Coding Challenge-1

Name: Harish Er

Database:

Customers:

```
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    CustomerName NVARCHAR(100),
    City NVARCHAR(50)
);
```

Orders:

```
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    OrderDate DATE,
    TotalAmount DECIMAL(10, 2),
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);
```

Couriers:

```
CREATE TABLE Couriers (
    CourierID INT PRIMARY KEY,
    CourierName NVARCHAR(100),
    City NVARCHAR(50)
);
```

Deliveries:

```
CREATE TABLE Deliveries (
    DeliveryID INT PRIMARY KEY,
    OrderID INT,
    CourierID INT,
    DeliveryDate DATE,
    Status NVARCHAR(50),
    FOREIGN KEY (OrderID) REFERENCES Orders(OrderID),
    FOREIGN KEY (CourierID) REFERENCES Couriers(CourierID)
);
```

Explanation:

1. **Joins**: Used to combine rows from two or more tables based on a related column between them.
    - **Inner Join**: Returns only matching rows from both tables.
    - **Left Join**: Returns all rows from the left table and matching rows from the right table; unmatched rows from the right are NULL.
    - **Right Join**: Returns all rows from the right table and matching rows from the left table; unmatched rows from the left are NULL.
    - **Cross Join**: Produces a Cartesian product of rows from both tables, combining each row from one table with all rows in the other.
    - **Self Join**: Joins a table with itself to compare rows within the same table.

2. **Subqueries**: Queries nested inside another query to retrieve specific data needed by the outer query.
    - **In SELECT Clause**: Used to calculate or retrieve specific values for each row in the main query.
    - **In WHERE Clause**: Filters rows in the main query based on the subquery's result.
    - **In FROM Clause**: Treats the subquery as a derived table for further querying.
    - **Correlated Subquery**: References columns from the outer query, executing once per row in the outer query.

3. **Subtotal**: Uses aggregation functions like SUM, AVG, or COUNT to calculate totals or averages over groups of data.

4. **GROUP BY**: Organizes rows into groups based on one or more columns, enabling aggregate functions like SUM or AVG on each group.

5. **HAVING**: Filters groups created by GROUP BY based on aggregate values, showing only groups that meet specific criteria.

1. Querying Data by Using Joins and Subqueries & subtotal

1. Get Each Customer's Latest Order and the Total Amount They Have Spent

```
SELECT C.CustomerID,
       C.CustomerName,
       (SELECT MAX(O.OrderDate)
       FROM Orders O
       WHERE O.CustomerID = C.CustomerID) AS LatestOrderDate,
       SUM(O.TotalAmount) AS TotalAmountSpent
FROM Customers C
JOIN Orders O ON C.CustomerID = O.CustomerID
GROUP BY C.CustomerID, C.CustomerName;
```

Output:

	CustomerID	CustomerName	LatestOrderDate	TotalAmountSpent
1	1	Amit Sharma	2024-01-05	500.00
2	2	Priya Singh	2024-01-10	750.00
3	3	Ravi Kumar	2024-02-15	1200.00
4	4	Anita Desai	2024-02-20	600.00
5	5	Sunil Verma	2024-03-25	450.00
6	6	Pooja Patel	2024-04-01	800.00
7	7	Rakesh Mehta	2024-05-10	1300.00
8	8	Sita Iyer	2024-06-15	400.00
9	9	Vikram Malhotra	2024-07-20	900.00
10	10	Kavita Rao	2024-08-25	1100.00

Explanation:

- The main query fetches each customer's CustomerID and CustomerName.
- A correlated subquery retrieves the LatestOrderDate for each customer.
- The SUM function provides the total amount spent by each customer (TotalAmountSpent), creating a subtotal for each customer.
- GROUP BY groups results by CustomerID to apply aggregation functions accurately.

2. **Subquery with Aggregation - Find the maximum order amount for each city.**

```
SELECT City,
      (SELECT MAX(TotalAmount) FROM Orders O WHERE C.CustomerID =
      O.CustomerID) AS MaxOrderAmount
FROM Customers C;
```

Output:

	City	MaxOrderAmount
1	Mumbai	500.00
2	Delhi	750.00
3	Bangalore	1200.00
4	Chennai	600.00
5	Hyderabad	450.00
6	Pune	800.00
7	Ahmedabad	1300.00
8	Kolkata	400.00
9	Mumbai	900.00
10	Jaipur	1100.00

Explanation: This query shows the maximum order amount for each customer's city. For each city, a subquery is executed to find the maximum TotalAmount based on matching CustomerIDs between Customers and Orders.

