Swiggy Database Schema Design Guide - Business Analyst
Project Name: Swiggy Database Schema Design
Date: 01-09-2024

Project Difficulty Level : Junior / Fresher Analyst

By:

@Learn_withmegtd

@**learnwithmegtd@gmail.com**

OVERVIEW

This guidebook provides an in-depth look into designing a database schema for Swiggy, a leading food delivery service. The aim is to equip students with the knowledge and skills to create a robust and scalable database that efficiently handles various aspects of the food delivery process, including customer interactions, restaurant management, and order processing.

GOALS

- **Design a Comprehensive Schema**: Learn how to develop a detailed schema that captures all essential entities and relationships involved in Swiggy's operations.
- **Implement Best Practices**: Understand normalization techniques and best practices to ensure data integrity, reduce redundancy, and support scalability.

SPECIFICATIONS

- **Core Entities**: The schema includes entities such as Customers, Restaurants, Menus, Orders, Deliveries, Payments, Reviews, Addresses, Coupons, and Categories.
- **Relationships and Constraints**: Define foreign keys, constraints, and relationships between tables to maintain data consistency and integrity.
- **Normalization**: Apply principles of normalization to structure the database efficiently, reducing duplication and ensuring reliable data.

MILESTONES

- **Design Phase**
  Develop an initial draft of the database schema. Identify and define all core entities and their attributes. Establish primary and foreign key relationships.
- **Review and Refinement**
  Present the schema to stakeholders or peers for feedback. Make necessary adjustments to improve the schema based on insights and suggestions.
- **Implementation and Testing**
  Implement the schema in a database management system. Perform thorough testing to ensure that data operations, such as queries, updates, and deletions, are functioning correctly.
- **Deployment and Documentation**
  Finalize the schema and prepare it for deployment in a production environment. Create detailed documentation outlining the schema design, relationships, and any special considerations for future maintenance.

This guidebook serves as a comprehensive resource for students to understand and implement a practical database schema design for a real-world application like Swiggy. It covers key concepts, design principles, and practical steps necessary for successful database management. This guide book also provides the interview simulation and sql query for better understanding.
Lacking : - visual ERD, Data flow ..

Swiggy Database Schema Design

1. Core Entities and Relationships

To effectively represent Swiggy's operations, the key entities in the food delivery process must be identified and clearly defined. The primary entities include:

- **Customers**
- **Restaurants**
- **Menus and Menu Items**
- **Orders and Order Items**
- **Deliveries**
- **Delivery Personnel**
- **Payments**
- **Reviews**
- **Addresses**
- **Coupons/Promotions**
- **Categories**

2. Detailed Table Definitions

A. Customers

- **Table Name:** customers
- **Columns:**
  - `customer_id` (INT, PRIMARY KEY, AUTO_INCREMENT)
  - `first_name` (VARCHAR(50))
  - `last_name` (VARCHAR(50))
  - `email` (VARCHAR(100), UNIQUE, NOT NULL)
  - `phone_number` (VARCHAR(15), UNIQUE, NOT NULL)
  - `password` (VARCHAR(255), NOT NULL) — Store hashed passwords for security.
  - `signup_date` (DATETIME, DEFAULT CURRENT_TIMESTAMP)
  - `last_login` (DATETIME)

Explanation:
The `customer_id` serves as the primary key to uniquely identify each customer. Unique constraints on email and phone number prevent duplicates, and passwords are stored in a hashed format for security.

B. Addresses

- **Table Name:** addresses
- **Columns:**
  - `address_id` (INT, PRIMARY KEY, AUTO_INCREMENT)
  - `customer_id` (INT, FOREIGN KEY REFERENCES customers(customer_id) ON DELETE CASCADE)
  - `address_line1` (VARCHAR(255), NOT NULL)
  - `address_line2` (VARCHAR(255))
  - `city` (VARCHAR(100), NOT NULL)
  - `state` (VARCHAR(100), NOT NULL)
  - `zip_code` (VARCHAR(10), NOT NULL)
  - `country` (VARCHAR(100), NOT NULL)
  - `is_primary` (BOOLEAN, DEFAULT FALSE)

Explanation:
This table establishes a one-to-many relationship with the `customers` table, linking each address to a specific customer. The foreign key constraint ensures that when a customer is deleted, all their associated addresses are also removed.

