Project Track 1 Stage 4 Final Report

Differences from Proposal

- Please list out changes in the directions of your project if the final project is different from your original proposal (based on your stage 1 proposal submission).

For the most part, our project direction stayed consistent when comparing our original proposal and our final application. While some of the proposed features were cut due to time or complexity constraints, we still delivered the core functionality of our application and provided an effective way for users to find ways to plan their trips when using electric vehicles.

Achievements and Shortcomings

- Discuss what you think your application achieved or failed to achieve regarding its usefulness.

Our application provides users an intuitive interface they can use to explore their options when it comes to using electric vehicles for trips. We provide information not only about the EV stations people have to choose from, but also about the EVs themselves. Combining this with a GUI which shows locations of stations on the map and the relevant information for each station allows people to better plan their trips.

While we believe that the project was an overall success, we believe that there is always room for improvement. Some of our UI elements could be a bit more intuitive or simple to use, and we don't provide 100% of the information that a user may want to know about a given EV station or stop, but these are all things that could be improved in the future since we have constructed a robust baseline application.

Changes to Schema & Data Sources

- Discuss if you changed the schema or source of the data for your application

Our schema remained consistent throughout development, and our data sources haven't changed. The final schema is mostly consistent with our stage 2/3 schema with the only real changes being minor adjustments of data types, the implementation of a few additional table checks and the other advanced database programs we were required to implement.

Change to ER diagram or Tables

- Discuss what you change to your ER diagram and/or your table implementations. What are some differences between the original design and the final design? Why? What do you think is a more suitable design?

Our tables and ER diagrams remained consistent between stages 2-4. We attribute this to our forward thinking to ensure that our data sources and our planned schema were well designed through normalization and any future expandability. In our final database schema, we did add a few additional tables to support the stored procedures we wrote, but these are fairly inconsequential and are only used for our advanced queries. In addition, we added some minor checks to help ensure data integrity, but many of these were also implemented in our earlier table versions.

Added/Removed functionalities

- Discuss what functionalities you added or removed. Why?

We mostly stuck to our proposal in terms of our advanced query functionality and our overall application layout and feature set. Some of the more real-time aspects of the project were removed due to the unnecessary complexity they would add. Things like viewing google location reviews of different EVStations or viewing nearby attractions are things that were removed from the final version so we could focus on the more core areas of the application, but these are all features that we could explore in the future via API calls or something similar.

Advanced Database Programming

- Explain how you think your advanced database programs complement your application.

Our advanced database programs are well suited to enhance the functionality of our application. The stored procedure for calculating the haversine distance between any 2 latitude and longitudes has proven extremely useful in both query functionality and readability, as it greatly simplified some of our advanced queries and improved their performance. Other complimentary database programming features include things like our robust checks for data integrity on things like location, datetime checks, and all of the foreign key constraints we have implemented. For the backend specifically, we ensured that all queries are handled by transactions and are given appropriate isolation levels to ensure that our database remains robust even in the event of a failure. Our granular approach to isolation level management (discussed later) gives us a way to fine tune database stability vs performance for each individual endpoint.

Technical Challenges

- Each team member should describe one technical challenge that the team encountered. This should be sufficiently detailed such that another future team could use this as helpful advice if they were to start a similar project or were to maintain your project.

Nicholas Keriotis: One of the technical challenges I encountered was figuring out how to write and index our advanced queries in a way that performed complex distance calculations for individual records. The distance functionality was a key portion of our application's functionality, and while we could outsource many of our calculations to the frontend or somewhere else, doing it in the query was an interesting experience. One of the main lessons we learned from these advanced queries is that functions or procedures should be used when possible. Many of our queries were dependent on a similar distance calculation and being able to abstract it to something like a stored procedure or function makes things much easier to manage when query readability is a concern. Indexing these queries also was a tedious task as we had to consider multiple indexing combinations and how they would affect performance, but ultimately found that an indexing scheme akin to spatial hashing proved effective in reducing query costs.

