DATABASE DESIGN AND IMPLEMENTATION OF A PASSENGER POSITIONING SYSTEM (DIGITEKISI)

COURSE: INTERNET AND MOBILE PROGRAMMING, CEF440

TASK 6

NAME	**MATRICULE NO**
YIMNAI NERUS ZAUMU	FE20A123
TABOT CHARLES BESSONG	FE20A106
BEBONGNCHU YANNICK NKWETTA	FE20A022
BALEMBA JUNIOR BALEMBA	FE20A021
TANDONGFOR SHALOM CHANGEH	FE20A111

Course Instructor: DR. NKEMENI VALERY

June 2023

Table of Contents

DATABASE DESIGN AND IMPLEMENTATION OF A PASSENGER POSITIONING SYSTEM (DIGITEKISI)

Introduction

The database design phase of a software system is the process of creating a database that will store the data for the system. This phase includes three main steps: conceptual design, logical design, and physical design.

- **Conceptual design** is the process of creating a high-level overview of the database. This includes identifying the entities (tables) that will be stored in the database, the relationships between those entities, and the attributes (columns) that will be stored in each entity.
- **Logical design** is the process of translating the conceptual design into a more detailed model. This includes specifying the data types for each attribute, the constraints on the data, and the indexes that will be used to improve performance.
- **Physical design** is the process of creating the physical database. This includes creating the tables, columns, indexes, and relationships in the database management system (DBMS).

The database design phase is an important part of the software development process. A well-designed database can improve the performance, scalability, and security of the software system.

Here are some of the key benefits of good database design:

- **Improved performance:** A well-designed database can be accessed and queried more efficiently, which can improve the performance of the software system.
- **Increased scalability:** A well-designed database can be scaled to handle more data and users, which can help the software system to grow and meet the needs of its users.
- **Enhanced security:** A well-designed database can be protected from unauthorized access, which can help to protect the data and privacy of the software system's users.

Here are some of the key challenges of database design:

- **Complexity**: Database design can be a complex and challenging process. There are many factors to consider, such as the data requirements of the software system, the performance requirements, and the security requirements.
- **Change**: The data requirements of a software system can change over time, which can require changes to the database design.
- **Cost**: The cost of database design can be significant, depending on the complexity of the software system and the database.

Despite the challenges, database design is an important part of the software development process. A well-designed database can improve the performance, scalability, and security of the software system.

2. Objective of our system

This is an app that is used by both passengers and drivers, where the passengers use it to specify their positions at a given time, whereas the drivers use it to locate where the passengers are found in a given
time. This can help both drivers and passengers in the following ways:

- It reduces the amount of time passengers have to wait for a taxi.
- Drivers can optimize fuel consumption as the app will guide them to move to locations where there are more potential customers.