C. Restaurants

- **Table Name:** restaurants
- **Columns:**
  - `restaurant_id` (INT, PRIMARY KEY, AUTO_INCREMENT)
  - `name` (VARCHAR(100), NOT NULL)
  - `owner_name` (VARCHAR(100))
  - `phone_number` (VARCHAR(15), UNIQUE)
  - `email` (VARCHAR(100), UNIQUE)
  - `address` (VARCHAR(255))
  - `rating` (DECIMAL(3,2), DEFAULT 0.00)
  - `created_at` (DATETIME, DEFAULT CURRENT_TIMESTAMP)
  - `updated_at` (DATETIME, DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP)

Explanation:
The `restaurant_id` serves as the primary key for unique identification. The `rating` column captures the average rating based on customer reviews, and timestamps track the creation and updates of records.

D. Categories

- **Table Name:** categories
- **Columns:**
  - `category_id` (INT, PRIMARY KEY, AUTO_INCREMENT)
  - `category_name` (VARCHAR(50), UNIQUE, NOT NULL)

Explanation:
This table categorizes restaurants into various types like Italian, Chinese, etc., which can help in filtering and searching.

E. Restaurant Categories (Junction Table)

- **Table Name:** restaurant_categories
- **Columns:**
  - `restaurant_id` (INT, FOREIGN KEY REFERENCES restaurants(restaurant_id) ON DELETE CASCADE)
  - `category_id` (INT, FOREIGN KEY REFERENCES categories(category_id) ON DELETE CASCADE)
- **Primary Key:** (restaurant_id, category_id)

Explanation:
This table manages the many-to-many relationship between restaurants and categories, allowing a restaurant to belong to multiple categories and a category to include multiple restaurants.

F. Menus

- **Table Name:** menus
- **Columns:**
  - `menu_id` (INT, PRIMARY KEY, AUTO_INCREMENT)
  - `restaurant_id` (INT, FOREIGN KEY REFERENCES restaurants(restaurant_id) ON DELETE CASCADE)
  - `menu_name` (VARCHAR(100), NOT NULL)
  - `description` (TEXT)

Explanation:
Each restaurant can offer multiple menus (e.g., breakfast, lunch, dinner), forming a one-to-many relationship between `restaurants` and `menus`.

G. Menu Items

- **Table Name:** menu_items
- **Columns:**
  - `item_id` (INT, PRIMARY KEY, AUTO_INCREMENT)
  - `menu_id` (INT, FOREIGN KEY REFERENCES menus(menu_id) ON DELETE CASCADE)
  - `item_name` (VARCHAR(100), NOT NULL)
  - `description` (TEXT)
  - `price` (DECIMAL(10,2), NOT NULL)
  - `available` (BOOLEAN, DEFAULT TRUE)
  - `category_id` (INT, FOREIGN KEY REFERENCES categories(category_id))

Explanation:
This table links each menu to its specific items, including details like price and availability. The `available` column indicates if an item is currently offered.

H. Orders

- **Table Name:** orders
- **Columns:**
  - `order_id` (INT, PRIMARY KEY, AUTO_INCREMENT)
  - `customer_id` (INT, FOREIGN KEY REFERENCES customers(customer_id) ON DELETE SET NULL)
  - `restaurant_id` (INT, FOREIGN KEY REFERENCES restaurants(restaurant_id) ON DELETE SET NULL)
  - `address_id` (INT, FOREIGN KEY REFERENCES addresses(address_id) ON DELETE SET NULL)
  - `order_status` (ENUM('Placed', 'Confirmed', 'Preparing', 'Dispatched', 'Delivered', 'Cancelled'), DEFAULT 'Placed')
  - `order_date` (DATETIME, DEFAULT CURRENT_TIMESTAMP)
  - `total_amount` (DECIMAL(10,2), NOT NULL)
  - `payment_id` (INT, FOREIGN KEY REFERENCES payments(payment_id))

Explanation:
This table captures all orders, with foreign keys linking to the customer, restaurant, delivery address, and payment details. The `order_status` column tracks the progress of each order.

I. Order Items

- **Table Name:** order_items
- **Columns:**
  - `order_item_id` (INT, PRIMARY KEY, AUTO_INCREMENT)
  - `order_id` (INT, FOREIGN KEY REFERENCES orders(order_id) ON DELETE CASCADE)
  - `item_id` (INT, FOREIGN KEY REFERENCES menu_items(item_id) ON DELETE SET NULL)
  - `quantity` (INT, NOT NULL)
  - `price` (DECIMAL(10,2), NOT NULL)

Explanation:
This table captures the items included in each order, allowing for multiple items per order. The `price` column records the cost at the time of order, accounting for any future price changes.

