Fleet Management System

Abhishek Jadhav
UBID: ajadhav8

Aishwarya Salvi
UBID: aishvary

MILESTONE I: SYNOPSYS

I. INTRODUCTION:

The main goal of this project is building a scalable and dependable database system to oversee a fleet of self-driving cars. The technology is intended to make ride booking, vehicle tracking, maintenance scheduling, and payment processing more efficient. It guarantees real-time data processing, facilitates better decision-making, and raises the general effectiveness of fleet management.

II. PROBLEM STATEMENT:

Current systems struggle with fragmented data management of autonomous fleets, causing delays and inefficiencies in handling real-time ride requests, telemetry, maintenance, and payments.

The system will:

- **Centralize Data Storage**
- **Enable Real-Time Updates**
- **Facilitate Querying**
- **Provide Analytical Insight**

III. REASONS FOR CHOOSING DATABASE OVER EXCEL

We selected a database over excel because of its capacity to manage the intricacy of structured, relational data and scale effectively for huge datasets. By enforcing restrictions and linkages, it minimizes errors and redundancy while ensuring data integrity. Multiple users can work on the data at once without encountering any issues because to the system's support for concurrent access. Role-based access restrictions in databases further improve security by guaranteeing that only individuals with permission can see or alter data. Powerful data retrieval and analysis are made possible by the ability to execute sophisticated querying in SQL, which is difficult to accomplish in Excel. Last but not least, database systems with automated backups assist prevent data loss and provide dependable recovery in the event of an outage.

IV. INTENDED USERS OF THE DATABASE

A number of important user groups are intended to be served by the fleet management database system. It will be used by fleet managers to plan maintenance, allocate trips, and keep an eye on vehicles. To schedule rides, check the progress of their trips, and make payments, customers will communicate with the system. Vehicle data will be accessed by maintenance crews in order to record repairs and determine servicing requirements. To improve operations and produce performance insights, data analysts will make use of previous data. The database will be used by payment processors for financial reporting and safe transaction processing.

V. WHO WILL ADMINISTER THE DATABASE?

Database administrators (DBAs) and data engineers will be in charge of maintaining the database, handling duties such data entry, backups, security management, performance tuning, and schema design. Additionally, data analysts will work with the database to retrieve and examine data, offering insights to help different stakeholders and business decisions.

VI. COMPANY IMPLEMENTATION

The organization used a number of crucial tactics for better fleet management as a result of the investigation. These include dynamic ride allocation, which lowers vehicle idle time and increases utilization; customer feedback integration, which uses ride data to improve service quality and user satisfaction; and predictive maintenance, which uses telemetry data to schedule vehicle servicing proactively and prevent breakdowns.

VII. WHAT HAPPENS WHEN THE PRIMARY KEY IS DELETED

With Constraints: ON DELETE CASCADE, SET NULL, or Restrict. Without Constraints, it leads to orphaned records and data inconsistency.This was main points from Milestone1.

MILESTONE II

I. PROJECT OVERVIEW

The project aims to build a normalized, query-efficient, and scalable database for a fleet and ride-sharing management system. Vehicles, batteries, telemetry, maintenance, alerts, ride requests, payments, and customers are all tracked by this system. After domain analysis and normalization, we made changes to the original schema to improve efficiency and provide a more distinct division of responsibilities.

II. FINAL SCHEMA DESIGN

<u>Schema 1</u>: Fleet Schema
Tables: vehicles, batteries, maintenance, trips
<u>Schema 2</u>: User_Ride Schema
Tables: customers, ride_requests, payments
<u>Schema 3</u>: Operational Schema
Tables: telemetry, alerts, battery_updates

III. TRANSFORMATION AND JUSTIFICATION

For greater coherence, the initial schema was restructured from constrained domains (batteries, fleet, rides) into more expansive functional domains: fleet (vehicles and batteries), operational (real-time telemetry, alerts, updates), and user_ride (payment and ride data). A 1:1 link between ride requests and journeys, telemetry and alarms being high-volume logs, and each vehicle having a single battery at a time are important presumptions. This reorganization streamlines dependencies and centralizes relevant data.

