**Window functions in mysql**

**What is a Window Function?**

A window function performs calculations across a set of rows related to the current row. Unlike regular aggregate functions (like SUM(), AVG(), MAX(), etc.), which reduce the result set to a single value per group, window functions retain the original row structure and provide additional calculated columns based on the window of rows around each row.

### **Key Concepts**

1. **Window Frame**:
   * A window function operates over a "window frame" of rows defined by the OVER() clause. This frame is a subset of the result set that is specified in relation to the current row.
2. **Partitioning**:
   * **PARTITION BY**: Divides the result set into partitions (groups) where the window function calculation is performed separately for each partition.
   * Example: PARTITION BY department\_id groups rows by department\_id, and calculations are done within each department.
3. **Ordering**:
   * **ORDER BY**: Specifies the order in which rows are processed within each partition. The ordering affects calculations that depend on the row's position within the partition.
   * Example: ORDER BY salary DESC orders rows by salary in descending order within each partition.
4. **Window Frame Specification**:
   * You can specify a window frame to define which rows are included in the calculation relative to the current row. This can be defined using ROWS or RANGE options.
   * Example: ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW specifies that the window frame includes all rows from the start of the partition up to the current row.

Window functions in MySQL are used to perform calculations across a set of table rows that are related to the current row. These functions are very useful for complex analytical queries. As of MySQL 8.0, several window functions are available, but they can be broadly categorized into three main types:

1. **Aggregate Window Functions**
2. **Ranking Window Functions**
3. **Value Window Functions**
4. **Statistical Window Functions**

### **1. Aggregate Window Functions**

**Examples:**

* SUM()
* AVG()
* MIN()
* MAX()
* COUNT()

**Usage:** Aggregate window functions compute aggregates over a specified window of rows, providing an aggregate value that can vary depending on the current row's position within the window.

**Syntax Example:**

SELECT

employee\_id,

department\_id,

salary,

SUM(salary) OVER (PARTITION BY department\_id) AS department\_salary\_total

FROM employees;

**When to Use:** Use aggregate window functions when you need to calculate summary statistics (like sums, averages) that apply to a subset of rows, partitioned by some criteria, while still showing individual row details.

**Why to Use:** They allow you to perform calculations without collapsing the result set into a single row per group, as with regular aggregate functions. This means you can see the aggregated values alongside the individual row data.

Aggregate window functions in SQL allow you to compute aggregate values over a specified window of rows while still retaining individual row details. These functions provide a way to calculate statistics over a set of rows that relate to each individual row within the result set.

### **Common Aggregate Window Functions**

Here’s a detailed explanation of some commonly used aggregate window functions, along with examples:

#### **1. SUM()**

**Function**: Calculates the total sum of a numeric column within the specified window.

**Syntax**:

SUM(expression) OVER (PARTITION BY partition\_expression ORDER BY order\_expression ROWS frame\_specification)

**Example**: Suppose you have a table of sales data and you want to calculate the running total of sales for each region.

CREATE TABLE sales (

sale\_id INT,

region VARCHAR(50),

sale\_amount DECIMAL(10, 2),

sale\_date DATE

);

INSERT INTO sales (sale\_id, region, sale\_amount, sale\_date) VALUES

(1, 'North', 100, '2024-01-01'),

(2, 'North', 150, '2024-01-02'),

(3, 'North', 200, '2024-01-03'),

(4, 'South', 300, '2024-01-01'),

(5, 'South', 400, '2024-01-02'),

(6, 'South', 500, '2024-01-03');

SELECT

sale\_id,

region,

sale\_amount,

SUM(sale\_amount) OVER (PARTITION BY region ORDER BY sale\_date) AS running\_total

FROM sales;

**Explanation**:

* **SUM(sale\_amount) OVER (PARTITION BY region ORDER BY sale\_date)**: Computes the running total of sales within each region, ordered by sale date.

#### **2. AVG()**

**Function**: Computes the average value of a numeric column within the specified window.

**Syntax**:

AVG(expression) OVER (PARTITION BY partition\_expression ORDER BY order\_expression ROWS frame\_specification)

**Example**: Suppose you want to calculate the average sale amount per day for each region.

SELECT

sale\_id,

region,

sale\_amount,

AVG(sale\_amount) OVER (PARTITION BY region ORDER BY sale\_date ROWS BETWEEN 1 PRECEDING AND CURRENT ROW) AS moving\_avg

FROM sales;

**Explanation**:

* **AVG(sale\_amount) OVER (PARTITION BY region ORDER BY sale\_date ROWS BETWEEN 1 PRECEDING AND CURRENT ROW)**: Calculates a moving average of sales amounts, considering the current row and the previous row within each region.