J. Payments

- **Table Name:** payments
- **Columns:**
  - `payment_id` (INT, PRIMARY KEY, AUTO_INCREMENT)
  - `customer_id` (INT, FOREIGN KEY REFERENCES customers(customer_id) ON DELETE SET NULL)
  - `order_id` (INT, FOREIGN KEY REFERENCES orders(order_id) ON DELETE SET NULL)
  - `payment_method` (ENUM('Credit Card', 'Debit Card', 'Net Banking', 'UPI', 'COD'), NOT NULL)
  - `payment_status` (ENUM('Pending', 'Completed', 'Failed'), DEFAULT 'Pending')
  - `payment_date` (DATETIME, DEFAULT CURRENT_TIMESTAMP)
  - `amount` (DECIMAL(10,2), NOT NULL)

Explanation:
This table manages payment details for each order, including method and status, to handle various payment scenarios.

K. Delivery Personnel

- **Table Name:** delivery_personnel
- **Columns:**
  - `delivery_person_id` (INT, PRIMARY KEY, AUTO_INCREMENT)
  - `name` (VARCHAR(100), NOT NULL)
  - `phone_number` (VARCHAR(15), UNIQUE, NOT NULL)
  - `email` (VARCHAR(100), UNIQUE)
  - `vehicle_details` (VARCHAR(100))
  - `status` (ENUM('Available', 'Busy', 'Offline'), DEFAULT 'Offline')
  - `rating` (DECIMAL(3,2), DEFAULT 0.00)
  - `assigned_orders` (INT, DEFAULT 0)

Explanation:
The `status` column reflects whether a delivery person is available, and `rating` tracks performance based on customer feedback.

L. Deliveries

- **Table Name:** deliveries
- **Columns:**
  - `delivery_id` (INT, PRIMARY KEY, AUTO_INCREMENT)
  - `order_id` (INT, FOREIGN KEY REFERENCES orders(order_id) ON DELETE CASCADE)
  - `delivery_person_id` (INT, FOREIGN KEY REFERENCES delivery_personnel(delivery_person_id) ON DELETE SET NULL)
  - `delivery_status` (ENUM('Assigned', 'Picked', 'On the way', 'Delivered', 'Cancelled', 'Late'), DEFAULT 'Assigned')
  - `pickup_time` (DATETIME)
  - `delivery_time` (DATETIME)

Explanation:
Each delivery is linked to an order and a delivery person, with `delivery_status` tracking the process from assignment to completion.

M. Reviews (continued)

- **Columns:**
  - `restaurant_id` (INT, FOREIGN KEY REFERENCES restaurants(restaurant_id) ON DELETE SET NULL)
  - `order_id` (INT, FOREIGN KEY REFERENCES orders(order_id) ON DELETE SET NULL)
  - `rating` (INT, CHECK (rating BETWEEN 1 AND 5), NOT NULL)
  - `review_text` (TEXT)
  - `created_at` (DATETIME, DEFAULT CURRENT_TIMESTAMP)
  - `updated_at` (DATETIME, DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP)

Explanation:
This table links reviews to specific customers, restaurants, and orders. The `rating` field ensures that the rating value is within the valid range, and timestamps manage the review's creation and updates.

N. Coupons/Promotions

- **Table Name:** coupons
- **Columns:**
  - `coupon_id` (INT, PRIMARY KEY, AUTO_INCREMENT)
  - `code` (VARCHAR(50), UNIQUE, NOT NULL)
  - `description` (TEXT)
  - `discount_percentage` (DECIMAL(5,2), NOT NULL)
  - `expiry_date` (DATETIME, NOT NULL)
  - `usage_limit` (INT, DEFAULT 1)
- **Table Name:** customer_coupons
- **Columns:**
  - `customer_coupon_id` (INT, PRIMARY KEY, AUTO_INCREMENT)
  - `coupon_id` (INT, FOREIGN KEY REFERENCES coupons(coupon_id) ON DELETE CASCADE)
  - `customer_id` (INT, FOREIGN KEY REFERENCES customers(customer_id) ON DELETE CASCADE)
  - `used` (BOOLEAN, DEFAULT FALSE)

Explanation:
The `coupons` table stores details about each coupon or promotion, including its code and discount. The `customer_coupons` table tracks which coupons have been used by which customers, enforcing usage limits and handling coupon redemption.