IV. NORMALIZATION

Normalization Is a process that systematically refines table structures to uphold data integrity, and it helps to minimize redundancy. Databases with poor structure frequently experience inefficiencies and update anomalies. We initially confirmed 1NF for our schema by making sure all attributes were atomic and that the tables had primary keys. We then verified 2NF by making sure there were no partial dependencies (for example, non-key values in the vehicles table, such as model or status, are totally dependent on vehichle_id). By making sure there were no transitive dependencies, we were able to validate 3NF. For example, non-prime attributes, such as charge_level in batteries, depend only on battery_id and not indirectly through another attribute.

1. BCNF Normalization and Functional Dependencies

The database schema was carefully analyzed to ensure all relations satisfy Boyce-Codd Normal Form (BCNF). The functional dependencies (FDs) and normalization status for each table are summarized below:

a)fleet.vehicles
Functional Dependencies: vehicle_id → model, status, latitude, longitude, battery_id, last_updated
Normalization Status: The table is in BCNF. The primary key fully determines all other attributes, with no partial or transitive dependencies.

b)fleet.batteries
Functional Dependencies: battery_id → capacity_kwh, health_status, charge_level, last_replacement_date, last_updated
Normalization Status: The table is in BCNF.

c)fleet.maintenance
Functional Dependencies: maintenance_id → vehicle_id, maintenance_type, scheduled_date, cost
Normalization Status: The table is in BCNF.

d)fleet.trips
Functional Dependencies: trip_id → vehicle_id, start_time, end_time, start_latitude, start_longitude, end_latitude, end_longitude, distance_km
Normalization Status: The table is in BCNF.

e)user_ride.customers
Functional Dependencies: customer_id → name, phone_number, email, default_payment_method
Normalization Status: The table is in BCNF.

f)user_ride.ride_requests
Functional Dependencies: ride_id → customer_id, vehicle_id, request_time, pickup_latitude, pickup_longitude, dropoff_latitude, dropoff_longitude, status, estimated_fare
Observations: Although (customer_id, request_time) could functionally determine ride_id, it is not a candidate key.
Normalization Status: The table is in BCNF.

g)user_ride.payments
Functional Dependencies: payment_id → ride_id, customer_id, amount, payment_method, payment_status

Normalization Status: The table is in BCNF.
operational.telemetry
Functional Dependencies: telemetry_id → vehicle_id, timestamp, speed_kmh, battery_level, latitude, longitude
Normalization Status: The table is in BCNF.

h)operational.alerts
Functional Dependencies: alert_id → vehicle_id, issue, severity, timestamp
Normalization Status: The table is in BCNF.

i)operational.battery_updates
Functional Dependencies: update_id → battery_id, vehicle_id, charge_level_before, charge_level_after, timestamp
Normalization Status: The table is in BCNF.

Thus, all relations are successfully normalized to BCNF. This design minimizes redundancy, eliminates update anomalies, and ensures high data integrity across the database.

Table 1: vehicles
Attributes: vehicle_id (PK), model, status, latitude, longitude, battery_id (FK), last_updated
Relations:

One-to-many with: fleet.maintenance, fleet.trips, operational.telemetry, operational.alerts, user_ride.ride_requests (via vehicle_id)
Many-to-one with: fleet.batteries (via battery_id)

Table 2: batteries
Attributes: battery_id (PK), capacity_kwh, health_status, charge_level, last_replacement_date, last_updated
Relations:

One-to-many with: operational.battery_updates (via battery_id)
Referenced by: fleet.vehicles (via battery_id)

Table 3: BatteryUpdates Table
Attributes: update_id (PK), battery_id (FK), vehicle_id (FK), charge_level_before, charge_level_after, timestamp
Relations:

Many-to-one with: fleet.batteries
Many-to-one with: fleet.vehicles

Table 4: Telemetry Table

Attributes: update_id (PK), battery_id (FK), vehicle_id (FK), charge_level_before, charge_level_after, timestamp
Relations:
Many-to-one with: fleet.batteries
Many-to-one with: fleet.vehicles

