a) Problem Description

GlobalRides, a company similar to Uber and Uber Eats, operates a platform connecting riders, drivers, food delivery customers, and restaurants. The company seeks to design a relational database to effectively manage its multifaceted operations, including Riders, Drivers, Customers, Restaurants, Rides, Food Orders, Payments, and Reviews.

A user can serve multiple roles, such as being a Rider, a Customer, a Driver, and a Restaurant Owner. All users share common attributes, including User ID, Name (First, Middle, Last), Contact Details, Address, Gender, and Date of Birth. A user may provide multiple contact numbers.

For Rides, each booking has attributes like Ride ID, Pickup and Drop-off Locations, Pickup Time, Ride Fare, and Payment Status. Rides are associated with Drivers, who can be assigned multiple rides, while each ride can only have one driver. Drivers have specific attributes such as Driver ID, License Details, Vehicle Information, and Experience.

For Food Orders, customers can place orders with Restaurants listed on the platform. Each order records Order ID, Restaurant ID, Customer ID, Order Date, Delivery Status, Total Amount, and Payment Method. Orders consist of multiple items from the restaurant's menu. Each restaurant manages its menu, including attributes like Item ID, Name, Description, Price, and Food Category (e.g., Appetizers, Main Course, Desserts). Restaurants have attributes such as Restaurant ID, Name, Address, Cuisine, Operational Hours, and Ownership details. Restaurants can also run promotions tied to specific menu items. These promotions are unique within the restaurant and include Promotion ID, Description, and Validity Period.

Riders and Customers can leave reviews. Reviews have attributes like Review ID, Rating, Feedback Text, and Date, linked to the specific ride, food item, or restaurant being reviewed.

Employees of GlobalRides are integral to the company's operations and are categorized into several distinct roles: Platform Managers, Support Agents, and Delivery Coordinators. Each employee has an Employee ID, a unique identifier following a predefined format such as "E###," where "###" represents a sequence of digits. Employees must be at least 18 years old. Additional general attributes include their Start Date, which records when the employee joined the organization, and Department, signifying the specific area within the company they are associated with.

Delivery Coordinators are tasked with managing the logistics of delivery drivers, focusing on ensuring timely and efficient order deliveries. They are responsible for assigning drivers to orders, optimizing routes, and addressing operational challenges such as delays or vehicle issues. Their role is crucial in maintaining the smooth flow of delivery operations and upholding service quality standards.

b) Project Questions

1. Would a superclass/subclass relationship be beneficial in the GlobalRides database design? Why or why not?

A superclass/subclass relationship would be beneficial in the GlobalRides database design as all Users, Employees, and Reviews share a collection of the same attributes with slight variations to individual tuples based on the role in the database. However, since these roles have their own collections of the same attributes, a superclass/subclass design for the database is suitable. For example, in the database itself, Users, Employees, and Reviews can be divided as follows.

Users: Rider, Customer, Driver, and Restaurant Owner

Employees: Platform Managers, Support Agents, and Delivery Coordinators

Reviews: Restaurant Review, Menu Item Review, and Ride Review

If I wanted to query information specific to a type of user, then I can simply query on the individual user type table. If I am trying to find a general attribute such as Address or Date of Birth, then I can query across all users on the users table and do not have to join all the individual table types.

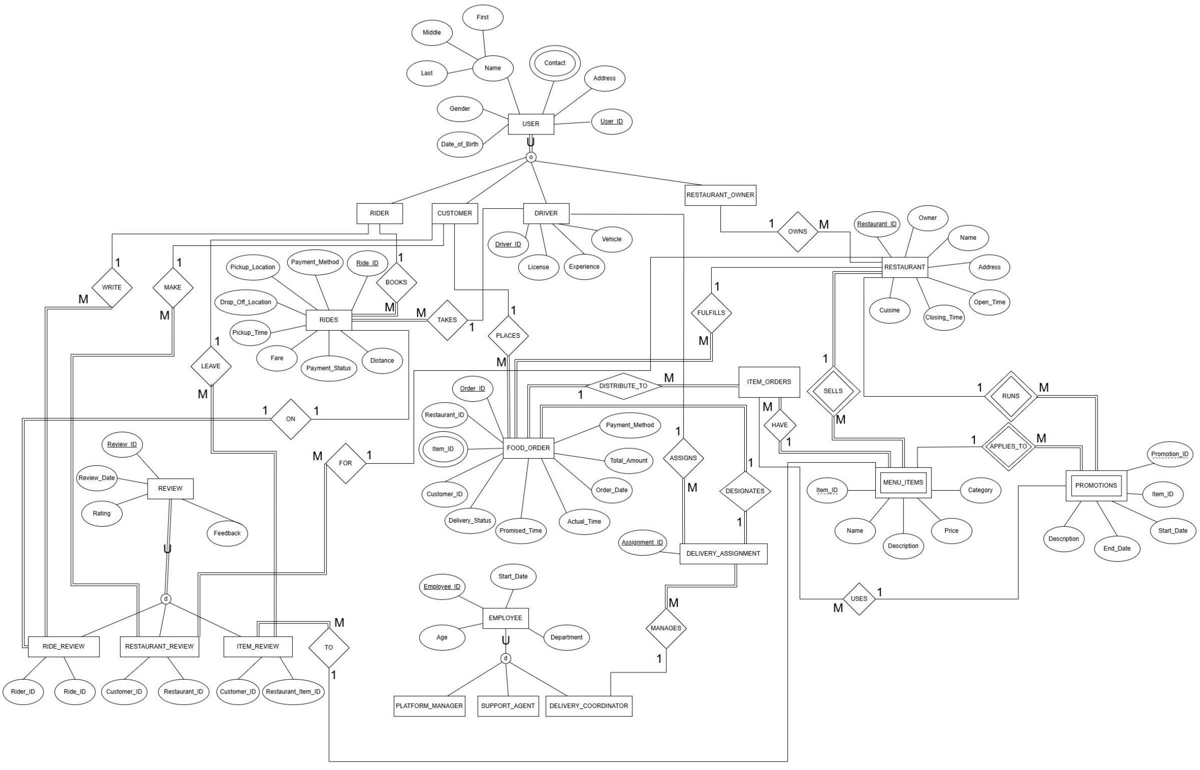
2. Can you think of 5 additional rules (other than those described above) that would likely be used in this environment? How would your design change to accommodate these rules?