3. Entity-Relationship (ER) Diagram Overview

A visual ER diagram would provide a clearer view, but here's a textual overview:

- **Customers** can have multiple **Addresses**.
- **Customers** can place multiple **Orders**.
- **Orders** are associated with one **Restaurant**, one **Address**, and one **Payment**.
- **Orders** can contain multiple **Order Items**, each linked to a **Menu Item**.
- **Restaurants** can belong to multiple **Categories** through the `restaurant_categories` table.
- **Menus** belong to **Restaurants**, and **Menu Items** belong to **Menus**.
- **Deliveries** are linked to **Orders** and **Delivery Personnel**.
- **Customers** can write multiple **Reviews** for **Restaurants** based on **Orders**.
- **Customers** can use multiple **Coupons** through the `customer_coupons` table.

4. Relationship Explanations

One-to-Many Relationships:

- **Customers ↔ Addresses:** Each customer can have multiple addresses.
- **Restaurants ↔ Menus:** A restaurant can offer multiple menus.
- **Menus ↔ Menu Items:** Each menu can have multiple items.
- **Customers ↔ Orders:** A customer can place multiple orders.
- **Orders ↔ Order Items:** Each order can include multiple items.
- **Restaurants ↔ Orders:** A restaurant can receive multiple orders.
- **Customers ↔ Reviews:** A customer can write multiple reviews.
- **Restaurants ↔ Reviews:** A restaurant can have multiple reviews.

Many-to-Many Relationships:

- **Restaurants ↔ Categories:** Managed through the `restaurant_categories` table.
- **Customers ↔ Coupons:** Handled via the `customer_coupons` table.

One-to-One Relationships:

- **Orders ↔ Payments:** Each order is linked to a single payment record.
- **Orders ↔ Deliveries:** Each order has one associated delivery record.

Foreign Keys and Referential Actions:

- **ON DELETE CASCADE:** Ensures related records (e.g., addresses, orders) are removed when a parent record (e.g., customer) is deleted.
- **ON DELETE SET NULL:** Maintains historical data by setting foreign keys to NULL if the referenced parent record is deleted.

5. Sample SQL Table Creation Scripts

Below are sample SQL scripts to create some of the key tables with relationships:

```sql
-- Customers Table
CREATE TABLE customers (
    customer_id INT AUTO_INCREMENT PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50),
    email VARCHAR(100) NOT NULL UNIQUE,
    phone_number VARCHAR(15) NOT NULL UNIQUE,
    password VARCHAR(255) NOT NULL,
    signup_date DATETIME DEFAULT CURRENT_TIMESTAMP,
    last_login DATETIME
);

-- Addresses Table
CREATE TABLE addresses (
    address_id INT AUTO_INCREMENT PRIMARY KEY,
    customer_id INT,
```

```
    address_line1 VARCHAR(255) NOT NULL,
    address_line2 VARCHAR(255),
    city VARCHAR(100) NOT NULL,
    state VARCHAR(100) NOT NULL,
    zip_code VARCHAR(10) NOT NULL,
    country VARCHAR(100) NOT NULL,
    is_primary BOOLEAN DEFAULT FALSE,
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id) ON DELETE CASCADE
);

-- Restaurants Table
CREATE TABLE restaurants (
    restaurant_id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    owner_name VARCHAR(100),
    phone_number VARCHAR(15) UNIQUE,
    email VARCHAR(100) UNIQUE,
    address VARCHAR(255),
    rating DECIMAL(3,2) DEFAULT 0.00,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    updated_at DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

```
-- Categories Table
CREATE TABLE categories (
    category_id INT AUTO_INCREMENT PRIMARY KEY,
    category_name VARCHAR(50) NOT NULL UNIQUE
);

-- Restaurant Categories Junction Table
CREATE TABLE restaurant_categories (
    restaurant_id INT,
    category_id INT,
    PRIMARY KEY (restaurant_id, category_id),
    FOREIGN KEY (restaurant_id) REFERENCES restaurants(restaurant_id) ON DELETE CASCADE,
    FOREIGN KEY (category_id) REFERENCES categories(category_id) ON DELETE CASCADE
);

-- Menus Table
CREATE TABLE menus (
    menu_id INT AUTO_INCREMENT PRIMARY KEY,
    restaurant_id INT,
    menu_name VARCHAR(100) NOT NULL,
    description TEXT,
    FOREIGN KEY (restaurant_id) REFERENCES restaurants(restaurant_id) ON DELETE CASCADE
);
```