#### 

#### **3. MIN()**

**Function**: Returns the minimum value of a column within the specified window.

**Syntax**:

MIN(expression) OVER (PARTITION BY partition\_expression ORDER BY order\_expression ROWS frame\_specification)

**Example**: To find the minimum sale amount for each region up to the current date:

SELECT

sale\_id,

region,

sale\_amount,

MIN(sale\_amount) OVER (PARTITION BY region ORDER BY sale\_date ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS min\_amount

FROM sales;

**Explanation**:

* **MIN(sale\_amount) OVER (PARTITION BY region ORDER BY sale\_date ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)**: Provides the minimum sale amount seen up to the current row in each region.

#### 

#### 

#### 

#### **4. MAX()**

**Function**: Computes the maximum value of a column within the specified window.

**Syntax**:

MAX(expression) OVER (PARTITION BY partition\_expression ORDER BY order\_expression ROWS frame\_specification)

**Example**: To calculate the maximum sale amount for each region up to the current date:

SELECT

sale\_id,

region,

sale\_amount,

MAX(sale\_amount) OVER (PARTITION BY region ORDER BY sale\_date ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS max\_amount

FROM sales;

**Explanation**:

* **MAX(sale\_amount) OVER (PARTITION BY region ORDER BY sale\_date ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)**: Finds the maximum sale amount observed up to the current row within each region.

#### **5. COUNT()**

**Function**: Counts the number of rows within the specified window.

**Syntax**:

COUNT(expression) OVER (PARTITION BY partition\_expression ORDER BY order\_expression ROWS frame\_specification)

**Example**: To count the number of sales entries up to the current date for each region:

SELECT

sale\_id,

region,

sale\_amount,

COUNT(sale\_id) OVER (PARTITION BY region ORDER BY sale\_date ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS sale\_count

FROM sales;

**Explanation**:

* **COUNT(sale\_id) OVER (PARTITION BY region ORDER BY sale\_date ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)**: Counts the number of sales entries up to the current row within each region.

### **Summary**

* **Aggregate Window Functions**: Compute aggregate values over a defined window of rows.
* **SUM()**: Total sum of values.
* **AVG()**: Average of values.
* **MIN()**: Minimum value.
* **MAX()**: Maximum value.
* **COUNT()**: Count of rows.

Each function is powerful for generating detailed analytics and summaries directly in SQL queries, allowing for sophisticated data analysis while keeping the row-level detail intact.

### **2. Ranking Window Functions**

**Examples:**

* ROW\_NUMBER()
* RANK()
* DENSE\_RANK()
* NTILE()

**Usage:** Ranking functions assign a rank to each row within a partition of the result set. They are useful for sorting data and providing rank-based information.

**Syntax Example:**

SELECT

employee\_id,

department\_id,

salary,

RANK() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS salary\_rank

FROM employees;

**When to Use:** Use ranking functions when you need to assign a rank to rows based on their order within a partition, such as identifying the top performers in each department.

**Why to Use:** They help in scenarios where you need to sort and rank data, providing insight into the relative position of each row within a specified group.

Ranking window functions in SQL are used to assign a rank or order to rows within a result set, often based on some criteria or column values. Unlike aggregate functions that summarize data, ranking functions provide a way to assign a rank or ordinal value to rows, which can be useful for tasks such as determining the top performers, finding the highest or lowest values, or simply sorting data within partitions.

### **Common Ranking Window Functions**

Here’s a detailed explanation of some commonly used ranking window functions, along with examples:

#### **1. ROW\_NUMBER()**

**Function**: Assigns a unique sequential integer to rows within a partition, based on the order specified.

**Syntax**:

ROW\_NUMBER() OVER (PARTITION BY partition\_expression ORDER BY order\_expression)

**Example**: Suppose you have a table of employees and you want to assign a unique row number to each employee within their department based on their salary.

CREATE TABLE employees (

employee\_id INT,

department\_id INT,

salary DECIMAL(10, 2)

);

INSERT INTO employees (employee\_id, department\_id, salary) VALUES

(1, 1, 50000),

(2, 1, 55000),

(3, 1, 60000),

(4, 2, 70000),

(5, 2, 75000),

(6, 2, 80000);

SELECT

employee\_id,

department\_id,

salary,

ROW\_NUMBER() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS row\_num

FROM employees;

**Explanation**:

* **ROW\_NUMBER() OVER (PARTITION BY department\_id ORDER BY salary DESC)**: Assigns a unique row number to each employee within their department, ordered by salary in descending order. The highest salary gets a rank of 1 within each department.

