We have 2 different variants of this query, as we may want to know specifics about the exact prices for each plug instance we could use within some range, while other times we may only want a summary of the compatible plugs in our area.

Complexity Requirements

The query involves multiple joins between tables `EVStation`, `HasPlugs`, `PlugInstance`, `ElectricVehicle`. It also uses subqueries to retrieve the `battery_capacity` from the `ElectricVehicle` table. It has some calculations of charging time and cost and implements Haversine formula. The second query also uses a `GROUP BY` to group EVStations and their distances to get the average cost of all the available plugs.

Justification for Usage

This query is unique because it combines the spatial filtering with compatibility checks specific to the user's vehicle and is essential because it allows users to find suitable charging stations efficiently. Unlike other queries, this one focuses on personalized results based on both location and vehicle specifications.

Query Results

RESULTS

ev_station_id	distance_km	type_id	power_output	in_use	base_price	usage_price	address	state	zip	time_to_charge_hr	price_to_charge
63694	1.82	1	50	0	1.16	0.47	515 south Flower st	CA	90071	4	95.28
63694	1.82	1	50	0	1.48	0.21	515 south Flower st	CA	90071	4	43.53
5776	2.125	1	50	0	1.62	0.36	1230 W. 3rd St	CA	90017	4	73.71
1895	2.661	1	50	0	0.00	0.45	111 S Grand Ave	CA	90012	4	90.11
1895	2.661	1	50	0	0.00	0.45	111 S Grand Ave	CA	90012	4	90.11
1895	2.661	1	50	0	0.00	0.45	111 S Grand Ave	CA	90012	4	90.11
30555	2.764	1	50	0	1.40	0.38	201 N Figueroa St	CA	90012	4	77.5
30555	2.764	1	50	0	1.98	0.44	201 N Figueroa St	CA	90012	4	90.09
30555	2.764	1	50	0	0.91	0.30	201 N Figueroa St	CA	90012	4	60.98
1898	2.893	1	50	0	0.00	0.30	313 N Figueroa St	CA	90012	4	60.08
1898	2.893	1	50	0	0.00	0.30	313 N Figueroa St	CA	90012	4	60.08
1898	2.893	1	50	0	0.00	0.30	313 N Figueroa St	CA	90012	4	60.08
1898	2.893	1	50	0	0.00	0.30	313 N Figueroa St	CA	90012	4	60.08
1898	2.893	1	50	0	0.00	0.30	313 N Figueroa St	CA	90012	4	60.08
1898	2.893	1	50	0	0.00	0.30	313 N Figueroa St	CA	90012	4	60.08

Rows per page: 20 1 – 15 of 15

These results are from the first query variant. We return multiple rows per ev_station_id as these represent the individual plugs with their individual different pricing schemes.

RESULTS

ev_station_id	distance_km	type_id	avg_power_output	available_plugs	avg_base_price	avg_usage_price	address	state	zip	avg_time_to_charge_hr	avg_price_to_charge
63694	1.82	1	50	2	1.320000	0.340000	515 south Flower st	CA	90071	4	69.4
5776	2.125	1	50	1	1.620000	0.360000	1230 W. 3rd St	CA	90017	4	73.71
1895	2.661	1	50	3	0.000000	0.450000	111 S Grand Ave	CA	90012	4	90.11
30555	2.764	1	50	3	1.430000	0.373333	201 N Figueroa St	CA	90012	4	76.19
1898	2.893	1	50	6	0.000000	0.300000	313 N Figueroa St	CA	90012	4	60.08
12456	2.947	1	50	2	0.000000	0.000000	111 N Hope St	CA	90012	4	0
42699	3.083	1	50	1	1.540000	0.320000	101 Judge John Aiso St	CA	90012	4	65.62
42700	3.351	1	50	2	0.890000	0.415000	525 S Hewitt St	CA	90013	4	83.99
34834	3.503	1	50	3	0.963333	0.390000	1820 Industrial St	CA	90021	4	79.06
63488	3.59	1	50	1	1.650000	0.450000	1115 Sunset Blvd	CA	90012	4	91.76
38979	3.609	1	50	4	0.912500	0.182500	3410EV W 3RD ST	CA	90020	4	37.46
53908	3.671	1	50	6	1.221667	0.325000	3461 W 3rd St	CA	90020	4	66.3
13255	3.682	1	50	1	1.040000	0.210000	661 Imperial St	CA	90021	4	43.09
18970	4.043	1	50	3	1.290000	0.383333	850 N Broadway	CA	90012	4	78.05
59207	4.227	1	50	1	1.230000	0.120000	1810 E 25th St	CA	90058	4	25.26

Rows per page: 20 1 – 15 of 15

These results are from the second variant. Note how we instead return an average of prices for all plugs as well as the plug count for each station.

Q2: Get Congestion Score For Location@My_latitude: User's Latitude

```
SET @my_latitude = 34.040539;
SET @my_longitude = -118.271387;
SET @distance_threshold = 40; -- (in km)
SET @hour_range = 2;
SET @current_hour = 12;

SELECT DISTINCT
    Distances.station_id,
    Distances.state_code,
    Distances.distance_km,
  ROUND(AVG(HourVolumeData.volume), 1) AS avg_volume,
  ROUND(
  1/(1+EXP((distance_km-@distance_threshold/2.0)/10)) *
ROUND(AVG(HourVolumeData.volume), 1)
  ,2) AS CongestionScore
FROM (
    SELECT
        latitude,
        longitude,
        station_id,
        state_code,
        (
            6371 * 2 * ASIN(
                SQRT(
                    POWER(SIN(RADIANS(latitude - (@my_latitude)) / 2), 2) +
                    COS(RADIANS(@my_latitude)) *
                    COS(RADIANS(latitude)) *
                    POWER(SIN(RADIANS(longitude - (@my_longitude)) / 2), 2)
                )
            )
        ) AS distance_km
    FROM TrafficStation
    WHERE
        latitude
        BETWEEN @my_latitude - (@distance_threshold/111.0)
            AND @my_latitude + (@distance_threshold/111.0)
        AND
        longitude
        BETWEEN (@my_longitude) - (@distance_threshold/(111.0 *
```

```
COS(RADIANS(@my_latitude))))
            AND (@my_longitude) + (@distance_threshold/(111.0 *
COS(RADIANS(@my_latitude))))
) AS Distances
JOIN TrafficStation
    ON Distances.station_id = TrafficStation.station_id
    AND Distances.state_code = TrafficStation.state_code
JOIN TrafficVolume
  ON Distances.station_id = TrafficVolume.station_id
  AND Distances.state_code = TrafficVolume.state_code
JOIN HourVolumeData
  ON TrafficVolume.volume_id = HourVolumeData.volume_id
WHERE
    distance_km <= @distance_threshold
  AND HourVolumeData.hour BETWEEN @current_hour-@hour_range AND
@current_hour+@hour_range
GROUP BY
  TrafficStation.station_id,
  TrafficVolume.state_code,
  TrafficVolume.station_id
ORDER BY
    Distances.distance_km ASC
LIMIT 15;
```

Arguments

- @my_latitude: User's latitude
- @my_longitude: User's longitude
- @distance_threshold: Max distance to return results from (used in prefiltering)
- @hour_range: How many hours before and after hour 12 to average over
- @current_hour: The current hour of the day as an integer.

Functionality

This is a query we can use to compute a "congestion" score based on a traffic data portion of our database. This congestion score is calculated based on the average traffic volume reported by each TrafficStation over a given range of hours. The congestion score for a given station is higher if we are close to that station and it is lower if we are far from it. We can use the computed congestions for all TrafficStations in an area to get estimates of EV charging wait times at a higher level in the application, or to display a heat-map of traffic so users can determine areas they'd like to avoid when looking for places to charge.

$$\text{Congestion Score} = \frac{1}{1+e^{X}} \cdot V$$

$$X = \frac{d - \frac{d_t}{2}}{10}$$

- V: Average Volume for a TrafficStation
- d: Distance to each TrafficStation
- d_t: Distance threshold

Note on calculations

We use $d - \frac{d_t}{2}$ to scale how large the magnitude of X is where large positive values of X indicate that d is large, meaning the value of our scaling factor will be small reducing the amount V contributes to the congestion score. Conversely, if $d = 0$, then X will have the largest possible *negative* magnitude minimizing denominator and allowing V to have a higher weight.

Our congestion score calculation is fairly arbitrary and not based on any rigorous analysis of traffic patterns and congestion, but was simple and reasonable enough for our purposes.

Complexity Requirements

Multiple Joins between TrafficVolume, HourVolumeData, TrafficStation, and the derived Distances Table

Aggregation via average over a given hour range. This lets us get the average volume depending on time instead of average traffic volume for the entire day.

Justification for Usage

This query is significantly different from other listed queries, as it is one of the primary queries we have planned that will use the Traffic Volume portion of our dataset. We believe that the computation of a congestion score weighted by distance makes this query particularly unique and complex. The averaging of traffic volumes based on time of day is a unique aggregation not present in any other queries.

Additionally, this query performs unique calculations to estimate the congestion in any location which can be used at a higher level in our application to estimate the wait times for any individual EV station (although this could be done as a separate query, it makes more sense to perform these calculations at a higher level to reduce the load on the database).

Query Results Image

RESULTS

station_id	state_code	distance_km	avg_volume	CongestionScore
073600	06	11.857765143373138	2453.6	1700.37
077210	06	14.60989305523597	6786.3	4286.11
077590	06	17.770002448479538	3631.2	2017.2
070390	06	17.87350148316806	6925.8	3829.71
075510	06	18.44899763642157	7864.5	4236.59
071260	06	20.789110758777102	7111.4	3415.48
074210	06	21.428247555547415	1040.3	483.07
072110	06	25.344284997900466	8878.7	3280.54
079020	06	26.365577312270428	1265	437.72
074650	06	28.235123699512712	9059	2763.16
079140	06	29.517291509590603	115.4	32.14
122260	06	30.95128930205072	7042.8	1765.3
122190	06	31.88268243554411	4887.7	1141.61
121960	06	38.49614403145237	5756.3	782.39
126590	06	38.85580689858249	1300.5	171.34

Rows per page: 20 1 – 15 of 15

Q3: Find Users Who Own Multiple ElectricVehicles

```
SELECT
  U.first_name,
  U.last_name,
  U.email,
  GROUP_CONCAT(EV.make,
      ' ',
      EV.model) AS vehicles_owned,
  COUNT(*) AS num_vehicles
FROM
  User U
JOIN
  OwnsEV OE
ON
  U.user_id = OE.user_id
JOIN
  ElectricVehicle EV
ON
  OE.ev_id = EV.ev_id
GROUP BY
  U.user_id
HAVING
  COUNT(*) > 1
ORDER BY
  num_vehicles DESC
LIMIT
  15;
```

Arguments

None

Functionality

This query lets us identify all users that have multiple ElectricVehicles tied to their account. This is particularly useful for family-based settings or EV enthusiasts who may change the car they drive often. We return a list of all EVs that someone owns which can easily be displayed in the front end. This query can easily be modified to filter for a variety of features, like EVs owned, user location, etc.