3. Conceptual Database Design

3.1 Entity-Relationship (ER) Diagram

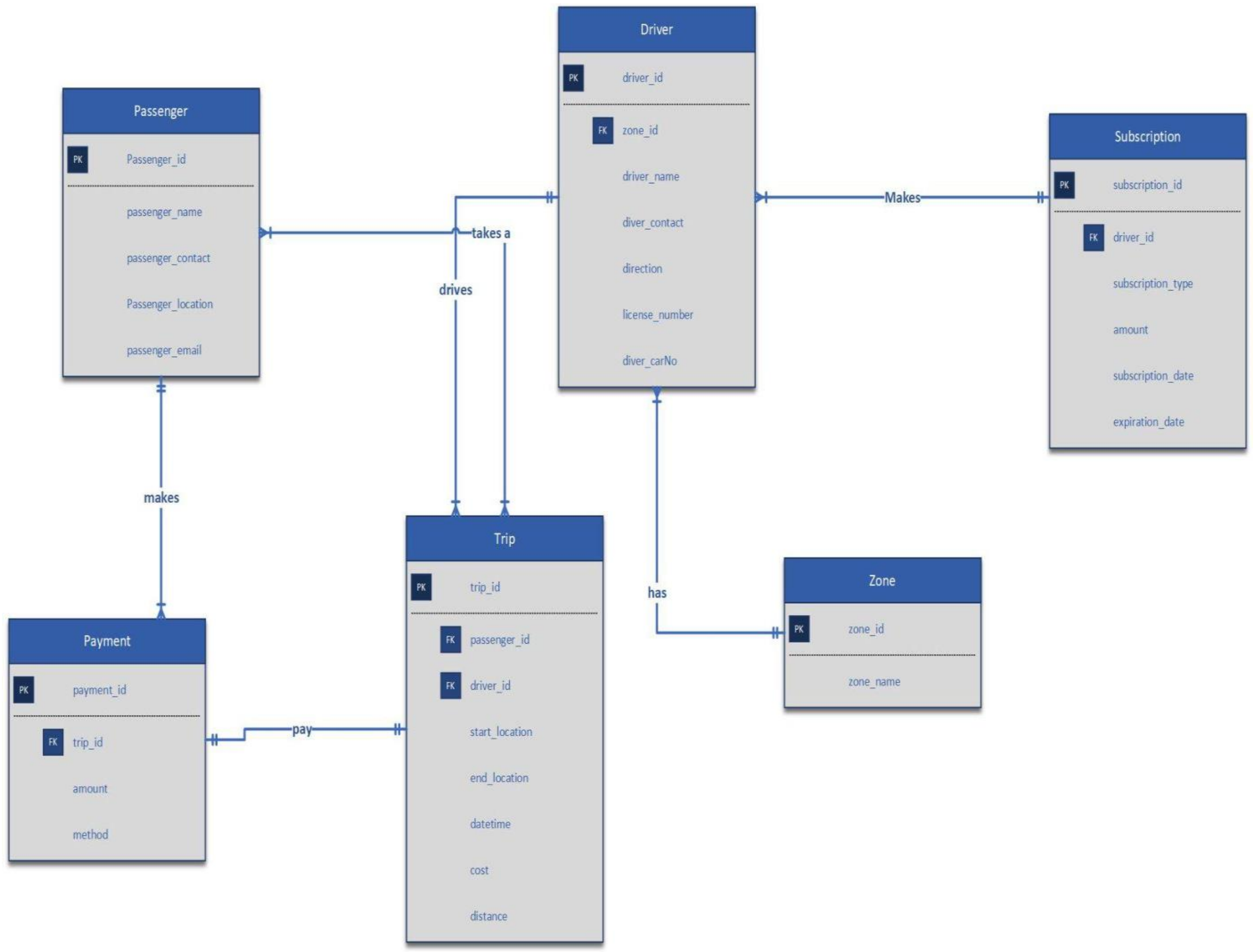


Figure 1: The Entity Relationship diagram of our system showing all the tables and their attributes.

3.2 Entity Descriptions

Our system has six main entities which are; “Passenger”, “Driver”, “Trip”, “Subscription”, “Payment” and “Zone”.

- The “Passenger” entity includes attributes such as “passenger_id”, “passenger_name”, “passenger_email”, “passenger_contact”, “passenger_location”.
- The “Driver” entity includes the attributes; “driver_id”, “driver_name”, “driver_contact”, “license_number”, “driver_carNo”, “direction”, “zone_id”.
- The “Trip” entity includes the following attributes; “trip_id”, “passenger_id”, “driver_id”, “start_location”, “end_location”, “datetime”, “distance”.
- The “Subscription” entity contains the attributes; “subscription_id”, “driver_id”, “subscription_type”, “amount”, “subscription_date”, “expiration_date”.
- The “Payment” entity has the attributes; “payment_id”, “trip_id”, “method”, “amount”.
- The “Zone” entity has the following attributes; “zone_id”, “zone_name”.

3.3 Relationships Description between different Entities

Passenger and Payment: We have a one to many relationship between the passenger and the payment. As a site that will love to generate some revenue , we will love to have our users make use of the system as many times as possible. Our passengers can book as many rides as they want and even on the same day. So in the process of booking a ride many times, our passengers tend to pay every ride and that makes a one to many relationship between the passenger and the payment tables. So we have a passengerId in every record of the payment table and this shows which passenger made which payment at what time.

Payment and Trip: Every trip will have just one payment. As a result we will have just a one to one relationship between the trip and the payment. Since each trip takes place at a different time, and can have different passengers. So it makes sense to link a single trip to one payment cause each trip will be different so can not be linked to another payment

Passenger and Trip: Here we have many to many relationships. Taking into consideration our country Cameroon and how transportation has already been handled here. In a single trip, as Tekisi's have 5 passenger seats, so as a Tekisi company to make profit, they tend to have more than one passenger and less than 5 passengers. So in a single trip we have many trip and we know as human beings are very mobile, they can travel many times to different locations even in a single day. So reason for a many to many relationship

Trip and Driver: Now a trip can be done by only one car and our system is designed in such a way that when our passengers register, they add a license plate number for their Tekisi. And that tells us that one driver can have one car, as a consequent, We have a one to many relationship between the driver and the trip. Cause a trip is something is a journey or a movement from one cluster to another. So we have one driver having many trips