David Liang: One significant technical challenge we encountered involved managing frontend-backend interoperability while trying to keep the ability to scale and be responsive. While integrating the heatmap calculations and congestion score mapping features that required querying the backend, we quickly realized that handling asynchronous data is quite difficult in React. We ended up using React Query to manage cache invalidation and invalidating state. This also fixed another major issue that arose from synchronizing real-time data updates with the backend while avoiding too many redundant API calls, which caused noticeable performance bottlenecks during development. Abstracting the backend knowledge for the frontend consumption needed careful maintenance of our MSVC design pattern, especially with the amount of nested queries we were handling. Future teams should prioritize design clear API contracts early on and consider using tools that can provide caching layers or background workers to decouple the model logic, business logic, and so on for efficiency and especially maintainability.

Rajas Gandhi: One technical challenge we encountered was ensuring that data passed through efficiently between different pages in our React app. For example, passing the congestion score query results to the map. The way we did it originally worked, but was a little cumbersome and not very efficient. Instead, we used function props to send the data between the backend query processing result and the map page. Future teams should prioritize efficient data flow to ensure a clean codebase and application.

Michael Ko: One significant technical challenge we encountered was errors on the backend part. We had some trouble implementing error handling making it difficult to diagnose and debug the issues. For the query to find the nearby EV station with the plug types, when we were checking it in the front-end, we initially got an error that gave a generic internal error response which forced us to trace through the entire query to identify the problem manually. There could've been an error with an invalid input, during the haversine distance calculation, or while it was joining the tables. For the future team, they should try to implement specific error types for the errors that are occurring and structured error logging with the complex join tables and stored procedures. Perhaps, they can add automated error monitoring where it detects the error and alerts the team to prevent them from manually digging to find the problem.

Changes between Final Version & Proposal

- Are there other things that changed comparing the final application with the original proposal?

None not discussed previously.

Future Work

- Describe future work that you think, other than the interface, that the application can improve on

An interesting direction for future work is potential integration with some kind of NoSQL database like Neo4j. As discussed in class, one of the main limitations that SQL faces is that we desire normalization across the database for consistency, but this often comes at the expense of performance due to the large number of intermediate joins required. For example, if we wanted to link all of a user's vehicles to a set of compatible plug instances, we would have the following join order: User > OwnsEV > ElectricVehicle > PlugInstance > HasPlugs > EVStation. This sequence of joins is expensive both from a database perspective and from a development perspective as we must maintain all these relations. If we chose to go with a NoSQL approach, we could easily avoid many of these intermediate joins since the graph database structure allows for direct linkage of many different collections. We could go from User > ElectricVehicle > PlugInstance > EVStation which avoids many of the intermediate tables we would otherwise require.

Additionally for the backend specifically, there is much more we can do to refactor the codebase and make it more extensible. Things like basic CRUD operations were abstracted to some extent, but a significant amount of time was involved in adding the basic CRUD functionality to tables as needed. Instead, we could pull the table schemas from the database at startup and use those local table representations to define things like primary keys for the backend in real time. An example of this would be our generic update query function which requires the primary key of each table so we only update a single record. This primary key is defined manually as a string each time it is used, but we could instead use a theoretical Table object which would have the primary keys already defined which would enable an additional layer of abstraction.

Another area for improvement is error handling on the backend. While our application works well for the intended use case, we didn't spend enough time ensuring our code was robust to all possible scenarios, so more verbose logging and meaningful error messages would improve the development process should this project continue.

There is a whole laundry list of other improvements that could be made in the future that would take too long to fully write out, but we will list a few of them for the reader's pleasure.

- Granular trip planning with intermediate stops
- View attractions near EV stations
- More real-time traffic information
- Proper user login (no more deleting anyone's EVs!)
- Add user-specific favorite stops
- Real-time estimation of wait times for PlugInstances based on traffic

- Modification of in_use field for PlugInstance to a status field with the addition of a StatusCode table for additional information about whether a plug is in use, under repair, deprecated, etc.
- Images for the different types of EVs
- Develop a separate UI for analytics to let companies view statistics about EV ownership (Query 3)
- Add text-based location entry as opposed to dragging markers around the map.
- Refactor all of the backend to use standard data models (with pydantic or something)
- Refactor all of the frontend to use standard data models (instead of three different paradigms of validating objects)
- Refactor the backend query management to use a queue + connection reuse

Division of Labor

- Describe the final division of labor and how well you managed teamwork.