Complexity Requirements

Multiple Joins through User, OwnsEV and ElectricVehicle
Aggregation on the make and model of each EV.

Justification for Usage

This is a very unique query in that it primarily deals with the Users table. The aggregation over vehicle make and model is something unique that we haven't used elsewhere and allows for immediate display straight onto the front end, or it allows for easy analytics for business partners interested in how EVs are distributed through our Users.

Query Results Image

RESULTS

first_name	last_name	email	vehicles_owned	num_vehicles
Stephanie	Pierce	StePierce95@yahoo.com	Volkswagen ID.3 Pro S,Tesla Model S Long ...	3
Christopher	Hancock	ChrHancock98@hotmail.com	Lightyear One,Renault Zoe ZE40 R110,Byton...	3
Valerie	Smith	ValSmith81@hotmail.com	Volkswagen ID.4,Kia e-Niro 64 kWh,Tesla Cy...	3
Kathy	Church	KatChurch94@gmail.com	MG ZS EV,Skoda Enyaq iV 60,Nissan Ariya 6...	3
Rebecca	Washington	RebWashington18@hotmail.com	Volkswagen e-Up!,Audi e-tron GT,Skoda Eny...	3
Peggy	Mayo	PegMayo51@gmail.com	DS 3 Crossback E-Tense,Tesla Model Y Lon...	3
Veronica	Kelley	VerKelley94@gmail.com	Skoda Enyaq iV 50,Volkswagen ID.3 Pro,For...	3
Matthew	Washington	MatWashington52@yahoo.com	Ford Mustang Mach-E ER RWD,Smart EQ for...	3
Bailey	Spencer	BaiSpencer94@hotmail.com	Volkswagen e-Golf,Audi e-tron GT,Ford Must...	3
Stefanie	Chavez	SteChavez99@gmail.com	BMW i4,Honda e Advance,Renault Zoe ZE40...	3
David	Reed	DavReed45@yahoo.com	Tesla Model 3 Standard Range Plus,Tesla C...	3
Samuel	Wong	SamWong89@gmail.com	Mercedes EQA,Skoda Enyaq iV 80X,Nissan ...	3
Melanie	Short	MelShort24@yahoo.com	Nissan Leaf,Hyundai Kona Electric 64 kWh,...	3
Jeremy	Stanley	JerStanley79@hotmail.com	Audi e-tron GT,Volvo XC40 P8 AWD Recharg...	3
Paul	Burke	PauBurke71@yahoo.com	Volkswagen ID.4,Jaguar I-Pace,Nissan Ariya...	3

Rows per page: 20 1 – 15 of 15

Q4: List EVStations With Highest Number Of Available Plugs

```
SET @my_latitude = 34.040539;
SET @my_longitude = -118.271387;
SET @distance_threshold = 40;

SELECT
  EVS.station_id,
  EVS.name AS station_name,
  EVS.address,
  EVS.city,
  state,
  COUNT(PI.instance_id) AS available_plugs,
  Distances.distance_km
FROM (
  SELECT
    latitude,
    longitude,
    EVStation.station_id AS ev_station_id,
    ( 6371 * 2 * ASIN( SQRT( POWER(SIN(RADIANS(latitude - (@my_latitude)) /
2), 2) + COS(RADIANS(@my_latitude)) * COS(RADIANS(latitude)) *
POWER(SIN(RADIANS(longitude - (@my_longitude)) / 2), 2) ) ) ) AS
distance_km
  FROM
    EVStation -- EVStation || TrafficStation
  WHERE
    latitude BETWEEN @my_latitude - (@distance_threshold/111.0)
    AND @my_latitude + (@distance_threshold/111.0)
    AND longitude BETWEEN @my_longitude - (@distance_threshold/(111.0 *
COS(RADIANS(@my_latitude))))
    AND @my_longitude + (@distance_threshold/(111.0 *
COS(RADIANS(@my_latitude)))) ) AS Distances
JOIN EVStation EVS
      ON Distances.ev_station_id = EVS.station_id
JOIN HasPlugs HP
      ON EVS.station_id = HP.station_id
JOIN PlugInstance PI
      ON HP.instance_id = PI.instance_id
WHERE
  PI.in_use = FALSE
  AND Distances.distance_km <= @distance_threshold
GROUP BY
  EVS.station_id,
```

```
  EVS.name,
  EVS.address,
  EVS.city,
  EVS.state
ORDER BY
  Distances.distance_km ASC,
  available_plugs DESC
LIMIT
  15;
```

Arguments

- @my_latitude: User’s latitude
- @my_longitude: User's longitude
- @distance_threshold: Max distance to return results from (used in prefiltering)

Functionality

This query lists nearby EV charging stations sorted by the highest number of available (not in_use) plugs. It calculates the distance to each station, filters based on availability and proximity, and provides station details. This helps users find stations where they are most likely to find an available charger.

Complexity Requirements

There are multiple joins on `EVStation`, `HasPlugs`, and `PlugInstance`. It uses `COUNT()` to count available plugs per station which is aggregation and groups results by station to aggregate plug counts. It also performs spatial calculations using the Haversine formula and filters out plugs that are currently in use.

Justification for Usage

While other queries may list stations or plugs, this one specifically focuses on availability, combining it with proximity to provide practical recommendations when compared to results from Q1 for example. Instead of providing compatible plugs and pricing information like Q1, we instead provide much higher level information about the *number* of available plugs each station has which is more useful for users to find the largest EV stations in general which can make trip planning easier. Large EV stations often have attractions or other points of interest around them, so some may be inclined to search for larger stations or smaller stations if they wish to avoid a crowd of people.

Query Results Image

RESULTS

station_id	station_name	address	city	state	available_plugs	distance_km
1	Los Angeles Conventi...	1201 S Figuero...	Los Ang...	CA	7	0
15405	ESSEX AVANT STATIO...	1362-1458 S Fi...	Los Ang...	CA	2	0.287402091618131
15396	ESSEX AVANT STATIO...	1420 S Figuero...	Los Ang...	CA	2	0.2923426377772659
27293	CLF SPACE 138	1300 S Figuero...	Los Ang...	CA	1	0.34682469995103576
68037	Moxy Hotel	1248 S Figuero...	Los Ang...	CA	16	0.3860423646348323
22911	CIRCA COMMERICA S...	1200 S Figuero...	Los Ang...	CA	1	0.41850741244065687
14293	CIRCA COMMERICA S...	1200 S Figuero...	Los Ang...	CA	2	0.42381567857470687
5434	LADWP - Georgia Street	1728 7/8 Georg...	Los Ang...	CA	1	0.42535739376954196
22908	CIRCA COMMERICA S...	1200 S Figuero...	Los Ang...	CA	2	0.4331619696179834
22910	CIRCA COMMERICA S...	1200 S Figuero...	Los Ang...	CA	2	0.43638062307045555
22909	CIRCA COMMERICA S...	1200 S Figuero...	Los Ang...	CA	1	0.4402806677294739
22912	CIRCA COMMERICA S...	1200 S Figuero...	Los Ang...	CA	1	0.44745628752926636
33343	HOSPITAL CHMC PAR...	1347 S Hope St	Los Ang...	CA	2	0.49709248925450916
33344	HOSPITAL CHMC PAR...	1347 S Hope St	Los Ang...	CA	2	0.4989294429772248
24858	L.A. LIVE EV CH UNIT 4	1005 Chick Hea...	Los Ang...	CA	2	0.543593649751629

Rows per page: 20 1 – 15 of 15

Q5: Query To Get Statistics For All Pluginstances In A Range

```
SET @my_latitude = 34.040539;
SET @my_longitude = -118.271387;
SET @distance_threshold = 40;

SELECT
  PI.type_id,
  COUNT(PI.type_id) AS type_count,
  ROUND(AVG(base_price), 2) AS avg_base_price,
  ROUND(AVG(usage_price), 2) AS avg_useage_price
FROM
  (
    SELECT
      latitude,
      longitude,
      (
        6371 * 2 * ASIN(
          SQRT(
            POWER(SIN(RADIANS(latitude - (@my_latitude)) / 2), 2) +
            COS(RADIANS(@my_latitude)) *
            COS(RADIANS(latitude)) *
            POWER(SIN(RADIANS(longitude - (@my_longitude)) / 2), 2)
          )
        )
      ) AS distance_km,
      EVStation.station_id AS ev_station_id
    FROM EVStation -- EVStation || TrafficStation
    WHERE
      latitude
      BETWEEN @my_latitude - (@distance_threshold/111.0)
        AND @my_latitude + (@distance_threshold/111.0)
      AND
      longitude
      BETWEEN @my_longitude - (@distance_threshold/(111.0 *
COS(RADIANS(@my_latitude))))
        AND @my_longitude + (@distance_threshold/(111.0 *
COS(RADIANS(@my_latitude))))
  ) AS Distances
JOIN HasPlugs
  ON Distances.ev_station_id = HasPlugs.station_id
JOIN PlugInstance PI
  ON HasPlugs.instance_id = PI.instance_id
```

```
GROUP BY
  PI.type_id
LIMIT 15;
```

Arguments

- @my_latitude: User's latitude
- @my_longitude: User's longitude
- @distance_threshold: Max distance to return results from (used in prefiltering)

Functionality

This query identifies all the PlugInstances within a range based on their type as well as collecting statistics about the average base and usage price for using these plugs. This query is a useful addition to our application's functionality since not everyone may be particularly savvy with the common plug types or costs in their area. This is especially true for those who are renting or borrowing an EV, and this query will allow them to easily get an overview of the kinds of plugs there are in their area as well as the prices they can expect for charging.

