Database Design for a Taxi hailing App

Michael Ayesa Momo

Database Tables

Customers Drivers Rides Payments Cars

Introduction

The proposed database design for a taxi hailing app would consist of several tables that store information about customers, drivers, rides, and payments.

Customers table

The Customers table would contain information about the app's customers, including their unique identifier, first name, last name, email address, and phone number.

Drivers table

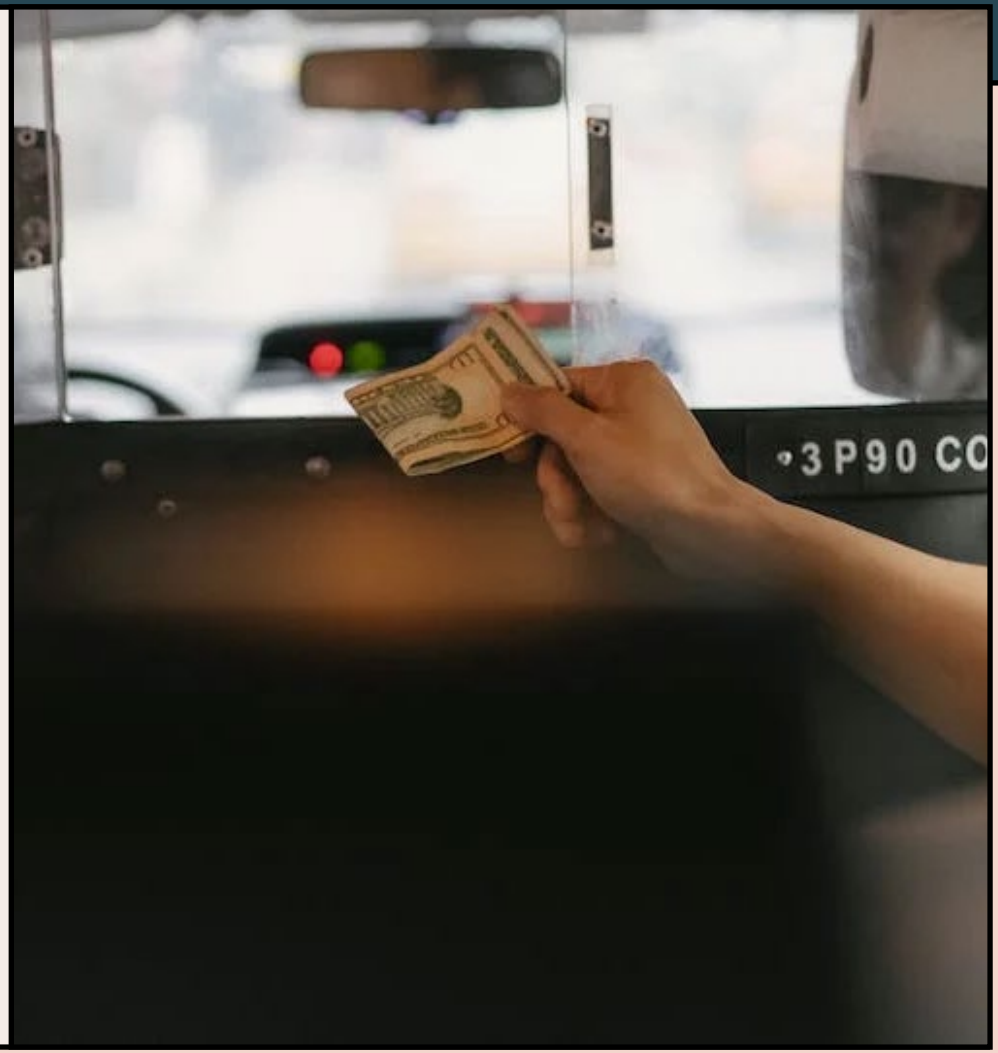
Drivers table

Drivers table

The Drivers table would contain information about the app's drivers, including their unique identifier, first name, last name, email address, phone number, car make, car model, car year, and car license plate number.

Rides table

The Rides table would contain information about each ride that is requested and completed through the app, including the ride's unique identifier, the customer who requested the ride, the driver who completed the ride, the pickup location, the drop-off location, the date and time of the ride, and the fare charged for the ride



Payments table

Payments table

The Payments table would contain information about each payment made through the app, including the payment's unique identifier, the customer who made the payment, the ride for which the payment was made, the amount paid, and the date and time of the payment

Payments table

Cars Table

Payments table

Payments table

The Car table in the normalized database design for the taxi hailing app contains information about the cars that drivers can use to provide rides to customers.
The table has a primary key "car_id" which uniquely identifies each car in the table. It also has a foreign key "driver_id" that references the Drivers table to establish a one-to-one relationship between cars and drivers.
The table has four additional columns: "make", "model", "year", and "license_plate". These columns store information about the car's make, model, year, and license plate number respectively.
By having a separate Cars table, rather than including car information directly in the Drivers table, we can avoid repeating car data for each driver that uses the same car. This allows for efficient storage of car data and makes it easy to update car information for multiple drivers at once.

Primary goals

The primary goal of the proposed database design for a taxi hailing app is to facilitate the interactions between customers and drivers within the app.

The database stores and manages data related to customers, drivers, rides, and payments, enabling the app to provide efficient and reliable transportation services. The data stored in the database can be used to match customers with available drivers, track the progress of rides, calculate fares and process payments, and maintain records of past rides and payments.