1. Separating the total order amount in the food orders table and fare amount in the rides table into an item/ride total, tip total, and tax total. This would allow us to not only query on revenue but also tips, perhaps to see which area customers tip best or common tip percentages. Total amounts can become calculated values and can now include tax. Tax would not be calculated in queries as tax varies by state/region.
    a. In FOOD_ORDER, instead of a total amount attribute, there would be an item order attribute and tip attribute. Similarly, in the RIDES table, instead of fare, the table would have ride fare and tip attributes.
2. Adding a quantity attribute to the item orders table. The current implementation just shows one quantity of each item that a customer orders. However, if there are multiple of the same items, then they should be accounted for to get an accurate number of menu items ordered. The implementation allows us to specify the quantity without redundant rows. This allows restaurants to see their item revenue break down as well since the quantities can be multiplied by the item price to get a subtotal.

    a. Add a quantity attribute to the ITEM_ORDERS table.
3. Drivers should have an age check to determine if they are permitted to deliver alcohol. This allows us to see how many drivers are permitted to have delivered alcohol and helps delivery coordinators know which employees are permitted. Also, only customers over the age of 21 should be able to purchase alcohol.
    a. The implementation may involve a Boolean attribute of legal_drinkers and checking if the difference between the current date and date of birth of the user is greater than or equal to 21. If the user is at least 21, then the legal_drinker attribute in the USERS table would be true.
4. All drivers must be at least 18 years old as this is the minimum unrestricted legal driving age for all states.
    a. This would involve checking if the difference between the current date and date_of_birth attribute if the USERS table is at least 18 years.
5. Rides and food orders should include a refund attribute in case there was a missing item in the order or issue with the ride. This allows us to calculate company compensation and see which drivers are problematic all while seeing the revenue nuances.
    a. The RIDES and FOOD_ORDERS tables would include a refund_amount attribute whose default value is 0.

3. Justify the use of a Relational DBMS like Oracle for this project (Successfully design a relational database system, and show all implementation in the final report at Phase IV).

GlobalRides uses structured entities with consistent attributes and well-defined relationships among each other. A relational model allows us to use primary and foreign keys to express these relationships in a consistent, organized, non-redundant, and simple way. The use of a relational database management system allows us to enforce referential integrity, check input values, and enforce business rules at the database level. This means that we can verify data, relations, and logic in the tables and relationships themselves. DBMSs also promotes performance and security as the management system can handle many users and data. Through role-based access, the actual editing and consistency of the data is maintained. DBMSs also allow for querying of the data to analyze sales and other business factors. Other forms of database management, such as files, have very little control over data integrity or are inconvenient to use when handling a lot of data.