Complexity Requirements

Multiple Joins through derived Distances table, HasPlugs, and PlugInstance.
Aggregation to calculate counts, base prices, and usage prices for all PlugInstances

Justification for Usage

This query is specifically tailored to users who may be curious about the distribution of EV charging type availability. Although it again uses the backbone distance subquery present in many of our operations, it is still different in its function and provides results that we believe to be adequate for a separate query. Comparing again to Q1, this provides some of the highest level information which allows users to gauge the average price and availability of plug types for whatever EV they may be considering using.

Query Results Image

This query has less than 15 results.

RESULTS

type_id	type_count	avg_base_price	avg_useage_price
3	10428	1.05	0.29
2	582	1.10	0.28
1	1045	1.16	0.29

Q6: Query to get the “Best” EV for a trip from one city to another

```
-- We have to do this because the formatting is stupid and dumb and stupid
SET @city1 = 'Los Angeles' COLLATE utf8mb4_0900_ai_ci;
SET @city2 = 'San Francisco' COLLATE utf8mb4_0900_ai_ci;

SET @city1_latitude = 34.0549;
SET @city1_longitude = -118.2426;

SET @city2_latitude = 37.7749;
SET @city2_longitude = -122.4194;

SET @distance_threshold = 40;

SELECT
      EVStationCounts.ev_station_city,
      EVStationCounts.type_id,
      EVStationCounts.type_count,
      ElectricVehicle.make,
      ElectricVehicle.model,
      ElectricVehicle.range_km,
      ROUND(ElectricVehicle.battery_capacity /
EVStationCounts.avg_power_output, 2) AS time_to_charge_hr,
      ROUND(
      (EVStationCounts.avg_base_price +
      ElectricVehicle.battery_capacity * EVStationCounts.avg_useage_price)
/
      ElectricVehicle.range_km * 100
      , 2) AS expected_charge_cost_per_hundred_km
FROM
(
      SELECT * FROM
      (
      SELECT
            Distances1.ev_station_city,
            PI.type_id,
            COUNT(PI.type_id) AS type_count,
            ROUND(AVG(PI.usage_price), 2) AS avg_useage_price,
            ROUND(AVG(PI.base_price), 2) AS avg_base_price,
            ROUND(AVG(PI.power_output), 2) AS avg_power_output
      FROM (
            SELECT
```

```
                latitude,
                longitude,
                (
                6371 * 2 * ASIN(
                      SQRT(
                            POWER(SIN(RADIANS(latitude -
(@city1_latitude)) / 2), 2) +
                            COS(RADIANS(@city1_latitude)) *
                            COS(RADIANS(latitude)) *
                            POWER(SIN(RADIANS(longitude -
(@city1_longitude)) / 2), 2)
                            )
                )
                ) AS distance_km,
                EVStation.station_id AS ev_station_id,
                EVStation.city AS ev_station_city
          FROM EVStation -- EVStation || TrafficStation
          WHERE
                latitude
                BETWEEN @city1_latitude - (@distance_threshold/111.0)
                AND @city1_latitude + (@distance_threshold/111.0)
                AND
                longitude
                BETWEEN @city1_longitude - (@distance_threshold/(111.0 *
COS(RADIANS(@city1_latitude))))
                AND @city1_longitude + (@distance_threshold/(111.0 *
COS(RADIANS(@city1_latitude))))
                AND EVStation.city = @city1
     ) AS Distances1
     JOIN HasPlugs HP
          ON HP.station_id = Distances1.ev_station_id
     JOIN PlugInstance PI
          ON HP.instance_id = PI.instance_id
     GROUP BY
          PI.type_id,
          Distances1.ev_station_city
     ) AS City1Info
     UNION
     (
     SELECT
          Distances2.ev_station_city,
          PI.type_id,
          COUNT(PI.type_id) AS type_count,
```

```
            ROUND(AVG(PI.usage_price), 2) AS avg_useage_price,
            ROUND(AVG(PI.base_price), 2) AS avg_base_price,
            ROUND(AVG(PI.power_output), 2) AS avg_power_output
      FROM (
            SELECT
                  latitude,
                  longitude,
                  (
                  6371 * 2 * ASIN(
                        SQRT(
                              POWER(SIN(RADIANS(latitude -
(@city2_latitude)) / 2), 2) +
                              COS(RADIANS(@city2_latitude)) *
                              COS(RADIANS(latitude)) *
                              POWER(SIN(RADIANS(longitude -
(@city2_longitude)) / 2), 2)
                        )
                  )
                  ) AS distance_km,
                  EVStation.station_id AS ev_station_id,
                  EVStation.city AS ev_station_city
            FROM EVStation -- EVStation || TrafficStation
            WHERE
                  latitude
                  BETWEEN @city2_latitude - (@distance_threshold/111.0)
                  AND @city2_latitude + (@distance_threshold/111.0)
                  AND
                  longitude
                  BETWEEN @city2_longitude - (@distance_threshold/(111.0 *
COS(RADIANS(@city2_latitude))))
                  AND @city2_longitude + (@distance_threshold/(111.0 *
COS(RADIANS(@city2_latitude))))
                  AND EVStation.city = @city2
      ) AS Distances2
      JOIN HasPlugs HP
            ON HP.station_id = Distances2.ev_station_id
      JOIN PlugInstance PI
            ON HP.instance_id = PI.instance_id
      GROUP BY
            PI.type_id,
            Distances2.ev_station_city
      )
) AS EVStationCounts
```

```
JOIN ElectricVehicle
      ON EVStationCounts.type_id = ElectricVehicle.plug_type


ORDER BY
      expected_charge_cost_per_hundred_km ASC,
      type_count DESC
LIMIT 15;
```

Arguments

- @city1: Name of the first city
- @city1_latitude: User's latitude
- @city1_longitude: User's longitude
- @city2: Name of the second city
- @city2_latitude: Latitude of the second city
- @city2_longitude; longitude of the second city
- @distance_threshold: Max distance to return results from (used in prefiltering)

Functionality

This query is responsible for finding the "best" ElectricVehicles for traveling between 2 cities. When traveling to different cities using an EV, it is important to understand what charging environments you can expect in both cities. To provide this information to our users, we use this query to provide a list of the number of usable PlugInstances for each type of EV, as well as a variety of statistics about the EV itself and the expected charging times and costs for charging per 100 km as a measure of efficiency. Users may value different things, especially if they are given a choice to rent an EV, so this query provides a diverse and verbose result set which allows people to determine what the best EV for their use case may be.

Complexity Requirements

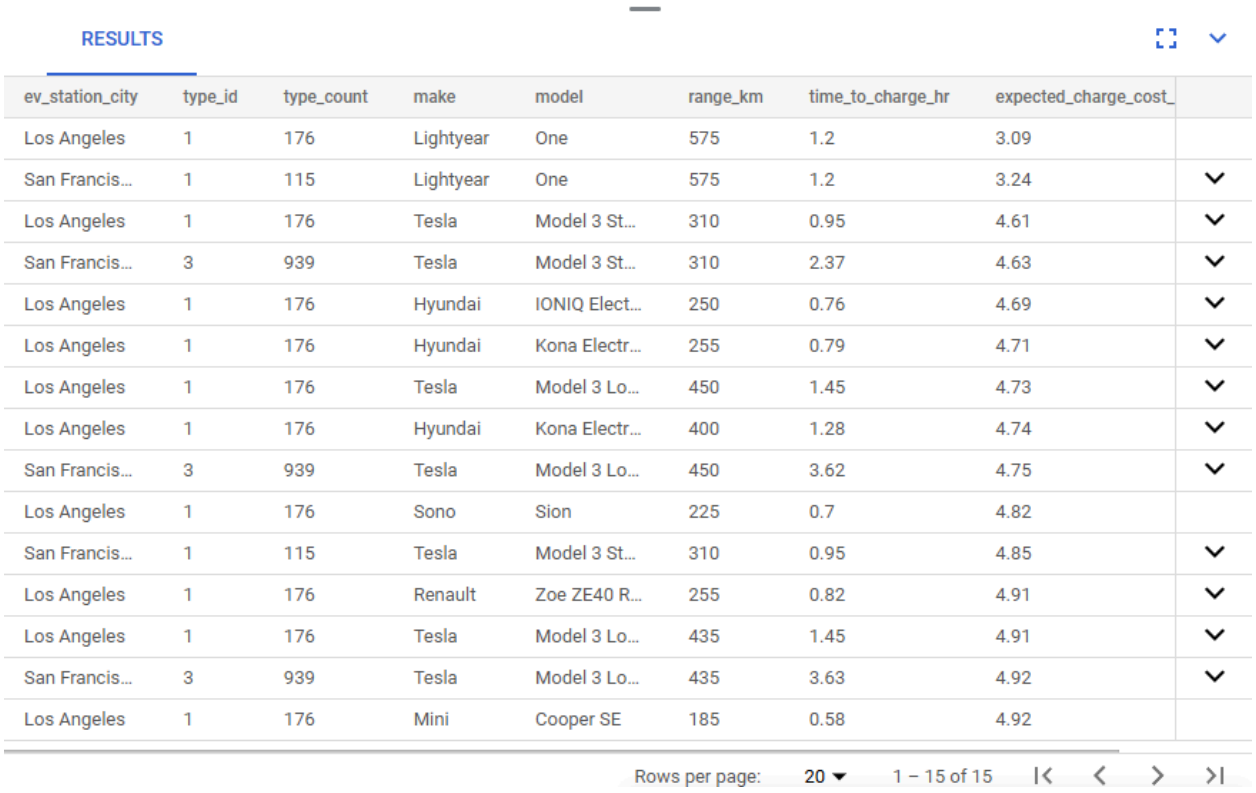
It combines data from two cities using `UNION`. It has multiple joins between `HasPlugs`, `PlugInstance`, and `ElectricVehicle`. It uses `COUNT()` and `AVG()` to compute statistics.

Justification for Usage

This query is unique because it provides a comparative analysis across two geographic locations, helping users make informed decisions about vehicle selection based on infrastructure and cost considerations for long-distance travel. It is distinct due to cross-city analysis and focuses on recommending electric vehicles rather than stations or user data.Unique parts of the query include the Union of statistics for different locations and plug type

compatibility checks via aggregation over plug types to ensure correct EV statistic computations.