Table 5: Alerts table
Attributes: alert_id (PK), vehicle_id (FK), issue, severity, timestamp
Relations:

Many-to-one with: fleet.vehicles

Table 6: Maintenance Table
Attributes: maintenance_id (PK), vehicle_id (FK), maintenance_type, scheduled_date, cost
Relations:

Many-to-one with: fleet.vehicles

Table 7: Trips Table
Attributes: trip_id (PK), vehicle_id (FK), start_time, end_time, start_latitude, start_longitude, end_latitude, end_longitude, distance_km
Relations:

Many-to-one with: fleet.vehicles

Table 8: Customers Table
Attributes: customer_id (PK), name, phone_number, email, default_payment_method
Relations:

One-to-many with: user_ride.ride_requests, user_ride.payments (via customer_id)

Table 9: RideRequests Table
Attributes: ride_id (PK), customer_id (FK), vehicle_id (FK), request_time, start_latitude, start_longitude, end_latitude, end_longitude, estimated_fare, status
Relations:

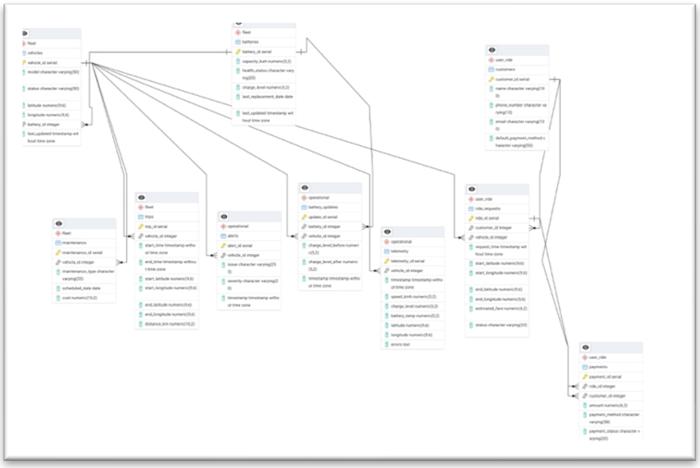
Many-to-one with: user_ride.customers
Many-to-one with: fleet.vehicles
One-to-many with: user_ride.payments (via ride_id)

Table 10: Payments Table
Attributes: payment_id (PK), ride_id (FK), customer_id (FK), amount, payment_method, payment_status
Relations:

Many-to-one with: user_ride.ride_requests
Many-to-one with: user_ride.customers

V. FINALIZED ER DIAGRAM

Based on the normalization, we made a few changes and this is how our updated ER diagram looks like:
Link to Final ER Diagram:
https://buffalo.box.com/s/56dc4ckhafenojf0krsq92alk7yclp7j



The E/R diagram represents a well-normalized fleet management database with clear relationships across the fleet, user_ride, and operational schemas. Primary and foreign key constraints ensure data consistency, such as linking vehicles to trips, maintenance, telemetry, and alerts. Domain constraints restrict values for fields like vehicle status, alert severity, and payment status to valid categories. NOT NULL and UNIQUE constraints on essential fields like customer email and phone number help maintain data quality. Cascading actions on foreign keys, like ON DELETE CASCADE and ON DELETE SET NULL, manage dependent records effectively. All things considered, these limitations promote a dependable and scalable system, guard against invalid entries, and guarantee referential integrity.

<u>Previous E/R Diagram Issues for Scale</u>
Scattered related tables like batteries and battery updates across different schemas made relationships less intuitive. Although they exist in different schemas, telemetry and alerts are closely related to vehicles. Joins across different schemas for frequent queries (e.g., battery updates for a vehicle) caused inefficiencies. Constraint enforcement was harder across schema boundaries.