#### 

#### 

#### 

#### **2. RANK()**

**Function**: Assigns a rank to rows within a partition, with the same rank assigned to rows with equal values. The next rank value is skipped if there are ties.

**Syntax**:

RANK() OVER (PARTITION BY partition\_expression ORDER BY order\_expression)

**Example**: To rank employees within each department by salary, allowing for ties:

SELECT

employee\_id,

department\_id,

salary,

RANK() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS rank

FROM employees;

**Explanation**:

* **RANK() OVER (PARTITION BY department\_id ORDER BY salary DESC)**: Ranks employees within each department by salary in descending order. Ties receive the same rank, and the rank number is incremented by the number of tied rows.

#### **3. DENSE\_RANK()**

**Function**: Similar to RANK(), but does not skip rank values when there are ties. The next rank value is consecutive.

**Syntax**:

DENSE\_RANK() OVER (PARTITION BY partition\_expression ORDER BY order\_expression)

**Example**: To rank employees within each department by salary, ensuring consecutive ranks even with ties:

SELECT

employee\_id,

department\_id,

salary,

DENSE\_RANK() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS dense\_rank

FROM employees;

**Explanation**:

* **DENSE\_RANK() OVER (PARTITION BY department\_id ORDER BY salary DESC)**: Assigns ranks to employees within each department. Ties receive the same rank, but the next rank value is not skipped (i.e., consecutive ranks).

#### **4. NTILE()**

**Function**: Divides the result set into a specified number of approximately equal parts (buckets) and assigns a unique bucket number to each row.

**Syntax**:

NTILE(number\_of\_buckets) OVER (PARTITION BY partition\_expression ORDER BY order\_expression)

**Example**: To divide employees within each department into 3 salary buckets:

SELECT

employee\_id,

department\_id,

salary,

NTILE(3) OVER (PARTITION BY department\_id ORDER BY salary DESC) AS salary\_bucket

FROM employees;

**Explanation**:

* **NTILE(3) OVER (PARTITION BY department\_id ORDER BY salary DESC)**: Divides employees within each department into 3 groups based on their salary. Each employee is assigned a bucket number (1, 2, or 3) representing their rank within the department.

### **Summary**

* **Ranking Window Functions**: Assign ranks or orders to rows within a result set based on a specified ordering.
* **ROW\_NUMBER()**: Provides a unique sequential number to rows, based on the specified order.
* **RANK()**: Assigns ranks, allowing for ties, with gaps in rank values if there are ties.
* **DENSE\_RANK()**: Similar to RANK(), but does not skip rank values in the presence of ties.
* **NTILE()**: Divides the result set into a specified number of groups (buckets) and assigns a bucket number to each row.

These functions are powerful tools for analyzing and sorting data within groups, allowing for sophisticated ranking and partitioning directly in SQL queries.

### **3. Value Window Functions**

**Examples:**

* LEAD()
* LAG()
* FIRST\_VALUE()
* LAST\_VALUE()
* **NTH\_VALUE()**:

**Usage:** Value window functions access data from other rows within the same result set to provide comparative or sequential values. They are useful for comparing values between rows.

**Syntax Example:**

SELECT

employee\_id,

salary,

LAG(salary, 1) OVER (ORDER BY employee\_id) AS previous\_salary

FROM employees;

**When to Use:** Use value functions when you need to refer to values from preceding or following rows, or get the first or last value within a partition.

**Why to Use:** They are particularly useful for trend analysis, comparing current row values with those of previous or next rows, and for more advanced analytical queries.

Value window functions in SQL are used to access data from other rows within the result set relative to the current row. They are particularly useful for comparative analysis, such as finding previous or next values, or retrieving the first or last values within a specified window. Unlike aggregate or ranking functions, which provide summary statistics or ranks, value functions focus on the relative position of data within the result set.

### 

### **Common Value Window Functions**

Here’s a detailed explanation of some commonly used value window functions, along with examples:

#### **1. LAG()**

**Function**: Provides access to a value from a preceding row in the same result set.

**Syntax**:

LAG(expression, offset, default) OVER (PARTITION BY partition\_expression ORDER BY order\_expression)

* **expression**: The column or expression whose value you want to retrieve.
* **offset**: Number of rows back from the current row to access. Defaults to 1 if not specified.
* **default**: Value to return if the offset goes out of bounds. Defaults to NULL if not specified.

