

SUMMARY: WINDOWS FUNCTIONS IN SQL-II

SESSION OVERVIEW:

By the end of this session the students will be able to:

- Understand the functionalities of analytical window functions.
- Understand the advanced concepts of Windows functions.
- Perform advanced functions while writing queries generating important insights.

KEY TOPICS AND EXAMPLES:

In the previous session, we understood the basic concepts of Windows functions, which included understanding how Windows function works, why it is important, and when we should use such functions. Additionally, we have also learned some of the aggregate Windows functions and ranking functions and their uses.

Understanding the functionalities of analytical window functions:

- **Concepts on LEAD():**

The LEAD() window function in SQL is used to access data from subsequent rows in the same result set without the need for a self-join. This function allows you to look ahead in your data and retrieve values from rows that are a specified number of rows ahead of the current row. This function is beneficial if we want to compare and find the difference between the current row and the next rows.

Imagine a race where each runner is represented by a row in a table. The Lead function would show the distance (or time) to the next runner ahead. For example, if runner A is in first place and runner B is in second place, the Lead function for runner A would show the distance (or time) between them.

Syntax:

```
LEAD (expression, offset, default_value) OVER (  
    PARTITION BY (expression)  
    ORDER BY (expression)  
)
```

NOTE:

- *Expression: can be any column of the table or a built-in function.*
- *Offset: tells the number of rows to be succeeded from the current row. This value must be a positive integer. If we specify 0, then the next row is considered the current row. If no offset is specified, then by default, it is taken as 1.*
- *Default_value: contains the value to be returned if there is no subsequent row. By default, it returns the null value.*
- *OVER: is in charge of categorizing rows into groups. If it is not present, the function operates on all rows.*

- **PARTITION BY:** divides the result set's rows into partitions to which a function is applied. If this clause is not specified, all rows in the result set are treated as a single row.
- **ORDER BY** tells the sequence of rows in the partitions before applying the function.

Let's create a separate table that will help us to understand these concepts thoroughly.

```
CREATE TABLE SALES (  
    sales_id int primary key,  
    customer_id int,  
    sales_date date,  
    sales_amount decimal(16, 2)  
);
```

After creating this table let's insert some of the data in it.

```
INSERT INTO SALES VALUES  
    (1, 1, '20200201', 500),  
    (2, 1, '20200301', 7200),  
    (3, 1, '20200401', 3440),  
    (4, 2, '20200315', 29990),  
    (5, 2, '20200921', 6700),  
    (6, 3, '20201026', 4500),  
    (7, 3, '20200611', 30000),  
    (8, 4, '20201229', 8560);
```

Now suppose we want the details of the next sales of each customer, so for that purpose, we will use the LEAD function as follows.

Example 1:

```
SELECT customer_id, sales_date, sales_amount, LEAD (sales_amount) OVER (  
    PARTITION BY customer_id  
    ORDER BY sales_date ) next_sale  
FROM SALES;
```

Let's understand the above query.

The above query will first divide the SALES table into partitions according to the customer_id column. Since we have four distinct values of customer_id, our table will be divided into four partitions (one for each customer_id).

Then each partition will be sorted in the increasing order of their sales_date column.

Finally, the LEAD function will be applied to the sales_amount column for each partition.

Output:

Result Grid Filter Rows: Export:				
	customer_id	sales_date	sales_amount	next_sale
▶	1	2020-02-01	500.00	7200.00
	1	2020-03-01	7200.00	3440.00
	1	2020-04-01	3440.00	NULL
	2	2020-03-15	29990.00	6700.00
	2	2020-09-21	6700.00	NULL
	3	2020-06-11	30000.00	4500.00
	3	2020-10-26	4500.00	NULL
	4	2020-12-29	8560.00	NULL

Now, you must be wondering why there are some NULL values in the next_sale column. So, to understand this, consider the customer_id 1. There are three sales for customer_id 1 with the sales_amount 500.00, 7200.00 and 3440.00. The next sale for the first sale amount, i.e. 500.00, will be 7200.00, and for the second sale amount, 7200.00 will be 3440.00. Since we have no fourth sale amount, so, for the third sale amount, the next sale will be NULL.