Our division of labor deviated from our original proposed roles, mainly because everyone had different strengths and weaknesses so we had to adapt as the project progressed. We often had disjoint schedules but would try to meet somewhat regularly to work on major stages of the project or at least have short discussions about the current state of the project before and after class.

Original Division of Labor (From proposal)

Nicholas Keriotis: Work on data pre-processing, database consistency, heatmap calculations and backend query support.
David Liang: Work on frontend, backend interoperability; External API interfacing; Maintain design pattern (MSVC)
Michael Ko: Work on the UI interfaces, frontend backend integration, backend authentication and authorization (passwords, 2-factor authentication)
Rajas Gandhi: Work on database management, advanced SQL queries, data post-processing

Actual Division of Labor

Nicholas Keriotis: As team captain, I handled a variety of tasks throughout the project like helping write the reports but primarily worked on backend parts of the project since frontend development is something I've never done before. I developed major backend features like the data preprocessing, database schema, advanced queries/indexes, and the backend service itself. I also did the heatmap calculations and a *tiny* bit of frontend.

David Liang: As the frontend lead, I developed an intuitive user interface with a responsive layout. This included managing data fetching and mutation, implementing the frontend portion of the heatmap to visualize traffic data, and ensuring frontend-backend interoperability. Along with that, I implemented the shadow of a terrible MSVC design pattern that helped with architecting the project and streamlined adding new features.

Rajas Gandhi: As a member of the frontend team, I developed various UI elements needed for the functionality of our project. Additionally, I implemented Q3 into the frontend by creating a new interface to represent the CongestionScore and adding additional elements to the frontend as needed. I worked with David to make sure that the code was up to par and features were implemented as needed.

Michael Ko: As a member of the backend team, I supported Nicholas with the backend developments. I contributed to the frontend-backend interoperability making sure all the backend components were working correctly and could be implemented on the frontend. I worked with Nicholas on the advanced queries and indexes along with the query performance for each and made sure there weren't any errors when converting from the backend to the frontend.

Final Table Schema with Checks

Our finalized table schema is shown below with additional notes for the reasoning behind given checks on a per-table basis. We will try to only mention the major changes for each table (if any) from our stage 2/3 versions.

EVStation

```sql
CREATE TABLE `EVStation` (
  `station_id` int NOT NULL AUTO_INCREMENT,
  `name` varchar(100) DEFAULT NULL,
  `latitude` decimal(8,6) DEFAULT NULL,
  `longitude` decimal(9,6) DEFAULT NULL,
  `state` varchar(2) DEFAULT NULL,
  `zip` varchar(10) DEFAULT NULL,
  `city` varchar(100) DEFAULT NULL,
  `address` varchar(100) DEFAULT NULL,
  `phone` varchar(100) DEFAULT NULL,
  PRIMARY KEY (`station_id`),
  KEY `idx_EVStation_lat` (`latitude`),
  KEY `idx_EVStation_long` (`longitude`),
  KEY `idx_EVStation_city` (`city`),
  CONSTRAINT `EVStation_chk_1` CHECK ((`latitude` between -(90) and 90)),
  CONSTRAINT `EVStation_chk_2` CHECK ((`longitude` between -(180) and 180)),
  CONSTRAINT `EVStation_chk_3` CHECK ((`state` in
(_utf8mb4'CA',_utf8mb4'VT',_utf8mb4'WA',_utf8mb4'OR',_utf8mb4'IL',_utf8mb4'I
D',_utf8mb4'FL',_utf8mb4'WI',_utf8mb4'IA',_utf8mb4'AZ',_utf8mb4'NJ',_utf8mb4
'TX',_utf8mb4'SC',_utf8mb4'CT',_utf8mb4'OH',_utf8mb4'WV',_utf8mb4'MO',_utf8m
b4'UT',_utf8mb4'KS',_utf8mb4'MA',_utf8mb4'CO',_utf8mb4'MI',_utf8mb4'LA',_utf
8mb4'NC',_utf8mb4'VA',_utf8mb4'TN',_utf8mb4'AL',_utf8mb4'GA',_utf8mb4'HI',_u
tf8mb4'MD',_utf8mb4'MN',_utf8mb4'NV',_utf8mb4'AR',_utf8mb4'RI',_utf8mb4'PA',
_utf8mb4'OK',_utf8mb4'DC',_utf8mb4'NY',_utf8mb4'ME',_utf8mb4'NH',_utf8mb4'KY
',_utf8mb4'NE',_utf8mb4'MS',_utf8mb4'SD',_utf8mb4'DE',_utf8mb4'IN',_utf8mb4'
NM',_utf8mb4'MT',_utf8mb4'ND',_utf8mb4'WY',_utf8mb4'AK',_utf8mb4'PR')))
)
```

