Name: Krushnakumar Patle

Email: krishnapatle128@gmail.com

Batch: Data Engineering Batch-1

1. Total Aggregation Functions

Total Sales Amount:

The SUM() aggregate function returns the sum of all the numeric values from the column.

```
-- Total Sales Amount:
SELECT SUM(Quantity * Price) AS TotalSalesAmount
FROM Sales;
```

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

TotalSalesAmount
681.00

Average Price per Product:

The AVG() aggregate function returns the mean average of the numeric values in the specified column.

```
-- Average Price per Product:
SELECT ProductName, AVG(Price) AS AvgPrice
FROM Sales
GROUP BY ProductName;
```

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

ProductName	AvgPrice
Product A	21.250000
Product B	15.000000
Product C	18.750000

Maximum Quantity Sold:

The MAX() aggregate function returns the largest value from the column.

```sql
-- Maximum Quantity Sold:
SELECT MAX(Quantity) AS MaxQuantity
FROM Sales;
```

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

MaxQuantity
12

Product with the Highest Sale Amount:

```sql
-- Product with the Highest Sale Amount:
SELECT ProductName, SUM(Quantity * Price) AS TotalSaleAmount
FROM Sales
GROUP BY ProductName
ORDER BY TotalSaleAmount DESC
LIMIT 1;
```

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

ProductName	TotalSaleAmount
Product A	381.00

Average price per distinct product:

The DISTINCT() aggregate function returns the set of unique values in the column, discarding multiples.

```sql
-- Average price per distinct product
SELECT AVG(DISTINCT Price) AS AvgPrice
FROM Sales;
```

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

AvgPrice
19.062500

Count of distinct products:

The COUNT() aggregate function returns the number of items in the column.

```sql
-- Count of distinct products
SELECT COUNT(DISTINCT ProductName) AS DistinctProductCount
FROM Sales;
```

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

DistinctProductCount
3

2. OVER and PARTITION BY Clause in SQL Queries& Total Aggregation using OVER and PARTITION BY in SQL Queries

The OVER and PARTITION BY clauses in SQL are used in conjunction with window functions to perform calculations across a specific subset of rows within a result set. These clauses are commonly used for analytical functions that require operations on groups of rows.

```sql
-- Total Aggregation using OVER and PARTITION BY:
SELECT
    SaleID,
    ProductName,
    Category,
    Quantity,
    Price,
    SUM(Quantity) OVER (PARTITION BY Category) AS TotalQuantityInCategory,
    SUM(Price) OVER (PARTITION BY Category) AS TotalPriceInCategory
FROM
    Sales;
```

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

SaleID	ProductName	Category	Quantity	Price	TotalQuantityInCategory	TotalPriceInCategory
4	Product C	Clothing	12	18.75	12	18.75
1	Product A	Electronics	10	20.50	23	57.50
2	Product B	Electronics	5	15.00	23	57.50
3	Product A	Electronics	8	22.00	23	57.50

3. Rules and Restrictions to Group and Filter Data in SQL queries

```sql
-- Average Deal Value by Sales Agent
SELECT sales_agent, AVG(close_value) AS avg_deal_value
FROM sales_pipeline
WHERE deal_stage = 'Won'
GROUP BY sales_agent
ORDER BY avg_deal_value DESC;
```

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

sales_agent	avg_deal_value
Darcel Schlecht	11666.666667
Donn Cantrell	9333.333333
Elease Gluck	5250.000000

Count of Deals and Count of Deals > 1000 by Sales Agent

```sql
-- Count of Deals and Count of Deals > 1000 by Sales Agent
SELECT sales_agent,
       COUNT(close_value) AS total_deals,
       SUM(CASE WHEN close_value > 1000 THEN 1 ELSE 0 END) AS deals_over_1000
FROM sales_pipeline
WHERE deal_stage = 'Won'
GROUP BY sales_agent;
```

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

sales_agent	total_deals	deals_over_1000
Elease Gluck	4	4
Darcel Schlecht	3	3
Donn Cantrell	3	3

Multiple Filters in a Single Query

```sql
-- Multiple Filters in a Single Query
SELECT
    sales_agent,
    COUNT(close_value) AS total_deals,
    SUM(CASE WHEN close_value > 1000 THEN 1 ELSE 0 END) AS deals_over_1000,
    AVG(close_value) AS avg_deal_value,
    AVG(CASE WHEN close_value > 1000 THEN close_value END) AS avg_deal_value_over_1000
FROM sales_pipeline
WHERE deal_stage = 'Won'
GROUP BY sales_agent;
```

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

sales_agent	total_deals	deals_over_1000	avg_deal_value	avg_deal_value_over_1000
Elease Gluck	4	4	5250.000000	5250.000000
Darcel Schlecht	3	3	11666.666667	11666.666667
Donn Cantrell	3	3	9333.333333	9333.333333

4. **What is SQL Order of Execution?**

SQL order of execution refers to **the sequence in which different clauses and operations within a SQL query are processed** by the database management system.