Example 2:

```
SELECT customer_id, sales_date, sales_amount, LEAD (sales_amount,2) OVER
(
PARTITION BY customer_id
ORDER BY sales_date ) next_to_next_sale
FROM SALES;
```

Output:

Result Grid Filter Rows: Export:				
	customer_id	sales_date	sales_amount	next_to_next_sale
▶	1	2020-02-01	500.00	3440.00
	1	2020-03-01	7200.00	NULL
	1	2020-04-01	3440.00	NULL
	2	2020-03-15	29990.00	NULL
	2	2020-09-21	6700.00	NULL
	3	2020-06-11	30000.00	NULL
	3	2020-10-26	4500.00	NULL
	4	2020-12-29	8560.00	NULL

Since we have the next-to-next sale record of only one customer_id,i.e. 1, the rest values will be NULL.



Now we will look into some of the examples on the dataset that we have been using from the beginning of this module.

Example 4:

```
SELECT t.SalesTerritoryKey, t.Region, t.Country, t.Continent,
       LEAD(t.Region) OVER (ORDER BY t.SalesTerritoryKey) AS
NextRegion
FROM territories t;
```


Output:

Result Grid





Filter Rows:

Export:



Wrap Cell Content:



	SalesTerritoryKey	Region	Country	Continent	NextRegion
▶	1	Northwest	United States	North America	Northeast
	2	Northeast	United States	North America	Central
	3	Central	United States	North America	Southwest
	4	Southwest	United States	North America	Southeast
	5	Southeast	United States	North America	Canada
	6	Canada	Canada	North America	France
	7	France	France	Europe	Germany
	8	Germany	Germany	Europe	Australia
	9	Australia	Australia	Pacific	United Kingdom
	10	United Kingdom	United Kingdom	Europe	NULL

Example 2:

```

SELECT
    DATE_FORMAT(s.orderdate, '%Y-%m') AS year_and_month,
    Round(SUM(s.orderquantity * p.productprice), 2) AS
total_sales_amount,
    LEAD(round(SUM(s.orderquantity * p.productprice), 2)) OVER (ORDER BY
DATE_FORMAT(s.orderdate, '%Y-%m')) AS next_month_total_sales_amount
FROM
    sales_2016 s
    JOIN products p ON s.productkey = p.productkey
GROUP BY
    DATE_FORMAT(s.orderdate, '%Y-%m')
ORDER BY
    year_and_month;

```

(Comment for Instructors: The instructor must explain the above example using default_value. The intent is to show the students what happens when different parts of the parameter are used.)

Output:

Result Grid

Filter Rows:

Export:

Wrap Cell Content:

	year_and_month	total_sales_amount	next_month_total_sales_amount
▶	2016-01	432425.74	474162.79
	2016-02	474162.79	471961.88
	2016-03	471961.88	494957.42
	2016-04	494957.42	545534.74
	2016-05	545534.74	533824.98
	2016-06	533824.98	815356.47
	2016-07	815356.47	804193.39
	2016-08	804193.39	952743.49
	2016-09	952743.49	1029821.05
	2016-10	1029821.05	1133913.05
	2016-11	1133913.05	1635308.8
	2016-12	1635308.8	NULL

Result 8

- **Concepts on LAG():**

The LAG() window function in SQL is used to access data from previous rows in the result set without using a self-join. It allows you to retrieve data from a specific row that precedes the current row within the same result set. This function is useful for comparing values between consecutive rows or calculating the difference between current and previous values.

Similarly, the Lag function would show the distance (or time) to the runner behind. Using the same race analogy, if runner B is in second place and runner A is in first place, the Lag function for runner B would show the distance (or time) between them.