Query Results Image



RESULTS

ev_station_city	type_id	type_count	make	model	range_km	time_to_charge_hr	expected_charge_cost_
Los Angeles	1	176	Lightyear	One	575	1.2	3.09
San Francis...	1	115	Lightyear	One	575	1.2	3.24
Los Angeles	1	176	Tesla	Model 3 St...	310	0.95	4.61
San Francis...	3	939	Tesla	Model 3 St...	310	2.37	4.63
Los Angeles	1	176	Hyundai	IONIQ Elect...	250	0.76	4.69
Los Angeles	1	176	Hyundai	Kona Electr...	255	0.79	4.71
Los Angeles	1	176	Tesla	Model 3 Lo...	450	1.45	4.73
Los Angeles	1	176	Hyundai	Kona Electr...	400	1.28	4.74
San Francis...	3	939	Tesla	Model 3 Lo...	450	3.62	4.75
Los Angeles	1	176	Sono	Sion	225	0.7	4.82
San Francis...	1	115	Tesla	Model 3 St...	310	0.95	4.85
Los Angeles	1	176	Renault	Zoe ZE40 R...	255	0.82	4.91
Los Angeles	1	176	Tesla	Model 3 Lo...	435	1.45	4.91
San Francis...	3	939	Tesla	Model 3 Lo...	435	3.63	4.92
Los Angeles	1	176	Mini	Cooper SE	185	0.58	4.92

Rows per page: 20 1 – 15 of 15

Q7: Mutated Mega-Recursive Halting-Problem-Vulnerable Super Query for Finding Shortest Number of EVStations from Start Location to Finish Location Along a Path Entirely in SQL

This one is a joke. Please don't include it in the grading.

```
-- Adapted from the Post Below
--
https://jamesmccaffrey.wordpress.com/2010/07/20/sql-graph-shortest-path-part-ii/

CREATE PROCEDURE usp_shortestPath (
    @startLatitude DECIMAL(8,6),
    @startLongitude DECIMAL(9,6),
    @endLatitude DECIMAL(8,6),
    @endLongitude DECIMAL(9,6)
)
AS
BEGIN
    SET NOCOUNT ON;

    -- Create a temporary table to hold all nodes
    CREATE TABLE #tblAllNodes (
        nodeid VARCHAR(50) NOT NULL,
        distance_km FLOAT NULL,
        previous VARCHAR(50) NULL,
        done BIT NULL
    );

    -- Insert start, end, and all EVStation nodes into #tblAllNodes
    INSERT INTO #tblAllNodes (nodeid)
    VALUES ('start'), ('end');

    INSERT INTO #tblAllNodes (nodeid)
    SELECT CAST(station_id AS VARCHAR(50)) FROM EVStation;

    -- Initialize distances and flags
    UPDATE #tblAllNodes
    SET distance_km = 2147483647, previous = NULL, done = 0;
```

```sql
-- Set the distance for the start node to 0
UPDATE #tblAllNodes
SET distance_km = 0
WHERE nodeid = 'start';

-- Create temporary tables for node coordinates
CREATE TABLE #NodeCoords (
    nodeid VARCHAR(50) NOT NULL,
    latitude DECIMAL(8,6),
    longitude DECIMAL(9,6)
);

-- Insert coordinates for start, end, and EVStations
INSERT INTO #NodeCoords (nodeid, latitude, longitude)
VALUES ('start', @startLatitude, @startLongitude),
       ('end', @endLatitude, @endLongitude);

INSERT INTO #NodeCoords (nodeid, latitude, longitude)
SELECT CAST(station_id AS VARCHAR(50)), latitude, longitude FROM EVStation;

-- Create a temporary table for edges
CREATE TABLE #tblEdges (
    fromNode VARCHAR(50) NOT NULL,
    toNode VARCHAR(50) NOT NULL,
    edgeWeight FLOAT NOT NULL
);

-- Calculate distances between nodes and insert into #tblEdges
INSERT INTO #tblEdges (fromNode, toNode, edgeWeight)
SELECT
    n1.nodeid AS fromNode,
    n2.nodeid AS toNode,
    6371 * 2 * ASIN(
        SQRT(
            POWER(SIN(RADIANS(n2.latitude - n1.latitude) / 2.0), 2) +
            COS(RADIANS(n1.latitude)) * COS(RADIANS(n2.latitude)) *
            POWER(SIN(RADIANS(n2.longitude - n1.longitude) / 2.0), 2)
        )
    ) AS edgeWeight
```

```
    FROM #NodeCoords n1
    CROSS JOIN #NodeCoords n2
    WHERE n1.nodeid <> n2.nodeid
      AND 6371 * 2 * ASIN(
            SQRT(
                POWER(SIN(RADIANS(n2.latitude - n1.latitude) / 2.0), 2) +
                COS(RADIANS(n1.latitude)) * COS(RADIANS(n2.latitude)) *
                POWER(SIN(RADIANS(n2.longitude - n1.longitude) / 2.0), 2)
            )
        ) <= 150;

    -- Variables for the main loop
    DECLARE @dist FLOAT, @smallestDistance FLOAT, @nodeWithSmallestDistance
VARCHAR(50);

    -- Main loop implementing Dijkstra's algorithm
    WHILE 1 = 1
    BEGIN
        SET @nodeWithSmallestDistance = NULL;

        SELECT TOP 1
            @nodeWithSmallestDistance = nodeid,
            @smallestDistance = distance_km
        FROM #tblAllNodes
        WHERE done = 0
        ORDER BY distance_km;

        IF @nodeWithSmallestDistance IS NULL BREAK;
        IF @nodeWithSmallestDistance = 'end' BREAK;

        UPDATE #tblAllNodes
        SET done = 1
        WHERE nodeid = @nodeWithSmallestDistance;

        UPDATE #tblAllNodes
        SET
            distance_km = @smallestDistance + e.edgeWeight,
            previous = @nodeWithSmallestDistance
        FROM #tblAllNodes n
```

```
        INNER JOIN #tblEdges e ON n.nodeid = e.toNode
        WHERE e.fromNode = @nodeWithSmallestDistance
          AND (@smallestDistance + e.edgeWeight) < n.distance_km
          AND n.done = 0;
    END

    -- Retrieve the shortest path if it exists
    DECLARE @totalDistance FLOAT;
    SELECT @totalDistance = distance_km
    FROM #tblAllNodes
    WHERE nodeid = 'end';

    IF @totalDistance = 2147483647
    BEGIN
        SELECT 'No path found' AS Message;
    END
    ELSE
    BEGIN
        -- Reconstruct the path from end to start
        DECLARE @path VARCHAR(MAX) = '';
        DECLARE @currentNode VARCHAR(50) = 'end';

        WHILE @currentNode IS NOT NULL
        BEGIN
            SET @path = @currentNode + CASE WHEN @path = '' THEN '' ELSE ',' END
+ @path;
            SELECT @currentNode = previous FROM #tblAllNodes WHERE nodeid =
@currentNode;
        END

        SELECT @path AS Path, @totalDistance AS TotalDistance_km;
    END
END;
```

Part 2 Indexing

A	B	C	D	E	F	G
Index Not Applicable						
Indexing Scheme:	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**
No Indexing	122721.54	29614.15	56139.56	42129.03	33521.89	55058.3
EVStation.latitude/longitude	39954.96			20248.85	17276.52	33172.58
TrafficStation.latitude/longitude		10844.54				
HourVolumeData.hour		29614.23				
EVStation.latitude/longitude (Non composite)	29448.62			19160.06	15713.87	22950.66
EVStation.city						3959.13
TrafficStation.latitude/longitude (Non composite)		1703.02				
FInal Indexing Scheme	**38079.49**	**2187.22**	**56139.56**	**23151.22**	**21706.36**	**6511.75**

Indexing schemes tried are listed above in a table for all the different queries. Each index scheme was tried in isolation except for the Final Indexing Scheme which used our combination of 5 indices. All tests below the 4th row used all previous indexes, so row 4 only had the EVStation latitude and longitude indexes, row 5 added the EVStation city index, etc. Above, we have highlighted the optimal query costs for each indexing scheme all of which had their respective indexes in isolation which makes sense from a database performance perspective since more indexes mean less space in memory for actual DB operations.

The non-indexed queries all had relatively high costs where many of the costs came from very expensive joins between tables. Since our database schema is highly normalized with 3NF, we have multiple tables which reduces data redundancy but hurts query performance significantly due to the required joins between normalized tables.

Some candidate columns we theorized would make good columns for indexing included the following:

- Latitude and Longitude columns: These make sense to index over since many of our distance thresholding methods use the BETWEEN operator. As covered in class, creating indexes with B Trees is especially efficient when using this operator since we can traverse to the leaf node and perform a linear search to get all records that are valid. For these columns, we tried composite indexing with both latitude and longitude as a composite attribute in a single index and separate indexing with each attribute in its own index.
- City: This attribute was decided upon because it would allow a very efficient filtering operation for Q6 in particular. Although including this one attribute as an index would

only help one of our queries, we found that it was an effective way to improve the performance of this query by nearly an order of magnitude.
- Hour: We believed this would be an effective indexing attribute as it appeared in the computation for the Congestion scores in Q2. This attribute also appeared inside a BETWEEN clause which further made us believe it would be an effective candidate for indexing over.

Unfortunately, we couldn't index over every possible combination of columns due to time constraints, but we believe the columns listed represent a good set of indexable attributes. Many of the attributes present in JOINS within our queries were primary keys or foreign keys as a result of our normalization process, so a lot of potential attributes were ignored simply because they already have indexes due to their nature as keys.

Composite EVStation lat/long indexes & Composite TrafficStation lat/long indexes
We observed a lower cost compared to when we had no index. The queries had a lower cost specifically from joining on Users, OwnsEV, TrafficStation, etc. Most of the work was from joining tables which made the index cost-efficient.

HourVolumeData.hour Indexes
While we initially thought this to be a useful column to index over, experimental results showed that this actually had no effect on the overall performance of the query. We believe this is mainly caused by the fact that the majority of the cost for this query comes from the table scan to find the TrafficStations themselves, and a relatively small amount of work goes into the actual averaging since the HourVolumeData records are already filtered using the volume_id by the time we even start selecting by hour. This makes the hour column relatively ineffective and would only serve to waste DB memory since the HourVolumeData table is our largest by far.