Driver and Zone: We have a one to many relationship between the driver and their respective Zone. As one of the objectives of our system was to help drivers save fuel, we decided to make the system in such a way that our drivers can move only within a certain cluster or Zone. The zone will be of a certain geometric surface area. So as a consequent, every zone will have its own group of drivers making the relationship between driver and zone a many to one

Driver and Subscription: In order to generate some revenue for our system to keep running, we decided to add a subscription system so that our drivers will have to subscribe to different subscriptions where they will be charged based on how much of the system they use. Now we have different subscription plan, like monthly plans, weekly plans, monthly and quarterly plans. A driver can not select 2 different types of users and as a system with many users, we will have different users selecting their particular subscription. So we will have a one to many relationship between the driver and subscription table.

4. Logical Database Design

4.1 Logical Schema

```
Passenger (passenger_id, passenger_name,passenger_contact, passenger_location passenger_email)

Driver (driver_id, driver_name,driver_contact, *zone_id, direction, license_number, driver_carNo)

Zone (zone_id, zone_name)

Trip (trip_id, *passenger_id,* driver_id, start_location, end_location, datetime,distance)

Payment (payment_id, *trip_id, amount, method)

Subscription (subscription_id,*driver_id,subscription_type,amount, subscription_date, expiration_date)
```

In this logical schema:

- The "Passenger" table contains information about passengers, including attributes such as "PassengerID," "Name," "Phone," and all attributes specific to passengers.
- The "Driver" table includes details about drivers, including attributes such as "DriverID," "Name," "Phone," "ZoneID," "Direction," and all attributes specific to drivers.
- The "Zone" table represents the zones in which drivers operate. It includes attributes such as "ZoneID" and "ZoneName."
- The "Trip" table represents specific trips and contains attributes such as "TripID," "PassengerID," "DriverID," "StartLocation," "EndLocation," "distance," "datetime," .
- The "Payment" table stores payment information for each trip. It includes attributes such as "PaymentID," "TripID," "Amount," "method,".
- The “subscription” table stores information on the drivers’ subscriptions.

4.2 Data Types and Constraints

Here are the data types and constraints used in our passenger positioning system:

Passenger:
- passenger_id: Integer, Primary Key
- passenger_name: String
- passenger_contact: String
- passenger_email: String
- passenger_location: String

Zone:
- zone_id: Integer, Primary Key
- zone_name: String

Driver:
- driver_id: Integer, Primary Key
- driver_name: String
- driver_contact: String
- zone_id: Integer, Foreign Key (referencing Zone table)
- direction: String
- license_numbe: String
- driver_carNo: String

Trip:
- trip_id: Integer, Primary Key
- start_location: String
- end_location: String
- datetime: Date/Time
- distance: String
- driver_id: Integer, Foreign Key (referencing Driver table)
- passenger_id: Integer, Foreign Key (referencing Passenger table)

Payment:
- payment_id: Integer, Primary Key
- amount: Float/Double
- Method: String
- trip_id: Integer, Foreign Key (referencing Trip table)

Subscription:
- subscription_id: Integer, Primary Key
- amount: Float/Double
- subscription_type: String
- subscription_date: Date/Time
- expiration_date: Date/Time
- driver_id: Integer, Foreign Key (referencing Driver table)

These data types and constraints are suggestions based on the given information. Depending on the specific requirements of your system and the database management system (DBMS) you are using, you may need to adjust or add additional data types and constraints.

It's important to consult the documentation of your chosen DBMS to ensure the compatibility and accuracy of the data types and constraints used in your system.

4.3 Normalization

To normalize our passenger positioning system, we follow a step-by-step process based on the given entities and attributes. Here's the normalization:

1st Normal Form (1NF):
- Driver (DriverID, Name, Phone, Zone, Direction)
- Passenger (PassengerID, Name, Phone)
- Trip (TripID, StartLocation, EndLocation, StartTime, EndTime)

2nd Normal Form (2NF):
- Driver (DriverID, Name, Phone)
- Zone (DriverID, Zone) [New table for Zone and Driver relationship]
- Direction (DriverID, Direction) [New table for Direction and Driver relationship]
- Passenger (PassengerID, Name, Phone)
- Trip (TripID, StartLocation, EndLocation, StartTime, EndTime, DriverID)

3rd Normal Form (3NF):
- Driver (DriverID, Name, Phone)
- Zone (DriverID, Zone)

- Direction (DriverID, Direction)
- Passenger (PassengerID, Name, Phone)
- Trip (TripID, StartLocation, EndLocation, StartTime, EndTime, DriverID)
- Payment (PaymentID, Amount, Method, tripID)
- Subscription (subscription_id, amount, type)

In this normalization:
- The Driver entity is separated from the Zone and Direction attributes, creating two new tables (Zone and Direction) to avoid redundancy and handle potential updates independently.
- The Payment entity is created to store payment-related information and includes a foreign key reference to the Trip entity.