<u>New E/R Diagram Advantages</u>
Logical Domain Separation:
All fleet-related entities, including vehicles, batteries, maintenance, and trips, are included in the fleet schema.
User_Ride Schema: customer interactions (profiles, requests for rides, and payments)
Operational Schema: real-time data (telemetry, alerts, battery updates)
Query Efficiency:

<u>Common queries involve intra-schema joins</u>

Vehicle performance → vehicles, batteries, maintenance
Customer history → customers, ride_requests, payments
Data Integrity:

Related entities grouped together → easier FK enforcement
Scalability:

Operational schema can scale independently, use stream-based ingestion or partitioning

VI. INDEXING

The Fleet Management System dataset contains operational data about vehicles, batteries, and customer ride requests. As the dataset grows, query performance becomes critical for real-time analytics. We identified several performance bottlenecks, particularly in queries involving multi-table joins and timestamp-based filtering.

Challenges Faced Without Indexing
Slow query execution due to full sequential scans on large tables like alerts, ride_requests etc.
Inefficient joins between vehicles, batteries, and alerts tables, leading to high memory usage.
Poor filtering performance on timestamp-based conditions (request_time, last_replacement_date).
Self-joins on ride requests were computationally expensive due to lack of optimized access paths.
To address these issues, we implemented strategic indexing on frequently queried columns, significantly improving query performance.

Indexing Added:

Fleet Schema (Batteries & Vehicles)
<u>Table:</u> fleet.batteries
<u>Index</u>: battery_id (Primary Key already indexed)
<u>Reason</u>: Speeds up joins with vehicles and alerts.

<u>Table</u>: fleet.vehicles
<u>Index</u>: battery_id (Foreign Key)
<u>Reason</u>: Optimizes joins between vehicles and batteries.

2. Operational Schema (Alerts)
Table: operational.alerts
Composite Index: (vehicle_id, timestamp)
Reason: Accelerates filtering on timestamp ranges and Improves join performance with vehicles.

User Ride Schema (Customers & Ride Requests)
Table: user_ride.customers
Index: customer_id (Primary Key already indexed)

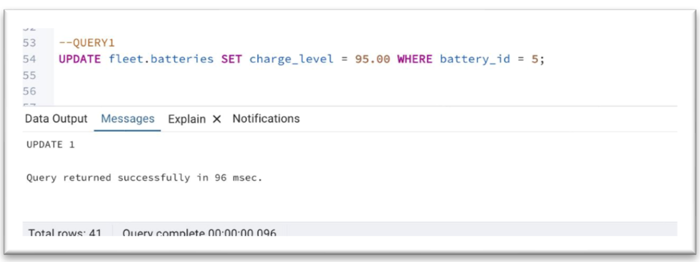
Table: user_ride.ride_requests
Composite Index: (customer_id, request_time)
Reason: Optimizes self-joins for detecting idle periods between rides.
Enables efficient range queries on request_time.

VII. SQL QUERIES

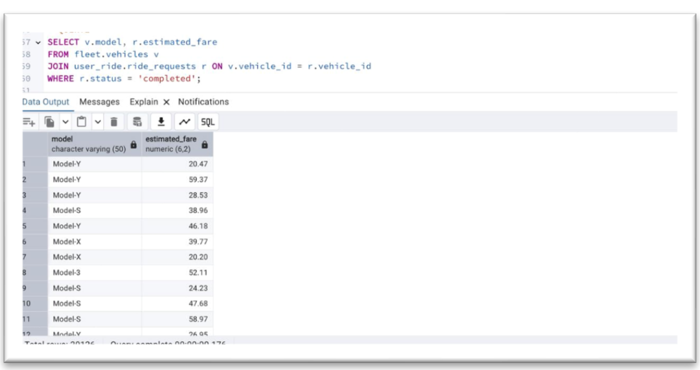
Link to SQL Queries : https://buffalo.box.com/s/yxpkpwrzzpxysg89e5mqeikee69m6ww2

The queries that we implemented are:

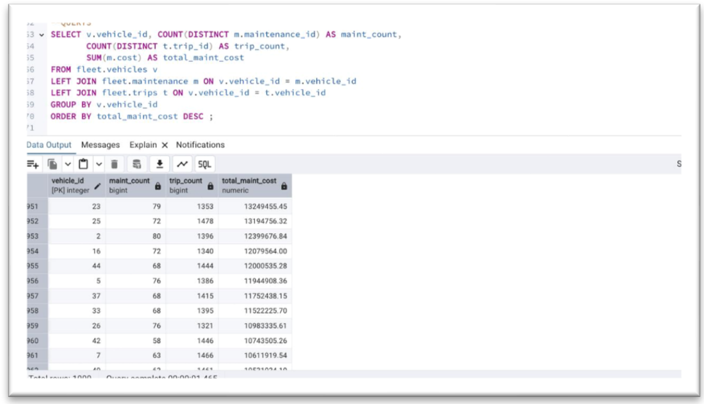
1) Update query to update charge percent of certain battery



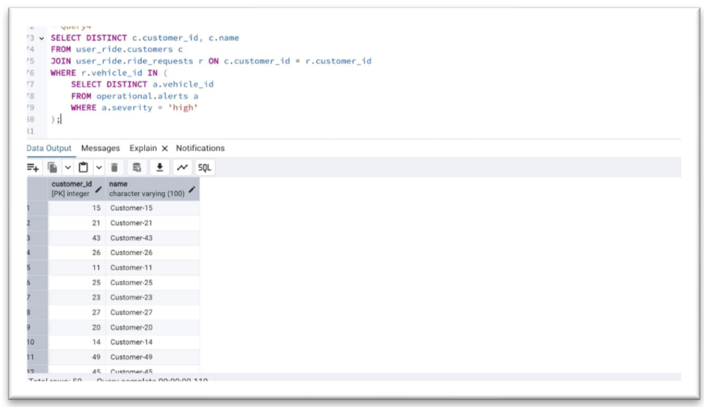
2) Query to filter completed rides with focus on model_type and its fares that it accumulated.



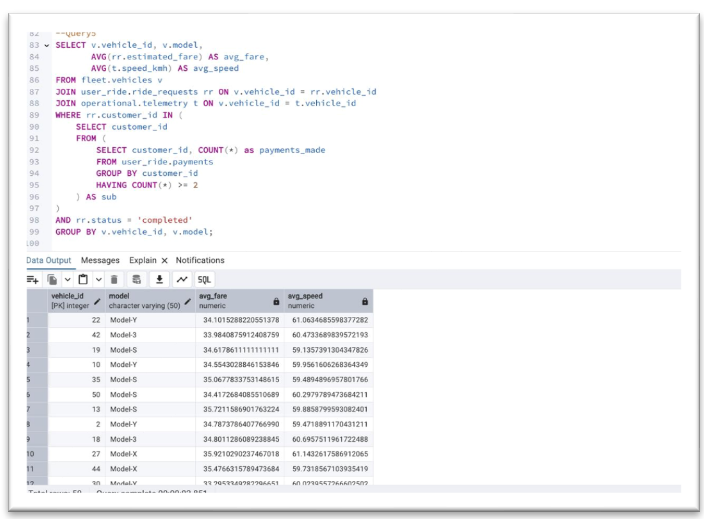
3) Maintenance Cost Summary per Vehicle with Trip Data



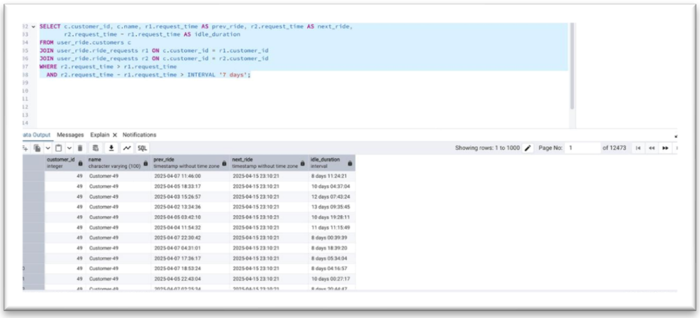
4) Query to List customers who only took trips in vehicles that have ever triggered a "high" severity alert.