Syntax:

```
LAG (column_name, offset, default_value) OVER (  
    PARTITION BY partition_column  
    ORDER BY order_column  
)
```

NOTE:

- *column_name: The column from which to retrieve the value.*
- *offset: The number of rows before the current row from which to retrieve the value. If omitted, the default is 1.*
- *default_value: The value to return if the LAG() function tries to go before the first row of the partition. If omitted, the default is NULL.*
- *PARTITION BY: Divides the result set into partitions to which the LAG() function is applied. If omitted, the function considers the entire result set as a single partition.*
- *ORDER BY: Specifies the order of the rows within each partition.*

Example 1:

Let's understand the concepts of the LAG() function. We will consider the same SALES table that we created above while understanding LEAD().

```
SELECT customer_id, sales_date, sales_amount, LAG (sales_amount) OVER (  
    PARTITION BY customer_id  
    ORDER BY sales_date) last_sale  
FROM SALES;
```

Let's understand the above query.

The above query will first divide the SALES table into partitions according to the customer_id column. Since we have four distinct values of customer_id, our table will be divided into four partitions (one for each customer_id).

Then each partition will be sorted in the increasing order of their sales_date column.

Finally, the LAG function will be applied to the sales_amount column for each division.

One point to note here is we haven't specified the offset parameter of the LAG function (since we want the immediately preceding value). So, by default, it will be taken as 1.

Output:

Result Grid Filter Rows: Export:				
	customer_id	sales_date	sales_amount	last_sale
▶	1	2020-02-01	500.00	NULL
	1	2020-03-01	7200.00	500.00
	1	2020-04-01	3440.00	7200.00
	2	2020-03-15	29990.00	NULL
	2	2020-09-21	6700.00	29990.00
	3	2020-06-11	30000.00	NULL
	3	2020-10-26	4500.00	30000.00
	4	2020-12-29	8560.00	NULL

We have displayed our result in a new column named last_sale.

Now, you must be wondering why there are some NULL values in the last_sale column. The same concept of the LEAD function applies here. To understand this, consider the customer_id 1. There are three sales for customer_id 1 with the sales_amounts of 500.00, 7200.00, and 3440.00. The last sale for the third sale amount, i.e. 3440.00, will be 7200.00, and for the second sale amount, 7200.00 will be 500.00. Since we have no zeroth sale amount (not possible) or default value, for the first sale amount, the last sale will be NULL.

Similar is the case for the rest of the customer_id.

Now that the fundamentals have been clear, we can move towards understanding the LAG() function concept using examples on the dataset that we have been using throughout the module.

In the below examples we have taken the monthly trend analysis for the year 2016, you can use the sales_2015 and sales_2017 tables to figure out the monthly trends of those years.

Example 2: Comparing Monthly Sales Amount with the Previous Month

```
SELECT
    DATE_FORMAT(s.orderdate, '%Y-%m') AS month,
    Round(SUM(s.orderquantity * p.productprice), 2) AS
total_sales_amount,
    LAG(round(SUM(s.orderquantity * p.productprice),2)) OVER (ORDER BY
DATE_FORMAT(s.orderdate, '%Y-%m')) AS previous_month_total_sales_amount
FROM
    sales_2016 s
    JOIN products p ON s.productkey = p.productkey
GROUP BY
    DATE_FORMAT(s.orderdate, '%Y-%m')
ORDER BY
    month;
```

Output:

Result Grid			
	Filter Rows:		Export: Wrap
month	total_sales_amount	previous_month_total_sales_amount	
2016-01	432425.74	NULL	
2016-02	474162.79	432425.74	
2016-03	471961.88	474162.79	
2016-04	494957.42	471961.88	
2016-05	545534.74	494957.42	
2016-06	533824.98	545534.74	
2016-07	815356.47	533824.98	
2016-08	804193.39	815356.47	
2016-09	952743.49	804193.39	
2016-10	1029821.05	952743.49	
2016-11	1133913.05	1029821.05	
2016-12	1635308.8	1133913.05	