Non-composite indexing for both EVStation and TrafficStation
One interesting point found was that using a composite index on the latitude and longitude attributes for both the EVStation and TrafficStation tables was less efficient than using 2 separate indexes for each attribute instead. We believe this happens because the single attribute indexing allows for more efficient lookups and traversal of the B Tree since there is no need to condition on the secondary attribute, however, more experimentation is needed to verify this.

City indexing for EVStation
City indexing for the EVStation table proved highly useful. Since we are only considering EVStations from certain cities in this query, it makes sense to index over this attribute to pre-filter any records before the more expensive portions of the query are run. This indexing allowed us to see a speedup of an order of magnitude in terms of cost and was deemed worth including even if it were only applicable to this one query.

Final Indexing Scheme

Our final indexing scheme was simply the collection of indexes that improved our performances. These include the separate indexing of latitude and longitude for both EVStation and TrafficStation as well as the city index in EVStation.

Looking at the table, the green highlighted columns are the most cost-efficient indexing for each query. Smaller indexes were more efficient and to maintain a low cost, we had to avoid any redundant indexes that covered similar columns.

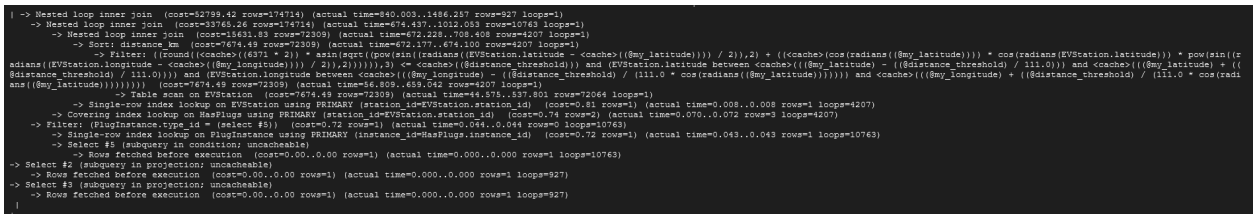
Query Performance Raw Data

Project Track 1 Phase 3 Data

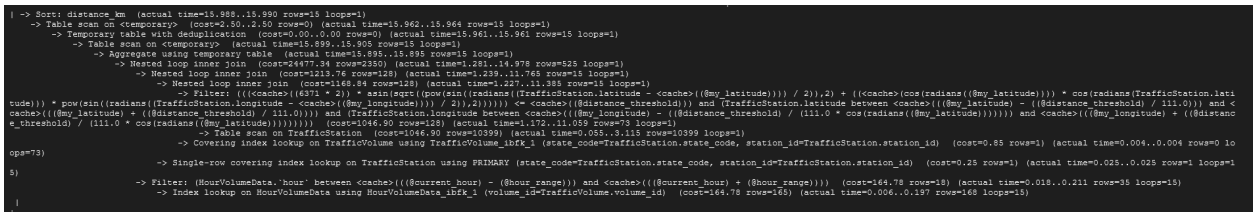
Analysis Notes

Raw text outputs and the table itself are visible in the Query Performance Raw Data Google Sheets document linked above.

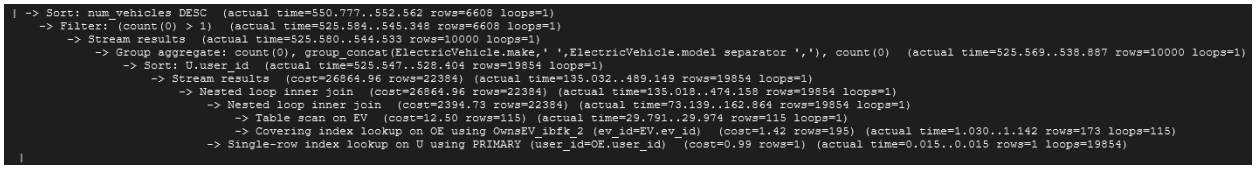
Q1 No Indexing



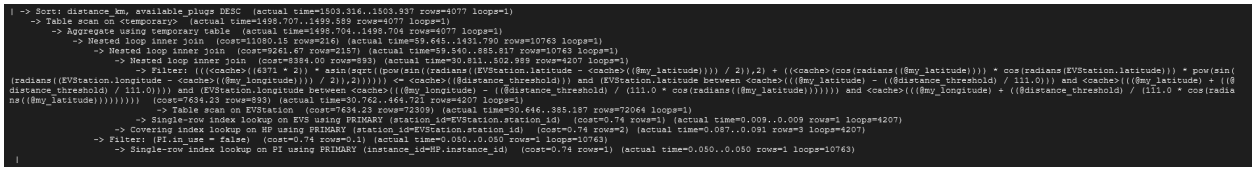
Q2 No Indexing



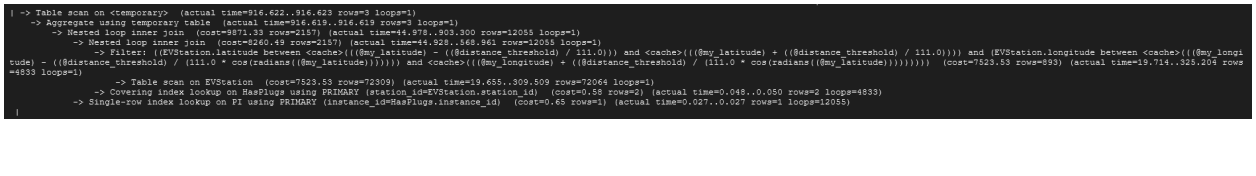
Q3 No Indexing



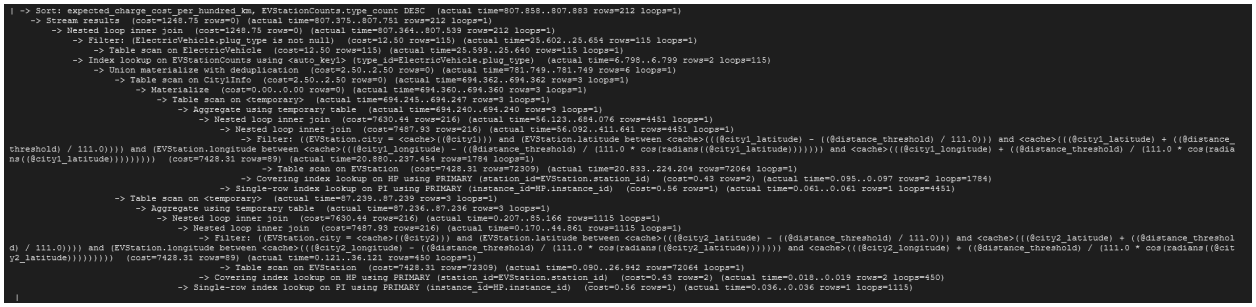
Q4 No Indexing



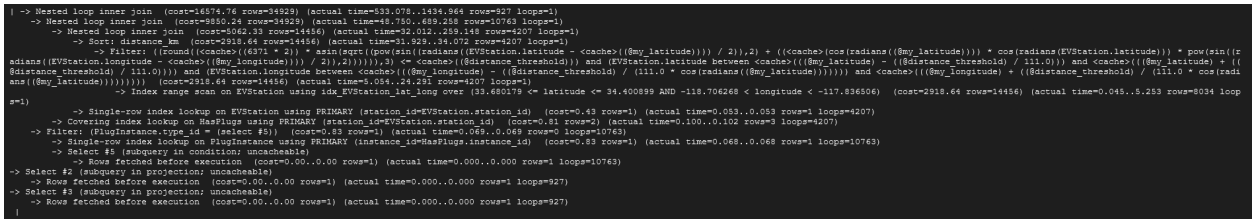
Q5 No Indexing



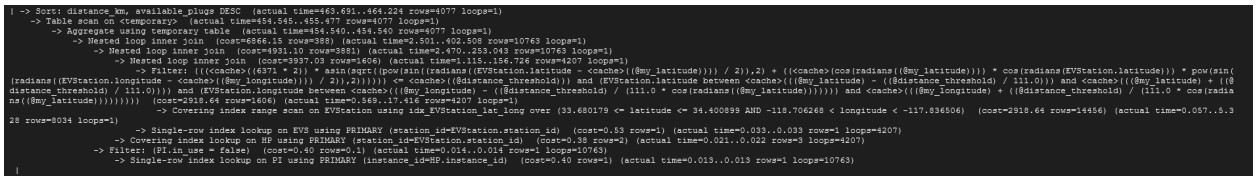
Q6 No Indexing



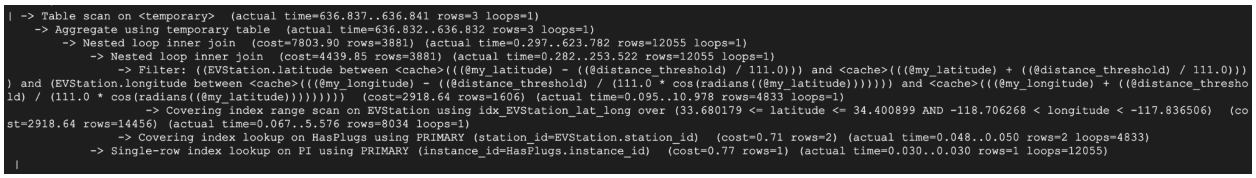
Q1 Composite Indexing EVStation lat/long



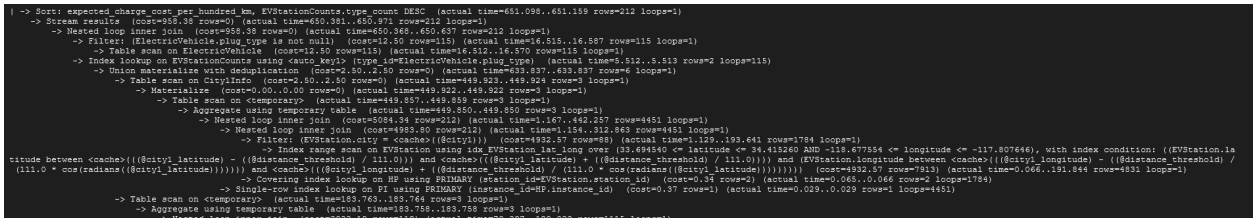
Q4 Composite Indexing EVStation lat/long



