SQL Window Functions

LESSON

Generate a quiz about this unit

Summarize this unit

Have any doubt about this content? Ask!

Learning Objectives

By the end of this lesson, you will be able to:

- Explain how window functions work and describe their utility.
- Apply various window functions in MySQL to perform calculations across sets of table rows related to the current row.
- Utilize different window frame specifications to refine calculations.
- Combine window functions with other SQL clauses for advanced data analysis.

Introduction to Window Functions

Window functions, sometimes referred to as analytical or OLAP (Online Analytical Processing) functions, allow you to perform calculations across a set of rows related to the current row, without collapsing groups of rows into a single output row like an aggregate function does. They operate within a "window" of rows defined by the `OVER` clause.

Basic Syntax

Copy

```sql
SELECT column1, column2,
       WINDOW_FUNCTION(column3) OVER (
           PARTITION BY column4
           ORDER BY column5
           [window_frame_clause]
       ) AS alias_name
FROM tablename;
```

Explain this code

1. **PARTITION BY**: Divides the result set into partitions (groups) to which the window function is applied. Think of it as a way to apply the window function to "subgroups" of your result set.
2. **ORDER BY**: Defines an order within each partition.
3. **Frame Specification**: Defines which rows are included in the window to be processed by the window function relative to the current row.

Common Window Functions

1. ROW_NUMBER()

Assigns a unique sequential integer to each row within its partition.

```sql
SELECT column1, column2,
       ROW_NUMBER() OVER (ORDER BY column3) AS row_num
FROM tablename;
```

Copy

Explain this code

Example:

Here is a table `sales` with columns `date`, `salesperson_id`, and `sales_amount`.

date	salesperson_id	sales_amount
2023-09-01	1	100
2023-09-02	1	150
2023-09-01	2	200

Here is how to assign a unique sequential number to each sale made by each salesperson ordered by date:

```
SELECT date, salesperson_id, sales_amount,
       ROW_NUMBER() OVER (PARTITION BY salesperson_id ORDER BY date) as r
FROM sales;
```

Copy

Explain this code

This is the result:

date	salesperson_id	sales_amount	row_num
2023-09-01	1	100	1
2023-09-02	1	150	2
2023-09-01	2	200	1

▶ Detailed explanation

2. RANK() and DENSE_RANK()

Assigns a unique rank to each distinct row, with the same rank assigned to rows with the same values. If two rows have the same rank, the next rank is skipped. `DENSE_RANK()` is similar to `RANK()`, but without skipping any ranks.

```
SELECT column1, column2,
       RANK() OVER (ORDER BY column3) AS rank_num,
       DENSE_RANK() OVER (ORDER BY column3) AS dense_rank_num
FROM tablename;
```

Copy

Explain this code

Example:

Take a look at this table:

salesperson_id	total_sales
1	250
2	200

salesperson_id	total_sales
3	250

Let's calculate the total sales for each salesperson and then rank them based on the total sales in descending order.

```
SELECT salesperson_id, SUM(sales_amount) as total_sales,
       RANK() OVER (ORDER BY SUM(sales_amount) DESC) as rank,
       DENSE_RANK() OVER (ORDER BY SUM(sales_amount) DESC) as dense_rank
FROM sales
GROUP BY salesperson_id;
```

Copy

Explain this code

This is the result:

salesperson_id	total_sales	rank	dense_rank
1	250	1	1
3	250	1	1
2	200	3	2

▶ Detailed explanation

3. NTILE(n)

Divides the result set into 'n' number of parts with approximately equal number of rows.

```
SELECT column1, column2,
       NTILE(4) OVER (ORDER BY column3) AS quartile
FROM tablename;
```

Copy

Explain this code

Example:

Take a look at this table:

salesperson_id	total_sales
1	300
2	200
3	400
4	100

Let's categorize salespeople into two groups based on their total sales, dividing them into lower and upper halves. This can be useful for a quick bifurcation (split into two groups) to understand which salespeople fall below the median in terms of sales and which ones are above the median.

```
SELECT salesperson_id, SUM(sales_amount) as total_sales,
       NTILE(2) OVER (ORDER BY SUM(sales_amount)) as half
FROM sales
GROUP BY salesperson_id;
```

Copy

Explain this code

The result would be:

salesperson_id	total_sales	quartile
4	100	1
2	200	1
1	300	2
3	400	2

▶ Detailed explanation

4. LEAD() and LAG()

LEAD() and LAG() functions allow you to access data from subsequent or preceding rows, respectively.

```
SELECT column1, column2,
       LEAD(column3, 1) OVER (ORDER BY column4) AS next_value,
       LAG(column3, 1) OVER (ORDER BY column4) AS prev_value
FROM tablename;
```

Copy

Explain this code

Example:

Imagine you have a sales team and you want to analyze the monthly performance of each salesperson. Specifically, you would like to understand how the sales of each individual has changed from one month to the next.

Here's a sample dataset from the `monthly_sales` table:

month	salesperson_id	monthly_sales
January	1	100
February	1	120
March	1	110
January	2	200
February	2	210
March	2	195

You want to determine the difference in `monthly_sales` for each `salesperson_id` from one month to the next.

Using `LAG()`

To achieve this, we can use the `LAG()` function to compare the `monthly_sales` of the current month with the `monthly_sales` of the previous month:

```
SELECT month, salesperson_id, monthly_sales,
       monthly_sales - LAG(monthly_sales) OVER (PARTITION BY salesperson_
FROM monthly_sales
ORDER BY salesperson_id, month;
```

Copy

Explain this code

This is the result:

month	salesperson_id	monthly_sales	sales_difference
January	1	100	NULL

month	salesperson_id	monthly_sales	sales_difference
February	1	120	20
March	1	110	-10
January	2	200	NULL
February	2	210	10
March	2	195	-15

This result table illustrates:

- For `salesperson_id` 1, sales increased by 20 from January to February and then decreased by 10 from February to March.
- For `salesperson_id` 2, sales increased by 10 from January to February and then decreased by 15 from February to March.

The `LAG()` function provides insights into the month-over-month sales difference, helping to quickly identify trends and performance changes for each salesperson.

Window Frame Specification

Window frame specifications allow you to define which rows are included in the frame for each row's calculation. Common specifications include:

- **ROWS BETWEEN n PRECEDING AND m FOLLOWING**: Includes the 'n' rows before the current row and the 'm' rows after the current row.
- **ROWS UNBOUNDED PRECEDING/FOLLOWING**: Includes all rows from the start to the current row or from the current row to the end, respectively.

Example:

Here is how to calculate a moving average:

```
SELECT date, sales,
       AVG(sales) OVER (
           ORDER BY date
           ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
       ) AS moving_avg
FROM sales_data;
```

Copy

Explain this code

Summary

Window functions are powerful tools for analytics and complex calculations in SQL without collapsing multiple rows into a single output row. They let you perform calculations across related data points, making them indispensable for time series analyzes, rankings, and other analytical challenges. Effectively using window functions in MySQL enables you to derive deeper insights from your data and perform advanced data processing tasks with ease.

Any doubt? Ask our AI Chatbot!

Mark as completed

How would you rate the content of this unit?

PREVIOUS
← SQL Temporary Tables, Views and CTEs Hands On

NEXT
SQL Window Functions Hands On →