**Example**: To find each employee’s salary compared to their previous salary in the same department:

SELECT

employee\_id,

department\_id,

salary,

LAG(salary, 1) OVER (PARTITION BY department\_id ORDER BY salary) AS previous\_salary

FROM employees;

**Explanation**:

* **LAG(salary, 1)**: Retrieves the salary of the previous row within the same department, ordered by salary. If there is no previous row, it returns NULL.

#### **2. LEAD()**

**Function**: Provides access to a value from a following row in the same result set.

**Syntax**:

LEAD(expression, offset, default) OVER (PARTITION BY partition\_expression ORDER BY order\_expression)

* **expression**: The column or expression whose value you want to retrieve.
* **offset**: Number of rows forward from the current row to access. Defaults to 1 if not specified.
* **default**: Value to return if the offset goes out of bounds. Defaults to NULL if not specified.

**Example**: To find each employee’s salary compared to their next salary in the same department:

SELECT

employee\_id,

department\_id,

salary,

LEAD(salary, 1) OVER (PARTITION BY department\_id ORDER BY salary) AS next\_salary

FROM employees;

**Explanation**:

* **LEAD(salary, 1)**: Retrieves the salary of the next row within the same department, ordered by salary. If there is no next row, it returns NULL.

#### **3. FIRST\_VALUE()**

**Function**: Retrieves the first value in the window frame based on the specified ordering.

**Syntax**:

FIRST\_VALUE(expression) OVER (PARTITION BY partition\_expression ORDER BY order\_expression ROWS frame\_specification)

**Example**: To get the highest salary in each department for every employee:

SELECT

employee\_id,

department\_id,

salary,

FIRST\_VALUE(salary) OVER (PARTITION BY department\_id ORDER BY salary DESC ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS highest\_salary

FROM employees;

**Explanation**:

* **FIRST\_VALUE(salary)**: Retrieves the highest salary in the department. The window frame is defined from the start to the end of the partition to ensure it captures the maximum salary for comparison.

#### 

#### **4. LAST\_VALUE()**

**Function**: Retrieves the last value in the window frame based on the specified ordering.

**Syntax**:

LAST\_VALUE(expression) OVER (PARTITION BY partition\_expression ORDER BY order\_expression ROWS frame\_specification)

**Example**: To get the lowest salary in each department for every employee:

SELECT

employee\_id,

department\_id,

salary,

LAST\_VALUE(salary) OVER (PARTITION BY department\_id ORDER BY salary ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS lowest\_salary

FROM employees;

**Explanation**:

* **LAST\_VALUE(salary)**: Retrieves the lowest salary in the department. The window frame is defined from the start to the end of the partition to ensure it captures the minimum salary for comparison.

### **3. NTH\_VALUE()**

**Function**: Retrieves the value of the nth row in the specified window frame. Useful for getting a specific value from a sequence of rows.

**Syntax**:

NTH\_VALUE(expression, n) OVER (PARTITION BY partition\_expression ORDER BY order\_expression ROWS frame\_specification)

* **expression**: The column or expression from which to retrieve the value.
* **n**: The position of the value in the window frame.

**Example**: To find the third highest salary within each department:

SELECT

employee\_id,

department\_id,

salary,

NTH\_VALUE(salary, 3) OVER (PARTITION BY department\_id ORDER BY salary DESC ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS third\_highest\_salary

FROM employees;

**Explanation**:

* **NTH\_VALUE(salary, 3)**: Retrieves the third highest salary within each department, considering the entire window frame.

**Summary**

* **Value Window Functions**: Allow access to data from other rows relative to the current row, enabling comparative and trend analysis.
* **LAG()**: Provides a value from a preceding row within the result set.
* **LEAD()**: Provides a value from a following row within the result set.
* **FIRST\_VALUE()**: Retrieves the first value in the window frame.
* **LAST\_VALUE()**: Retrieves the last value in the window frame.

These functions are essential for performing detailed, row-by-row comparative analysis and for generating insights that involve sequences and trends in your data. They are powerful tools for enhancing your SQL queries with sophisticated data analysis capabilities.

4 . **Statistical window functions**

Statistical window functions in SQL are used to perform advanced statistical analyses within partitions of a dataset. They are particularly useful for understanding data distributions, calculating percentiles, and evaluating relative positions within a dataset. Here’s a detailed explanation of some common statistical window functions:

### **1. CUME\_DIST()**

**Function**: Computes the cumulative distribution of a value within a partition. It calculates the relative rank of a value as a fraction of the number of values less than or equal to it.

