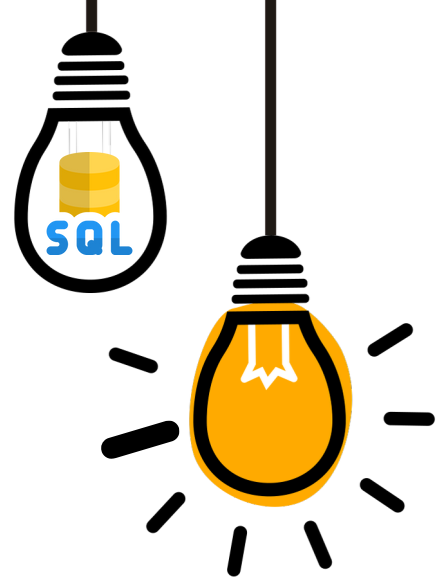
Tarini Prasad Das



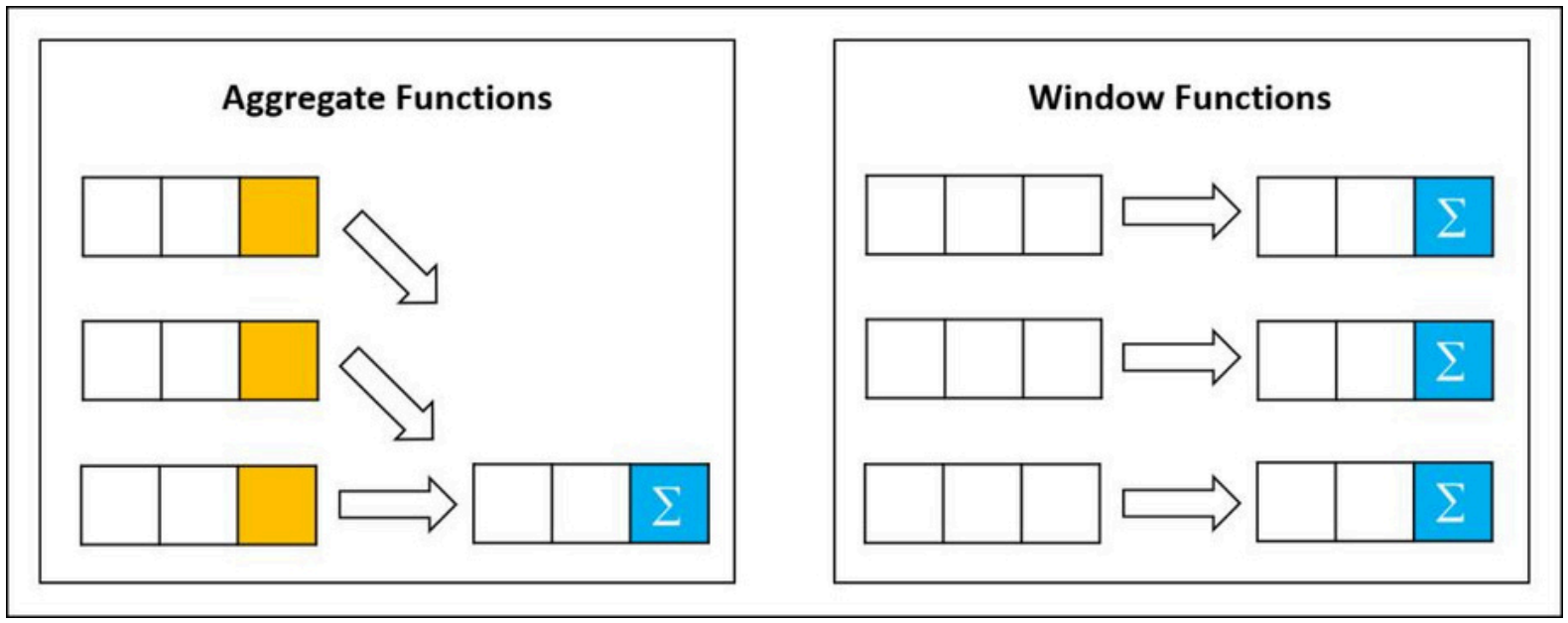
SQL

WINDOW FUNCTIONS

MySQL®

What is a Window Function?