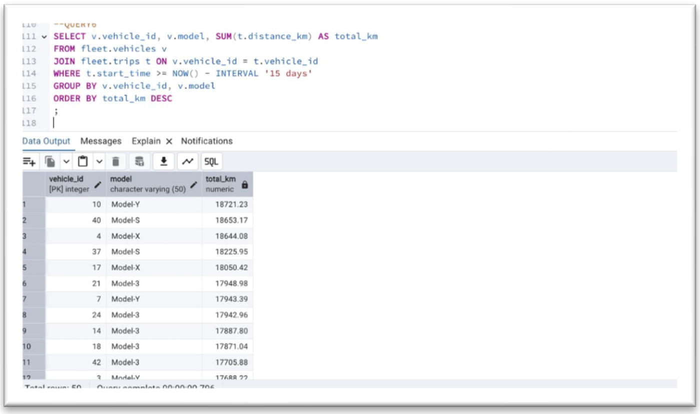
5) Query to get the average fare of completed rides requested by customers who have made at least two payments, and also average speed during those rides using telemetry.



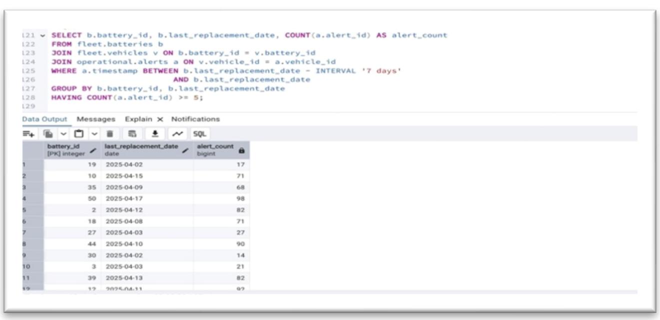
6) Active Customers with Long Idle Gaps Between Rides



7) Vehicles with Most Distance Covered in Last 15 Days

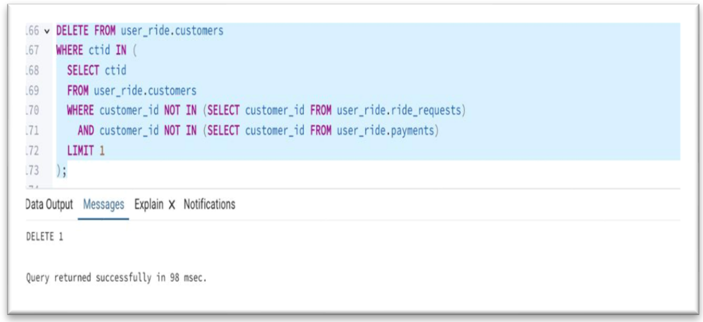


8) Battery Replacements That Happened After Frequent Alerts



9) INSERT into fleet.vehicles

10) Delete customer who never made a ride_request or payment



VIII. PROBLEMATIC QUERIES

We worked with tables containing 70,000+ rows and did encounter performance degradation as we increase data hugely. we're aware that in real-world industry settings, data volume continues to grow rapidly, which can eventually lead to slow queries and heavy sequential scans. To proactively address this, we adopted indexing strategies on frequently queried columns such as vehicle_id, customer_id, and ride_id. These indexes help the database engine locate rows faster, significantly improving query performance. By anticipating scalability challenges, we ensured that our system remains efficient and responsive as data grows.

Considering queries 5,7,8.

Performance Improvements After Indexing
https://buffalo.app.box.com/s/dpviy0h491uuhm4tnk7zpfc7ee2em2u3

Query 1: Battery Alerts After Replacement
Before Indexing:
Full sequential scans, slow filtering (~100ms+).

After Indexing:
Hash joins used instead of nested loops.
Execution time reduced to 8ms (12x faster).
Buffer usage minimized (shared hit=106).

Query 2: Customer Idle Time Analysis
Before Indexing: Heavy sequential scans, slow self-joins (~3000ms+).

After Indexing:

Parallel Index Scans used for ride_requests.
Execution time reduced to ~2300ms (despite 80,000+ rows).
I/O cost reduced due to efficient index-only scans.