By ensuring that data is stored accurately and efficiently, the database design enables the app to provide a seamless and hassle-free experience for its users. Overall, the primary goal of the database is to support the functionality of the taxi hailing app, enabling it to provide convenient, safe, and efficient transportation services to its users.

Views

Views

- A view would be created to show the ongoing rides for a specific driver. This view would join the Rides and Drivers tables on the driver_id column and only show rows where the ride_status is ongoing.

Views sample code

```sql
CREATE VIEW driver_ongoing_rides AS
SELECT r.ride_id, r.customer_id, r.driver_id, r.pickup_location, r.dropoff_location, r.ride_
FROM Rides r
INNER JOIN Drivers d ON r.driver_id = d.driver_id
WHERE r.ride_status = 'ongoing' AND d.driver_id = <driver_id>;
```

Logging table

A trigger would be created to automatically log data deleted from the Rides and Payments tables. The trigger would insert the deleted data into a separate logging table, which would have the following columns:

table_name: The name of the table from which the data was deleted.

deleted_by: The user who deleted the data.

delete_timestamp: The timestamp when the data was deleted.

deleted_data: A JSON object containing the data that was deleted.

Logging table sample code

```
CREATE OR REPLACE FUNCTION log_deleted_data()
RETURNS TRIGGER AS $$
DECLARE
  deleted_table_name text;
  deleted_by_user text := user;
  delete_timestamp timestamp := current_timestamp;
  deleted_data jsonb;
BEGIN
  IF TG_OP = 'DELETE' THEN
    IF (TG_TABLE_NAME = 'Rides' OR TG_TABLE_NAME = 'Payments') THEN
      deleted_table_name := TG_TABLE_NAME;
      deleted_data := to_jsonb(OLD.*);
      INSERT INTO Deleted_Data_Logs(table_name, deleted_by, delete_timestam
      VALUES (deleted_table_name, deleted_by_user, delete_timestamp, delete
    END IF;
  END IF;
  RETURN OLD;
END;
```

Procedure

A stored procedure sample code

```
CREATE OR REPLACE FUNCTION get_total_fare(driver_id integer, start_date date, end_date date)
RETURNS numeric AS $$
DECLARE
  total_fare numeric := 0;
BEGIN
  SELECT SUM(fare_amount) INTO total_fare
  FROM Rides
  WHERE driver_id = driver_id
    AND ride_status = 'completed'
    AND start_time BETWEEN start_date AND end_date;

  RETURN total_fare;
END;
```

procedure

A stored procedure would be created to calculate the total fare earned by a driver over a specified time period. The procedure would take a driver_id, start_date, and end_date as parameters and return the total fare amount earned by the driver during that time period.

Justification of this Approach and Design

The proposed database design follows the principles of normalization to ensure data integrity and avoid data duplication. The use of foreign keys and referential actions ensures that data relationships are maintained and prevents orphaned records. The logging table ensures that deleted data can be recovered if necessary, while the view and stored procedure provide convenient access to commonly requested data.

Table Attributes

Customers Table

- customer_id (primary key)
- first_name
- last_name
- email
- phone_number

Drivers Table

- driver_id (primary key)
- first_name
- last_name
- email
- phone_number
- car_id (foreign key referencing Cars.car_id)

Rides Table

ride_id (primary key)
customer_id (foreign key referencing Customers.customer_id)
driver_id (foreign key referencing Drivers.driver_id)
pickup_location
dropoff_location
ride_date_time
fare

Why This Database

One reason for choosing this database design over the animal Sanctuary is because it is simpler and easier to work with. It has fewer tables and relationships compared to the more complex database designs of the animal sancruary, which could make it easier to develop, maintain, and scale. Additionally, the use of normalization ensures that data is stored efficiently and prevents redundancy, which can help optimize the database's performance.

Another reason for choosing this database design is because it aligns well with the needs of a microservice project. Microservices are a software architecture pattern in which an application is broken down into smaller, independent components that can be developed, deployed, and scaled independently. This database design could be a good fit for a microservice project I am planning on working on, because it separates the different components of the app (customers, drivers, rides, payments, etc.) into distinct tables, which could make it easier to develop and maintain independent microservices for each component.

Table Attributes

Payments Table

- payment_id (primary key)
- customer_id (foreign key referencing Customers.customer_id)
- ride_id (foreign key referencing Rides.ride_id)
- amount
- payment_date_time

Cars Table

- car_id (primary key)
- driver_id (foreign key referencing Drivers.driver_id)
- make
- model
- year
- license_plate

Summary

The database design for the taxi hailing app consists of five tables: Customers, Cars, Drivers, Rides, and Payments.
The Customers table stores information about the app's customers, including their first and last names, email addresses, and phone numbers. The Cars table stores information about the cars that drivers use to provide rides to customers, including make, model, year, and license plate number. The Drivers table contains information about the app's drivers, including their contact information and the car they use. The Rides table stores information about each ride, including the customer and driver IDs, pickup and dropoff locations, ride date and time, and fare. The Payments table tracks payment information, including the amount paid and the payment date and time, for each ride.
The design uses normalization to prevent data redundancy and ensure efficient storage of information. For example, by having a separate Cars table, we can avoid repeating car data for each driver that uses the same car. The use of primary and foreign keys establishes relationships between tables, allowing for easy retrieval and analysis of data. Additionally, the design includes a table to log deleted data and at least one function or procedure for added functionality. Overall, this design provides a solid foundation for a taxi hailing app database that can efficiently store and manage information about customers, drivers, cars, rides, and payments.

Thank you

Michael Momo Ayesa

mayesamomo1997@gmail.com