A **Window Function** in SQL is a powerful tool that allows you to perform calculations across a set of table rows related to the current row. Unlike aggregate functions that return a single result for a group, window functions return a value for each row while still being able to access related rows' data. This makes them ideal for complex analytics, ranking, running totals, and more.



Syntax for Window Functions

The basic syntax for a window function is:

```sql
<window_function>() OVER (
    [PARTITION BY partition_expression]
    [ORDER BY sort_expression]
    [window_frame_clause]
)
```

- **<window_function>():** This is the function you are applying, like ROW_NUMBER(), RANK(), etc.
- **PARTITION BY:** Divides the result set into partitions to which the window function is applied. (optional)
- **ORDER BY:** Specifies the order of rows in each partition. It's mandatory for ranking functions.(optional)
- **window_frame_clause:** Defines a subset of rows within the partition that the function operates on, like ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.(optional)

Sample Table : Sales

Let's consider a sample Sales table:

order_id	order_date	product_code	quantity	sale_price
1	2024-08-01	P001	10	100
2	2024-08-01	P002	5	200
3	2024-08-02	P001	8	100
4	2024-08-02	P003	12	150
5	2024-08-03	P002	7	200

Different Types of Window Functions

1. Aggregate Functions

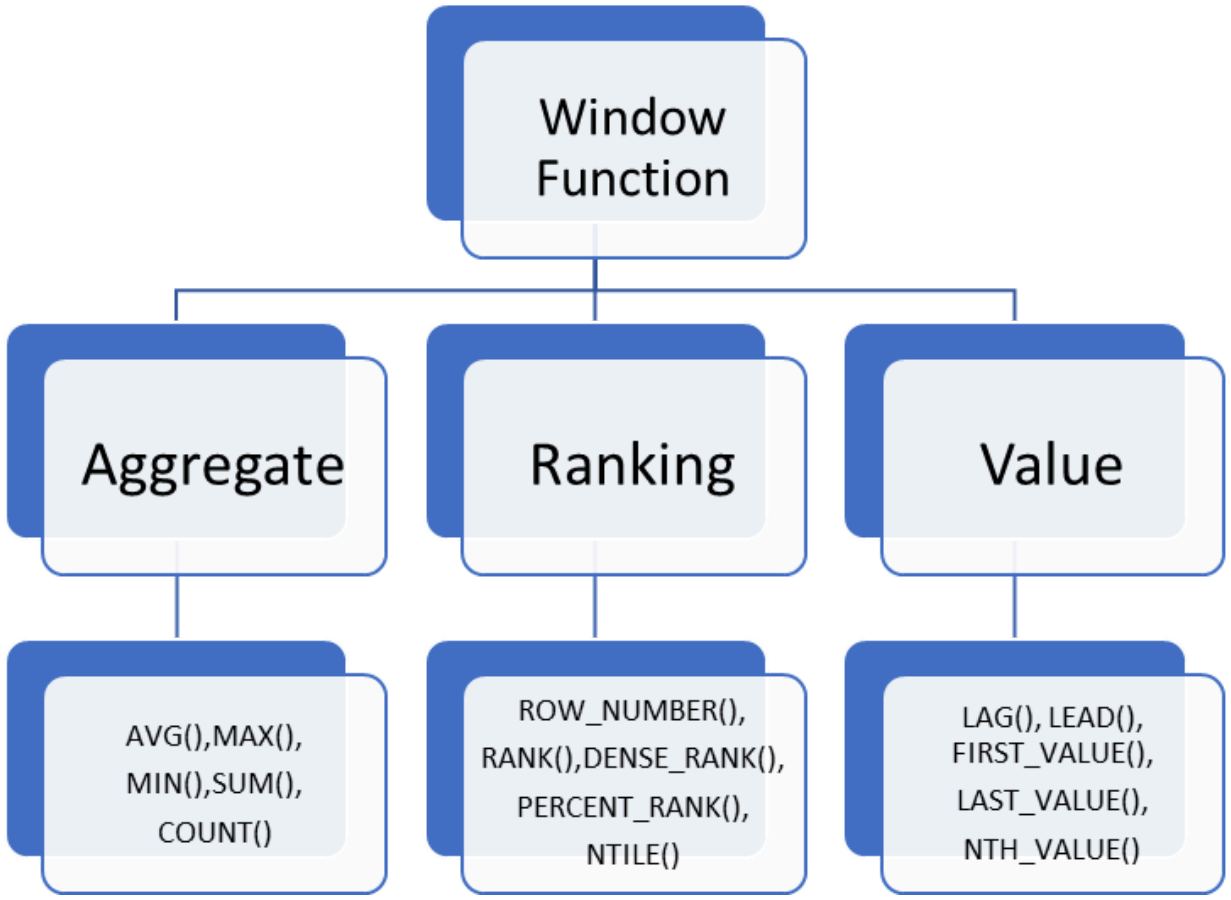
- **SUM()**
- **AVG()**
- **MIN() and MAX()**
- **COUNT()**