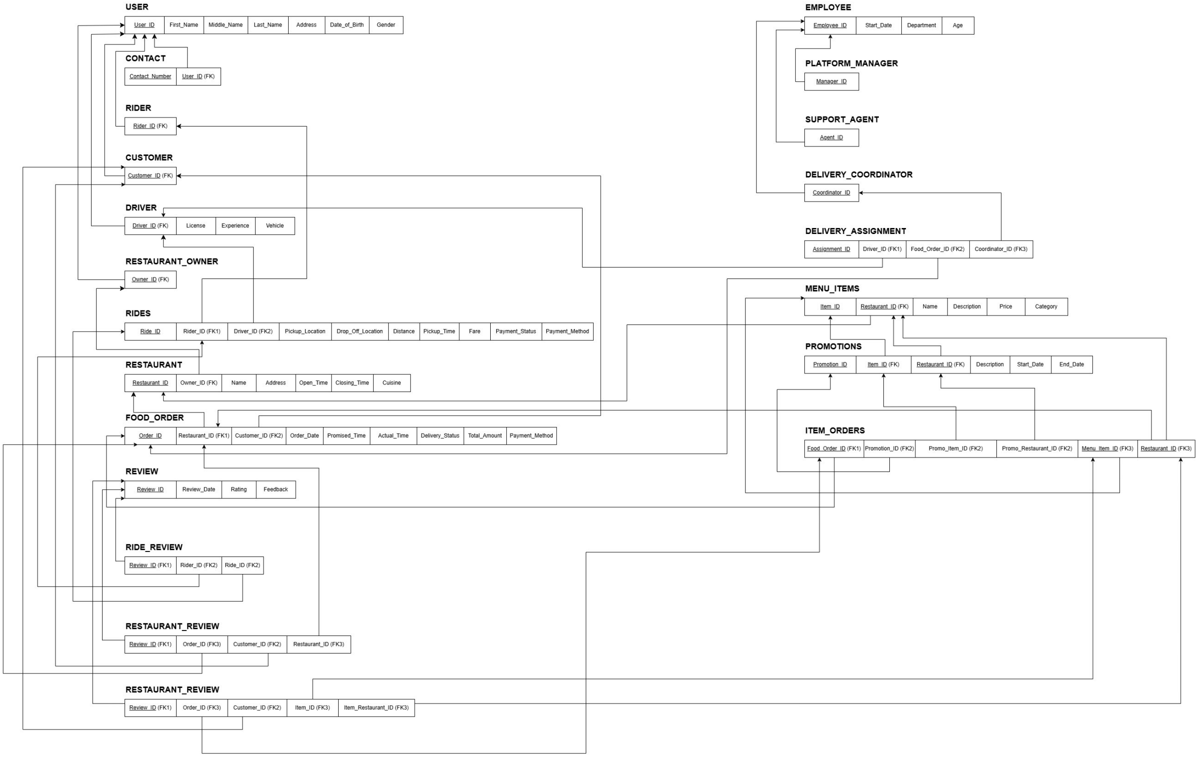
First
Middle
Last
Name
Contact
Address
Gender
USER
User_ID
Date_of_Birth
RIDER
CUSTOMER
DRIVER
RESTAURANT_OWNER
OWNS
Restaurant_ID
Owner
Name
RESTAURANT
Address
Cuisine
Closing_Time
Open_Time
Driver_ID
License
Experience
Vehicle
WRITE
MAKE
BOOKS
Pickup_Location
Payment_Method
Ride_ID
Drop_Off_Location
RIDES
TAKES
Pickup_Time
Fare
Payment_Status
Distance
PLACES
FULFILLS
LEAVE
SELLS
RUNS
ITEM_ORDERS
DISTRIBUTE_TO
ON
Order_ID
Restaurant_ID
Payment_Method
HAVE
Item_ID
FOOD_ORDER
Total_Amount
ASSIGNS
APPLIES_TO
Promotion_ID
FOR
Customer_ID
Order_Date
DESIGNATES
MENU_ITEMS
Item_ID
Category
PROMOTIONS
Item_ID
Review_ID
Review_Date
REVIEW
Rating
Feedback
Delivery_Status
Promised_Time
Actual_Time
Name
Description
Price
Description
End_Date
Start_Date
Assignment_ID
DELIVERY_ASSIGNMENT
Employee_ID
Start_Date
EMPLOYEE
Age
Department
USES
MANAGES
RIDE_REVIEW
RESTAURANT_REVIEW
ITEM_REVIEW
TO
Rider_ID
Ride_ID
Customer_ID
Restaurant_ID
Customer_ID
Restaurant_Item_ID
PLATFORM_MANAGER
SUPPORT_AGENT
DELIVERY_COORDINATOR

First, I created all the given entities and their attributes. I assumed that users and employees were superclasses of their given subclasses. I assumed that there were no other roles for users so there was total participation and that a user could assume multiple roles. I assumed that there were other roles and departments not listed and thus employees do not have to assume one of the three roles in the instructions. I assumed employees could only have one job title, making the inheritance disjoint. I also assumed that a Restaurant Owner owns a Restaurant.

Next, I focused on the relations between entity types. I assumed a ride needed a driver and a rider, so I created a relation between the Rider and Ride and made the Ride total participation. Similarly, Food Orders needed to be placed by a customer, for a restaurant, with a menu item, and assigned to a driver. I assumed Menu Items and Promotions were unique to the Restaurant that offered them, so I made them both weak entity types with partial keys. Since the Promotions were tied to Menu Items of the restaurant, I made Promotions related to both Menu Items and Restaurant. Since these promotions would change the total of the order, I connected the Promotions to the Food Order.

With Reviews, I created an attribute that shows which type of rating is being offered so that the link to the specific reviewed thing is clear. With the Delivery Coordinator, since they are assigning Drivers to Food Orders, I thought the implementation may be easier if there was an Assignments entity type that kept track of the Driver, Food Order, and Coordinator together rather than a ternary relationship type. I also assumed the other descriptors of the Delivery Coordinator were more role and task oriented than attributes of the Employee themselves.

Lastly, I worked on Cardinality and Participation. I assumed all User types could book, own, write reviews for, and take multiple orders and items since that was a premise of the company. This is similar to the Delivery Coordinator and the Delivery Assignments. I also assumed the booked and ordered entities could only be made by one person each time. For a Review or Food Order to exist, it had to be made by one person thus total participation. I assumed if Drivers could be assigned multiple rides, then they could also be assigned multiple Food Orders. However, they are not required since they could be Ride drivers. For the rest, I used common logic such as Restaurant Owners could own multiple Restaurants, Restaurants could fulfill multiple Food Orders, and multiple Promotions of Multiple Menu Items could occur at once for a Restaurant. I assumed a Restaurant cannot exist without a Menu Item but could exist without a Promotion.