- For the PK, we use autoincrement to ensure we don't get any null values. We maintain the same indexing and FK constraints, but we also ensure that any station latitudes and longitudes are valid, as well as ensuring the state is also valid.

EVStationDistances

```
EVStationDistances | CREATE TABLE `EVStationDistances` (
  `latitude` decimal(8,6) DEFAULT NULL,
  `longitude` decimal(9,6) DEFAULT NULL,
  `ev_station_id` int DEFAULT NULL,
  `distance_km` float DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

This is a temporary table used for our stored procedure.

ElectricVehicle

```
ElectricVehicle | CREATE TABLE `ElectricVehicle` (
  `ev_id` int NOT NULL AUTO_INCREMENT,
  `make` varchar(100) DEFAULT NULL,
  `model` varchar(100) DEFAULT NULL,
  `plug_type` int DEFAULT NULL,
  `range_km` double DEFAULT NULL,
  `battery_capacity` double DEFAULT NULL,
  PRIMARY KEY (`ev_id`),
  KEY `ElectricVehicle_ibfk_1` (`plug_type`),
  CONSTRAINT `ElectricVehicle_ibfk_1` FOREIGN KEY (`plug_type`) REFERENCES
`PlugType` (`type_id`)
) ENGINE=InnoDB AUTO_INCREMENT=121 DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci
```

The only change here is that we removed the ON DELETE CASCADE functionality from the FK constraint to PlugType to replace it with our own stored procedure discussed later.

HasPlugs

```
CREATE TABLE `HasPlugs` (
  `station_id` int NOT NULL,
  `instance_id` int NOT NULL,
  PRIMARY KEY (`station_id`,`instance_id`),
  KEY `HasPlugs_ibfk_2` (`instance_id`),
  CONSTRAINT `HasPlugs_ibfk_1` FOREIGN KEY (`station_id`) REFERENCES
`EVStation` (`station_id`) ON DELETE CASCADE,
  CONSTRAINT `HasPlugs_ibfk_2` FOREIGN KEY (`instance_id`) REFERENCES
`PlugInstance` (`instance_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

The only change here is removing the ON DELETE CASCADE functionality from the PlugInstance table. While this does mean that we can't directly delete a PlugInstance without an additional trigger, we believe this to be an acceptable trade, as we could easily expand on the functionality of our application by not needing to delete from the PlugInstance table. Instead, we could replace the in_use field to a status_code field which would allow more diverse behavior and allow us to keep a record of all previous plugs and reactivate them if needed.

HourVolumeData

```
CREATE TABLE `HourVolumeData` (
  `volume_id` int DEFAULT NULL,
  `month_record` int NOT NULL,
  `year_record` int NOT NULL,
  `day_of_week` int NOT NULL,
  `hour` int NOT NULL,
  `volume` double DEFAULT NULL,
  KEY `HourVolumeData_ibfk_1` (`volume_id`),
  CONSTRAINT `HourVolumeData_ibfk_1` FOREIGN KEY (`volume_id`) REFERENCES
`TrafficVolume` (`volume_id`),
  CONSTRAINT `HourVolumeData_chk_1` CHECK ((`month_record` between 1 and
12)),
  CONSTRAINT `HourVolumeData_chk_2` CHECK ((`day_of_week` between 1 and 7)),
  CONSTRAINT `HourVolumeData_chk_3` CHECK ((`hour` between 0 and 23))
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

No major changes from the initial version. Checks include validating FK constraints and ensuring times are valid for dates and times.

OwnsEV