5. Physical Database Design

5.1 Database Management System Selection

After several researches and brainstorming sessions, we decided to go with the postgresql DBMS for our system.

PostgreSQL is a suitable choice for our passenger positioning system for several reasons:

1. Relational Database Management System (RDBMS): PostgreSQL is a powerful and feature-rich open-source RDBMS that adheres to the relational database model. It provides robust support for managing structured data, relationships, and enforcing data integrity.
2. Scalability and Performance: PostgreSQL is known for its scalability, allowing you to handle a large number of drivers, passengers, trips, and payments efficiently. It offers various performance optimization techniques, including indexing, query optimization, and parallel processing.
3. ACID Compliance: PostgreSQL follows the principles of ACID (Atomicity, Consistency, Isolation, Durability), ensuring that your data remains consistent and reliable even in the presence of concurrent transactions or system failures. This is important for maintaining data integrity and preserving the accuracy of the passenger and driver information.

4. Data Types and Constraints: PostgreSQL provides a wide range of data types and constraints that can suit the needs of your system. It supports standard data types such as integer, string, date/time, and also allows you to define custom data types if necessary. Additionally, it offers various constraints, including primary key, foreign key, not null, unique, and check constraints, which can help enforce data consistency and integrity.
5. Advanced Features: PostgreSQL offers advanced features like stored procedures, triggers, views, and full-text search capabilities. These features allow you to implement complex business logic, automate tasks, and enhance the functionality of your system.
6. Extensibility and Community Support: PostgreSQL has a vibrant and active community of developers and users who contribute to its ongoing development and provide support. It also offers extensive documentation and resources to help you effectively work with the database.
7. Cross-Platform Compatibility: PostgreSQL is cross-platform and can be deployed on various operating systems, including Windows, macOS, and different Linux distributions. This ensures flexibility and compatibility with your preferred deployment environment.

We utilized supabase for the implementation of our database.

Supabase is a popular open-source alternative to traditional database management systems, offering a range of features that make it suitable for our passenger positioning system. Here are some reasons why Supabase is a good fit:

1. PostgreSQL Compatibility: Supabase is built on top of PostgreSQL, which means it provides full compatibility with PostgreSQL features, including advanced querying, data types, and ACID compliance. This allows us to leverage the rich functionality of PostgreSQL while benefiting from the additional features and ease of use provided by Supabase.

2. Real-time Capabilities: Supabase includes real-time functionality through its support for WebSockets. This is particularly useful for our passenger positioning system, as it enables instant updates and notifications to be sent to drivers and passengers when relevant events occur, such as new trip requests or driver location updates.

3. Scalability and Performance: Supabase is designed to scale horizontally by automatically distributing data across multiple nodes. This ensures that our system can handle increasing workloads and deliver consistent performance even as the user base grows.

4. Authentication and Authorization: Supabase offers built-in authentication and authorization mechanisms. This allows you to easily manage user access and permissions, ensuring that only

authorized individuals can perform actions such as creating trips, making payments, or accessing sensitive data.

5. Real-time Database Triggers: Supabase allows you to define database triggers that automatically execute serverless functions in response to specified events. This feature can be utilized to implement custom business logic, perform data validations, or trigger actions based on specific database events.

6. Serverless Functions: Supabase supports serverless functions, allowing you to run custom code in response to API requests or database events. This feature enables you to extend the functionality of your system by implementing custom APIs, integrating with third-party services, or performing complex calculations or validations.

7. Developer-Friendly Tools: Supabase provides a user-friendly web interface and a rich set of developer tools, including a SQL editor, authentication management, and data management features. This helps streamline the development process, allowing you to focus on building your passenger positioning system rather than managing infrastructure.

8. Open Source and Community Support: Supabase is an open-source project with an active and growing community. This means you can benefit from community-contributed features, bug fixes, and ongoing development. The community also provides support and resources to help you troubleshoot issues and explore best practices.