USER
User_ID
First_Name
Middle_Name
Last_Name
Address
Date_of_Birth
Gender
CONTACT
Contact_Number
User_ID (FK)
RIDER
Rider_ID (FK)
CUSTOMER
Customer_ID (FK)
DRIVER
Driver_ID (FK)
License
Experience
Vehicle
RESTAURANT_OWNER
Owner_ID (FK)
RIDES
Ride_ID
Rider_ID (FK1)
Driver_ID (FK2)
Pickup_Location
Drop_Off_Location
Distance
Pickup_Time
Fare
Payment_Status
Payment_Method
RESTAURANT
Restaurant_ID
Owner_ID (FK)
Name
Address
Open_Time
Closing_Time
Cuisine
FOOD_ORDER
Order_ID
Restaurant_ID (FK1)
Customer_ID (FK2)
Order_Date
Promised_Time
Actual_Time
Delivery_Status
Total_Amount
Payment_Method
REVIEW
Review_ID
Review_Date
Rating
Feedback
RIDE_REVIEW
Review_ID (FK1)
Rider_ID (FK2)
Ride_ID (FK2)
RESTAURANT_REVIEW
Review_ID (FK1)
Order_ID (FK3)
Customer_ID (FK2)
Restaurant_ID (FK3)
RESTAURANT_REVIEW
Review_ID (FK1)
Order_ID (FK3)
Customer_ID (FK2)
Item_ID (FK3)
Item_Restaurant_ID (FK3)
EMPLOYEE
Employee_ID
Start_Date
Department
Age
PLATFORM_MANAGER
Manager_ID
SUPPORT_AGENT
Agent_ID
DELIVERY_COORDINATOR
Coordinator_ID
DELIVERY_ASSIGNMENT
Assignment_ID
Driver_ID (FK1)
Food_Order_ID (FK2)
Coordinator_ID (FK3)
MENU_ITEMS
Item_ID
Restaurant_ID (FK)
Name
Description
Price
Category
PROMOTIONS
Promotion_ID
Item_ID (FK)
Restaurant_ID (FK)
Description
Start_Date
End_Date
ITEM_ORDERS
Food_Order_ID (FK1)
Promotion_ID (FK2)
Promo_Item_ID (FK2)
Promo_Restaurant_ID (FK2)
Menu_Item_ID (FK3)
Restaurant_ID (FK3)

The first thing I did was create relations for all the entity types and their associated attributes. This included identifying the primary keys for the strong entities and mapping the owner's primary key to the weak entity types to form composite primary keys. This also included mapping subclasses to their superclasses through the primary key in the superclass as a foreign and primary key in the subclass. I underlined primary keys and indicated foreign keys using parentheses.

For multivalued attributes, I treated them like weak entities by creating a new relation. Each multivalued relation had a composite primary key consisting of the multivalued attribute and a foreign key referencing the primary key of the original entity.

Next, I mapped the relationships. For 1:M relationships, I placed a foreign key on the "many" side referencing the primary key on the "one" side. For M:N relationships, I created a new relation (bridge table) containing foreign keys to both participating entity types. These foreign keys also formed the composite primary key of the bridge table.

After building out the relations and mapping all keys, I adjusted the schema based on how I imagined the actual SQL implementation would work. For example, in the REVIEW relation, I included multiple foreign keys to allow later use of SQL constraints to ensure that each review references only one reviewer and one item being reviewed.

Once all relations and attributes were in place, I drew arrows from each foreign key to the corresponding primary key in the referenced relation to clearly show referential integrity.

USER

User_ID	First_Name	Middle_Name	Last_Name	Address	Date_of_Birth	Gender

CONTACT

Contact_Number	User_ID (FK)

RIDER

Rider_ID (FK)

CUSTOMER

Customer_ID (FK)

DRIVER

Driver_ID (FK)	License	Experience	Vehicle

RESTAURANT_OWNER

Owner_ID (FK)

RIDES

Ride_ID	Rider_ID (FK1)	Driver_ID (FK2)	Pickup_Location	Drop_Off_Location	Distance	Pickup_Time	Fare	Payment_Status	Payment_Method

RESTAURANT

Restaurant_ID	Owner_ID (FK)	Name	Address	Open_Time	Closing_Time	Cuisine

FOOD_ORDER

Order_ID	Restaurant_ID (FK1)	Customer_ID (FK2)	Order_Date	Promised_Time	Actual_Time	Delivery_Status	Total_Amount	Payment_Method

REVIEW

Review_ID	Review_Date	Rating	Feedback

RIDE_REVIEW

Review_ID (FK1)	Rider_ID (FK2)	Ride_ID (FK2)

RESTAURANT_REVIEW

Review_ID (FK1)	Order_ID (FK3)	Customer_ID (FK2)	Restaurant_ID (FK3)

RESTAURANT_REVIEW

Review_ID (FK1)	Order_ID (FK3)	Customer_ID (FK2)	Item_ID (FK3)	Item_Restaurant_ID (FK3)

EMPLOYEE

Employee_ID	Start_Date	Department	Age

PLATFORM_MANAGER

Manager_ID

SUPPORT_AGENT

Agent_ID

DELIVERY_COORDINATOR

Coordinator_ID

DELIVERY_ASSIGNMENT

Assignment_ID	Driver_ID (FK1)	Food_Order_ID (FK2)	Coordinator_ID (FK3)