```
CREATE TABLE `OwnsEV` (
  `user_id` int NOT NULL,
  `ev_id` int NOT NULL,
  PRIMARY KEY (`user_id`,`ev_id`),
  KEY `OwnsEV_ibfk_2` (`ev_id`),
  CONSTRAINT `OwnsEV_ibfk_1` FOREIGN KEY (`user_id`) REFERENCES `User`
(`user_id`) ON DELETE CASCADE,
  CONSTRAINT `OwnsEV_ibfk_2` FOREIGN KEY (`ev_id`) REFERENCES
`ElectricVehicle` (`ev_id`) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

No changes.

PlugInstance

```
CREATE TABLE `PlugInstance` (
  `instance_id` int NOT NULL AUTO_INCREMENT,
  `type_id` int DEFAULT NULL,
  `power_output` double DEFAULT NULL,
  `in_use` tinyint(1) DEFAULT NULL,
  `base_price` decimal(4,2) DEFAULT NULL,
  `usage_price` decimal(4,2) DEFAULT NULL,
  PRIMARY KEY (`instance_id`),
  KEY `PlugInstance_ibfk_1` (`type_id`),
  CONSTRAINT `PlugInstance_ibfk_1` FOREIGN KEY (`type_id`) REFERENCES
`PlugType` (`type_id`)
) ENGINE=InnoDB AUTO_INCREMENT=166670 DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci
```

Here we remove the ON DELETE CASCADE constraint on PlugType to be replaced with our trigger, other than that, no changes have been made.

PlugType

```
CREATE TABLE `PlugType` (
  `type_id` int NOT NULL AUTO_INCREMENT,
  `type_name` varchar(100) DEFAULT NULL,
  PRIMARY KEY (`type_id`)
) ENGINE=InnoDB AUTO_INCREMENT=1000 DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci
```

No changes.

TrafficStation

```
CREATE TABLE `TrafficStation` (
  `state_code` varchar(2) NOT NULL,
  `station_id` varchar(6) NOT NULL,
  `latitude` decimal(8,6) DEFAULT NULL,
  `longitude` decimal(9,6) DEFAULT NULL,
  PRIMARY KEY (`state_code`,`station_id`),
  KEY `idx_TrafficStation_lat` (`latitude`),
  KEY `idx_TrafficStation_long` (`longitude`),
  CONSTRAINT `TrafficStation_chk_1` CHECK ((`latitude` between -(90) and 
90)),
  CONSTRAINT `TrafficStation_chk_2` CHECK ((`longitude` between -(180) and 
180))
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

Included additional checks to ensure valid latitude and longitude values (the DoT somehow had some invalid ones). We don't check the state codes since these are defined by the US DoT since they provided the data source.

TrafficStationDistances

```
CREATE TABLE `TrafficStationDistances` (
  `latitude` decimal(8,6) DEFAULT NULL,
  `longitude` decimal(9,6) DEFAULT NULL,
  `station_id` int DEFAULT NULL,
  `state_code` varchar(2) DEFAULT NULL,
  `distance_km` float DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

Utility table for our stored procedure

TrafficVolume

```
CREATE TABLE `TrafficVolume` (
  `volume_id` int NOT NULL AUTO_INCREMENT,
  `state_code` varchar(2) DEFAULT NULL,
  `station_id` varchar(6) DEFAULT NULL,
  PRIMARY KEY (`volume_id`),
  KEY `TrafficVolume_ibfk_1` (`state_code`,`station_id`),
  CONSTRAINT `TrafficVolume_ibfk_1` FOREIGN KEY (`state_code`, `station_id`) 
REFERENCES `TrafficStation` (`state_code`, `station_id`)
) ENGINE=InnoDB AUTO_INCREMENT=7016 DEFAULT CHARSET=utf8mb4 
COLLATE=utf8mb4_0900_ai_ci
```

No changes.