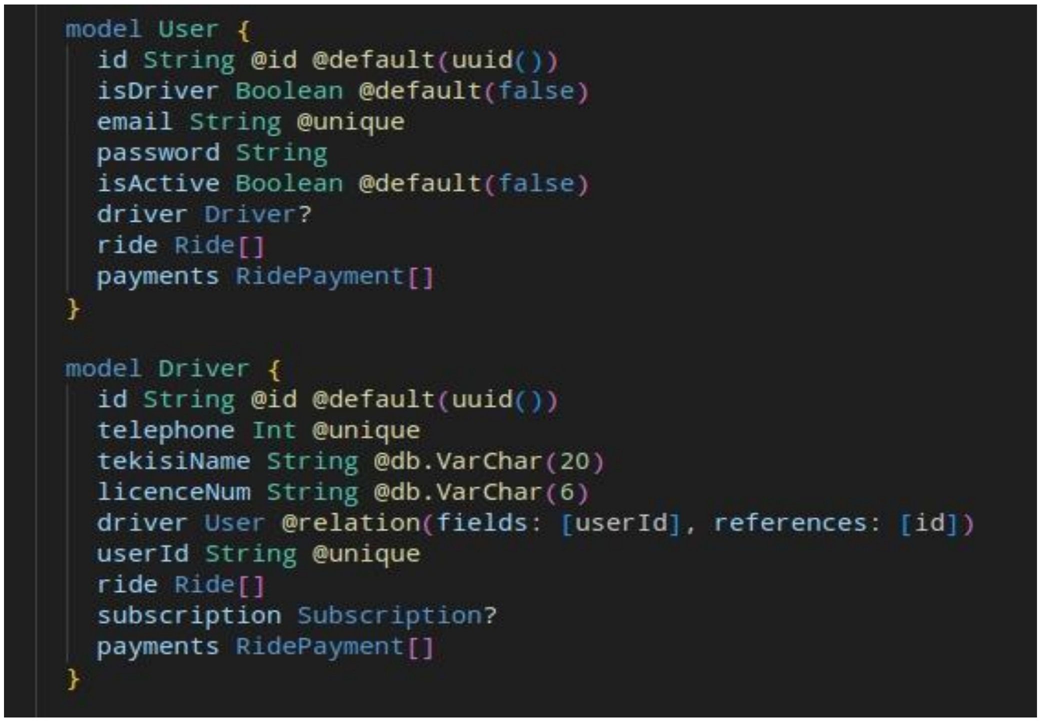
Considering these reasons, Supabase's PostgreSQL compatibility, real-time capabilities, scalability, authentication features, serverless functions, developer-friendly tools, and active community support make it a suitable choice for implementing our passenger positioning system.

5.2 Physical Schema

```
        -- Driver table
CREATE TABLE Driver (
  DriverID SERIAL PRIMARY KEY,
  Name VARCHAR(255) NOT NULL,
  Phone VARCHAR(20) NOT NULL
);

-- Zone table
CREATE TABLE Zone (
  ZoneID SERIAL PRIMARY KEY,
  DriverID INT NOT NULL REFERENCES Driver(DriverID),
  Zone VARCHAR(255) NOT NULL
);

-- Direction table
CREATE TABLE Direction (
  DirectionID SERIAL PRIMARY KEY,
  DriverID INT NOT NULL REFERENCES Driver(DriverID),
  Direction VARCHAR(255) NOT NULL
);

-- Passenger table
CREATE TABLE Passenger (
  PassengerID SERIAL PRIMARY KEY,
  Name VARCHAR(255) NOT NULL,
  Phone VARCHAR(20) NOT NULL
);

-- Trip table
CREATE TABLE Trip (
  TripID SERIAL PRIMARY KEY,
  StartLocation VARCHAR(255) NOT NULL,
  EndLocation VARCHAR(255) NOT NULL,
  StartTime TIMESTAMPTZ NOT NULL,
  EndTime TIMESTAMPTZ,
  DriverID INT NOT NULL REFERENCES Driver(DriverID),
  PassengerID INT NOT NULL REFERENCES Passenger(PassengerID)
);

-- Payment table
```