Example 2: Calculating the Sales Amount Change from the Previous Month

```

SELECT
    DATE_FORMAT(s.orderdate, '%Y-%m') AS month,
    SUM(s.orderquantity * p.productprice) AS total_sales_amount,
    LAG(SUM(s.orderquantity * p.productprice)) OVER (ORDER BY
DATE_FORMAT(s.orderdate, '%Y-%m')) AS previous_month_total_sales_amount,
    SUM(s.orderquantity * p.productprice) - LAG(SUM(s.orderquantity *
p.productprice)) OVER (ORDER BY DATE_FORMAT(s.orderdate, '%Y-%m')) AS
sales_amount_change
FROM
    sales_2016 s
    JOIN products p ON s.productkey = p.productkey
GROUP BY
    DATE_FORMAT(s.orderdate, '%Y-%m')
ORDER BY
    month;

```


Output:

Result Grid				
	Filter Rows:		Export: Wrap Cell Content: I A	
month	total_sales_amount	previous_month_total_sales_amount	sales_amount_change	
2016-01	432425.74	NULL	NULL	
2016-02	474162.79	432425.74	41737.05	
2016-03	471961.88	474162.79	-2200.91	
2016-04	494957.42	471961.88	22995.54	
2016-05	545534.74	494957.42	50577.33	
2016-06	533824.98	545534.74	-11709.76	
2016-07	815356.47	533824.98	281531.49	
2016-08	804193.39	815356.47	-11163.08	
2016-09	952743.49	804193.39	148550.11	
2016-10	1029821.05	952743.49	77077.56	
2016-11	1133913.05	1029821.05	104092	
2016-12	1635308.8	1133913.05	501395.76	

Example 3:

```
SELECT
    DATE_FORMAT(s.orderdate, '%Y-%m') AS month,
    round(SUM(s.orderquantity * p.productprice), 2) AS
total_sales_amount,
    LAG(round(SUM(s.orderquantity * p.productprice), 2)) OVER (ORDER BY
DATE_FORMAT(s.orderdate, '%Y-%m')) AS previous_month_total_sales_amount,
    CASE
        WHEN SUM(s.orderquantity * p.productprice) <
LAG(SUM(s.orderquantity * p.productprice)) OVER (ORDER BY
DATE_FORMAT(s.orderdate, '%Y-%m')) THEN 'Decreased'
        WHEN SUM(s.orderquantity * p.productprice) >
LAG(SUM(s.orderquantity * p.productprice)) OVER (ORDER BY
DATE_FORMAT(s.orderdate, '%Y-%m')) THEN 'Increased'
        ELSE 'No Change'
    END AS sales_trend
FROM
    sales_2016 s
    JOIN products p ON s.productkey = p.productkey
GROUP BY
    DATE_FORMAT(s.orderdate, '%Y-%m')
ORDER BY
    month;
```

Output:

Result Grid				
Filter Rows:		Export:  Wrap Cell Content:		
	month	total_sales_amount	previous_month_total_sales_amount	sales_trend
▶	2016-01	432425.74	NULL	No Change
	2016-02	474162.79	432425.74	Increased
	2016-03	471961.88	474162.79	Decreased
	2016-04	494957.42	471961.88	Increased
	2016-05	545534.74	494957.42	Increased
	2016-06	533824.98	545534.74	Decreased
	2016-07	815356.47	533824.98	Increased
	2016-08	804193.39	815356.47	Decreased
	2016-09	952743.49	804193.39	Increased
	2016-10	1029821.05	952743.49	Increased
	2016-11	1133913.05	1029821.05	Increased
	2016-12	1635308.8	1133913.05	Increased

Here is one query that will help analyze the company's yearly sales trend.

```
SELECT year,
    SUM(sales_amount) AS total_sales_amount,
    LAG(SUM(sales_amount)) OVER (ORDER BY year) AS
```



```
previous_year_total_sales_amount,
CASE
    WHEN SUM(sales_amount) < LAG(SUM(sales_amount)) OVER (ORDER
BY year) THEN 'Decreased'
    WHEN SUM(sales_amount) > LAG(SUM(sales_amount)) OVER (ORDER
BY year) THEN 'Increased'
    ELSE 'No Change'
END AS sales_trend
FROM combined_sales
GROUP BY year
ORDER BY year;
```

