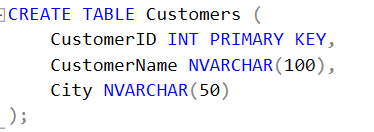
**Coding Challenge-1**

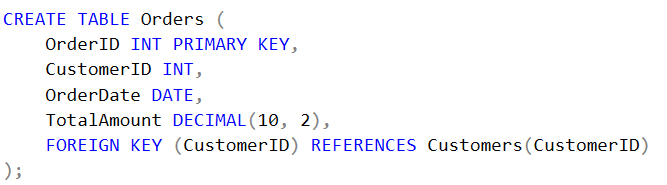
Name: Harish Er

**Database:**

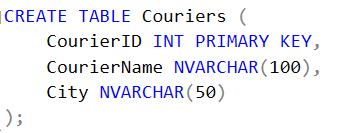
**Customers:**

****

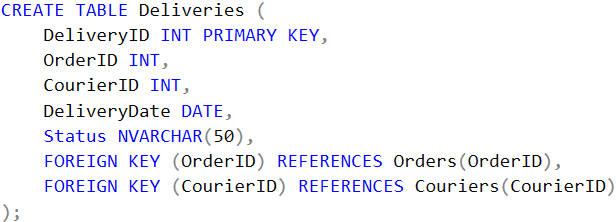
**Orders:**

****

**Couriers:**

****

**Deliveries:**

****

**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.

1. **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.

1. **Subtotal**: Uses aggregation functions like SUM, AVG, or COUNT to calculate totals or averages over groups of data.
2. **GROUP BY**: Organizes rows into groups based on one or more columns, enabling aggregate functions like SUM or AVG on each group.
3. **HAVING**: Filters groups created by GROUP BY based on aggregate values, showing only groups that meet specific criteria.
4. **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,**

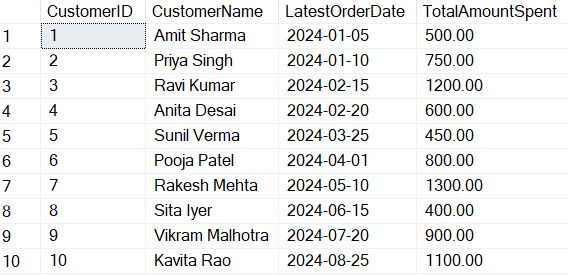
**SUM(O.TotalAmount) AS TotalAmountSpent**

**FROM Customers C**

**JOIN Orders O ON C.CustomerID = O.CustomerID**

**GROUP BY C.CustomerID, C.CustomerName;**

**Output:**

****

**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.

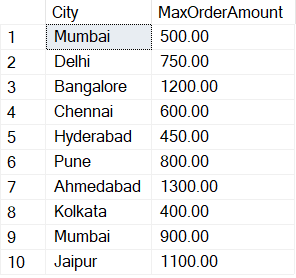
1. **Subquery with Aggregation - Find the maximum order amount for each city.**

**SELECT City,**

**(SELECT MAX(TotalAmount) FROM Orders O WHERE C.CustomerID = O.CustomerID) AS MaxOrderAmount**

**FROM Customers C;**

**Output:**

****

**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.

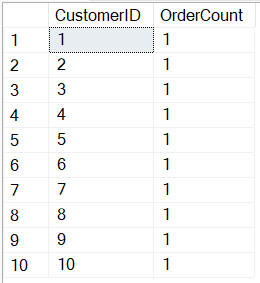
1. **Subquery in FROM Clause - Get the count of orders per customer using a subquery.**

**SELECT CustomerID, COUNT(\*) AS OrderCount**

**FROM (SELECT CustomerID, OrderID FROM Orders) AS SubOrders**

**GROUP BY CustomerID;**

**Output:**

****

**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.

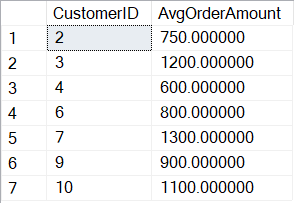
**2. Manipulate Data Using GROUP BY and HAVING**

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:**

****

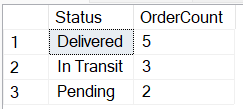
**Explanation:** This query calculates the average TotalAmount per CustomerID using AVG() and groups by CustomerID. The HAVING clause filters to only show customers with an average order amount above $500.

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

**SELECT Status, COUNT(OrderID) AS OrderCount**

**FROM Deliveries GROUP BY Status;**

**Output:**

****

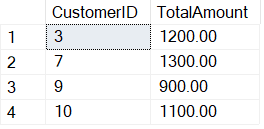
**Explanation:** This query counts orders grouped by Status in Deliveries. The COUNT() function finds how many orders fall into each status category.

1. **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:**

****

**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.

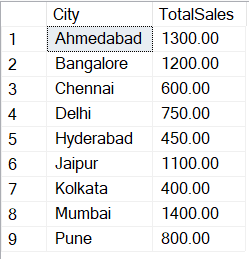
1. **Total Sales by Courier City - Calculate total order amount delivered by couriers in each city.**

**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:**

****

**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.

1. **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**

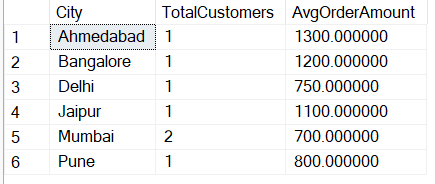
**FROM Customers C**

**JOIN Orders O ON C.CustomerID = O.CustomerID**

**GROUP BY C.City**

**HAVING AVG(O.TotalAmount) > 600;**

**Output:**

****

**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.