Q5 Composite Indexing EVStation lat/long



Q6 Composite Indexing EVStation lat/long



Q2 Composite Indexing TrafficStation lat/long

```
| -> Sort: distance_km  (actual time=301.885..301.887 rows=15 loops=1)
    -> Table scan on <temporary>  (cost=2.50..2.50 rows=0) (actual time=301.848..301.852 rows=15 loops=1)
        -> Temporary table with deduplication  (cost=0.00..0.00 rows=0) (actual time=301.847..301.847 rows=15 loops=1)
            -> Table scan on <temporary>  (actual time=301.666..301.675 rows=15 loops=1)
                -> Aggregate using temporary table  (actual time=301.661..301.661 rows=15 loops=1)
                    -> Nested loop inner join  (cost=9899.62 rows=982) (actual time=196.306..299.075 rows=525 loops=1)
                        -> Nested loop inner join  (cost=173.98 rows=54) (actual time=57.089..59.024 rows=15 loops=1)
                            -> Nested loop inner join  (cost=155.20 rows=54) (actual time=56.989..58.699 rows=15 loops=1)
                                -> Filter: (((<cache>((6371 * 2)) * asin(sqrt((pow(sin((radians((TrafficStation.latitude - <cache>((@my_latitude)))) / 2)),2) + ((<cache>(cos(radia
ns((@my_latitude)))) * cos(radians(TrafficStation.latitude))) * pow(sin((radians((TrafficStation.longitude - <cache>((@my_longitude)))) / 2)),2)))))) <= <cache>((@distance_thresho
ld))) and (TrafficStation.latitude between <cache>(((@my_latitude) - ((@distance_threshold) / 111.0))) and <cache>(((@my_latitude) + ((@distance_threshold) / 111.0)))) and (Traffi
cStation.longitude between <cache>(((@my_longitude) - ((@distance_threshold) / (111.0 * cos(radians((@my_latitude)))))))) and <cache>(((@my_longitude) + ((@distance_threshold) / (1
11.0 * cos(radians((@my_latitude)))))))))  (cost=100.20 rows=54) (actual time=0.328..1.465 rows=73 loops=1)
                                    -> Covering index range scan on TrafficStation using idx_TrafficStation_lat_long over (33.680179 <= latitude <= 34.400899 AND -118.706268 < lon
gitude < -117.836506)  (cost=100.20 rows=483) (actual time=0.096..0.446 rows=483 loops=1)
                                -> Covering index lookup on TrafficVolume using TrafficVolume_ibfk_1 (state_code=TrafficStation.state_code, station_id=TrafficStation.station_id)
(cost=0.93 rows=1) (actual time=0.782..0.784 rows=0 loops=73)
                            -> Single-row covering index lookup on TrafficStation using PRIMARY (state_code=TrafficStation.state_code, station_id=TrafficStation.station_id)  (cost
=0.25 rows=1) (actual time=0.020..0.020 rows=1 loops=15)
                        -> Filter: (HourVolumeData.`hour` between <cache>(((@current_hour) - (@hour_range))) and <cache>(((@current_hour) + (@hour_range))))  (cost=164.80 rows=18)
 (actual time=14.156..15.998 rows=35 loops=15)
                            -> Index lookup on HourVolumeData using HourVolumeData_ibfk_1 (volume_id=TrafficVolume.volume_id)  (cost=164.80 rows=165) (actual time=14.113..15.972 r
ows=168 loops=15)
 |
```

Q2 HourVolumeData Hour Indexing

```
| -> Sort: distance_km  (actual time=269.930..269.932 rows=15 loops=1)
    -> Table scan on <temporary>  (cost=2.50..2.50 rows=0) (actual time=269.849..269.851 rows=15 loops=1)
        -> Temporary table with deduplication  (cost=0.00..0.00 rows=0) (actual time=269.847..269.847 rows=15 loops=1)
            -> Table scan on <temporary>  (actual time=268.400..268.406 rows=15 loops=1)
                -> Aggregate using temporary table  (actual time=268.397..268.397 rows=15 loops=1)
                    -> Nested loop inner join  (cost=9940.03 rows=3486) (actual time=223.952..265.843 rows=525 loops=1)
                        -> Nested loop inner join  (cost=214.39 rows=54) (actual time=112.355..114.073 rows=15 loops=1)
                            -> Nested loop inner join  (cost=155.37 rows=54) (actual time=76.775..78.321 rows=15 loops=1)
                                -> Filter: (((<cache>((6371 * 2)) * asin(sqrt((pow(sin((radians((TrafficStation.latitude - <cache>((@my_latitude)))) / 2)),2) + ((<cache>(cos(radia
ns((@my_latitude)))) * cos(radians(TrafficStation.latitude))) * pow(sin((radians((TrafficStation.longitude - <cache>((@my_longitude)))) / 2)),2)))))) <= <cache>((@distance_thresho
ld))) and (TrafficStation.latitude between <cache>(((@my_latitude) - ((@distance_threshold) / 111.0))) and <cache>(((@my_latitude) + ((@distance_threshold) / 111.0)))) and (Traffi
cStation.longitude between <cache>(((@my_longitude) - ((@distance_threshold) / (111.0 * cos(radians((@my_latitude)))))))) and <cache>(((@my_longitude) + ((@distance_threshold) / (1
11.0 * cos(radians((@my_latitude)))))))))  (cost=100.36 rows=54) (actual time=0.216..1.176 rows=73 loops=1)
                                    -> Covering index range scan on TrafficStation using idx_TrafficStation_lat_long over (33.680179 <= latitude <= 34.400899 AND -118.706268 < lon
gitude < -117.836506)  (cost=100.36 rows=483) (actual time=0.070..0.384 rows=483 loops=1)
                                -> Covering index lookup on TrafficVolume using TrafficVolume_ibfk_1 (state_code=TrafficStation.state_code, station_id=TrafficStation.station_id)
(cost=0.93 rows=1) (actual time=1.055..1.057 rows=0 loops=73)
                            -> Single-row covering index lookup on TrafficStation using PRIMARY (state_code=TrafficStation.state_code, station_id=TrafficStation.station_id)  (cost
=1.00 rows=1) (actual time=2.382..2.382 rows=1 loops=15)
                        -> Filter: (HourVolumeData.`hour` between <cache>(((@current_hour) - (@hour_range))) and <cache>(((@current_hour) + (@hour_range))))  (cost=164.89 rows=65)
 (actual time=9.352..10.114 rows=35 loops=15)
                            -> Index lookup on HourVolumeData using HourVolumeData_ibfk_1 (volume_id=TrafficVolume.volume_id)  (cost=164.89 rows=165) (actual time=9.318..10.089 ro
ws=168 loops=15)
 |
```

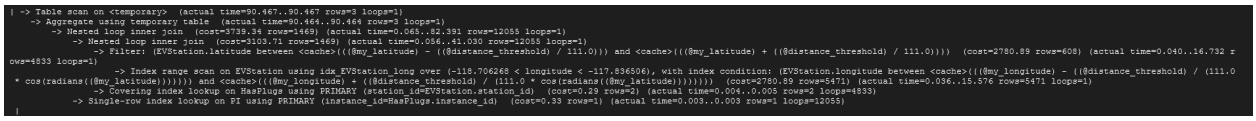
Q1 Separate Indexing EVStation lat/long

```
| -> Nested loop inner join  (cost=9104.95 rows=13219) (actual time=364.772..1150.078 rows=927 loops=1)
    -> Nested loop inner join  (cost=6500.38 rows=13219) (actual time=233.599..616.180 rows=10763 loops=1)
        -> Nested loop inner join  (cost=4655.33 rows=5471) (actual time=232.082..260.425 rows=4207 loops=1)
            -> Sort: distance_km  (cost=3806.89 rows=5471) (actual time=232.046..234.482 rows=4207 loops=1)
                -> Filter: ((round((<cache>((6371 * 2)) * asin(sqrt((pow(sin((radians((EVStation.latitude - <cache>((@my_latitude)))) / 2)),2) + ((<cache>(cos(radians((@my_latitud
e)))) * cos(radians(EVStation.latitude))) * pow(sin((radians((EVStation.longitude - <cache>((@my_longitude)))) / 2)),2)))))),3) <= <cache>((@distance_threshold))) and (EVStation.l
atitude between <cache>(((@my_latitude) - ((@distance_threshold) / 111.0))) and <cache>(((@my_latitude) + ((@distance_threshold) / 111.0))))  (cost=3806.89 rows=5471) (actual tim
e=0.195..225.209 rows=4207 loops=1)
                    -> Index range scan on EVStation using idx_EVStation_long over (-118.706268 < longitude < -117.836506), with index condition: (EVStation.longitude between <cac
he>(((@my_longitude) - ((@distance_threshold) / (111.0 * cos(radians((@my_latitude)))))))) and <cache>(((@my_longitude) + ((@distance_threshold) / (111.0 * cos(radians((@my_latitud
e)))))))))  (cost=3806.89 rows=5471) (actual time=0.075..215.661 rows=5471 loops=1)
            -> Single-row index lookup on EVStation using PRIMARY (station_id=EVStation.station_id)  (cost=0.50 rows=1) (actual time=0.006..0.006 rows=1 loops=4207)
        -> Covering index lookup on HasPlugs using PRIMARY (station_id=EVStation.station_id)  (cost=0.86 rows=2) (actual time=0.082..0.084 rows=3 loops=4207)
    -> Filter: (PlugInstance.type_id = (select #5))  (cost=0.87 rows=1) (actual time=0.049..0.049 rows=0 loops=10763)
        -> Single-row index lookup on PlugInstance using PRIMARY (instance_id=HasPlugs.instance_id)  (cost=0.87 rows=1) (actual time=0.048..0.048 rows=1 loops=10763)
        -> Select #5 (subquery in condition; uncacheable)
            -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=10763)
-> Select #2 (subquery in projection; uncacheable)
    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=927)
-> Select #3 (subquery in projection; uncacheable)
    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=927)
 |
```