Output:

Result Grid Filter Rows: Export: Wrap Cell Content:				
	year	total_sales_amount	previous_year_total_sales_amount	sales_trend
▶	2015	6404933.580299864	NULL	No Change
	2016	9324203.791703518	6404933.580299864	Increased
	2017	9185449.44730302	9324203.791703518	Decreased

- **Concepts on NTILE():**

The NTILE() window function in SQL is used to divide the result set into a specified number of groups, or "tiles," and assigns a bucket number to each row based on the specified number of tiles. This function is commonly used to create equal-sized buckets or percentile groups from a dataset.

The Ntile function can be compared to dividing the race into equal segments (tiles) based on some criteria (e.g., time intervals). Each runner would be placed in a tile based on their performance relative to others. For example, if there are 4 tiles (1st quartile, 2nd quartile, 3rd quartile, 4th quartile), runners in the 1st quartile are performing better than those in the 4th quartile.

Uses of NTILE functions:

- **Data Distribution:** NTILE can help analyze how data is distributed across different segments.
- **Percentiles and Quartiles:** By specifying 100 groups for percentiles or 4 groups for quartiles, you can determine which percentile or quartile each row falls into.

Syntax:

```
NTILE(number_of_buckets) OVER (ORDER BY column_name)
```

NOTE:

- **number_of_buckets:** The number of groups or "tiles" into which to divide the result set.
- **ORDER BY:** Specifies the column or expression by which to order the rows before dividing them into buckets.

Example 1:

```
SELECT ProductKey, ProductName, ProductPrice,
       NTILE(4) OVER (ORDER BY ProductPrice) AS price_quartile
FROM Products;
```

Output:

ProductKey	ProductName	ProductPrice	price_quartile
480	Patch Kit/8 Patches	2.29	1
529	Road Tire Tube	3.99	1
477	Water Bottle - 30 oz.	4.99	1
528	Mountain Tire Tube	4.99	1
530	Touring Tire Tube	4.99	1
484	Bike Wash - Dissolver	7.95	1
223	AWC Logo Cap	8.6442	1
479	Road Bottle Cage	8.99	1
481	Racing Socks, M	8.99	1
482	Racing Socks, L	8.99	1
218	Mountain Bike Socks, M	9.5	1

Example 2:

```
WITH monthly_sales AS (
    SELECT DATE_FORMAT(orderdate, '%Y-%m') AS month,
           Round(SUM(orderquantity * p.productprice), 2) AS
total_sales_amount
    FROM sales_2017 s
    JOIN products p ON s.productkey = p.productkey
    GROUP BY DATE_FORMAT(orderdate, '%Y-%m')
)
SELECT
    month,
    total_sales_amount,
    NTILE(3) OVER (ORDER BY total_sales_amount) AS sales_tercile
FROM
    monthly_sales;
```

Output:

Result Grid			
		Filter Rows:	
Export:			
	month	total_sales_amount	sales_tercile
▶	2017-01	1274378.67	1
	2017-02	1339241.29	1
	2017-03	1448596.12	2
	2017-04	1527813.72	2
	2017-05	1768432.51	3
	2017-06	1826987.14	3

Example 3: Analyze Sales Performance by Price Range

```
WITH product_sales AS (
    SELECT p.ProductKey,
           p.ProductName,
           p.ProductPrice,
           SUM(s.OrderQuantity) AS total_quantity,
           NTILE(5) OVER (ORDER BY p.ProductPrice) AS price_range
    FROM Products p
    JOIN Sales_2015 s ON p.ProductKey = s.ProductKey
    GROUP BY p.ProductKey, p.ProductName, p.ProductPrice
)
SELECT price_range,
       COUNT(*) AS num_products,
       SUM(total_quantity) AS total_quantity_sold,
       MIN(ProductPrice) AS min_price,
       MAX(ProductPrice) AS max_price
FROM product_sales
GROUP BY price_range;
```