IX. QUERIES:

1) SELECT b.battery_id, b.last_replacement_date,
COUNT(a.alert_id) AS alert_count
FROM fleet.batteries b
JOIN fleet.vehicles v ON b.battery_id = v.battery_id
JOIN operational.alerts a ON v.vehicle_id = a.vehicle_id
WHERE a.timestamp BETWEEN
b.last_replacement_date - INTERVAL '7 days' AND
b.last_replacement_date
GROUP BY b.battery_id, b.last_replacement_date
HAVING COUNT(a.alert_id) >= 5;

After Cost: 309.23..321.73

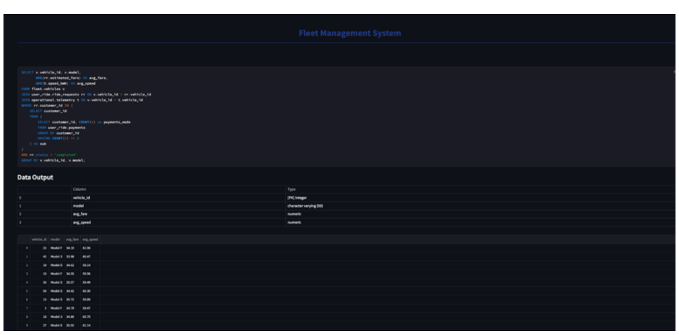
2) SELECT c.customer_id, c.name, r1.request_time AS prev_ride, r2.request_time AS next_ride, r2.request_time - r1.request_time AS idle_duration FROM user_ride.customers c
JOIN user_ride.requests r1 ON c.customer_id = r1.customer_id
JOIN user_ride.requests r2 ON c.customer_id = r2.customer_id
WHERE r2.request_time = r1.request_time + INTERVAL '7 days';

After Cost: 3035.51…15011

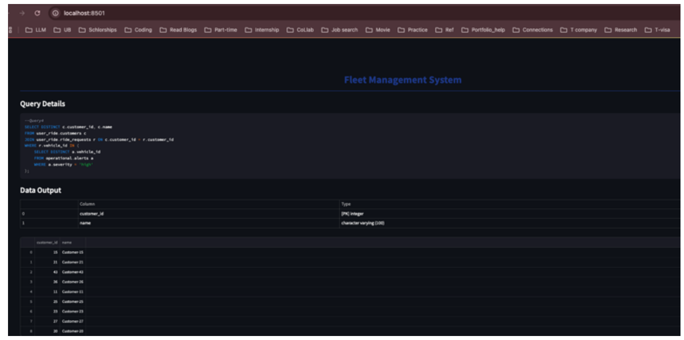
3) CREATE INDEX idx_ride_requests_customer_time
ON user_ride.ride_requests(customer_id, request_time);

X. WEBSITE

Link:https://buffalo.box.com/s/29v1eshiys8u6hcwx9frpt1pqya21vm7



https://buffalo.app.box.com/s/29v1eshiys8u6hcwx9frpt1pqya21vm7



XI. CONCLUSION

The Fleet Management System project effectively addressed the core issues of ride booking, maintenance tracking, and real-time data processing by creating a scalable and effective database solution for fleets of autonomous vehicles. The solution guarantees data integrity, quick query performance, and scalability through the use of BCNF normalization, optimal schema design, and strategic indexing. Efficiency is improved by the logical division into Fleet, User_Ride, and Operational schemas, and insightful analysis is made possible by strong SQL queries. For fleet managers, clients, and maintenance crews, this solution offers a centralized, safe, and effective platform that enhances overall operational efficiency and decision-making.

XII. CITATIONS

1)https://www.postgresql.org/docs/current/indexes.html

2) https://www.geeksforgeeks.org/how-to-design-database-for-fleet-management-systems/

3)https://www.geeksforgeeks.org/introduction-of-er-model/

4)https://www.researchgate.net/publication/298951839_Methodology_of_Introducing_Fleet_Management_System