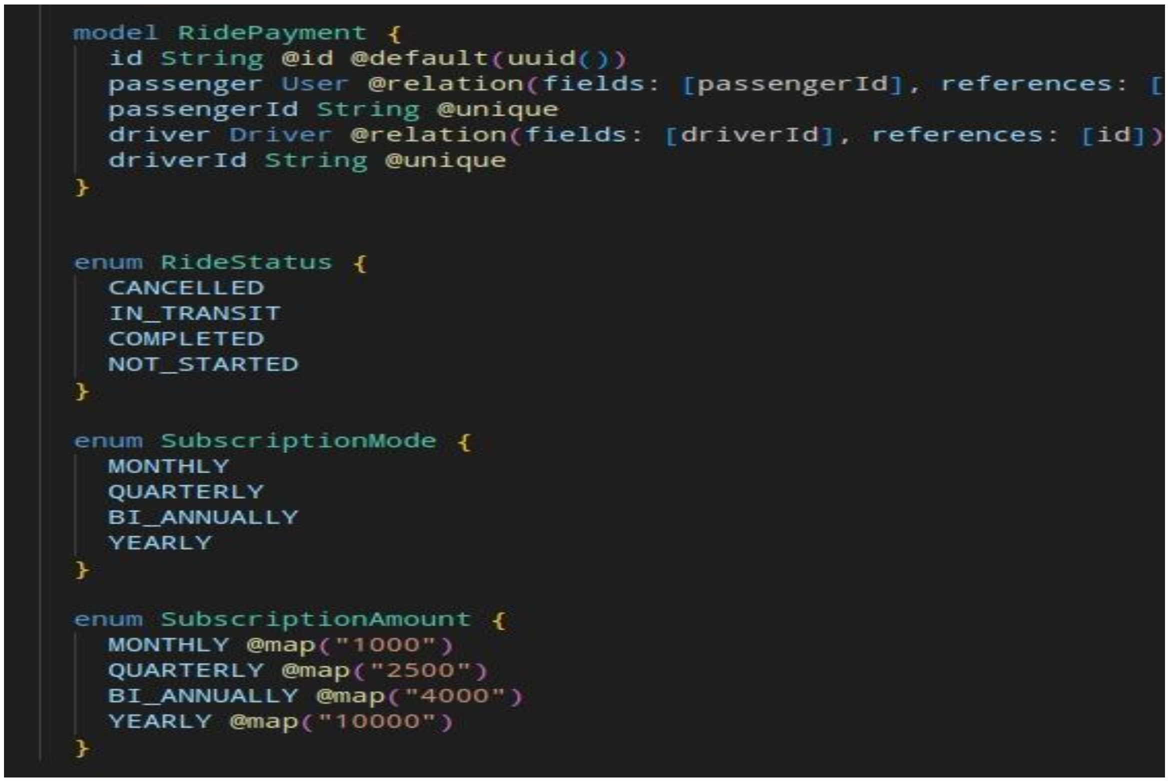
```
CREATE TABLE Payment (
  PaymentID SERIAL PRIMARY KEY,
  Amount NUMERIC(10, 2) NOT NULL,
  Method VARCHAR(50) NOT NULL,
  PassengerID INT NOT NULL REFERENCES Passenger(PassengerID)
);

CREATE TABLE Subscription(
  subscription_id SERIAL PRIMARY KEY,
  Amount NUMERIC(10, 2) NOT NULL,
  subscription_type VARCHAR(50) NOT NULL,
 driver_id INT NOT NULL REFERENCES Driver(PassengerID)
);
```

Display On Supabase:



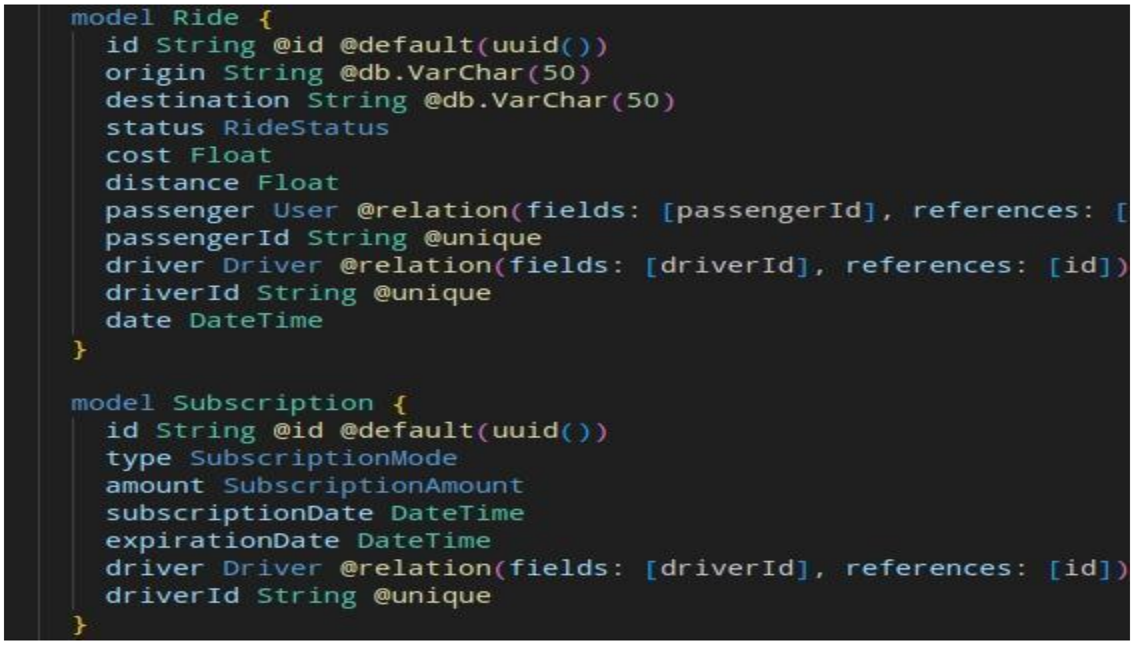
```
model User {
  id String @id @default(uuid())
  isDriver Boolean @default(false)
  email String @unique
  password String
  isActive Boolean @default(false)
  driver Driver?
  ride Ride[]
  payments RidePayment[]
}

model Driver {
  id String @id @default(uuid())
  telephone Int @unique
  tekisiName String @db.VarChar(20)
  licenceNum String @db.VarChar(6)
  driver User @relation(fields: [userId], references: [id])
  userId String @unique
  ride Ride[]
  subscription Subscription?
  payments RidePayment[]
}
```