MENU_ITEMS

Item_ID	Restaurant_ID (FK)	Name	Description	Price	Category

PROMOTIONS

Promotion_ID	Item_ID (FK)	Restaurant_ID (FK)	Description	Start_Date	End_Date

ITEM_ORDERS

Food_Order_ID (FK1)	Promotion_ID (FK2)	Promo_Item_ID (FK2)	Promo_Restaurant_ID (FK2)	Menu_Item_ID (FK3)	Restaurant_ID (FK3)

```sql
--TABLES--

CREATE DATABASE GlobalRides;

USE GlobalRides;


-- USERS and User Roles

CREATE TABLE USERS (
  User_ID INT PRIMARY KEY,
  First_Name VARCHAR(50) NOT NULL,
  Middle_Name VARCHAR(50),
  Last_Name VARCHAR(50) NOT NULL,
  Address VARCHAR(255) NOT NULL,
  Date_of_Birth DATE NOT NULL,
  Gender CHAR(1) NOT NULL,
  CHECK (Gender IN ('M', 'F', 'O'))
);


CREATE TABLE CONTACT (
  Contact_Number CHAR(10),
  User_ID INT,
  CHECK (Contact_Number REGEXP '^[0-9]{10}$'),
  FOREIGN KEY (User_ID) REFERENCES USERS(User_ID) ON DELETE CASCADE,
  PRIMARY KEY (Contact_Number, User_ID)
);


CREATE TABLE RIDER (
  Rider_ID INT PRIMARY KEY,
```

```
    FOREIGN KEY (Rider_ID) REFERENCES USERS(User_ID) ON DELETE CASCADE
);

CREATE TABLE CUSTOMER (
    Customer_ID INT PRIMARY KEY,
    FOREIGN KEY (Customer_ID) REFERENCES USERS(User_ID) ON DELETE CASCADE
);

CREATE TABLE DRIVER (
    Driver_ID INT PRIMARY KEY,
    License VARCHAR(10) NOT NULL,
    Experience VARCHAR(255) NOT NULL,
    Vehicle VARCHAR(50) NOT NULL,
    CHECK (Vehicle IN ('Sedan', 'SUV', 'Truck', 'Minivan', 'Luxury', 'Electric', 'Motorcycle',
'Hybrid', 'Convertible')),
    FOREIGN KEY (Driver_ID) REFERENCES USERS(User_ID) ON DELETE CASCADE
);

CREATE TABLE RESTAURANT_OWNER (
    Owner_ID INT PRIMARY KEY,
    FOREIGN KEY (Owner_ID) REFERENCES USERS(User_ID) ON DELETE CASCADE
);

-- Rides and Restaurants

CREATE TABLE RIDES (
```

```
  Ride_ID INT PRIMARY KEY,
  Rider_ID INT NOT NULL,
  Driver_ID INT NOT NULL,
  Pickup_Location VARCHAR(255) NOT NULL,
  Drop_Off_Location VARCHAR(255) NOT NULL,
  Distance DECIMAL(6, 2) NOT NULL,
        Pickup_Time DATETIME NOT NULL,
  Fare DECIMAL(8, 2) NOT NULL,
  Payment_Status VARCHAR(20) NOT NULL,
  Payment_Method VARCHAR(50) NOT NULL,
  UNIQUE (Ride_ID, Rider_ID),
  CHECK (Payment_Status IN ('Paid', 'Not Paid')),
  CHECK (Payment_Method IN ('Cash', 'Credit', 'Debit', 'Check', 'Other')),
  FOREIGN KEY (Rider_ID) REFERENCES RIDER(Rider_ID),
  FOREIGN KEY (Driver_ID) REFERENCES DRIVER(Driver_ID)
);

CREATE TABLE RESTAURANT (
  Restaurant_ID INT PRIMARY KEY,
  Owner_ID INT NOT NULL,
  Name VARCHAR(255) NOT NULL,
  Address VARCHAR(255) NOT NULL,
  Open_Time TIME NOT NULL,
  Closing_Time TIME NOT NULL,
  Cuisine VARCHAR(255) NOT NULL,
  CHECK (Closing_Time > Open_Time),
```

```sql
    FOREIGN KEY (Owner_ID) REFERENCES RESTAURANT_OWNER(Owner_ID)
);

-- Menu and Promotions

CREATE TABLE MENU_ITEMS (
    Item_ID INT,
    Restaurant_ID INT,
    Name VARCHAR(50) NOT NULL,
    Description VARCHAR(255) NOT NULL,
    Price DECIMAL(8, 2) NOT NULL,
    Category VARCHAR(50) NOT NULL,
    CHECK (Category IN ('Appetizer', 'Side', 'Main Course', 'Dessert', 'Beverage')),
    PRIMARY KEY (Item_ID, Restaurant_ID),
    FOREIGN KEY (Restaurant_ID) REFERENCES RESTAURANT(Restaurant_ID) ON DELETE CASCADE
);

CREATE TABLE PROMOTIONS (
    Promotion_ID INT,
    Item_ID INT NOT NULL,
    Restaurant_ID INT NOT NULL,
    Description VARCHAR(255) NOT NULL,
    Start_Date DATE NOT NULL,
    End_Date DATE NOT NULL,
    CHECK (End_Date >= Start_Date),
```

```sql
   PRIMARY KEY(Promotion_ID, Item_ID, Restaurant_ID),
   FOREIGN KEY (Item_ID, Restaurant_ID) REFERENCES MENU_ITEMS(Item_ID, 
Restaurant_ID)
);

-- Orders and Relationships
CREATE TABLE FOOD_ORDER (
   Order_ID INT PRIMARY KEY,
   Restaurant_ID INT NOT NULL,
   Customer_ID INT NOT NULL,
   Order_Date DATE NOT NULL,
   Promised_Delivery_Time DATETIME NOT NULL,
   Actual_Delivery_Time DATETIME,
   Delivery_Status VARCHAR(50) NOT NULL,
   Total_Amount DECIMAL(8, 2) NOT NULL,