```
-- Menu Items Table
CREATE TABLE menu_items (
    item_id INT AUTO_INCREMENT PRIMARY KEY,
    menu_id INT,
    item_name VARCHAR(100) NOT NULL,
    description TEXT,
    price DECIMAL(10,2) NOT NULL,
    available BOOLEAN DEFAULT TRUE,
    category_id INT,
    FOREIGN KEY (menu_id) REFERENCES menus(menu_id) ON DELETE CASCADE,
    FOREIGN KEY (category_id) REFERENCES categories(category_id)
);
```



```
-- Orders Table
CREATE TABLE orders (
    order_id INT AUTO_INCREMENT PRIMARY KEY,
    customer_id INT,
    restaurant_id INT,
    address_id INT,
    order_status ENUM('Placed', 'Confirmed', 'Preparing', 'Dispatched', 'Delivered', 'Cancelled')
DEFAULT 'Placed',
    order_date DATETIME DEFAULT CURRENT_TIMESTAMP,
    total_amount DECIMAL(10,2) NOT NULL,
    payment_id INT,
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id) ON DELETE SET NULL,
```

```
    FOREIGN KEY (restaurant_id) REFERENCES restaurants(restaurant_id) ON DELETE SET NULL,
    FOREIGN KEY (address_id) REFERENCES addresses(address_id) ON DELETE SET NULL,
    FOREIGN KEY (payment_id) REFERENCES payments(payment_id)
);

-- Order Items Table
CREATE TABLE order_items (
    order_item_id INT AUTO_INCREMENT PRIMARY KEY,
    order_id INT,
    item_id INT,
    quantity INT NOT NULL,
    price DECIMAL(10,2) NOT NULL,
    FOREIGN KEY (order_id) REFERENCES orders(order_id) ON DELETE CASCADE,
    FOREIGN KEY (item_id) REFERENCES menu_items(item_id) ON DELETE SET NULL
);

-- Payments Table
CREATE TABLE payments (
    payment_id INT AUTO_INCREMENT PRIMARY KEY,
    customer_id INT,
    order_id INT,
    payment_method ENUM('Credit Card', 'Debit Card', 'Net Banking', 'UPI', 'COD') NOT NULL,
    payment_status ENUM('Pending', 'Completed', 'Failed') DEFAULT 'Pending',
    payment_date DATETIME DEFAULT CURRENT_TIMESTAMP,
```

```
    amount DECIMAL(10,2) NOT NULL,
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id) ON DELETE SET NULL,
    FOREIGN KEY (order_id) REFERENCES orders(order_id) ON DELETE SET NULL
);

-- Delivery Personnel Table
CREATE TABLE delivery_personnel (
    delivery_person_id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    phone_number VARCHAR(15) UNIQUE NOT NULL,
    email VARCHAR(100) UNIQUE,
    vehicle_details VARCHAR(100),
    status ENUM('Available', 'Busy', 'Offline') DEFAULT 'Offline',
    rating DECIMAL(3,2) DEFAULT 0.00,
    assigned_orders INT DEFAULT 0
);

-- Deliveries Table
CREATE TABLE deliveries (
    delivery_id INT AUTO_INCREMENT PRIMARY KEY,
    order_id INT,
    delivery_person_id INT,
    delivery_status ENUM('Assigned', 'Picked', 'On the way', 'Delivered', 'Cancelled', 'Late') DEFAULT
'Assigned',
    pickup_time DATETIME,
```

```
    delivery_time DATETIME,
    FOREIGN KEY (order_id) REFERENCES orders(order_id) ON DELETE CASCADE,
    FOREIGN KEY (delivery_person_id) REFERENCES delivery_personnel(delivery_person_id) ON
DELETE SET NULL
);

-- Reviews Table
CREATE TABLE reviews (
    review_id INT AUTO_INCREMENT PRIMARY KEY,
    customer_id INT,
    restaurant_id INT,
    order_id INT,
    rating INT NOT NULL CHECK (rating BETWEEN 1 AND 5),
    review_text TEXT,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    updated_at DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id) ON DELETE SET NULL,
    FOREIGN KEY (restaurant_id) REFERENCES restaurants(restaurant_id) ON DELETE SET NULL,
    FOREIGN KEY (order_id) REFERENCES orders(order_id) ON DELETE SET NULL
);

-- Coupons Table
CREATE TABLE coupons (
    coupon_id INT AUTO_INCREMENT PRIMARY KEY,
    code VARCHAR(50) NOT NULL UNIQUE,
```