2. Ranking Functions

- **ROW_NUMBER()**
- **RANK()**
- **DENSE_RANK():**
- **NTILE(n)**

3. Value Functions

- **LEAD()**
- **LAG()**
- **FIRST_VALUE()**
- **LAST_VALUE()**
- **NTH_VALUE()**



Tarini Prasad Das

1. Aggregate Functions

An aggregate window function performs aggregation across a specified set of rows defined by the window frame. This window frame is determined by the PARTITION BY, ORDER BY, and ROWS or RANGE clauses.

Common Aggregate Window Functions

- SUM(): Calculates the sum of values.
- AVG(): Computes the average of values.
- COUNT(): Counts the number of rows.
- MIN(): Finds the minimum value.
- MAX(): Finds the maximum value.

Syntax :

```sql
aggregate_function(column) OVER (
    [PARTITION BY partition_expression]
    [ORDER BY sort_expression]
    [ROWS frame_specification]
)
```

Example Query :

Let’s calculate the running total and average sale price using aggregate window functions.

Tarini Prasad Das

a. Average Sale Price :

Here, AVG(sale_price) OVER () calculates the average of sale_price across all rows.

sql Copy code

```sql
SELECT
    order_id,
    order_date,
    product_code,
    quantity,
    sale_price,
    AVG(sale_price) OVER () AS avg_sale_price
FROM sales;
```

Output :

order_id	order_date	product_code	quantity	sale_price	avg_sale_price
1	2024-08-01	P001	10	100	150
2	2024-08-01	P002	5	200	150
3	2024-08-02	P001	8	100	150
4	2024-08-02	P003	12	150	150
5	2024-08-03	P002	7	200	150

Explanation:

- Global Average: Computes the average sale price across all rows in the dataset.
- Uniform Value: Provides the same average value for each row, reflecting the overall average price.

b.Using ORDER BY

Here, SUM(quantity) OVER (ORDER BY order_date) calculates the cumulative sum of quantity, ordered by order_date.

Example : Running Total of Quantity

sql Copy code

```sql
SELECT
    order_id,
    order_date,
    product_code,
    quantity,
    sale_price,
    SUM(quantity) OVER (ORDER BY order_date) AS running_total
FROM sales;
```

Output :

order_id	order_date	product_code	quantity	sale_price	running_total
1	2024-08-01	P001	10	100	10
2	2024-08-01	P002	5	200	15
3	2024-08-02	P001	8	100	23
4	2024-08-02	P003	12	150	35
5	2024-08-03	P002	7	200	42

Explanation:

- Cumulative Sum: Calculates a cumulative sum of the quantity column, adding up all quantities from the start up to the current row, based on the order of order_date.
- Preservation of Rows: Maintains the original rows and their data while computing the running total.

c.Using PARTITION BY

You can also use PARTITION BY to compute aggregates within specific partitions.

