Name: Krushnakumar Patle

Email: krishnapatle128@gmail.com

Batch: Data Engineering Batch-1

SQL Coding Challenge

Create Database and tables.

```sql
CREATE DATABASE SQL_Coding_Challenge;
USE SQL_Coding_Challenge;

CREATE TABLE orders (
    order_id INT PRIMARY KEY AUTO_INCREMENT,
    customer_id INT,
    order_date DATE,
    total_amount DECIMAL(10, 2)
);

CREATE TABLE order_details (
    detail_id INT PRIMARY KEY AUTO_INCREMENT,
    order_id INT,
    product_name VARCHAR(50),
    quantity INT,
    price DECIMAL(8, 2),
    FOREIGN KEY(order_id) REFERENCES orders(order_id)
);

INSERT INTO orders VALUES (1, 101, '2022-01-01', 150.00);
INSERT INTO orders VALUES (2, 102, '2022-01-02', 200.00);

INSERT INTO order_details VALUES (1, 1, 'Product A', 2, 50.00);
INSERT INTO order_details VALUES (2, 1, 'Product B', 3, 30.00);
INSERT INTO order_details VALUES (3, 2, 'Product A', 1, 80.00);
INSERT INTO order_details VALUES (4, 2, 'Product C', 2, 60.00);
```

1. **Over and Partition by Clause:**

The **PARTITION BY** clause in SQL is used with window functions to divide the result set into partitions to which the window function is applied. It is a way to perform calculations across subsets of rows within the result set, based on a specified column or set of columns. The **PARTITION BY** clause is commonly

used with aggregate functions to calculate values for each partition independently.

```sql
-- Over and Partition by Clause
SELECT
    order_id,
    product_name,
    quantity,
    SUM(quantity) OVER (PARTITION BY order_id) AS total_quantity
FROM
    order_details;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

order_id	product_name	quantity	total_quantity
1	Product A	2	5
1	Product B	3	5
2	Product A	1	3
2	Product C	2	3

This query calculates the total quantity for each order. The **PARTITION BY** clause divides the result set into partitions based on the **order_id**, and the **SUM(quantity) OVER** window function calculates the running sum of quantity within each partition.

2. **Create subtotals &Total Aggregations using SQL Queries.**

Subtotals represent intermediate calculations or sums within specific groups of data. They are often used when you want to see aggregate results for subsets of your data, not just the overall total. The **GROUP BY** clause is commonly used to create subtotals.

```sql
-- Calculate subtotal for each order
SELECT
    order_id,
    SUM(quantity) AS total_quantity,
    SUM(price * quantity) AS subtotal_amount
FROM
    order_details
GROUP BY
    order_id;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

order_id	total_quantity	subtotal_amount
1	5	190.00
2	3	200.00

This query calculates subtotals for each order. It uses the **GROUP BY** clause to group rows by **order_id**. The **SUM(quantity)** calculates the total quantity for each order, and **SUM(price * quantity)** calculates the subtotal amount.

Total aggregations:

Total aggregations involve calculating the overall sum, average, count, or other aggregate values across the entire dataset. Total aggregations are often achieved without the **GROUP BY** clause.

```sql
-- Calculate total quantity and total amount for all orders
SELECT
    SUM(quantity) AS total_quantity,
    SUM(price * quantity) AS total_amount
FROM
    order_details;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

total_quantity	total_amount
8	390.00

This query calculates total aggregations for all orders. It uses aggregate functions without the **GROUP BY** clause to get the overall sum of quantity and the total amount.

3. **Execute all the join with examples.**

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

Here are the different types of the JOINs in SQL:

INNER JOIN:

The **INNER JOIN** keyword selects records that have matching values in both tables. It returns only the rows where there is a match in the specified columns.