```
    description TEXT,
    discount_percentage DECIMAL(5,2) NOT NULL,
    expiry_date DATETIME NOT NULL,
    usage_limit INT DEFAULT 1
);

-- Customer Coupons Junction Table
CREATE TABLE customer_coupons (
    customer_coupon_id INT AUTO_INCREMENT PRIMARY KEY,
    coupon_id INT,
    customer_id INT,
    used BOOLEAN DEFAULT FALSE,
    FOREIGN KEY (coupon_id) REFERENCES coupons(coupon_id) ON DELETE CASCADE,
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id) ON DELETE CASCADE
);
```


6. Design Considerations

- **Normalization:** The schema is normalized up to the Third Normal Form (3NF) to eliminate data redundancy and ensure data integrity.
- **Scalability:**
    - **Partitioning:** For large tables like `orders` and `order_items`, consider partitioning based on date ranges or regions to enhance performance.
    - **Indexes:** Implement indexes on frequently queried columns such as `customer_id`, `restaurant_id`, `order_date`, and foreign keys to speed up data retrieval.
- **Data Integrity:**
    - **Constraints:** Use `CHECK` constraints (e.g., for ratings) and `NOT NULL` constraints to enforce data validity.
    - **Transactions:** Ensure that operations affecting multiple tables (e.g., placing an order) are handled within transactions to maintain consistency.

- **Security:**
    - **Sensitive Data Protection:** Hash passwords and consider encrypting sensitive information like payment details.
    - **Access Controls:** Implement role-based access controls to restrict data access based on user roles.
- **Performance Optimization:**
    - **Caching:** Utilize caching mechanisms for frequently accessed data such as restaurant menus.
    - **Efficient Queries:** Design queries to minimize resource consumption, leveraging joins and aggregations effectively.

7. Example Use Cases and Queries

A simple analysis on the data for a customer churn prediction.

A. Analyzing Customer Churn

Objective: Identify customers who have not placed an order in the last 30 days.

Steps and SQL Queries:

Define Active and Churned Customers:
sql

```
-- Churned Customers: No orders in the last 30 days

SELECT c.customer_id, c.first_name, c.last_name, MAX(o.order_date) AS
last_order_date

FROM customers c

LEFT JOIN orders o ON c.customer_id = o.customer_id

GROUP BY c.customer_id, c.first_name, c.last_name

HAVING MAX(o.order_date) < (CURDATE() - INTERVAL 30 DAY) OR
MAX(o.order_date) IS NULL;
```

Analyze Order Frequency:

- - - sql

```
-- Calculate total orders and average time between orders

SELECT c.customer_id, c.first_name, c.last_name, COUNT(o.order_id) AS
total_orders,

       AVG(DATEDIFF(o.order_date,
previous_orders.previous_order_date)) AS avg_days_between_orders

FROM customers c

JOIN orders o ON c.customer_id = o.customer_id

LEFT JOIN (

    SELECT customer_id, order_date,

           LAG(order_date) OVER (PARTITION BY customer_id ORDER BY
order_date) AS previous_order_date

    FROM orders
) AS previous_orders ON o.customer_id = previous_orders.customer_id

GROUP BY c.customer_id, c.first_name, c.last_name;
```

1.

Assess Customer Satisfaction:

sql