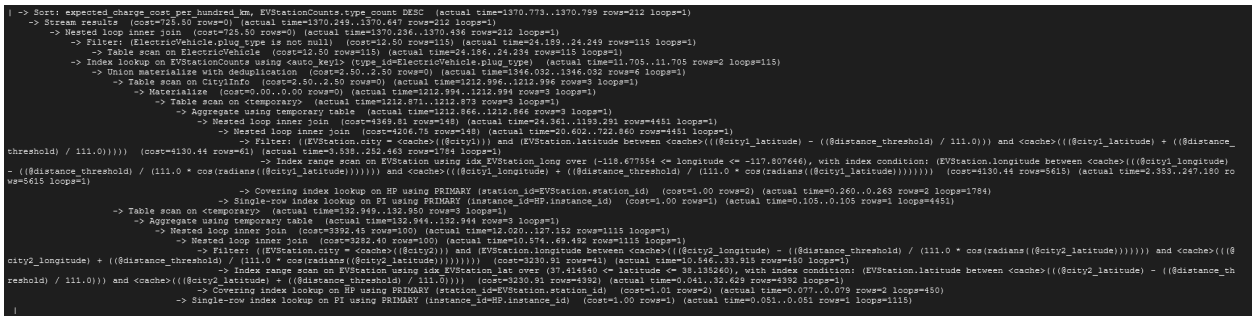
Q4 Separate Indexing EVStation lat/long

```
| -> Sort: distance_km, available_plugs DESC  (actual time=696.071..696.622 rows=4077 loops=1)
    -> Table scan on <temporary>  (actual time=690.752..691.663 rows=4077 loops=1)
        -> Aggregate using temporary table  (actual time=690.748..690.748 rows=4077 loops=1)
            -> Nested loop inner join  (cost=5196.65 rows=147) (actual time=0.135..641.476 rows=10763 loops=1)
                -> Nested loop inner join  (cost=4543.66 rows=1469) (actual time=0.124..536.454 rows=10763 loops=1)
                    -> Nested loop inner join  (cost=4220.84 rows=608) (actual time=0.106..502.633 rows=4207 loops=1)
                        -> Filter: (((<cache>((6371 * 2)) * asin(sqrt((pow(sin((radians((EVStation.latitude - <cache>((@my_latitude)))) / 2)),2) + ((<cache>(cos(radians((@my_latit
ude)))) * cos(radians(EVStation.latitude))) * pow(sin((radians((EVStation.longitude - <cache>((@my_longitude)))) / 2)),2)))))) <= <cache>((@distance_threshold))) and (EVStation.la
titude between <cache>(((@my_latitude) - ((@distance_threshold) / 111.0))) and <cache>(((@my_latitude) + ((@distance_threshold) / 111.0))))  (cost=3853.53 rows=608) (actual time=
0.096..488.077 rows=4207 loops=1)
                            -> Index range scan on EVStation using idx_EVStation_long over (-118.706268 < longitude < -117.836506), with index condition: (EVStation.longitude betw
een <cache>(((@my_longitude) - ((@distance_threshold) / (111.0 * cos(radians((@my_latitude)))))))) and <cache>(((@my_longitude) + ((@distance_threshold) / (111.0 * cos(radians((@my
_latitude)))))))))  (cost=3853.53 rows=5471) (actual time=0.063..476.877 rows=5471 loops=1)
                        -> Single-row index lookup on EVS using PRIMARY (station_id=EVStation.station_id)  (cost=0.50 rows=1) (actual time=0.003..0.003 rows=1 loops=4207)
                    -> Covering index lookup on HP using PRIMARY (station_id=EVStation.station_id)  (cost=0.29 rows=2) (actual time=0.006..0.008 rows=3 loops=4207)
                -> Filter: (PI.in_use = false)  (cost=0.34 rows=0.1) (actual time=0.009..0.010 rows=1 loops=10763)
                    -> Single-row index lookup on PI using PRIMARY (instance_id=HP.instance_id)  (cost=0.34 rows=1) (actual time=0.009..0.009 rows=1 loops=10763)
 |
```

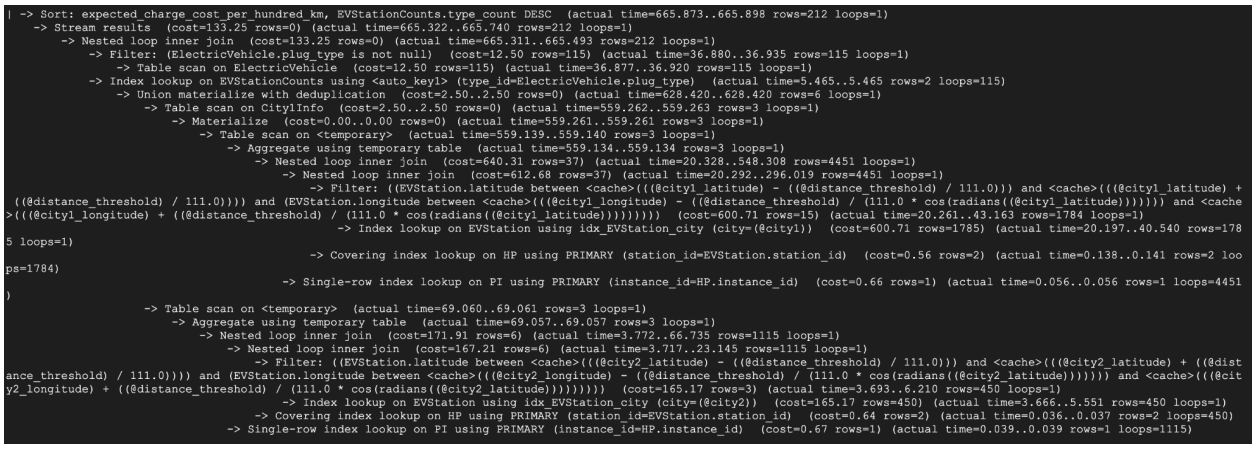
Q5 Separate Indexing EVStation lat/long



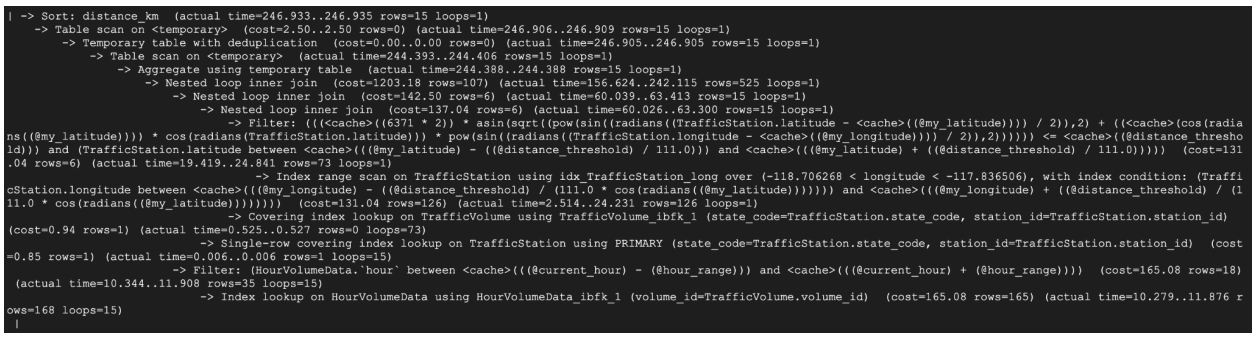
Q6 Separate Indexing EVStation lat/long



Q6 Separate Lat/long indexing + City indexing



Q2 Separate Lat/long indexing + EVStation Indexes



Q1 Final Indexing

```
| -> Nested loop inner join  (cost=7799.17 rows=13219) (actual time=209.345..455.557 rows=927 loops=1)
    -> Nested loop inner join  (cost=5737.94 rows=13219) (actual time=155.102..277.869 rows=10763 loops=1)
        -> Nested loop inner join  (cost=4111.25 rows=5471) (actual time=154.988..172.924 rows=4207 loops=1)
            -> Sort: distance_km  (cost=3317.21 rows=5471) (actual time=154.930..156.343 rows=4207 loops=1)
                -> Filter: ((round((<cache>((6371 * 2)) * asin(sqrt((pow(sin((radians((EVStation.latitude - <cache>((@my_latitude)))) / 2)),2) + ((<cache>(cos(radians((@my_latitud
e)))) * cos(radians(EVStation.latitude))) * pow(sin((radians((EVStation.longitude - <cache>((@my_longitude)))) / 2)),2))))),3) <= <cache>((@distance_threshold))) and (EVStation.l
atitude between <cache>(((@my_latitude) - ((@distance_threshold) / 111.0))) and <cache>(((@my_latitude) + ((@distance_threshold) / 111.0))))))  (cost=3317.21 rows=5471) (actual tim
e=0.104..145.852 rows=4207 loops=1)
                    -> Index range scan on EVStation using idx_EVStation_long over (-118.706268 < longitude < -117.836506), with index condition: (EVStation.longitude between <cac
he>((((@my_longitude) - ((@distance_threshold) / (111.0 * cos(radians((@my_latitude)))))))) and <cache>((((@my_longitude) + ((@distance_threshold) / (111.0 * cos(radians((@my_latitud
e)))))))))  (cost=3317.21 rows=5471) (actual time=0.050..138.628 rows=5471 loops=1)
            -> Single-row index lookup on EVStation using PRIMARY (station_id=EVStation.station_id)  (cost=0.41 rows=1) (actual time=0.004..0.004 rows=1 loops=4207)
        -> Covering index lookup on HasPlugs using PRIMARY (station_id=EVStation.station_id)  (cost=0.50 rows=2) (actual time=0.023..0.025 rows=3 loops=4207)
    -> Filter: (PlugInstance.type_id = (select #5))  (cost=0.50 rows=1) (actual time=0.016..0.016 rows=0 loops=10763)
        -> Single-row index lookup on PlugInstance using PRIMARY (instance_id=HasPlugs.instance_id)  (cost=0.50 rows=1) (actual time=0.015..0.016 rows=1 loops=10763)
        -> Select #5 (subquery in condition; uncacheable)
            -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=10763)
-> Select #2 (subquery in projection; uncacheable)
    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=927)
-> Select #3 (subquery in projection; uncacheable)
    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=927)
 |
```