Figure 2: The database creation on supabase with supabase syntax showing table models



```
model RidePayment {
  id String @id @default(uuid())
  passenger User @relation(fields: [passengerId], references: [
  passengerId String @unique
  driver Driver @relation(fields: [driverId], references: [id])
  driverId String @unique
}


enum RideStatus {
  CANCELLED
  IN_TRANSIT
  COMPLETED
  NOT_STARTED
}

enum SubscriptionMode {
  MONTHLY
  QUARTERLY
  BI_ANNUALLY
  YEARLY
}

enum SubscriptionAmount {
  MONTHLY @map("1000")
  QUARTERLY @map("2500")
  BI_ANNUALLY @map("4000")
  YEARLY @map("10000")
}
```

Figure 3: Continuation of database creation on supabase



```
model Ride {
  id String @id @default(uuid())
  origin String @db.VarChar(50)
  destination String @db.VarChar(50)
  status RideStatus
  cost Float
  distance Float
  passenger User @relation(fields: [passengerId], references: [
  passengerId String @unique
  driver Driver @relation(fields: [driverId], references: [id])
  driverId String @unique
  date DateTime
}

model Subscription {
  id String @id @default(uuid())
  type SubscriptionMode
  amount SubscriptionAmount
  subscriptionDate DateTime
  expirationDate DateTime
  driver Driver @relation(fields: [driverId], references: [id])
  driverId String @unique
}
```

Figure 4: More on the database on supabase

5.3 Indexing and Optimization

To improve query performance in our passenger positioning system, we employed indexing strategies and optimization techniques. We considered:

1. Indexing Strategies:
   - Identifying frequently queried columns: By analyzing the queries performed on our database and identify the columns frequently used in the WHERE, JOIN, or ORDER BY clauses.We created indexes on these columns to speed up query execution.
   - Primary and foreign key indexes:We ensured that primary key and foreign key columns are indexed. This helps optimize JOIN operations and maintain referential integrity.
   - Composite indexes: For queries involving multiple columns,we considered creating composite indexes that cover all the columns used in the query. This improves query performance by reducing the number of index lookups required.

2. Query Optimization Techniques:
   - Query planning and optimization: PostgreSQL has a sophisticated query planner that determines the most efficient execution plan for a given query. We updated statistics on our tables, as this helps the query planner make informed decisions.
   - Caching: We implemented caching mechanisms to store frequently accessed query results in memory. This significantly reduces the load on the database and improve response times, especially for read-heavy workloads.

5.4 Security and Access Control

Access Control and Authentication:

- The use of strong passwords: We enforced a password policy that requires users to create strong, unique passwords.
- Role-based access control (RBAC): We implemented RBAC to define roles and privileges for different user types (drivers, passengers). We assigned the minimum necessary privileges to each role to limit unauthorized access.
- Secure connections: We ensured that database connections are encrypted using SSL/TLS protocols to prevent eavesdropping and unauthorized access to sensitive data in transit.

2. Data Encryption:

- Encryption at rest: We encrypt the database files and backups to protect sensitive data even if unauthorized access occurs. PostgreSQL provides options for encrypting data at rest, such as using disk-level encryption or third-party encryption tools.