```sql
-- INNER JOIN:
SELECT
    o.order_id,
    od.product_name,
    od.quantity
FROM
    orders o
INNER JOIN
    order_details od ON o.order_id = od.order_id;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

order_id	product_name	quantity
1	Product A	2
1	Product B	3
2	Product A	1
2	Product C	2

This query performs an **INNER JOIN** to retrieve order details along with customer information. It connects the **orders** table with the **order_details** table using the common column **order_id.**

LEFT JOIN:

The **LEFT JOIN** keyword returns all records from the left table (table1) and the matched records from the right table (table2).

```sql
-- LEFT JOIN:
SELECT
    o.order_id,
    od.product_name,
    od.quantity
FROM
    orders o
LEFT JOIN
    order_details od ON o.order_id = od.order_id;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

order_id	product_name	quantity
1	Product A	2
1	Product B	3
2	Product A	1
2	Product C	2

This query performs a **LEFT JOIN** to get all orders and their details, including orders without details. It connects the **orders** table with the **order_details** table, and if there are no matching details for an order, NULL values will be included in the result set.

RIGHT JOIN:

The **RIGHT JOIN** keyword returns all records from the right table (table2) and the matched records from the left table (table1).

```sql
-- RIGHT JOIN
SELECT
    o.order_id,
    od.product_name,
    od.quantity
FROM
    orders o
RIGHT JOIN
    order_details od ON o.order_id = od.order_id;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

order_id	product_name	quantity
1	Product A	2
1	Product B	3
2	Product A	1
2	Product C	2

This query performs a **RIGHT JOIN** to get all details and include orders without details. It includes all rows from the **order_details** table and the matching rows from the **orders** table. If there are no matching details for an order, NULL values will be included.

CROSS JOIN:

The **CROSS JOIN** keyword returns the Cartesian product of the two tables, i.e., all possible combinations of rows from both tables. It does not require a matching condition.

```sql
-- CROSS JOIN:
SELECT
    o.order_id,
    od.product_name,
    od.quantity
FROM
    orders o
CROSS JOIN
    order_details od;
```

Result Grid | Filter Rows: | Export: | Wrap C

order_id	product_name	quantity
2	Product A	2
1	Product A	2
2	Product B	3
1	Product B	3
2	Product A	1
1	Product A	1
2	Product C	2
1	Product C	2

This query performs a **CROSS JOIN** to get all combinations of orders and details. It generates all possible combinations by pairing each row from the **orders** table with every row from the **order_details** table.

SELF JOIN:

A **SELF JOIN** is a regular join, but the table is joined with itself. This is useful when you want to relate rows within the same table.

```sql
-- SELF JOIN:
SELECT
    o1.order_id AS order1_id,
    o1.total_amount AS order1_total,
    o2.order_id AS order2_id,
    o2.total_amount AS order2_total
FROM
    orders o1
JOIN
    orders o2 ON o1.total_amount > o2.total_amount;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

order1_id	order1_total	order2_id	order2_total
2	200.00	1	150.00

This query performs a self-join to compare orders based on their total amounts. It connects the **orders** table to itself, comparing each order with every other order to find orders with higher total amounts.

FULL JOIN:

A **FULL JOIN** is a type of SQL join that combines the results of both the **LEFT JOIN** and the **RIGHT JOIN**. It returns all rows from both tables being joined, with matched rows from both sides where there is a match.

```sql
-- FULL JOIN:
SELECT o.order_id, od.product_name, od.quantity
FROM orders o
LEFT JOIN order_details od ON o.order_id = od.order_id
UNION
SELECT o.order_id, od.product_name, od.quantity
FROM orders o
RIGHT JOIN order_details od ON o.order_id = od.order_id
WHERE o.order_id IS NULL;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

order_id	product_name	quantity
1	Product B	3
1	Product A	2
2	Product C	2
2	Product A	1

This query performs a FULL JOIN to get all orders and details, including unmatched rows from both tables. It includes all rows from both the orders and order_details tables. If there is no matching detail for an order or no matching order for a detail, NULL values will be included.