Each SQL query consists of various components such as SELECT, FROM, WHERE, GROUP BY, HAVING, and ORDER BY clauses, along with functions and operators. Understanding the order in which these components are executed is vital for producing accurate and efficient query results.

```sql
-- Query: Get the average salary for each department with more than two employees
SELECT
    department,
    AVG(salary) AS avg_salary
FROM
    employees
WHERE
    salary > 50000
GROUP BY
    department
HAVING
    COUNT(employee_id) > 2
ORDER BY
    avg_salary DESC;
```

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

department	avg_salary

5. How to calculate Subtotals in SQL Queries:

Calculating subtotals in SQL queries involves using the **GROUP BY** clause to group rows based on a specific column or set of columns. Subtotals are then computed for each group using aggregate functions such as **SUM**, **AVG**, **COUNT**, etc. Here's a basic example to illustrate how you can calculate subtotals:

```sql
-- How to calculate Subtotals in SQL Queries
CREATE TABLE sales1 (
    order_id INT PRIMARY KEY,
    product_id INT,
    quantity INT,
    price DECIMAL(10, 2),
    order_date DATE
);

INSERT INTO sales1 VALUES
    (1, 101, 2, 50.00, '2022-01-01'),
    (2, 102, 3, 30.00, '2022-01-01'),
    (3, 101, 1, 50.00, '2022-01-02'),
    (4, 103, 2, 25.00, '2022-01-02'),
    (5, 102, 4, 30.00, '2022-01-03');
```

Calculate subtotals for each product

```sql
-- Query: Calculate subtotals for each product
SELECT
    product_id,
    SUM(quantity) AS total_quantity,
    SUM(price * quantity) AS total_sales
FROM
    sales1
GROUP BY
    product_id;
```

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

product_id	total_quantity	total_sales
101	3	150.00
102	7	210.00
103	2	50.00

Calculate subtotals for each product and each order date

```sql
-- Query: Calculate subtotals for each product and each order date
SELECT
    product_id,
    order_date,
    SUM(quantity) AS total_quantity,
    SUM(price * quantity) AS total_sales
FROM
    sales1
GROUP BY
    product_id, order_date;
```

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

product_id	order_date	total_quantity	total_sales
101	2022-01-01	2	100.00
102	2022-01-01	3	90.00
101	2022-01-02	1	50.00
103	2022-01-02	2	50.00
102	2022-01-03	4	120.00

The **UNION**, **EXCEPT**, and **INTERSECT** operators in SQL Server are used to combine or compare the results of two or more SELECT queries. Here are the key differences between these operators:

1. **UNION Operator:**
    - **Purpose:** Combines the result sets of two or more SELECT statements into a single result set.
    - **Syntax:**

SELECT column1, column2, ... FROM table1 UNION SELECT column1, column2, ... FROM table2;

    - **Result Set:** Includes all unique rows from the combined result sets. Duplicate rows are automatically eliminated.

2. **EXCEPT Operator (MINUS in some databases):**
    - **Purpose:** Returns the rows that are present in the result set of the first SELECT statement but not in the result set of the second SELECT statement.
    - **Syntax:**

SELECT column1, column2, ... FROM table1 EXCEPT SELECT column1, column2, ... FROM table2;

- **Result Set:** Contains only the rows that exist in the first result set but not in the second result set.

3. **INTERSECT Operator:**
   - **Purpose:** Returns the common rows that are present in both the result sets of the two SELECT statements.
   - **Syntax:**

SELECT column1, column2, ... FROM table1 INTERSECT SELECT column1, column2, ... FROM table2;

   - **Result Set:** Contains only the rows that are common to both result sets.

4. **Number of Columns:**
   - **UNION:** The number and data types of columns must match in both SELECT statements.
   - **EXCEPT and INTERSECT:** The number and data types of columns must match in both SELECT statements.

5. **Sorting:**
   - **UNION:** Automatically orders the result set and removes duplicates.
   - **EXCEPT and INTERSECT:** Do not automatically order the result set, and you may need to use additional sorting if desired.

6. **Usage Considerations:**
   - **UNION:** Typically used when you want to combine the results of two queries with similar structures.
   - **EXCEPT and INTERSECT:** Used when you want to compare or find differences between two result sets.