Example: Running Total by Product Code

```sql
SELECT
    order_id,
    order_date,
    product_code,
    quantity,
    sale_price,
    SUM(quantity) OVER (PARTITION BY product_code ORDER BY order_date) AS running_t
FROM sales;
```

Output :

order_id	order_date	product_code	quantity	sale_price	running_total_by_product
1	2024-08-01	P001	10	100	10
3	2024-08-02	P001	8	100	18
2	2024-08-01	P002	5	200	5
5	2024-08-03	P002	7	200	12
4	2024-08-02	P003	12	150	12

Explanation:

- Partitioned Running Total: Calculates the running total of quantities separately for each product_code, based on the order of order_date.
- Product-wise Calculation: Only sums quantities within the same product code partition, keeping totals distinct by product.

d.Query with All Aggregate Window Functions

Here's a query that demonstrates the use of several aggregate window functions on the sales dataset:

sql Copy code

```sql
SELECT
    order_id,
    order_date,
    product_code,
    quantity,
    sale_price,
    SUM(quantity) OVER (ORDER BY order_date) AS running_total,
    AVG(sale_price) OVER () AS avg_sale_price,
    COUNT(*) OVER () AS total_count,
    MIN(sale_price) OVER () AS min_sale_price,
    MAX(sale_price) OVER () AS max_sale_price
FROM sales;
```

Output :

order_id	order_date	product_code	quantity	sale_price	running_total	avg_sale_price	total_count	min_sale_price	max_sale_price
1	2024-08-01	P001	10	100	10	150	5	100	200
2	2024-08-01	P002	5	200	15	150	5	100	200
3	2024-08-02	P001	8	100	23	150	5	100	200
4	2024-08-02	P003	12	150	35	150	5	100	200
5	2024-08-03	P002	7	200	42	150	5	100	200

These output tables illustrate how each aggregate window function processes the data, providing various analytical insights while maintaining the detail of each row.

2. Ranking Window Functions

1. RANK()

The RANK() function assigns a unique rank to each row within a partition, with gaps in ranking values if there are ties. If two or more rows have the same rank, the next rank(s) are skipped.

2. DENSE_RANK()

The DENSE_RANK() function also assigns ranks to rows within a partition but does not leave gaps in the ranking values for ties. Each distinct rank is assigned consecutively without skipping.

3. ROW_NUMBER()

The ROW_NUMBER() function assigns a unique sequential integer to rows within a partition, with no gaps. It is the simplest ranking function and does not handle ties.

Example Query

Let's use the RANK(), DENSE_RANK(), and ROW_NUMBER() functions to rank orders based on sale_price.

sql Copy code

```sql
SELECT
    order_id,
    order_date,
    product_code,
    quantity,
    sale_price,
    RANK() OVER (ORDER BY sale_price DESC) AS rank,
    DENSE_RANK() OVER (ORDER BY sale_price DESC) AS dense_rank,
    ROW_NUMBER() OVER (ORDER BY sale_price DESC) AS row_num
FROM sales;
```

Explanation

- **RANK():** Assigns a rank to each row, with ties receiving the same rank and gaps in the ranking sequence.
- **DENSE_RANK():** Assigns a rank to each row, with ties receiving the same rank and no gaps in the ranking sequence.
- **ROW_NUMBER():** Assigns a unique sequential integer to each row, with no gaps, regardless of ties.

Output Table

Based on the above query, the output will be:

order_id	order_date	product_code	quantity	sale_price	rank	dense_rank	row_num
2	2024-08-01	P002	5	200	1	1	1
5	2024-08-03	P002	7	200	1	1	2
4	2024-08-02	P003	12	150	3	2	3
1	2024-08-01	P001	10	100	4	3	4
3	2024-08-02	P001	8	100	4	3	5

Summary

- RANK(): Rows with the same sale_price receive the same rank. The next rank after a tie has gaps (e.g., rank 1, 1, 3).
- DENSE_RANK(): Rows with the same sale_price receive the same rank, with no gaps in ranks (e.g., rank 1, 1, 2).
- ROW_NUMBER(): Each row receives a unique, sequential number (e.g., 1, 2, 3).

