Window Function In MySQL

What is window fuction ?

- Window functions in SQL are a type of analytical function that perform calculations across a set of rows that are related to the current row, called a "window".
- A window function calculates a value for each row in the result set based on a subset of the rows that are defined by a window specification.
- The window specification is defined using the OVER() clause in SQL, which specifies the partitioning and ordering of the rows that the window function will operate on. The partitioning divides the rows into groups based on a specific column or expression, while the ordering defines the order in which the rows are processed within each group.

In other word In – a **window function** is used to perform calculations across a set of table rows that are somehow related to the current row. It doesn’t group the rows into a single result like aggregate functions (e.g., SUM, AVG), but instead retains the rows and adds a new column with the calculated value.

In SQL, a **window function** is used to perform calculations across a set of table rows that are somehow related to the current row. It doesn’t group the rows into a single result like aggregate functions (e.g., SUM, AVG), but instead retains the rows and adds a new column with the calculated value.

Key Features of Window Functions:

1. **Operates Over a "Window"**
   A "window" is a subset of rows in a table. For each row, the window is defined based on criteria you specify (e.g., partitioning or ordering).

2. **Used With OVER() Clause**
   The OVER() clause defines the window over which the function operates. It can include:
   - PARTITION BY (to divide rows into groups)
   - ORDER BY (to order rows within each partition)

Example 1: Rank Employees by Salary

employee_id	department_id	salary
1	101	50000
2	101	70000
3	101	60000
4	102	80000
5	102	75000

```
SELECT
   employee_id,
   department_id,
   salary,
   RANK() OVER(PARTITION BY department_id ORDER BY salary DESC) AS rank
FROM employees;
```

Result:

employee_id	department_id	salary	rank
2	101	70000	1
3	101	60000	2
1	101	50000	3
4	102	80000	1
5	102	75000	2

- **PARTITION BY department_id:** Groups rows by department.
- **ORDER BY salary DESC:** Orders salaries within each department in descending order.
- **RANK()**: Assigns a rank to each employee based on their salary within their department.

Example 2: Running Total

customer_id	order_date	order_amount
101	2024-01-01	100
101	2024-01-03	200
102	2024-01-02	150
102	2024-01-04	300

```
SELECT
   customer_id,
   order_date,
   order_amount,
   SUM(order_amount) OVER(ORDER BY order_date) AS running_total
FROM orders;
```

Result:

customer_id	order_date	order_amount	running_total
101	2024-01-01	100	100
102	2024-01-02	150	250
101	2024-01-03	200	450
102	2024-01-04	300	750

- **ORDER BY order_date:** Orders rows by the date of the order.
- **SUM(order_amount) OVER(...):** Adds a cumulative total of the order amounts up to the current row.

Example 3: Average Salary in Each Department

```
SELECT
   employee_id,
   department_id,
   salary,
   AVG(salary) OVER(PARTITION BY department_id) AS avg_salary
FROM employees;
```

- **PARTITION BY department_id:** Groups employees by department.
- **AVG(salary) OVER(...):** Calculates the average salary for each department.

Why Use Window Functions?

- To calculate **rankings** (e.g., RANK, DENSE_RANK, ROW_NUMBER).
- To compute **running totals** or **cumulative values**.
- To calculate **moving averages**.
- To add insights without grouping or removing data rows.

➔ Window functions are powerful for analyzing and summarizing data while retaining all rows in the result set.

Aggregate Function with OVER()

1.find the student whose marks is greater than the avg marks of branch

```
select * from  (select *,avg(marks) over(partition by branch) as 'branch_avg' from marks ) t

where t.marks > t.branch_avg ;
```

RANK(), DENSE_RANK() and ROW_NUMBER()

Key Differences Between RANK, DENSE_RANK, and ROW_NUMBER:

- **RANK:** Skips numbers when there’s a tie.
- **DENSE_RANK:** No gaps in ranking when there’s a tie.
- **ROW_NUMBER:** Provides a unique sequential number for each row, even with ties.

1 . RANK(), DENSE_RANK() :