   Payment_Method VARCHAR(50) NOT NULL,
   UNIQUE (Order_ID, Restaurant_ID),
   CHECK (Delivery_Status IN ('Delivered', 'Canceled', 'In Progress', 'Not Started')),
   CHECK ((Delivery_Status IN ('In Progress', 'Not Started') AND Actual_Delivery_Time IS 
NULL) OR
                        (Delivery_Status = 'Canceled' AND Actual_Delivery_Time IS NULL AND 
Total_Amount = 0) OR
                        (Delivery_Status = 'Delivered' AND Actual_Delivery_Time IS NOT NULL)),
   CHECK (Payment_Method IN ('Cash', 'Credit', 'Debit', 'Check', 'Other')),
   FOREIGN KEY (Restaurant_ID) REFERENCES RESTAURANT(Restaurant_ID),
   FOREIGN KEY (Customer_ID) REFERENCES CUSTOMER(Customer_ID)
);
```

```sql
CREATE TABLE ITEM_ORDERS (
   Food_Order_ID INT,
   Promotion_ID INT DEFAULT NULL,
   Promo_Item_ID INT DEFAULT NULL,
   Promo_Restaurant_ID INT DEFAULT NULL,
   Menu_Item_ID INT NOT NULL,
   Restaurant_ID INT NOT NULL,
   CHECK (
               (Promotion_ID IS NULL AND Promo_Item_ID IS NULL AND
Promo_Restaurant_ID IS NULL)
               OR
               (Promotion_ID IS NOT NULL AND Promo_Item_ID = Menu_Item_ID AND
Promo_Restaurant_ID = Restaurant_ID)
       ),
   PRIMARY KEY (Food_Order_ID, Menu_Item_ID, Restaurant_ID),
   FOREIGN KEY (Food_Order_ID, Restaurant_ID) REFERENCES FOOD_ORDER(Order_ID,
Restaurant_ID) ON DELETE CASCADE,
   FOREIGN KEY (Menu_Item_ID, Restaurant_ID) REFERENCES MENU_ITEMS(Item_ID,
Restaurant_ID),
   FOREIGN KEY (Promotion_ID, Promo_Item_ID, Promo_Restaurant_ID) REFERENCES
PROMOTIONS(Promotion_ID, Item_ID, Restaurant_ID)
);

-- Reviews
CREATE TABLE REVIEW (
   Review_ID INT PRIMARY KEY,
```

```
    Review_Date DATE NOT NULL,
    Rating TINYINT NOT NULL,
    Feedback VARCHAR(255),
    CHECK (Rating BETWEEN 1 AND 5)
);

CREATE TABLE RIDE_REVIEW (
    Review_ID INT PRIMARY KEY,
    Ride_ID INT NOT NULL,
    Rider_ID INT NOT NULL,
    FOREIGN KEY (Review_ID) REFERENCES REVIEW(Review_ID) ON DELETE CASCADE,
    FOREIGN KEY (Ride_ID, Rider_ID) REFERENCES RIDES(Ride_ID, Rider_ID)
);

CREATE TABLE RESTAURANT_REVIEW (
    Review_ID INT PRIMARY KEY,
    Order_ID INT,
    Restaurant_ID INT NOT NULL,
    Customer_ID INT NOT NULL,
    FOREIGN KEY (Review_ID) REFERENCES REVIEW(Review_ID) ON DELETE CASCADE,
    FOREIGN KEY (Order_ID, Restaurant_ID) REFERENCES FOOD_ORDER(Order_ID,
Restaurant_ID),
    FOREIGN KEY (Customer_ID) REFERENCES CUSTOMER(Customer_ID)
);

CREATE TABLE ITEM_REVIEW (
```

```sql
    Review_ID INT PRIMARY KEY,
    Order_ID INT,
    Customer_ID INT NOT NULL,
    Item_ID INT NOT NULL,
    Restaurant_ID INT NOT NULL,
    FOREIGN KEY (Review_ID) REFERENCES REVIEW(Review_ID) ON DELETE CASCADE,
    FOREIGN KEY (Customer_ID) REFERENCES CUSTOMER(Customer_ID),
    FOREIGN KEY (Order_ID, Item_ID, Restaurant_ID) REFERENCES
ITEM_ORDERS(Food_Order_ID, Menu_Item_ID, Restaurant_ID)
);

-- Employees and Roles

CREATE TABLE EMPLOYEE (
    Employee_ID VARCHAR(10) PRIMARY KEY,
    Start_Date DATE NOT NULL,
    Department VARCHAR(50) NOT NULL,
    Age INT NOT NULL,
    CHECK (Employee_ID REGEXP '^E[0-9]+$'),
    CHECK (Age > 17)
);

CREATE TABLE PLATFORM_MANAGER (
    Manager_ID VARCHAR(10) PRIMARY KEY,
    FOREIGN KEY (Manager_ID) REFERENCES EMPLOYEE(Employee_ID) ON DELETE
CASCADE
);
```

```
CREATE TABLE SUPPORT_AGENT (
   Agent_ID VARCHAR(10) PRIMARY KEY,
   FOREIGN KEY (Agent_ID) REFERENCES EMPLOYEE(Employee_ID) ON DELETE CASCADE
);

CREATE TABLE DELIVERY_COORDINATOR (
   Coordinator_ID VARCHAR(10) PRIMARY KEY,
   FOREIGN KEY (Coordinator_ID) REFERENCES EMPLOYEE(Employee_ID) ON DELETE
CASCADE
);

CREATE TABLE DELIVERY_ASSIGNMENT (
   Assignment_ID INT PRIMARY KEY,
   Driver_ID INT NOT NULL,
   Food_Order_ID INT NOT NULL,
   Coordinator_ID VARCHAR(10) NOT NULL,
   FOREIGN KEY (Driver_ID) REFERENCES DRIVER(Driver_ID) ON DELETE CASCADE,
   FOREIGN KEY (Food_Order_ID) REFERENCES FOOD_ORDER(Order_ID),