4. PERCENT_RANK()

The PERCENT_RANK() function calculates the relative rank of a row within a partition as a percentage. It measures the position of a row in a sorted dataset as a percentage of the total number of rows.

Formula:

$$\text{PERCENT\_RANK} = \frac{\text{Rank of current row} - 1}{\text{Total number of rows} - 1}$$

Where:

- Rank of current row: The rank assigned to the current row.
- Total number of rows: The total number of rows in the partition.

5. NTILE()

The NTILE() function divides the dataset into a specified number of approximately equal-sized buckets or tiles. It assigns a bucket number to each row.

Formula:

$$\text{NTILE}(n) = \text{Bucket number assigned to the row}$$

Where:

- n: Number of buckets or tiles to divide the dataset into.

Example Query

sql Copy code

```sql
SELECT
    order_id,
    order_date,
    product_code,
    quantity,
    sale_price,
    PERCENT_RANK() OVER (ORDER BY sale_price DESC) AS percent_rank,
    NTILE(4) OVER (ORDER BY sale_price DESC) AS ntile_bucket
FROM sales;
```

Explanation

- PERCENT_RANK() OVER (ORDER BY sale_price DESC): Calculates the percentage rank of each row based on sale_price in descending order.
- NTILE(4) OVER (ORDER BY sale_price DESC): Divides the rows into 4 buckets based on sale_price in descending order.

Output Table

order_id	order_date	product_code	quantity	sale_price	percent_rank	ntile_bucket
2	2024-08-01	P002	5	200	0.00	1
5	2024-08-03	P002	7	200	0.00	1
4	2024-08-02	P003	12	150	0.50	2
1	2024-08-01	P001	10	100	0.75	3
3	2024-08-02	P001	8	100	0.75	3

Summary

- percent_rank: Shows the relative position of each sale price as a percentage of the total number of rows. For instance, the highest sale price (200) has a percent_rank of 0.00, indicating it is at the top of the distribution.
- ntile_bucket: Divides the data into 4 buckets based on sale_price. For instance, the highest sale prices fall into bucket 1, while the lower sale prices fall into buckets 2 and 3.

Tarini Prasad Das

3. Value Window Functions

Value window functions in SQL are used to retrieve values from a set of rows related to the current row, based on specific criteria. These functions are essential for performing calculations that involve looking at the data around the current row.

Value window functions include:

- **LEAD():** Provides access to the value of a column in a subsequent row.
- **LAG():** Provides access to the value of a column in a preceding row.
- **FIRST_VALUE():** Returns the first value in a window frame.
- **LAST_VALUE():** Returns the last value in a window frame.
- **NTH_VALUE():** Returns the value of a specific row within the window frame, where N is the row number.

1. LEAD( )

The LEAD() function provides access to a value from a subsequent row in the result set. It's useful for comparing a row with future rows.

Syntax:

sql Copy code

```sql
LEAD(column_name, offset, default) OVER (
    [PARTITION BY partition_expression]
    [ORDER BY sort_expression]
)
```

- **column_name:** The column whose value you want to retrieve.
- **offset:** The number of rows forward from the current row (default is 1).
- **default:** The value to return if the offset goes beyond the end of the result set (default is NULL).

Example Query 1:

Retrieve the next sale price for each order:

```sql
SELECT
    order_id,
    order_date,
    product_code,
    quantity,
    sale_price,
    LEAD(sale_price) OVER (ORDER BY order_date) AS next_sale_price
FROM sales;
```

Output Table:

order_id	order_date	product_code	quantity	sale_price	next_sale_price
1	2024-08-01	P001	10	100	200
2	2024-08-01	P002	5	200	100
3	2024-08-02	P001	8	100	150
4	2024-08-02	P003	12	150	200
5	2024-08-03	P002	7	200	NULL

Explanation:

LEAD(sale_price) retrieves the sale_price from the next row ordered by order_date.