User

```
CREATE TABLE `User` (
  `user_id` int NOT NULL AUTO_INCREMENT,
  `username` varchar(100) NOT NULL,
  `email` varchar(100) NOT NULL,
  `password` char(32) NOT NULL,
  `ssn` varchar(11) DEFAULT NULL,
  `address` varchar(100) DEFAULT NULL,
  `state` varchar(2) DEFAULT NULL,
  `city` varchar(100) DEFAULT NULL,
  `zip` varchar(10) DEFAULT NULL,
  `first_name` varchar(100) DEFAULT NULL,
  `last_name` varchar(100) DEFAULT NULL,
  `middle_initial` varchar(1) DEFAULT NULL,
  `creation_date` timestamp NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`user_id`)
) ENGINE=InnoDB AUTO_INCREMENT=10003 DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci
```

No changes.

Triggers

```
DELIMITER //
CREATE TRIGGER PlugTypeCascadeReplacement
BEFORE DELETE ON PlugType
FOR EACH ROW
BEGIN
      DELETE FROM HasPlugs
      WHERE instance_id IN (
      SELECT instance_id
      FROM PlugInstance
      WHERE type_id = OLD.type_id
      );

      DELETE FROM PlugInstance
      WHERE type_id = OLD.type_id;

      DELETE FROM OwnsEV
      WHERE ev_id IN (
      SELECT ev_id
      FROM ElectricVehicle
      WHERE plug_type = OLD.type_id
      );

      DELETE FROM ElectricVehicle
      WHERE plug_type = OLD.type_id;
END;
//
DELIMITER ;
```

Deciding on an effective trigger for our particular use case was difficult, as our application mainly serves to allow users to interact with raw data in a meaningful way. We don't often need to create records in complex ways, and our database is well designed enough that it is extremely unlikely that the database reaches an invalid state, so we instead decided to opt for replacing existing SQL functionality for our own version. This trigger effectively replaces the `ON DELETE CASCADE` functionality implemented in the PlugType table. Whenever a PlugType is deleted, instead of letting the database handle everything (as it should), we implement this functionality ourselves by manually finding and deleting any records with the associated PlugType in an order that doesn't break other FK constraints. While impractical and an obvious misstep in the mind of any sane developer, we decided to use this as it would satisfy one of our trigger requirements without being completely trivial.

Stored Procedure

```sql
DELIMITER //
CREATE PROCEDURE HaversineDistance (
      IN myLat DECIMAL(8,6),
      IN myLong DECIMAL(9,6),
      IN distance_threshold FLOAT,
      IN targetTable VARCHAR(20)
)
BEGIN

      CASE
      WHEN targetTable = 'EVStation' THEN
      DROP TABLE IF EXISTS EVStationDistances;
      -- Create TEMPORARY table for connection specific results
      CREATE TEMPORARY TABLE EVStationDistances(
            latitude DECIMAL(8,6),
            longitude DECIMAL(9,6),
            ev_station_id INT,
            distance_km FLOAT
      );

      INSERT INTO EVStationDistances
      SELECT
            latitude,
            longitude,
            station_id AS ev_station_id,
            ROUND(
                  (
                  6371 * 2 * ASIN(
                  SQRT(
                        POWER(SIN(RADIANS((latitude - myLat) / 2)), 2) +
                        COS(RADIANS(myLat)) *
                        COS(RADIANS(latitude)) *
                        POWER(SIN(RADIANS(longitude - myLong) / 2), 2)
                  )
                  )
            ), 3) AS distance_km
      FROM EVStation
      WHERE
            latitude
            BETWEEN myLat - (distance_threshold/111.0)
```

```
                  AND myLat + (distance_threshold/111.0)
            AND
            longitude
            BETWEEN myLong - (distance_threshold/(111.0 *
COS(RADIANS(myLat)) ))
                  AND myLong + (distance_threshold/(111.0 *
COS(RADIANS(myLat)) ))
      HAVING distance_km <= distance_threshold;

      WHEN targetTable = 'TrafficStation' THEN
      DROP TABLE IF EXISTS TrafficStationDistances;
      CREATE TEMPORARY TABLE TrafficStationDistances(
            latitude DECIMAL(8,6),
            longitude DECIMAL(9,6),
            station_id VARCHAR(100),
            state_code VARCHAR(2),
            distance_km FLOAT
      );
      INSERT INTO TrafficStationDistances
      SELECT
            latitude,
            longitude,
            station_id,
            state_code,
            ROUND(
                  (
                  6371 * 2 * ASIN(
                  SQRT(
                        POWER(SIN(RADIANS((latitude - myLat) / 2)), 2) +
                        COS(RADIANS(myLat)) *
                        COS(RADIANS(latitude)) *
                        POWER(SIN(RADIANS(longitude - myLong) / 2), 2)
                  )
                  )
            ), 3) AS distance_km
      FROM TrafficStation
      WHERE
            latitude
            BETWEEN myLat - (distance_threshold/111.0)
                  AND myLat + (distance_threshold/111.0)
            AND
            longitude
            BETWEEN myLong - (distance_threshold/(111.0 *
COS(RADIANS(myLat)) ))
                  AND myLong + (distance_threshold/(111.0 *
COS(RADIANS(myLat)) ))
      HAVING distance_km <= distance_threshold;
      END CASE;
END //
DELIMITER ;
```

```
CALL HaversineDistance(34.040539, -118.271387, 40, 'EVStation');
```