   FOREIGN KEY (Coordinator_ID) REFERENCES
DELIVERY_COORDINATOR(Coordinator_ID) ON DELETE CASCADE
);

--VIEWS--
-- LoyalCustomers: Which customers have placed orders consistently every month for the
past year?
CREATE VIEW LoyalCustomers AS
```

```sql
    SELECT c.Customer_ID, u.First_Name, u.Middle_Name, u.Last_Name
    FROM CUSTOMER c
        JOIN USERS u ON c.Customer_ID = u.User_ID
        JOIN FOOD_ORDER fo ON c.Customer_ID = fo.Customer_ID
    WHERE fo.Order_Date >= CURDATE() - INTERVAL 1 YEAR
    GROUP BY c.Customer_ID, u.First_Name, u.Middle_Name, u.Last_Name
    HAVING COUNT(DISTINCT MONTH(fo.Order_Date)) = 12;

-- TopRatedRestaurants: Which restaurants have an average review rating of 4.5 or higher over the past six months?
CREATE VIEW TopRatedRestaurants AS
    SELECT rt.Restaurant_ID, rt.Name, AVG(r.Rating) AS AVG_Rating
  FROM RESTAURANT rt
        JOIN RESTAURANT_REVIEW rr ON rt.Restaurant_ID = rr.Restaurant_ID
   JOIN REVIEW r ON rr.Review_ID = r.Review_ID
    WHERE r.Review_Date >= CURDATE() - INTERVAL 6 MONTH
  GROUP BY rt.Restaurant_ID, rt.Name
  HAVING AVG(r.Rating) >= 4.5;

-- ActiveDrivers: Which delivery drivers have completed at least 20 deliveries within the last two weeks?
CREATE VIEW ActiveDrivers AS
    SELECT d.Driver_ID, u.First_Name, u.Middle_Name, u.Last_Name, COUNT(*) AS
Completed_Deliveries
  FROM DELIVERY_ASSIGNMENT da
        JOIN FOOD_ORDER fo ON da.Food_Order_ID = fo.Order_ID
   JOIN DRIVER d ON da.Driver_ID = d.Driver_ID
```

```sql
    JOIN USERS u ON d.Driver_ID = u.User_ID
        WHERE fo.Order_Date >= CURDATE() - INTERVAL 14 DAY
            AND fo.Delivery_Status = 'Delivered'
        GROUP BY d.Driver_ID, u.First_Name, u.Middle_Name, u.Last_Name
  HAVING COUNT(*) >= 20;


-- PopularMenuItems: What are the top 10 most frequently ordered menu items across all restaurants in the past three months?
CREATE VIEW PopularMenuItems AS
        SELECT mi.Item_ID, mi.Restaurant_ID, mi.Name AS Item_Name, r.Name AS Restaurant_Name, COUNT(*) AS Times_Ordered
  FROM ITEM_ORDERS o
            JOIN MENU_ITEMS mi ON o.Menu_Item_ID = mi.Item_ID AND o.Restaurant_ID = mi.Restaurant_ID
    JOIN FOOD_ORDER fo ON o.Food_Order_ID = fo.Order_ID
    JOIN RESTAURANT r ON mi.Restaurant_ID = r.Restaurant_ID
        WHERE fo.Order_Date >= CURDATE() - INTERVAL 3 MONTH
  GROUP BY mi.Item_ID, mi.Restaurant_ID, mi.Name, r.Name
        ORDER BY Times_Ordered DESC
  LIMIT 10;


-- ProminentOwners: Which restaurant owners manage multiple restaurants with a combined total of at least 50 orders in the past month?
CREATE VIEW ProminentOwners AS
        SELECT r.Owner_ID, u.First_Name, u.Middle_Name, u.Last_Name, COUNT(fo.Order_ID) AS Total_Orders, COUNT(DISTINCT r.Restaurant_ID) AS Number_of_Restaurants
  FROM RESTAURANT r
```

```
JOIN FOOD_ORDER fo ON r.Restaurant_ID = fo.Restaurant_ID
JOIN USERS u ON r.Owner_ID = u.User_ID
WHERE fo.Order_Date >= CURDATE() - INTERVAL 1 MONTH
GROUP BY r.Owner_ID, u.First_Name, u.Middle_Name, u.Last_Name
HAVING COUNT(DISTINCT r.Restaurant_ID) > 1
AND COUNT(fo.Order_ID) >= 50;


--QUERIES--

-- TopEarningDrivers: List the names and total earnings of the top five drivers.

SELECT u.User_Id AS Driver_ID, u.First_Name, u.Middle_Name, u.Last_Name, SUM(r.Fare) AS Total_Earnings
FROM RIDES r
JOIN USERS u ON r.Driver_ID = u.User_ID
GROUP BY u.User_Id, u.First_Name, u.Middle_Name, u.Last_Name
ORDER BY Total_Earnings DESC
LIMIT 5;


-- HighSpendingCustomers: Identify customers who have spent more than $1,000 and list their total expenditure.

SELECT c.Customer_ID, u.First_Name, u.Middle_Name, u.Last_Name, SUM(fo.Total_Amount) AS Total_Expenditure
FROM FOOD_ORDER fo
JOIN CUSTOMER c ON fo.Customer_ID = c.Customer_ID
JOIN USERS u ON c.Customer_ID = u.User_ID
GROUP BY c.Customer_ID, u.First_Name, u.Middle_Name, u.Last_Name
HAVING SUM(fo.Total_Amount) > 1000;
```