Output:

Result Grid					
		Filter Rows:		Export:	
				Wrap Cell Content:	
	price_range	num_products	total_quantity_sold	min_price	max_price
▶	1	9	292	699.0982	699.0982
	2	9	319	699.0982	2049.0982
	3	9	664	2049.0982	2181.5625
	4	9	468	2181.5625	3399.99
	5	8	887	3399.99	3578.27

We have learned the concepts of quartiles in the previous module. Now we will look into the quartile concepts in MySQL and how with the help of the NTILE function we can remove the outliers in MySQL. So let's begin.

```
WITH product_stats AS (
    SELECT ProductPrice,
```

```

        NTILE(4) OVER (ORDER BY ProductPrice) AS price_quartile
    FROM Products
),
quartiles AS (
    SELECT MAX(CASE WHEN price_quartile = 1 THEN ProductPrice END) AS
Q1,
           MAX(CASE WHEN price_quartile = 3 THEN ProductPrice END) AS
Q3
    FROM product_stats
),
iqr_bounds AS (
    SELECT Q1, Q3,
           Q3 - Q1 AS IQR,
           Q1 - 1.5 * (Q3 - Q1) AS lower_bound,
           Q3 + 1.5 * (Q3 - Q1) AS upper_bound
    FROM quartiles
)
SELECT p.ProductKey, p.ProductName, p.ProductPrice
FROM Products p
JOIN iqr_bounds ib
ON p.ProductPrice >= ib.lower_bound AND p.ProductPrice <=
ib.upper_bound;

```

Output:

Result Grid	Filter Rows:	Export:
ProductKey	ProductName	ProductPrice
214	Sport-100 Helmet, Red	34.99
215	Sport-100 Helmet, Black	33.6442
218	Mountain Bike Socks, M	9.5
219	Mountain Bike Socks, L	9.5
220	Sport-100 Helmet, Blue	33.6442
223	AWC Logo Cap	8.6442
226	Long-Sleeve Logo Jersey, S	48.0673
229	Long-Sleeve Logo Jersey, M	48.0673
232	Long-Sleeve Logo Jersey, L	48.0673
235	Long-Sleeve Logo Jersey, XL	48.0673
238	HL Road Frame - Red, 62	1263.4598
241	HL Road Frame - Red, 44	1263.4598
244	HL Road Frame - Red, 48	1263.4598

Note: The output format might not contain all the returned records. The returned table above doesn't contain the outliers.

Concepts on FIRST_VALUE():

The FIRST_VALUE() function in MySQL is a window function that returns the first value in an ordered set of values within a specified partition. This function is useful for retrieving the initial value from a data subset, which can be particularly beneficial in trend analysis, reporting, and other

analytical tasks where the first occurrence of a value is significant.

Syntax:

```
FIRST_VALUE(expression) OVER (
  [PARTITION BY partition_expression]
  ORDER BY sort_expression
  [ROWS or RANGE window_frame_definition]
)
```



NOTE:

- **expression:** The column or expression from which to return the first value.
- **PARTITION BY partition_expression: (Optional)** Divides the result set into partitions to which the function is applied. If not specified, the function treats all rows as a single partition.
- **ORDER BY sort_expression:** Specifies the order of rows within each partition. *FIRST_VALUE()* returns the value of the first row in this order.
- **ROWS or RANGE window_frame_definition: (Optional)** Defines the frame of rows within which the function operates. This typically includes all rows from the start of the partition up to the current row.