Q2 Final Indexing

```
| -> Sort: distance_km  (actual time=265.743..265.745 rows=15 loops=1)
    -> Table scan on <temporary>  (cost=2.50..2.50 rows=0) (actual time=265.721..265.724 rows=15 loops=1)
        -> Temporary table with deduplication  (cost=0.00..0.00 rows=0) (actual time=265.719..265.719 rows=15 loops=1)
            -> Table scan on <temporary>  (actual time=265.658..265.665 rows=15 loops=1)
                -> Aggregate using temporary table  (actual time=265.653..265.653 rows=15 loops=1)
                    -> Nested loop inner join  (cost=1217.94 rows=107) (actual time=184.002..264.125 rows=525 loops=1)
                        -> Nested loop inner join  (cost=157.27 rows=6) (actual time=56.324..59.450 rows=15 loops=1)
                            -> Nested loop inner join  (cost=151.15 rows=6) (actual time=56.311..59.342 rows=15 loops=1)
                                -> Filter: (((<cache>((6371 * 2)) * asin(sqrt((pow(sin((radians((TrafficStation.latitude - <cache>((@my_latitude)))) / 2)),2) + ((<cache>(cos(radia
ns((@my_latitude)))) * cos(radians(TrafficStation.latitude))) * pow(sin((radians((TrafficStation.longitude - <cache>((@my_longitude)))) / 2)),2)))))) <= <cache>((@distance_thresho
ld))) and (TrafficStation.latitude between <cache>(((@my_latitude) - ((@distance_threshold) / 111.0))) and <cache>(((@my_latitude) + ((@distance_threshold) / 111.0)))))  (cost=145
.15 rows=6) (actual time=31.979..36.780 rows=73 loops=1)
                                    -> Index range scan on TrafficStation using idx_TrafficStation_long over (-118.706268 < longitude < -117.836506), with index condition: (TraffiStation.longitude between <cache>((((@my_longitude) - ((@distance_threshold) / (111.0 * cos(radians((@my_latitude)))))))) and <cache>(((@my_longitude) + ((@distance_threshold) / (1
11.0 * cos(radians((@my_latitude)))))))  (cost=145.15 rows=126) (actual time=23.309..36.297 rows=126 loops=1)
                            -> Covering index lookup on TrafficVolume using TrafficVolume_ibfk_1 (state_code=TrafficStation.state_code, station_id=TrafficStation.station_id)
(cost=0.94 rows=1) (actual time=0.308..0.309 rows=0 loops=73)
                        -> Single-row covering index lookup on TrafficStation using PRIMARY (state_code=TrafficStation.state_code, station_id=TrafficStation.station_id)  (cost
=0.96 rows=1) (actual time=0.006..0.006 rows=1 loops=15)
                    -> Filter: (HourVolumeData.`hour` between <cache>(((@current_hour) - (@hour_range))) and <cache>(((@current_hour) + (@hour_range))))  (cost=165.08 rows=18)
 (actual time=12.381..13.641 rows=35 loops=15)
                        -> Index lookup on HourVolumeData using HourVolumeData_ibfk_1 (volume_id=TrafficVolume.volume_id)  (cost=165.08 rows=165) (actual time=12.346..13.620 r
ows=168 loops=15)
 |
```

Q3 Final Indexing

```
| -> Sort: num_vehicles DESC  (actual time=647.788..649.297 rows=6608 loops=1)
    -> Filter: (count(0) > 1)  (actual time=620.764..640.868 rows=6608 loops=1)
        -> Stream results  (actual time=620.760..640.047 rows=10000 loops=1)
            -> Group aggregate: count(0), group_concat(ElectricVehicle.make,' ',ElectricVehicle.model separator ','), count(0)  (actual time=620.750..634.384 rows=10000 loops=1)
                -> Sort: U.user_id  (actual time=620.693..623.612 rows=19854 loops=1)
                    -> Stream results  (cost=26864.96 rows=22384) (actual time=129.488..588.101 rows=19854 loops=1)
                        -> Nested loop inner join  (cost=26864.96 rows=22384) (actual time=129.473..573.849 rows=19854 loops=1)
                            -> Nested loop inner join  (cost=2394.73 rows=22384) (actual time=84.022..236.467 rows=19854 loops=1)
                                -> Table scan on EV  (cost=12.50 rows=115) (actual time=15.172..15.411 rows=115 loops=1)
                                -> Covering index lookup on OE using OwnsEV_ibfk_2 (ev_id=EV.ev_id)  (cost=1.42 rows=195) (actual time=1.824..1.910 rows=173 loops=115)
                            -> Single-row index lookup on U using PRIMARY (user_id=OE.user_id)  (cost=0.99 rows=1) (actual time=0.017..0.017 rows=1 loops=19854)
 |
```

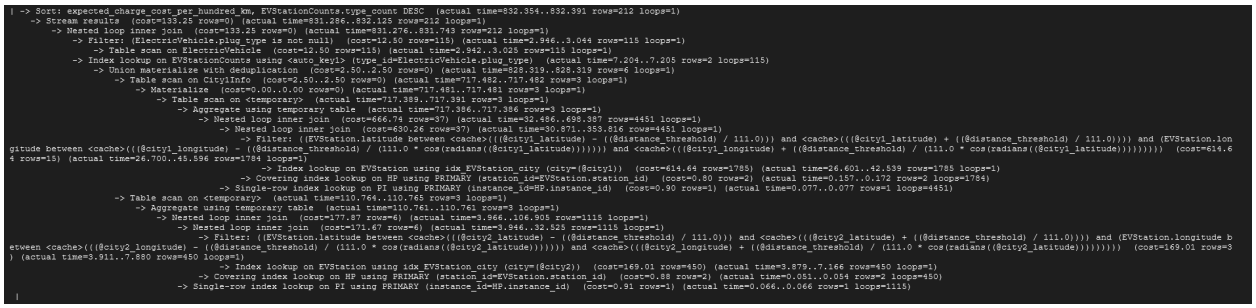
Q4 Final Indexing

```
| -> Sort: distance_km, available_plugs DESC  (actual time=511.429..511.976 rows=4077 loops=1)
    -> Table scan on <temporary>  (actual time=505.881..506.723 rows=4077 loops=1)
        -> Aggregate using temporary table  (actual time=505.877..505.877 rows=4077 loops=1)
            -> Nested loop inner join  (cost=4818.19 rows=147) (actual time=0.296..464.486 rows=10763 loops=1)
                -> Nested loop inner join  (cost=4073.41 rows=1469) (actual time=0.285..318.795 rows=10763 loops=1)
                    -> Nested loop inner join  (cost=3694.03 rows=608) (actual time=0.271..145.247 rows=4207 loops=1)
                        -> Filter: (((<cache>((6371 * 2)) * asin(sqrt((pow(sin((radians((EVStation.latitude - <cache>((@my_latitude)))) / 2)),2) + ((<cache>(cos(radians((@my_latitude)))) * cos(radians(EVStation.latitude))) * pow(sin(
(radians((EVStation.longitude - <cache>((@my_longitude)))) / 2)),2)))))) <= <cache>((@distance_threshold))) and (EVStation.latitude between <cache>(((@my_latitude) - ((@distance_threshold) / 111.0))) and <cache>(((@my_latitude) + ((@
distance_threshold) / 111.0))))  (cost=3379.39 rows=608) (actual time=0.254..133.860 rows=4207 loops=1)
                            -> Index range scan on EVStation using idx_EVStation_long over (-118.706268 < longitude < -117.836506), with index condition: (EVStation.longitude between <cache>((((@my_longitude) - ((@distance_threshold)
/ (111.0 * cos(radians((@my_latitude)))))))) and <cache>((((@my_longitude) + ((@distance_threshold) / (111.0 * cos(radians((@my_latitude))))))))  (cost=3379.39 rows=5471) (actual time=0.184..125.251 rows=5471 loops=1)
                        -> Single-row index lookup on EVS using PRIMARY (station_id=EVStation.station_id)  (cost=0.42 rows=1) (actual time=0.002..0.002 rows=1 loops=4207)
                    -> Covering index lookup on HP using PRIMARY (station_id=EVStation.station_id)  (cost=0.38 rows=2) (actual time=0.039..0.041 rows=3 loops=4207)
                -> Filter: (PI.in_use = false)  (cost=0.41 rows=0.1) (actual time=0.013..0.013 rows=1 loops=10763)
                    -> Single-row index lookup on PI using PRIMARY (instance_id=HP.instance_id)  (cost=0.41 rows=1) (actual time=0.013..0.013 rows=1 loops=10763)
 |
```

Q5 Final Indexing

```
| -> Table scan on <temporary>  (actual time=1039.930..1039.930 rows=3 loops=1)
    -> Aggregate using temporary table  (actual time=1039.926..1039.926 rows=3 loops=1)
        -> Nested loop inner join  (cost=5625.93 rows=1469) (actual time=6.210..1025.793 rows=12055 loops=1)
            -> Nested loop inner join  (cost=4432.09 rows=1469) (actual time=4.040..650.183 rows=12055 loops=1)
                -> Filter: (EVStation.latitude between <cache>(((@my_latitude) - ((@distance_threshold) / 111.0))) and <cache>(((@my_latitude) + ((@distance_threshold) / 111.0))))  (cost=3884.62 rows=608) (actual time=1.136..486.767
rows=4833 loops=1)
                    -> Index range scan on EVStation using idx_EVStation_long over (-118.706268 < longitude < -117.836506), with index condition: (EVStation.longitude between <cache>((((@my_longitude) - ((@distance_threshold) / (111.0
 * cos(radians((@my_latitude)))))))) and <cache>((((@my_longitude) + ((@distance_threshold) / (111.0 * cos(radians((@my_latitude))))))))  (cost=3884.62 rows=5471) (actual time=1.122..484.531 rows=5471 loops=1)
                -> Covering index lookup on HasPlugs using PRIMARY (station_id=EVStation.station_id)  (cost=0.66 rows=2) (actual time=0.032..0.033 rows=2 loops=4833)
            -> Single-row index lookup on PI using PRIMARY (instance_id=HasPlugs.instance_id)  (cost=0.71 rows=1) (actual time=0.031..0.031 rows=1 loops=12055)
 |
```

Q6 Final Indexing



Final Index Design

This was the final index design we chose as it used all the columns that conceptually made sense while minimizing the tradeoffs of having multiple indices. In particular, the indexing improvements for Q6 are especially good, but Q2 also saw notable improvements from its indexes alone. This will make running these queries in real time for end users much more efficient.

```
CREATE INDEX idx_EVStation_lat ON EVStation (latitude);
CREATE INDEX idx_EVStation_long ON EVStation (longitude);
CREATE INDEX idx_EVStation_city ON EVStation (city);
CREATE INDEX idx_TrafficStation_lat ON TrafficStation (latitude);
CREATE INDEX idx_TrafficStation_long ON TrafficStation (longitude);
```