```
-- Identify customers with low ratings

SELECT c.customer_id, c.first_name, c.last_name, AVG(r.rating) AS
avg_rating

FROM customers c

JOIN reviews r ON c.customer_id = r.customer_id

GROUP BY c.customer_id, c.first_name, c.last_name

HAVING AVG(r.rating) < 3;
```

2.

Combine Metrics for Churn Prediction:
sql

```
-- Flag high-risk churn customers
SELECT c.customer_id, c.first_name, c.last_name
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id
LEFT JOIN reviews r ON c.customer_id = r.customer_id
GROUP BY c.customer_id, c.first_name, c.last_name
HAVING MAX(o.order_date) < (CURDATE() - INTERVAL 30 DAY)
   OR AVG(r.rating) < 3;
```


Analyzing Customer Churn:

Churn Identification:

- **Customer Behavior:** Look for customers who haven't placed recent orders or have shown declining engagement. Low order frequency or consistently low ratings can indicate higher risk of churn.

Order Frequency Analysis:

- **Engagement Patterns:** Track how often customers place orders. A decrease in order frequency might signal potential disengagement or dissatisfaction.

Satisfaction Metrics:

- **Customer Feedback:** Analyze ratings and reviews. Low ratings or frequent complaints may suggest dissatisfaction, which can increase the likelihood of a customer churning.

B. Staying Updated with Data Analytics and SQL Trends

Response:

"To stay updated with the latest trends and technologies in data analytics and SQL, I adopt a multifaceted approach:

1. **Continuous Learning:**
    - *I regularly enroll in online courses on platforms like Coursera, Udemy, and LinkedIn Learning to learn about new SQL features, advanced analytics techniques, and emerging data visualization tools.*
2. **Industry Literature:**
    - *I follow leading data analytics blogs such as Towards Data Science, SQLServerCentral, and DataCamp to gain insights into best practices and innovative methodologies.*
3. **Professional Networking:**
    - *I actively participate in forums like Stack Overflow, Reddit's r/sql, and join LinkedIn groups focused on data analytics. Engaging with these communities allows me to exchange knowledge and stay informed about real-world applications.*
4. **Hands-On Practice:**
    - *I work on personal projects and contribute to open-source initiatives on GitHub. This practical experience helps me apply new concepts and tools in a controlled environment.*
5. **Conferences and Webinars:**
    - *I attend webinars, workshops, and conferences such as the PASS Summit and Data Science Conferences to learn from industry experts and network with peers.*
6. **Certification and Specialization:**
    - *I pursue certifications like Microsoft's Azure Data Scientist or Google's Data Analytics Professional to validate my skills and stay committed to professional growth.*
7. **Reading Research Papers and Case Studies:**
    - *I read academic papers and case studies to understand the theoretical underpinnings and practical implementations of advanced data analytics techniques.*

This comprehensive strategy ensures that I remain proficient and adaptable in the rapidly evolving field of data analytics and SQL."

7. Summary of the Database Schema

Normalization:

- **Data Integrity:** Ensures the database structure reduces redundancy and maintains data consistency.

Scalability:

- **Growth Management:** Designed to manage increasing volumes of transactions and data effectively.

Security:

- **Data Protection:** Incorporates best practices for safeguarding data and controlling access.

Performance Optimization:

- **Efficiency:** Employs indexing and optimized query design to enhance performance and query execution.

---

8. Visual Representation

For a comprehensive understanding, a visual ER (Entity-Relationship) diagram is beneficial. Tools like MySQL Workbench, Lucidchart, or Draw.io can be used to create detailed ER diagrams that illustrate the relationships and structure of the database schema. Use tableau or Power BI for visually analyzing and representing the data for analysis

---

INTERVIEW SIMULATION - SQL ROUND

Interviewer: Hi gtd , welcome to the interview for the Business Analyst role at Swiggy. To start, could you briefly introduce yourself and explain why you're interested in this position?

Interviewer: Great, let's dive into some SQL-related questions since that will be a significant part of your role.

SQL Questions

1. **Basic SQL Querying**:
    - Imagine you have a table called `orders` with columns: `order_id`, `customer_id`, `restaurant_id`, `order_date`, `total_amount`. How would you write a query to find the total number of orders placed by each customer?
2. **Data Aggregation**:
    - Suppose we want to see the average order value per restaurant. How would you write that query using the `orders` table?
3. **Joins**:
    - You have two tables: `customers` (`customer_id`, `customer_name`, `address`) and `orders` (`order_id`, `customer_id`, `restaurant_id`, `order_date`, `total_amount`). How would you write a query to fetch the customer names and their respective order totals?
4. **Subqueries**:
    - How would you find the customers who have placed more than 10 orders? Assume there is a table `customers` with `customer_id`, `customer_name`.
5. **Case Statement**:
    - Write a query that classifies orders as 'High', 'Medium', or 'Low' based on the `total_amount` (e.g., greater than 1000 is 'High', between 500 and 1000 is 'Medium', and less than 500 is 'Low').
6. **Performance Considerations**:
    - If a table has millions of rows, what indexing strategies might you use to improve query performance? How would you decide which columns to index?
7. **Handling Nulls**:
    - How would you modify a query to handle cases where certain fields in your results may be `NULL`? For example, how would you handle a scenario where some `order_date` fields might be `NULL` in your results?
8. **Complex Joins**:

- Let's say you have three tables: `orders`, `restaurants` (with `restaurant_id`, `restaurant_name`), and `deliveries` (with `order_id`, `delivery_person_id`, `delivery_status`). How would you write a query to find all restaurants that have never had a late delivery? Assume `delivery_status = 'Late'` indicates a late delivery.

9. **Database Design**:
    - Swiggy collects a vast amount of data. Suppose you were tasked with designing a database schema to track customer reviews for restaurants. What tables would you create, and how would they relate to each other?
10. **Data Quality**:
    - How would you ensure data quality and consistency when dealing with multiple sources of data in your SQL queries?

Interviewer: Thank you for your responses. Let's shift focus for a moment. Can you describe a situation where you used data analysis to solve a business problem? How did you approach the problem, and what tools did you use?

Interviewer: Excellent. Now, thinking specifically about Swiggy, how would you approach analyzing customer churn using SQL? What data would you need, and how would you structure your analysis?

Interviewer: Lastly, how do you stay updated with the latest trends and technologies in data analytics and SQL?

Interviewer: Thank you, gtd. That concludes our interview. Do you have any questions for me?

End of Interview Simulation

SQL - ANSWERS

1. Counting Orders by Customer

