Assignment/Task-5

WECP-Style SQL Assessment Scenario:

Scenario:
A food delivery platform *FoodDash* stores data about restaurants, customers, orders, and order items for revenue analytics.
You are given the following schema:

Restaurants:

- restaurant_id — INT — *Primary Key.* Unique restaurant identifier.
- name — VARCHAR(150) — Restaurant name.
- category — VARCHAR(50) — Cuisine type (e.g., Chinese, Indian, Italian).
- rating — DECIMAL(2,1) — Rating 0.0–5.0.
- city — VARCHAR(100) — Restaurant city/location (used in Q1).

Notes / Constraints:

- PK: restaurant_id

Customers:

- customer_id — INT — *Primary Key.* Unique customer identifier.
- name — VARCHAR(150) — Customer name.
- city — VARCHAR(100) — Customer city (used in Q1 to compare with restaurant city).
- created_at — DATE or TIMESTAMP — When the customer registered (optional but handy).

Notes / Constraints:

- PK: customer_id

Orders:

- order_id — INT — *Primary Key.* Unique order identifier.
- customer_id — INT — *Foreign Key → customers.customer_id.* Who placed the order.
- restaurant_id — INT — *Foreign Key → restaurants.restaurant_id.* Restaurant fulfilling the order.
- order_date — DATE or DATETIME — When the order was placed (used in Q5, Q7, Q10, etc.).
- status — VARCHAR(30) — (optional) order status like 'completed', 'cancelled'.

Notes / Constraints:

- PK: order_id
- FKs: customer_id references customers(customer_id), restaurant_id references restaurants(restaurant_id)

order_items:

- item_id — INT — *Primary Key.* Unique order-item row id.
- order_id — INT — *Foreign Key → orders.order_id.* Which order this item belongs to.
- item_name — VARCHAR(150) — Name of the menu item.
- price — DECIMAL(10,2) — Price per unit (supports revenue calc in Q3, Q6).
- quantity — INT — Units ordered (used in Q4, Q6, Q8, Q9).

Notes / Constraints:

- PK: item_id
- FK: order_id references orders(order_id)

1. Find customers who placed orders from restaurants located in cities different from their own.
Use:

- subquery joining orders + restaurants,
- outer query joining customers.

2. List all customers who ordered an item priced above ₹500.
Requirements:

- Subquery joins order_items + orders
- Outer query checks customer_id IN (subquery).

3. Find customers who have *never* ordered from Indian cuisine restaurants.
Requirements:

- Subquery joins orders + restaurants
- Outer query uses:

customer_id NOT IN (subquery)

4. Show restaurants that have at least one order containing an item quantity greater than 3.

- Subquery joins orders + order_items
- EXISTS must be used.

5. List restaurants that do NOT have any orders placed in the month of '2024-12'.

- Subquery joins orders + customers
- Use NOT EXISTS.

6. For each restaurant, display total revenue and classify it as:

- "High Revenue" if revenue > 1,00,000
- "Medium Revenue" if revenue between 50,000 and 1,00,000

- "Low Revenue" otherwise

Revenue must be computed using a subquery joining:
orders → order_items.

7. For each customer, find the restaurant from which they placed their MOST recent order.

- Correlated subquery on orders
- JOIN with restaurants
- Must return: customer_name, restaurant_name, latest_date.

8. Find the top 3 customers based on total spending.
Use:

- Subquery to join order_items → orders
- Outer subquery to group by customer
- Final query to limit top 3.

9. Show each customer with their total number of items ordered across all restaurants.

Use:

```
SELECT c.name,
    (SELECT SUM(quantity)
     FROM orders o
     JOIN order_items oi ON o.order_id = oi.order_id
     WHERE o.customer_id = c.customer_id) AS total_items
FROM customers c;
```

10. List customers who ordered items ONLY from a single restaurant (i.e., they have not ordered from ANY other restaurant).

Must use:

- NOT EXISTS
- Join orders, order_items, restaurants

Condition:

Customer never ordered from more than one restaurant.

Short notes on usage :

- city on both restaurants and customers supports Q1 (compare cities).
- price & quantity in order_items enable revenue and spending calculations (Q3, Q6, Q8, Q9).
- order_date in orders is necessary for date-range and recent-order queries (Q5, Q7).
- PK/FK constraints and indexes make JOINs and subqueries efficient for learning and practice.