Technical Report

Dataset analysis

We had 3 data sources:

- **Taxi zones**: Was a zip folder, but none of its files were readable and had any real data. So we skipped this one.
- **Yellow trip data:** showed the trips that took place
- **Taxi zone look-up:** showed us the areas across the United States where the service is operational.

Challenges

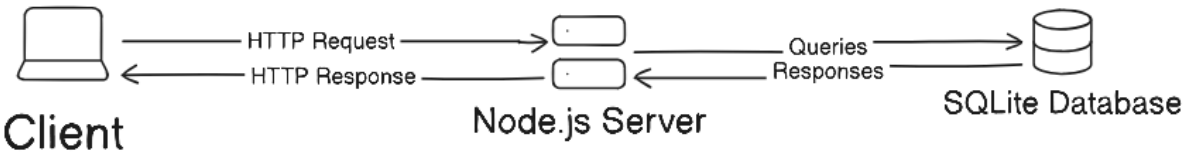
- The first challenge was to sort this huge mess of data and try to find some that are truly insightful, and figure out what kind of app we can build with it.
- One file of Yello Trip Data had an excess of over 7 million rows. We were not expecting that at all and tried to push it for the first time. But we ended up narrowing it down to just 1000 rows and cleaning that, and making it possible to push on GitHub because originally it occupied over 1 GB of disk space, and it's too big to push.

Assumptions made during data clean-up

- Not all columns are needed. Especially in yellow trip data, like rate code id and so we ended up ignoring them to meet our application's needs.
- We needed to round off some values that had a lot of decimal points to just 2 decimal points
- Empty rows were removed

System architecture

- We have a Node.js HTTP server handling requests to and from the client
- We are using a lightweight SQLite database for data storage
- For the web client, we used vanilla HTML, CSS, and JavaScript.



Stack choices

- We used SQLite because it's lightweight and doesn't require a database server like Postgresql and MySQL do. It enabled an easy setup.
- We used Node.js because most of the team members are more comfortable with JavaScript than Python, which enabled us to iterate faster.

Trade offs

- By moving away from PostgreSQL and MySQL, we lost the option of using something like Docker to make it easy for all of us to set up the database with the same settings
- Sometimes, depending on the Node.js version, installing SQLite would work or not. And so the team members had to upgrade their Node.js version or at least bump up the version ofthe SQLite driver, which was a bit annoying but worth it for the simplicity SQLite gave us.

Algorithm

Algorithm Documentation

This document explains the custom sorting, filtering, and searching logic implemented in the Trips Dashboard (client/assets/Js/trips.js).

The application manages client-side data processing for the trips table. When the user interacts with any filter or sort control, the `applyFiltersAndSort()` function is triggered, which executes a two-step pipeline:

1. Filtering & Searching (`matchesFilters`)

The `matchesFilters` function works by iterating through the cached dataset (`tripsCache`) and applying multiple checks to determine whether each trip should be included in the final results.

1. Filtering Logic

For every trip in the dataset, four conditional checks are applied:

- **Status:**
  The trip status must match the selected status using a case-insensitive comparison (for example, `"Completed"`).

- **Fare:**
  The trip's `total_amount` must fall within a specified numeric range (for example, 5000–10000).

- **Distance:**
  The trip's `trip_distance` must fall within a specified numeric range (for example,

5–10 miles).

2. Search Logic

The search functionality is built directly into the filtering process and works as follows:

- The search term is converted to lowercase.
- A case-insensitive substring search is performed.
- The system checks whether the search term exists in either:
    - Pickup Zone, or
    - Dropoff Zone
- If the search term is found in at least one of these fields, the trip is included (as long as it also passes all other filters).

3. Sorting (sortTripsArray):

- Takes the filtered result and sorts it based on the selected criteria.
- Standard JavaScript `Array.prototype.sort()` is used with custom comparator functions for:
    - **ID** (chronological order).
    - **Fare Amount** (numeric value).
    - **Trip Distance** (numeric value).

---

2. Pseudo Code

```
FUNCTION applyFiltersAndSort():
    filtered_list = []

   // Step 1: Filtering
    FOR EACH trip IN tripsCache:
       // Status Filter
```

```
        IF filter.status != "all" AND trip.status != filter.status:
            CONTINUE

        // Fare Filter (Range Check)
        IF filter.fare == "0-5000" AND (trip.amount < 0 OR trip.amount > 5000):
            CONTINUE
        // ... (other fare ranges)

        // Distance Filter (Range Check)
        IF filter.distance == "0-5" AND (trip.distance < 0 OR trip.distance > 5):
            CONTINUE
        // ... (other distance ranges)

        // Location Search
        IF filter.location IS NOT EMPTY:
            search_term = filter.location.toLowerCase()
            pickup = trip.pickup_zone.toLowerCase()
            dropoff = trip.dropoff_zone.toLowerCase()

            IF NOT (pickup contains search_term OR dropoff contains search_term):
                CONTINUE

        ADD trip TO filtered_list

    // Step 2: Sorting
    SWITCH currentSort:
        CASE "recent":
            SORT filtered_list BY trip.id DESCENDING
        CASE "oldest":
            SORT filtered_list BY trip.id ASCENDING
        CASE "fare-high":
            SORT filtered_list BY trip.amount DESCENDING
        CASE "fare-low":
            SORT filtered_list BY trip.amount ASCENDING
        CASE "distance-high":
            SORT filtered_list BY trip.distance DESCENDING
        CASE "distance-low":
            SORT filtered_list BY trip.distance ASCENDING

    RETURN filtered_list
```

---

3. Complexity Analysis

Time Complexity

Filtering: O(N)

- We iterate through all N trips once.
- Inside the loop, string matching (`includes`) depends on string length K, making it O(K). Since location names are short, this is effectively constant time.

Sorting: O(M log M)

- Where M is the number of filtered items (worst case M = N).
- JavaScript's `sort()` typically uses Timsort or Merge Sort, which run in O(M log M).

Total Time Complexity: O(N log N)

- Dominated by the sorting step in the worst case where no items are filtered out.

Space Complexity

Space Complexity: O(M)

- Where M is the number of filtered items.
- We create a new array (`filtered_list`) to store the matching trips, separate from the original cache.

Reflection

Technical Challenges

- We had a lot of merge conflicts
- Integrating the API server with our client

Improvements

- We encouraged the use of PRs and reviews to reduce merge conflicts
- We made the codebase more modular to make it easy to understand its structure