```
SELECT customer_id, COUNT(order_id) FROM orders GROUP BY customer_id;
```

2. Average Order Value per Restaurant

```
SELECT restaurant_id, AVG(total_amount) FROM orders GROUP BY restaurant_id;
```

3. Summing Order Amounts by Customer

```
SELECT c.customer_name, SUM(o.total_amount)
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_name;
```


4. Customers with More Than 10 Orders

```
SELECT customer_name, sorder
FROM (
    SELECT c.customer_id, c.customer_name, COUNT(o.order_id) AS sorder
    FROM customers c
    JOIN orders o ON c.customer_id = o.customer_id
    GROUP BY c.customer_name
) AS subquery
WHERE sorder > 10
ORDER BY sorder LIMIT 10;
```

5. Classifying Orders by Total Amount

```
SELECT order_id,
CASE
    WHEN total_amount > 1000 THEN 'HIGH'
    WHEN total_amount > 500 AND total_amount <= 1000 THEN 'Medium'
    WHEN total_amount <= 500 THEN 'Low'
END AS order_classification
FROM orders;
```

6. Indexing Strategy

Index Strategy Considerations: Index the columns that are frequently used in `WHERE`, `JOIN`, `ORDER BY`, and `GROUP BY` clauses. Additionally, consider creating composite indexes if multiple columns are often used together in queries. Avoid over-indexing as it can impact write performance.

7. Handling Nulls

Identify the root cause for missing data, it can be while insertion, storing, duplicating, or in any process and report it for solving the problem. Then implementing changes to work with current data,

Exclude NULL values from results when they are critical (`WHERE order_date IS NOT NULL`).

Fill NULL values with a default or average when dealing with non-critical data (`COALESCE(order_date, 'default_value')` or `COALESCE(total_amount, AVG(total_amount))`).

8. Restaurants with No Late Deliveries

```
SELECT r.restaurant_name, COUNT(d.delivery_status) AS status
FROM restaurants r
```

JOIN orders o ON o.restaurant_id = r.restaurant_id

JOIN deliveries d ON o.order_id = d.order_id

GROUP BY r.restaurant_name

HAVING COUNT(CASE WHEN d.delivery_status = 'Late' THEN 1 END) = 0

ORDER BY r.restaurant_name;

9. Database Design for Customer Reviews

refined schema:

- **Tables**:
  - `customers` (`customer_id`, `customer_name`, etc.)
  - `restaurants` (`restaurant_id`, `restaurant_name`, etc.)
  - `reviews` (`review_id`, `customer_id`, `restaurant_id`, `review_text`, `review_rating`, `created_on`, `updated_on`)

This structure keeps the relationships clear and allows for easy queries to find reviews by customer, restaurant, or both.

10. Ensuring Data Quality

the essential points:

- Proper table relationships via ERDs.
- Foreign keys, cascading, and triggers.
- Appropriate normalization and indexing.
- Documenting column information.

BONUS : **Analyzing Customer Churn**: “To analyze customer churn at Swiggy, I would begin by identifying key metrics like order frequency, average order value, and customer feedback scores. Using SQL, I would query the historical data to identify patterns in customer behavior leading up to churn. For instance, I might look for a decline in order frequency or satisfaction ratings over time. I would then segment customers based on these metrics and build a model to predict churn risk, enabling targeted interventions.”

Disclaimer : this is a not a final book and is under verification . Information intended for educational purpose only,