3. **Subquery in FROM Clause - Get the count of orders per customer using a subquery.**

```
SELECT CustomerID, COUNT(*) AS OrderCount
FROM (SELECT CustomerID, OrderID FROM Orders) AS SubOrders
GROUP BY CustomerID;
```

Output:

	CustomerID	OrderCount
1	1	1
2	2	1
3	3	1
4	4	1
5	5	1
6	6	1
7	7	1
8	8	1
9	9	1
10	10	1

Explanation: The subquery fetches CustomerID and OrderID from Orders. The main query then counts the number of orders per customer by grouping results from the subquery.

2. Manipulate Data Using GROUP BY and HAVING

1. **Average Order Amount by Customer - Find the average order amount for each customer, showing only those with an average order above $500.**

```
SELECT CustomerID, AVG(TotalAmount) AS AvgOrderAmount
FROM Orders GROUP BY CustomerID HAVING AVG(TotalAmount) > 500;
```

Output:

	CustomerID	AvgOrderAmount
1	2	750.000000
2	3	1200.000000
3	4	600.000000
4	6	800.000000
5	7	1300.000000
6	9	900.000000
7	10	1100.000000

Explanation: This query calculates the average TotalAmount per CustomerID using AVG() and groups by CustomerID. The HAVING clause filters to only show customers with an average order amount above $500.

2. **Order Count by Delivery Status - Count orders based on their delivery status.**

```
SELECT Status, COUNT(OrderID) AS OrderCount
FROM Deliveries GROUP BY Status;
```

Output:

	Status	OrderCount
1	Delivered	5
2	In Transit	3
3	Pending	2

Explanation: This query counts orders grouped by Status in Deliveries. The COUNT() function finds how many orders fall into each status category.

3. **Customers with Large Orders - Show customers who have placed orders totaling more than $800.**

```
SELECT CustomerID, SUM(TotalAmount) AS TotalAmount
FROM Orders GROUP BY CustomerID HAVING SUM(TotalAmount) > 800;
```

Output:

	CustomerID	TotalAmount
1	3	1200.00
2	7	1300.00
3	9	900.00
4	10	1100.00

Explanation: This query groups by CustomerID and uses SUM() to calculate total order value per customer. The HAVING clause filters results, only showing customers whose total order amount exceeds $800.

4. **Total Sales by Courier City - Calculate total order amount delivered by couriers in each city.**

```
SELECT C.City, SUM(O.TotalAmount) AS TotalSales FROM Orders O
JOIN Deliveries D ON O.OrderID = D.OrderID
JOIN Couriers C ON D.CourierID = C.CourierID GROUP BY C.City;
```

Output:

	City	TotalSales
1	Ahmedabad	1300.00
2	Bangalore	1200.00
3	Chennai	600.00
4	Delhi	750.00
5	Hyderabad	450.00
6	Jaipur	1100.00
7	Kolkata	400.00
8	Mumbai	1400.00
9	Pune	800.00

Explanation: By joining tables, this query matches orders with couriers. It groups by City and uses SUM() on TotalAmount to find total sales delivered by couriers from each city.

5. **Identify Cities with an Average Order Amount Above 600, and Show the Total Number of Customers per City Who Have Placed Orders**

```
SELECT C.City,
   COUNT(DISTINCT O.CustomerID) AS TotalCustomers,
   AVG(O.TotalAmount) AS AvgOrderAmount
FROM Customers C
JOIN Orders O ON C.CustomerID = O.CustomerID
GROUP BY C.City
 HAVING AVG(O.TotalAmount) > 600;
```

Output:

	City	TotalCustomers	AvgOrderAmount
1	Ahmedabad	1	1300.000000
2	Bangalore	1	1200.000000
3	Delhi	1	750.000000
4	Jaipur	1	1100.000000
5	Mumbai	2	700.000000
6	Pune	1	800.000000

Explanation:

- The JOIN operation links Customers and Orders to connect each customer's details with their orders.
- GROUP BY groups results by City, allowing aggregation across customers and their orders within each city.
- COUNT(DISTINCT O.CustomerID) calculates the total number of unique customers per city who have placed at least one order.
- The HAVING clause filters for cities where the average order amount exceeds 600. This ensures we only see cities with higher average spending.