**Syntax**:

CUME\_DIST() OVER (PARTITION BY partition\_expression ORDER BY order\_expression)

**Example**: To find the cumulative distribution of each employee’s salary within their department:

SELECT

employee\_id,

department\_id,

salary,

CUME\_DIST() OVER (PARTITION BY department\_id ORDER BY salary) AS cum\_dist

FROM employees;

**Explanation**:

* **CUME\_DIST()**: Returns a value between 0 and 1 indicating the proportion of rows with values less than or equal to the current row's value within the specified partition. For instance, if an employee's salary is at the 90th percentile, CUME\_DIST() would return 0.9, indicating that 90% of the salaries are less than or equal to this employee’s salary.

### **2. PERCENT\_RANK()**

**Function**: Calculates the relative rank of a value as a percentage within a partition. It shows the rank of a value relative to other values in the same partition.

**Syntax**:

PERCENT\_RANK() OVER (PARTITION BY partition\_expression ORDER BY order\_expression)

**Example**: To determine the percent rank of each employee’s salary within their department:

SELECT

employee\_id,

department\_id,

salary,

PERCENT\_RANK() OVER (PARTITION BY department\_id ORDER BY salary) AS percent\_rank

FROM employees;

**Explanation**:

* **PERCENT\_RANK()**: Calculates the relative rank of each salary as a percentage of the total number of salaries in the partition. It ranges from 0 to 1. For example, if an employee's salary is at the 50th percentile, PERCENT\_RANK() would return 0.5, indicating that the salary is in the middle of the distribution.

### **3. PERCENTILE\_CONT()**

**Function**: Computes a specific percentile (continuous percentile) based on interpolation. This function returns the value at the specified percentile by interpolating between values if necessary.

**Syntax**:

PERCENTILE\_CONT(fraction) WITHIN GROUP (ORDER BY order\_expression) OVER (PARTITION BY partition\_expression)

* **fraction**: A number between 0 and 1 representing the percentile to calculate (e.g., 0.5 for the 50th percentile or median).
* **WITHIN GROUP (ORDER BY order\_expression)**: Defines the order in which the percentile is calculated.

**Example**: To find the median salary (50th percentile) within each department:

SELECT

employee\_id,

department\_id,

salary,

PERCENTILE\_CONT(0.5) WITHIN GROUP (ORDER BY salary) OVER (PARTITION BY department\_id) AS median\_salary

FROM employees;

**Explanation**:

* **PERCENTILE\_CONT(0.5)**: Calculates the median salary within each department. If there is no exact median value, it interpolates between the nearest values to determine the median.

### **4. PERCENTILE\_DISC()**

**Function**: Computes a specific percentile (discrete percentile) by selecting an actual value from the set. It does not interpolate but picks the nearest value corresponding to the specified percentile.

**Syntax**:

PERCENTILE\_DISC(fraction) WITHIN GROUP (ORDER BY order\_expression) OVER (PARTITION BY partition\_expression)

* **fraction**: A number between 0 and 1 representing the percentile to calculate.
* **WITHIN GROUP (ORDER BY order\_expression)**: Defines the order for the calculation.

**Example**: To find the 90th percentile salary within each department:

SELECT

employee\_id,

department\_id,

salary,

PERCENTILE\_DISC(0.9) WITHIN GROUP (ORDER BY salary) OVER (PARTITION BY department\_id) AS percentile\_90\_salary

FROM employees;

**Explanation**:

* **PERCENTILE\_DISC(0.9)**: Calculates the salary at the 90th percentile within each department. It selects the salary value closest to the 90th percentile position without interpolation.

### 

### **Summary**

* **CUME\_DIST()**: Calculates the cumulative distribution of a value, showing the proportion of rows with values less than or equal to the current value within a partition.
* **PERCENT\_RANK()**: Provides the relative rank of a value as a percentage within a partition, indicating its position compared to other values.
* **PERCENTILE\_CONT()**: Computes continuous percentiles by interpolating between values, useful for finding specific percentiles like the median.
* **PERCENTILE\_DISC()**: Calculates discrete percentiles by selecting actual values from the dataset, without interpolation.

These statistical window functions help in analyzing data distributions, making them valuable for advanced data analysis tasks where understanding the position of values within a dataset is crucial.