- An example of how we can call this procedure

For our stored procedure, we knew we wanted to implement our Haversine distance function as it was something commonly used in almost all of our advanced queries and turning it into a reusable piece of code was an obvious solution, but the stage 3 requirements wouldn't allow for this, so we had to make a monstrous query that held this duplicate code into a much more streamlined version (For example, one of our queries went from 47 lines to only 12 after implementing this!). This stored procedure takes in the same parameters as it did when used in our advanced queries, taking a latitude, longitude, distance_threshold, and an additional table argument. Since we compute distances from both EVStations and TrafficStations, we needed to be able to pull from 2 different tables, and having this additional table argument lets us specify which table we want to pull from. Since the results we want to return are also different, we also added 2 additional temporary tables, `EVStationDistances` and `TrafficStationDistances` which both hold the distances and the associated data with either the EVStation or TrafficStation respectively. We use the TEMPORARY keyword here since these are connection specific and modifying them won't be blocking operations for other connections trying to perform the same operations in parallel. These tables are also dropped automatically whenever a connection closes.

Transaction & Isolation Level Documentation

For transactions and isolation levels, we don't implement these at the database level, instead opting to move it to the higher level of the backend. Much of our very high level transaction management is handled in `db_connection.py` where we always perform our queries inside a transaction. Should the underlying query fail for any reason, we always abort the transaction before returning an error code to ensure that we don't corrupt the database.

https://github.com/cs411-alawini/fa24-cs411-team087-2AndAHalfAsians/blob/c3919ff90532f8ddcf1c73809064ff6b54b74b79/app/backend/evproject-backend/src/db_connection.py#L73-L101

- Link to the primary transaction management

Our isolation levels needed to be controlled in a more granular way, as depending on what we were doing in the backend, we may want something to be SERIALIZABLE or READ_UNCOMMITTED. We achieved this by calling the appropriate query to set the transaction level before running the main query. After the main operations are done, we then ultimately close the cursor and connection which will restore the isolation level to its default value for the next connection. For many of our INSERT queries, we didn't need any guarantees on the stability of those records since they wouldn't affect anything else, and even if they did, it would be extremely rare, so these were often set to READ_COMMITTED to ensure things like referenced relations actually existed before adding the record. Other things like updates to the User table however are much more sensitive, so these often involved the SERIALIZABLE isolation level to ensure we didn't pull incorrect data or break other concurrent transactions. Overall, many of our backend API functions are fairly simple which allows us to be very granular in how we execute our queries. Each individual backend call gets its own transaction which ensures that everything related to that transaction occurs in an atomic fashion which greatly simplifies the overall logic of query execution, as we don't need to make multiple calls to different internal queries, each with their own isolation level.

https://github.com/cs411-alawini/fa24-cs411-team087-2AndAHalfAsians/blob/c3919ff90532f8ddcf1c73809064ff6b54b74b79/app/backend/evproject-backend/src/routers/customRouter.py#L149

- Example of setting the transaction level before a query, this is one example but we do this in pretty much every query