-- FrequentReviewers: Find customers who have left at least 10 reviews and their average review rating.

```
SELECT c.Customer_ID, u.First_Name, u.Middle_Name, u.Last_Name,
COUNT(r.Review_ID) AS Total_Reviews, AVG(r.Rating) AS AVG_Rating
FROM CUSTOMER c
        JOIN USERS u ON c.Customer_ID = u.User_ID
  JOIN (
                SELECT Customer_ID, Review_ID
    FROM RESTAURANT_REVIEW
    UNION ALL
    SELECT Customer_ID, Review_ID
    FROM ITEM_REVIEW
  ) AS all_reviews ON c.Customer_ID = all_reviews.Customer_ID
  JOIN REVIEW r ON all_reviews.Review_ID = r.Review_ID
GROUP BY c.Customer_ID, u.First_Name, u.Middle_Name, u.Last_Name
HAVING COUNT(r.Review_ID) >= 10;
```

-- InactiveRestaurants: List restaurants that have not received any orders in the past months.

```
SELECT r.Restaurant_ID, r.Name
FROM RESTAURANT r
        LEFT JOIN FOOD_ORDER fo ON r.Restaurant_ID = fo.Restaurant_ID
                AND fo.Order_Date >= CURDATE() - INTERVAL 1 MONTH
WHERE fo.Order_ID IS NULL;
```

-- PeakOrderDay: Identify the day of the week with the highest number of orders in the past month.

```
SELECT DAYNAME(Order_Date) AS Day_of_the_Week, COUNT(*) AS Total_Orders
FROM FOOD_ORDER
WHERE Order_Date >= CURDATE() - INTERVAL 1 MONTH
GROUP BY DAYNAME(Order_Date)
ORDER BY Total_Orders DESC
LIMIT 1;


-- HighEarningRestaurants: Find the top three restaurants with the highest total revenue in the past year.
SELECT r.Restaurant_ID, r.Name, SUM(fo.Total_Amount) AS Total_Revenue
FROM FOOD_ORDER fo
	JOIN RESTAURANT r ON fo.Restaurant_ID = r.Restaurant_ID
WHERE fo.Order_Date >= CURDATE() - INTERVAL 1 YEAR
GROUP BY r.Restaurant_ID, r.Name
ORDER BY Total_Revenue DESC
LIMIT 3;


-- PopularCuisineType: Identify the most frequently ordered cuisine type in the past six months.
SELECT r.Cuisine, COUNT(fo.Order_ID) AS Number_of_Orders
FROM FOOD_ORDER fo
	JOIN RESTAURANT r ON fo.Restaurant_ID = r.Restaurant_ID
WHERE fo.Order_Date >= CURDATE() - INTERVAL 6 MONTH
GROUP BY r.Cuisine
ORDER BY Number_of_Orders DESC
LIMIT 1;
```

```sql
-- LongestRideRoutes: Identify the top five ride routes with the longest travel distances.

SELECT r.Pickup_Location, r.Drop_Off_Location, r.Distance

FROM RIDES r

ORDER BY r.Distance DESC

LIMIT 5;


-- DriverRideCounts: Display the total number of rides delivered by each driver in the past three months.

SELECT r.Driver_ID, u.First_Name, u.Middle_Name, u.Last_Name, COUNT(r.Ride_ID) AS Total_Rides

FROM RIDES r

    JOIN USERS u ON r.Driver_ID = u.User_ID

WHERE r.Pickup_Time >= CURDATE() - INTERVAL 3 MONTH

GROUP BY r.Driver_ID, u.First_Name, u.Middle_Name, u.Last_Name;


-- UndeliveredOrders: Find all orders that were not delivered within the promised time window and their delay durations.

SELECT fo.Order_ID, fo.Order_Date, fo.Promised_Delivery_Time, fo.Actual_Delivery_Time, TIMESTAMPDIFF(MINUTE, fo.Promised_Delivery_Time, fo.Actual_Delivery_Time) AS Minutes_Delayed

FROM FOOD_ORDER fo

WHERE fo.Actual_Delivery_Time > fo.Promised_Delivery_Time;


-- MostCommonPaymentMethods: Identify the most frequently used payment method on the platform for both rides and food orders.

SELECT Payment_Method, COUNT(*) AS Usage_Count

FROM (

    SELECT r.Payment_Method
```

```
FROM RIDES r
UNION ALL
SELECT fo.Payment_Method
FROM Food_Order fo
) AS payment_methods
GROUP BY Payment_Method
ORDER BY Usage_Count DESC
LIMIT 1;
```

```
-- MultiRoleUsers: Identify users who simultaneously serve as both Drivers and Restaurant Owners, along with their details.
SELECT u.User_ID, u.First_Name, u.Middle_Name, u.Last_Name, u.Address, u.Date_of_Birth, u.Gender, d.Vehicle, COUNT(r.Restaurant_ID) AS Number_of_Restaurants_Owned
FROM USERS u
    JOIN DRIVER d ON u.User_ID = d.Driver_ID
    JOIN RESTAURANT_OWNER ro ON u.User_ID = ro.Owner_ID
  JOIN RESTAURANT r ON ro.Owner_ID = r.Owner_ID
GROUP BY u.User_ID, u.First_Name, u.Middle_Name, u.Last_Name, u.Address, u.Date_of_Birth, u.Gender, d.Vehicle;
```

```
-- DriverVehicleTypes: Display the distribution of drivers by vehicle type (Sedan, SUV, and etc.), including the total number for each type.
SELECT d.Vehicle, COUNT(*) AS Total_Drivers
FROM DRIVER d
GROUP BY d.Vehicle;
```