Example Query 2:

Retrieve the sale price of the row two steps ahead:

```sql
SELECT
    order_id,
    order_date,
    product_code,
    quantity,
    sale_price,
    LEAD(sale_price, 2) OVER (ORDER BY order_date) AS two_steps_ahead_sale_price
FROM sales;
```

Output Table:

order_id	order_date	product_code	quantity	sale_price	two_steps_ahead_sale_price
1	2024-08-01	P001	10	100	100
2	2024-08-01	P002	5	200	150
3	2024-08-02	P001	8	100	200
4	2024-08-02	P003	12	150	NULL
5	2024-08-03	P002	7	200	NULL

Explanation:

LEAD(sale_price, 2) retrieves the sale_price two rows ahead.

2. LAG()

The LAG() function provides access to a value from a preceding row in the result set. It's useful for comparing a row with previous rows.

Syntax:

```sql
LAG(column_name, offset, default) OVER (
    [PARTITION BY partition_expression]
    [ORDER BY sort_expression]
)
```

- **column_name:** The column whose value you want to retrieve.
- **offset:** The number of rows backward from the current row (default is 1).
- **default:** The value to return if the offset goes before the start of the result set (default is NULL).

Example Query 1:

Retrieve the previous sale price for each order:

```sql
SELECT
    order_id,
    order_date,
    product_code,
    quantity,
    sale_price,
    LAG(sale_price) OVER (ORDER BY order_date) AS previous_sale_price
FROM sales;
```

Output Table:

order_id	order_date	product_code	quantity	sale_price	previous_sale_price
1	2024-08-01	P001	10	100	NULL
2	2024-08-01	P002	5	200	100
3	2024-08-02	P001	8	100	200
4	2024-08-02	P003	12	150	100
5	2024-08-03	P002	7	200	150

Explanation:

LAG(sale_price) retrieves the sale_price from the previous row ordered by order_date.

Example Query 2:

Retrieve the sale price of the row two steps back:

sql Copy code

```sql
SELECT
    order_id,
    order_date,
    product_code,
    quantity,
    sale_price,
    LAG(sale_price, 2) OVER (ORDER BY order_date) AS two_steps_back_sale_price
FROM sales;
```

Output Table:

order_id	order_date	product_code	quantity	sale_price	two_steps_back_sale_price
1	2024-08-01	P001	10	100	NULL
2	2024-08-01	P002	5	200	NULL
3	2024-08-02	P001	8	100	100
4	2024-08-02	P003	12	150	200
5	2024-08-03	P002	7	200	150

Explanation:

LAG(sale_price, 2) retrieves the sale_price two rows back.

3. FIRST_VALUE()

The FIRST_VALUE() function returns the first value in the window frame, as defined by the ORDER BY clause.

Syntax:

```sql
FIRST_VALUE(column_name) OVER (
    [PARTITION BY partition_expression]
    [ORDER BY sort_expression]
    [ROWS frame_specification]
)
```

Example Query:

Retrieve the first sale price in the result set:

```sql
SELECT
    order_id,
    order_date,
    product_code,
    quantity,
    sale_price,
    FIRST_VALUE(sale_price) OVER (ORDER BY order_date) AS first_sale_price
FROM sales;
```

Output Table:

order_id	order_date	product_code	quantity	sale_price	first_sale_price
1	2024-08-01	P001	10	100	100
2	2024-08-01	P002	5	200	100
3	2024-08-02	P001	8	100	100
4	2024-08-02	P003	12	150	100
5	2024-08-03	P002	7	200	100

Explanation:

FIRST_VALUE(sale_price) returns the first sale_price according to the ORDER BY clause.

4. NTH_VALUE()

The NTH_VALUE() function returns the value of a specific row within the window frame, where N is the row number.