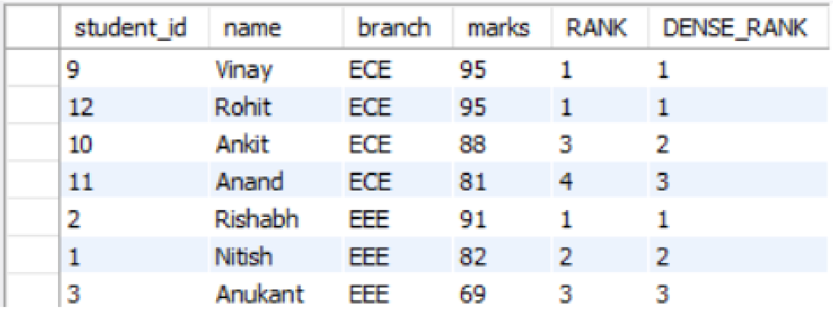
Q.1 RANKS STUDENT ON THE BASES OF THEIR MARKS IN BRANCH

```
SELECT * ,

RANK() OVER( PARTITION BY branch ORDER BY marks desc ) AS 'RANK',

DENSE_RANK() OVER( PARTITION BY branch ORDER BY marks desc ) AS 'DENSE_RANK'

FROM marks ;
```



student_id	name	branch	marks	RANK	DENSE_RANK
9	Vinay	ECE	95	1	1
12	Rohit	ECE	95	1	1
10	Ankit	ECE	88	3	2
11	Anand	ECE	81	4	3
2	Rishabh	EEE	91	1	1
1	Nitish	EEE	82	2	2
3	Anukant	EEE	69	3	3

Q2. FIND TWO MOST PAYING COSTUMER FROM EACH MONTH ?

```
SELECT t2.name, t1.user_id, t1.total, t1.RANK FROM (SELECT * FROM (SELECT
MONTHNAME(date) AS 'month',user_id,SUM(amount) AS 'TOTAL',

RANK() OVER(PARTITION BY MONTHNAME(date) ORDER BY SUM(amount) DESC) AS  'RANK'
FROM orders

GROUP BY user_id,MONTH) t

WHERE t.RANK < 3) t1

JOIN users t2 ON t1.user_id = t2.user_id

ORDER BY MONTH  desc ;
```

ROW_NUMBER : Provides a unique sequential number for each row, even with ties.

```
SELECT * ,

ROW_NUMBER()  OVER( PARTITION BY branch ) AS  'ROW_FOR_BRANCH'

FROM marks ;
```

Q.1 CREATE A STUDENT ROLLNUMBER BY ITS BRANCH LIKE CSE-1

```
SELECT * ,

CONCAT(branch,'-',ROW_NUMBER() OVER( PARTITION BY branch )) AS
'ROW_FOR_BRANCH' FROM marks;
```

FIRST_VALUE / LAST_VALUE / NTH_VALUE

1. FIRST_VALUE

→Fetches the first value in a specified window of rows.

```
SELECT *,
      FIRST_VALUE(name) OVER(ORDER BY marks DESC) AS first_name
FROM marks;
```

2. LAST_VALUE

→Fetches the last value in a specified window of rows.

```
SELECT *,
      LAST_VALUE(name) OVER(
         ORDER BY marks DESC
         ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
      ) AS last_name
FROM marks;
```

3. NTH_VALUE

→ Fetches the nth value in a specified window of rows.

```
SELECT *,
      LAST_VALUE(name) OVER(
         PARTITION BY branch
         ORDER BY marks DESC
         ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
      ) AS topper_name,
      LAST_VALUE(marks) OVER(
         PARTITION BY branch
      ORDER BY marks DESC
      ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) AS topper_marks
FROM marks;
```

Alternate Way (More Efficient):

```
SELECT *,
    LAST_VALUE(name) OVER w AS topper_name,
    LAST_VALUE(marks) OVER w AS topper_marks
FROM marks
WINDOW w AS (
  PARTITION BY branch
  ORDER BY marks DESC
  ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
);
```

LAG AND LEAD

Fetches previous and next row values in the result set.

```
SELECT *,
    LAG(marks) OVER(ORDER BY student_id) AS previous_marks,
    LEAD(marks) OVER(ORDER BY student_id) AS next_marks
FROM marks;
```

Practical Example: Monthly Profit Percentage

```
USE zomato;
SELECT
```

```
    MONTHNAME(date) AS MONTH,
    SUM(amount) AS TOTAL,
    ((SUM(amount) - LAG(SUM(amount)) OVER(ORDER BY MONTH(date)))
     / LAG(SUM(amount)) OVER(ORDER BY MONTH(date))) * 100 AS profit_percentage
FROM orders
GROUP BY MONTH(date), MONTHNAME(date)
ORDER BY MONTH(date);
```

PRACTICAL WINDOW FUNCTION EXAMPLE

Calculate Total Runs by "V Kohli" Across Matches Using Common Table Expressions (CTE):

```
WITH Kohli_runs AS (
  SELECT
      CONCAT('Match-', CAST((ROW_NUMBER() OVER (ORDER BY id)) AS CHAR)) AS Match_number,
      batter,
      SUM(batsman_run) AS total_runs
  FROM ipl
  WHERE batter = 'V Kohli'
  GROUP BY id
)
SELECT
    Match_number,
    total_runs AS runs_scored
```

```
FROM Kohli_runs;
```

Hope You Find this Valueble