• A short description of the final project, and what it accomplished

In our final project, We built a tourist attraction application, where users can login/signup, view locations and events, make bookings and write reviews. We also added features for filtering the locations/ events, like by rating and by category for Locations , and for events, being able to search up events, and select what attributes of the events they want to display. Users can visit their profile, and see their user info, be able to change their password, see Reviews and bookings they've made, and delete them as they wish.

.

• A description of how your final schema differed from the schema you turned in. If the final schema differed, explain why.

Our schema differed a little from milestone 2. In total, we deleted 3 tables, Time_of_Booking, Event_Time and Picture. We felt like Time_of_Booking and Event_Time made the schema really repetitive and made the frontend implementation a lot more difficult, so we just removed them. For the Picture table, we found it to not really have any use in our database schema as we represented pictures with Blobs which didn't really contain any useful data, so we just removed it as well. We also added 2 attributes to locations, locationID and locationName, locationName we felt was just necessary to add to describe a location, and locationID made some of the frontend routing a little less complex as our key for location is postalCode and Address.

• A list of all SQL queries used to satisfy the rubric items and where each query can be found in the code (file name and line number(s)).

Insert: userService.js at line 52
Update: userService.js at line 75, and 171
Delete: bookingService.js at line 39
Selection: eventDetailsService.js at line 44
Projection: eventDetailsService.js at line 192
Join: locationService.js at line 157

• For SQL queries 2.1.7 through 2.1.10 inclusive, include a copy of your SQL query and a maximum of 1-2 sentences describing what that query does.

Aggregation With Group By: ratingService.js at line 26
Query:
SELECT postalCode, address, AVG(score) as avg_rating, COUNT(*) as num_ratings from rating GROUP BY postalCode, address
Description:
This query gets the average rating and number of ratings for each location, displayed on the locations page.

Aggregation With Having: ratingService.js at line 72
Query:
SELECT postalCode, address, AVG(score) as avg_rating, COUNT(*) as num_ratings from rating GROUP BY postalCode, address HAVING AVG(score) > :num
Description:
This query gets the average rating and number of ratings for each location that has an average score > the inputted number, allowing users to filter for locations above a certain rating

Nested Aggregation With Group By: ratingService.js at line 47
Query:
SELECT postalCode, address, AVG(score) as avg_rating, COUNT(*) as num_ratings from rating GROUP BY postalCode, address HAVING AVG(score) > (SELECT AVG(score) FROM rating
Description:
This query gets the average rating and number of ratings for each location that has an average rating above the average rating of all locations, allowing users to filter for locations that have an above average rating

Division: locationService.js at line 304
Query:
SELECT L.postalCode, L.address FROM locations L WHERE NOT EXISTS (SELECT T.catName FROM category T WHERE T.catName IN (:catNames) MINUS SELECT C.catName FROM categorizes C WHERE C.postalCode = L.postalCode AND C.address = L.address)
Description:
This query gets all locations that are categorized by the inputted category.