Syntax:

```sql
NTH_VALUE(column_name, N) OVER (
    [PARTITION BY partition_expression]
    [ORDER BY sort_expression]
    [ROWS frame_specification]
)
```

Example Query:

Retrieve the first sale price in the result set:

```sql
SELECT
    order_id,
    order_date,
    product_code,
    quantity,
    sale_price,
    NTH_VALUE(sale_price, 2) OVER (ORDER BY order_date) AS second_sale_price
FROM sales;
```

Output Table:

order_id	order_date	product_code	quantity	sale_price	second_sale_price
1	2024-08-01	P001	10	100	200
2	2024-08-01	P002	5	200	200
3	2024-08-02	P001	8	100	100
4	2024-08-02	P003	12	150	150
5	2024-08-03	P002	7	200	200

Explanation:

NTH_VALUE(sale_price, 2) returns the second sale_price in the result set.

4. LAST_VALUE()

The LAST_VALUE() function returns the last value in the window frame, as defined by the ORDER BY clause. It's important to specify the correct window frame to get the expected results.

Syntax:

```sql
LAST_VALUE(column_name) OVER (
    [PARTITION BY partition_expression]
    [ORDER BY sort_expression]
    [ROWS frame_specification]
)
```

Example Query:

Retrieve the last sale price in the window frame:

```sql
SELECT
    order_id,
    order_date,
    product_code,
    quantity,
    sale_price,
    LAST_VALUE(sale_price) OVER (ORDER BY order_date) AS last_sale_price
FROM sales;
```

Output Table:

order_id	order_date	product_co...	quantity	sale_price	last_sale_price
1	2024-08-01	P001	10	100.00	200.00
2	2024-08-01	P002	5	200.00	200.00
3	2024-08-02	P001	8	100.00	150.00
4	2024-08-02	P003	12	150.00	150.00
5	2024-08-03	P002	7	200.00	200.00

Why This Output is Incorrect:

- In SQL, if you do not explicitly specify a RANGE frame, the default frame is usually **RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW**. This means that the window function considers all rows from the start of the partition up to the current row by default.
- **Issue with LAST_VALUE() Function:** Because the default frame does not extend beyond the current row, LAST_VALUE() will return the value of the **CURRENT ROW** if the frame is not specified correctly. To get the actual last value of the window frame, you need to explicitly define the frame to include all rows from the current row to the end of the partition.

Correct Usage of LAST_VALUE() with Explicit Frame Specification

To retrieve the last value of the partition or window frame, you should use ROWS to define the frame explicitly. For example, to get the last value from the current row to the end of the partition, you can use:

Corrected Query:

```sql
SELECT
    order_id,
    order_date,
    product_code,
    quantity,
    sale_price,
    LAST_VALUE(sale_price) OVER (
        ORDER BY order_date
        ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
    ) AS last_sale_price
FROM sales;
```

Output Table with Correct Frame Specification

order_id	order_date	product_code	quantity	sale_price	last_sale_price
1	2024-08-01	P001	10	100	200
2	2024-08-01	P002	5	200	200
3	2024-08-02	P001	8	100	200
4	2024-08-02	P003	12	150	200
5	2024-08-03	P002	7	200	200

Explanation:

By specifying ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING, you ensure that the LAST_VALUE() function considers all rows from the current row to the end of the partition. This way, the last_sale_price will accurately reflect the last sale_price value within the defined window frame.

SQL Frame Clause

The SQL frame clause is used with window functions to define the subset of rows within the partition that the function operates on. It controls the "window" or range of rows that are considered for the calculation of each row's result.

Frame Clause Syntax

```sql
window_function(column_name) OVER (
    [PARTITION BY partition_expression]
    [ORDER BY sort_expression]
    [ROWS|RANGE frame_specification]
)
```

- **PARTITION BY:** Divides the result set into partitions to which the window function is applied independently.
- **ORDER BY:** Determines the order of rows within each partition.
- **ROWS and RANGE:** Define the window frame for the function.

