BRAINALYST
A Data Driven Company

Brainalyst's

ALL YOU NEED TO KNOW SERIES

To Become a Successful Data Professional

SQL FOR DATA ANALYTICS

WINDOW FUNCTIONS

Window Function- Advance:

SQL window functions are analytical operations on a group of rows known as a "window" or "frame" within the result set. They enable the execution of calculations and aggregations on a per-row basis while taking the window's context into account. A thorough explanation of numerous window functions is provided below, along with a sample code: but lets first understand what is difference between aggregation function and window function.

In SQL, sets of rows in a table can be calculated using both aggregate functions and window functions. The methods they use to organise and interpret the data vary, though.

Aggregate functions:

- Calculation scope: Aggregate functions operate on a set of rows and return a single value for the entire set.
- Grouping: They are typically used with the GROUP BY clause to partition the data into groups and calculate a result for each group.
- Result granularity: Aggregate functions collapse multiple rows into a single result. For example, SUM, AVG, COUNT, MAX, and MIN are aggregate functions.
- Usage: They are used to obtain summary statistics or perform calculations such as calculating the total sales per category, average salary per department, or counting the number of orders per customer.

SYNTAX:

```
SELECT department, AVG(salary) AS average_salary
FROM employees
GROUP BY department;
```

Window functions:

- Calculation scope: Window functions perform calculations on a "window" or a subset of rows within a result set.
- Ordering: They are typically used with the ORDER BY clause to define the window's ordering within the result set.
- Result granularity: Window functions retain the individual rows in the result set but can calculate values based on the window's defined scope.
- Usage: They are used to compute values that are related to the current row but involve other rows in the result set. Examples of window functions include RANK, ROW_NUMBER, LAG, LEAD, and SUM with the OVER clause.

How window function works:

OVER clause, which specifies the window's borders and ordering, is frequently combined with window functions in programming.

SYNTAX:

```
function_name([arguments]) OVER (
[PARTITION BY partition_expression]
[ORDER BY order_expression [ASC|DESC]]
[ROWS/RANGE frame_clause]
```

- **Function_name:** Window function name you can use, such as ROW_NUMBER, SUM, AVG, etc.
- **rguments:** Its optional arguments that the window function accept. For example, SUM(column_name) would calculate the sum of the specified column.
- **PARTITION BY:** Optional clause that divides the rows into partitions or groups based on one or more expressions. The window function is applied separately to each partition.
- **ORDER BY**: Optional clause that specifies the order in which the rows within each partition should be processed by the window function. It can use one or more columns and can be sorted in ascending (ASC) or descending (DESC) order.
- **ROWS/RANGE frame_clause:** Optional clause that defines the window frame or the subset of rows within each partition over which the window function perates.

1. **ROW_NUMBER():** Each row in a window is given a distinct sequential number using the ROW NUMBER() function.

   Example:

```
SELECT ROW_NUMBER()OVER(ORDER BY PRICE DESC) as row_
no, Price
FROM aemf2
```

OUTPUT:

row_no	Price
Filter...	Filter...
1	18545.45028
2	16445.61469
3	13664.30592
4	13656.35883
5	12942.99138
6	8130.668104
7	7782.907225
8	6943.70098

2. **RANK():**

- Gives each row in a window a rank, leaving gaps for tied values.

Example:

```
SELECT RANK()OVER(ORDER BY PRICE DESC) as row_no,
Price
FROM aemf2;
```

3. **DENSE_RANK():**

- Gives each row in a window a rank, leaving no gaps for tied values.

Example:

```
SELECT DENSE_RANK()OVER(ORDER BY PRICE DESC) as row_no,
Price FROM ;
```

4. **NTILE():**

- Divides a window's rows into a predetermined number of groups or "tiles."

Example:

```
SELECT NTILE()OVER(ORDER BY PRICE DESC) as row_no,
Price FROM ;
```

5. **LAG():**
   - Accesses a previous row's value within a window.

Example:

```
SELECT PRICE,LAG(PRICE)OVER(ORDER BY DAY DESC) as row_
no, Price
FROM aemf2 ;
```

OUTPUT:

PRICE	row_no	Price
171.3297338	NULL	171.3297338
166.8887175	171.3297338	166.8887175
519.3651684	166.8887175	519.3651684
362.9946474	519.3651684	362.9946474
304.7939602	362.9946474	304.7939602
240.0486174	304.7939602	240.0486174
339.38713979999994	240.0486174	339.38713979999994
360.6572704	339.38713979999994	360.6572704
150.7608162	360.6572704	150.7608162
139.0739312	150.7608162	139.0739312

6. **LEAD():**
   - Accesses the value of a subsequent row within a window.

Example:

```
SELECT PRICE,LEAD(PRICE)OVER(ORDER BY DAY DESC) as row_
no, Price
FROM aemf2 ;
```

OUTPUT:

PRICE	row_no	Price
171.3297338	166.8887175	171.3297338
166.8887175	519.3651684	166.8887175
519.3651684	362.9946474	519.3651684
362.9946474	304.7939602	362.9946474
304.7939602	240.0486174	304.7939602
240.0486174	339.38713979999994	240.0486174
339.38713979999994	360.6572704	339.38713979999994
360.6572704	150.7608162	360.6572704

7. **FIRST_Value():**
   - Access previous row value within window.

   Example:

```
SELECT FIRST_VALUE(PRICE)OVER(ORDER BY DAY DESC) as
FIRST_V, Price
FROM aemf2 ;
```

OUTPUT:

FIRST_V	Price
Filter...	Filter...
171.3297338	171.3297338
171.3297338	166.8887175
171.3297338	519.3651684
171.3297338	362.9946474
171.3297338	304.7939602
171.3297338	240.0486174
171.3297338	339.38713979999994

```
SELECT Sales, PERCENT_RANK() OVER (ORDER BY Sales) AS
PercentileRank
FROM SalesData;
```