6. Data Backup and Recovery

Because our database is created and hosted on the Backend as a Service(BaaS) platform called supabase which itself uses postgres as its primary database, our database is automatically backed up for every operation or query being run on it. Because we used its in-built replication and sharding functionality, backup replicas are automatically created when we CREATE, DELETE or UPDATE data in our database. Thus, accessing the backed up data is as easy as accessing those replicas from our Supabase dashboard.

Here's an overview of the backup and restore process in Supabase:

6.1 Backup Process:

1. Access Supabase Dashboard: Log in to the Supabase dashboard at `https://app.supabase.io` using your credentials.

2. Select your Project: If you have multiple projects, select the appropriate project that contains the database you want to backup.

3. Navigate to the Database Section: In the dashboard, navigate to the "Database" section, which lists the databases associated with your project.

4. Select the Database: Choose the specific database that you want to backup from the list.

5. Initiate Backup: Within the selected database, locate the "Backups" tab or section. Click on the "Create Backup" or similar button to initiate the backup process.

6. Specify Backup Options: Depending on the available features, you may be able to specify options such as the backup name, retention period, and encryption settings. Configure these options as per your requirements.

7. Execute Backup: Start the backup process by clicking on the "Backup" or similar button. Supabase will create a backup of your PostgreSQL database.

8. Store the Backup: Once the backup process completes, Supabase typically stores the backup securely on their infrastructure. Ensure you have access to the backup file or the ability to retrieve it from the Supabase platform.

6.2 Restore Process:

1. Access Supabase Dashboard: Log in to the Supabase dashboard at `https://app.supabase.io` using your credentials.

2. Select your Project: Choose the appropriate project that contains the database you want to restore.

3. Navigate to the Database Section: In the dashboard, go to the "Database" section, which lists the databases associated with your project.

4. Select the Database: Choose the specific database to which you want to restore the backup.

5. Initiate Restore: Within the selected database, locate the "Backups" tab or section. Look for options to restore a backup and click on the "Restore" or similar button.

6. Specify Restore Options: Depending on the available features, you may be able to specify the backup file to restore, encryption settings, or other parameters. Provide the necessary information to proceed with the restore process.

7. Execute Restore: Start the restore process by clicking on the "Restore" or similar button. Supabase will restore the backup data to the specified PostgreSQL database.

8. Verify the Restoration: After the restore process completes, perform tests and checks to ensure the database has been successfully restored and is functioning as expected.

It's important to refer to the Supabase documentation and resources for detailed instructions and any specific considerations regarding backup and restore procedures within the Supabase platform.

7. Conclusion

In conclusion, the passenger positioning system presents an innovative solution for efficient and effective management of driver and passenger interactions. Throughout this report, we have examined various aspects of the system, including its requirements, entities, relationships, and functionalities.

The system provides a seamless experience for drivers, allowing them to define their zones and directions, facilitating optimal matching with passengers. Passengers, in turn, can easily request trips and make payments using cash or mobile money services. The system ensures data integrity and reliability by utilizing PostgreSQL as the underlying DBMS, which offers robust features, scalability, and cross-platform compatibility.

Furthermore, we have discussed the importance of normalization in the database design process, ensuring data consistency and reducing redundancy. The normalized entities and their relationships have been represented in the Entity-Relationship (ER) diagram, providing a clear visual representation of the system's structure.

Moreover, we have explored the suitability of Supabase as the database implementation platform, highlighting its compatibility with PostgreSQL, real-time capabilities, scalability, and developer-friendly tools. Supabase empowers the system with advanced features and security measures, ensuring efficient query performance and protection against unauthorized access.

To enhance the system's performance, we have suggested indexing strategies, query optimization techniques, and discussed the significance of monitoring and tuning the database. These measures contribute to faster query execution, improved responsiveness, and efficient resource utilization.

Lastly, we have emphasized the significance of implementing robust security measures to safeguard the database from unauthorized access. Measures such as access control, authentication, data encryption, regular updates, auditing, and employee training are vital for ensuring the confidentiality, integrity, and availability of the system's data.

In conclusion, the passenger positioning system offers a comprehensive solution for managing driver and passenger interactions efficiently. By leveraging advanced technologies, database design principles, and security measures, the system can provide a secure, reliable, and user-friendly experience for both drivers and passengers. As the system continues to evolve, it is essential to remain vigilant, adapt security practices, and incorporate user feedback to meet the dynamic needs of the users and maintain a high level of service quality.