Default Frame Clause

If no frame specification is provided, the default frame clause depends on the function and the SQL database system, but generally, it is:

- **For Aggregate Functions:** ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW is often the default, meaning the window includes all rows from the start of the partition up to the current row.
- **For Non-Aggregate Functions:** Some window functions may default to a frame that encompasses all rows in the partition.

Frame Specifications

1. ROWS Clause

The ROWS clause specifies a range of rows relative to the current row, allowing more control over the frame.

Syntax:

```sql
ROWS BETWEEN start_point AND end_point
```

- **ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING:** Includes all rows from the current row to the end of the partition.

- **ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW:** Includes all rows from the start of the partition up to the current row.

- **ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING:** Includes one row before and one row after the current row.

Example Query with ROWS:

Compute the average sale_price for the current row and the two preceding rows.

```sql
SELECT
    order_id,
    order_date,
    sale_price,
    AVG(sale_price) OVER (
        ORDER BY order_date
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
    ) AS avg_sale_price
FROM sales;
```

Output Table:

order_id	order_date	sale_price	avg_sale_price
1	2024-08-01	100.00	100.00
2	2024-08-01	200.00	150.00
3	2024-08-02	100.00	133.33
4	2024-08-02	150.00	116.67
5	2024-08-03	200.00	150.00

Explanation:

- For the first row, the window includes just the current row (since there are not enough preceding rows).
- For the second row, the window includes the current row and the 1 preceding row.
- For the third row, the window includes the current row and the 2 preceding rows.
- The same logic follows for subsequent rows

2. RANGE Clause

The RANGE clause specifies a range of values relative to the current row's value, rather than row numbers. This is used with ordered columns and is generally applied to numeric or date columns.

Syntax:

```sql
RANGE BETWEEN start_point AND end_point
```

- **RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW:** Includes all rows from the start of the partition up to the current row, based on the ordering column's value.
-
- **RANGE BETWEEN CURRENT ROW AND INTERVAL '1 DAY' FOLLOWING:** Includes rows within the range of the current row's value up to one day ahead.

Example Query with RANGE:

Compute the total sale_price for the current row and rows with order dates within one day prior.

```sql
SELECT
    order_id,
    order_date,
    sale_price,
    SUM(sale_price) OVER (
        ORDER BY order_date
        RANGE BETWEEN INTERVAL '1 DAY' PRECEDING AND CURRENT ROW
    ) AS total_sale_price
FROM sales;
```

Output Table:

order_id	order_date	sale_price	total_sale_price
1	2024-08-01	100.00	100.00
2	2024-08-01	200.00	300.00
3	2024-08-02	100.00	300.00
4	2024-08-02	150.00	250.00
5	2024-08-03	200.00	200.00

Explanation:

- For the first row, there are no rows within the specified range (1 day preceding) so the total is the same as the sale_price.
- For the second row, the window includes both the current row and the previous row.
- For the third row, the window includes the current row and rows within the specified range of 1 day before.
- The same logic follows for subsequent rows.

When to Use ROWS vs RANGE

- **ROWS:** Use ROWS when you need a fixed number of rows relative to the current row. It's ideal for operations that are sensitive to the actual row count, such as moving averages or sum calculations.
- **RANGE:** Use RANGE when you need to consider a range of values rather than a fixed number of rows. It's useful for operations that are sensitive to the value ranges, such as calculating sums or averages over a time period or numeric range.

Common Use Cases for SQL Window Functions

1. **Running Totals:** Calculate cumulative sums over a partitioned dataset, such as total sales over time.
2. **Moving Averages:** Compute averages over a sliding window of rows to smooth out fluctuations in data.
3. **Ranking:** Assign ranks to rows within a partition, such as ranking employees based on performance.
4. **Row Numbering:** Assign unique sequential numbers to rows within a partition or the entire result set.
5. **Lag and Lead Analysis**: Compare a row's value to a previous or subsequent row's value, useful for trend analysis.
6. **Percentiles:** Determine the percentile rank of a value within a set, often used in statistical analysis.