Examples:

```
SELECT
  s.productkey,
  c.firstname,
  c.lastname,
  FIRST_VALUE(c.firstname) OVER (PARTITION BY s.productkey ORDER BY
s.OrderDate) AS first_customer_firstname,
  FIRST_VALUE(c.lastname) OVER (PARTITION BY s.productkey ORDER BY
s.OrderDate) AS first_customer_lastname
FROM
  sales_2015 s
JOIN
  Customers c ON s.customerkey = c.customerkey
ORDER BY
  s.productkey;
```

Output:

Result Grid					
Filter Rows: <input type="text"/>					
Export:  Wrap Cell Content: 					
	productkey	firstname	lastname	first_customer_firstname	first_customer_lastname
▶	310	DEVIN	HENDERSON	DEVIN	HENDERSON
	310	GABRIEL	GREEN	DEVIN	HENDERSON
	310	JARED	COOK	DEVIN	HENDERSON
	310	NINA	YUAN	DEVIN	HENDERSON
	310	BRADLEY	NARA	DEVIN	HENDERSON
	310	BRANDY	SANCHEZ	DEVIN	HENDERSON
	310	ROBERTO	SANZ	DEVIN	HENDERSON
	310	ALFONSO	SANDRER	DEVIN	HENDERSON

Result 20 x

Concepts on LAST_VALUE():

The LAST_VALUE() function in MySQL is a window function that returns the last value in an ordered set of values within a specified partition. This function is particularly useful for retrieving the final value from a data subset, which can be beneficial in various analytical and reporting tasks where the last occurrence of a value is significant, such as determining the most recent transaction or the latest status update.

Syntax:

```
LAST_VALUE(expression) OVER (  
    [PARTITION BY partition_expression]  
    ORDER BY sort_expression  
    [ROWS or RANGE window_frame_definition]  
)
```

NOTE:

- **expression:** The column or expression from which to return the last value.
- **PARTITION BY partition_expression:** (Optional) Divides the result set into partitions to which the function is applied. If not specified, the function treats all rows as a single partition.
- **ORDER BY sort_expression:** Specifies the order of rows within each partition. LAST_VALUE() returns the value of the last row in this order.
- **ROWS or RANGE window_frame_definition:** (Optional) Defines the frame of rows within which the function operates. Typically, this includes all rows from the start of the partition up to the current row, but it can be customized as needed.

Example:

```
SELECT  
    s.productkey,  
    t.region,  
    LAST_VALUE(t.region) OVER (PARTITION BY s.productkey ORDER BY  
s.orderDate RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)  
AS last_region_sold  
FROM  
    sales_2017 s  
JOIN  
    territories t ON s.territorykey = t.salesterritorykey  
ORDER BY  
    s.productkey;
```

Output:

Result Grid			
	productkey	region	last_region_sold
▶	214	Northwest	Northwest
	214	United Kingdom	Northwest
	214	Southwest	Northwest
	214	United Kingdom	Northwest
	214	Australia	Northwest
	214	Southwest	Northwest
	214	Southwest	Northwest
	214	Australia	Northwest

Concepts on Nth_VALUE():

The NTH_VALUE() function in MySQL is a window function that returns the value of an expression from the N-th row of the window frame. This function is useful when you want to retrieve the N-th value from a set of ordered rows.

Syntax:

```
NTH_VALUE(expression, N) OVER (
    PARTITION BY column1, column2, ...
    ORDER BY column1 [ASC|DESC], column2 [ASC|DESC], ...
    [ROWS|RANGE BETWEEN ...]
)
```

NOTE:

- **expression:** The column or expression whose N-th value you want to retrieve.
- **N:** The position of the row within the window frame from which to retrieve the value. It must be a positive integer.
- **PARTITION BY:** Divides the result set into partitions to which the function is applied. If omitted, the function treats the entire result set as a single partition.
- **ORDER BY:** Defines the order of the rows within each partition. This is required to determine the N-th row.
- **ROWS|RANGE:** Defines the window frame. This clause is optional but can be used to limit the rows considered by the window function.

Example 1:

```
SELECT
    p.productkey,
    NTH_VALUE(productprice, 3) OVER (PARTITION BY p.productkey ORDER BY
p.productprice) AS third_sale_price
FROM
    products p
JOIN
    sales_2017 s7 ON p.ProductKey = s7.productkey
ORDER BY
```



```
p.productkey;
```

Output:

Result Grid		Filter Rows:
	productkey	third_sale_price
▶	214	34.99
	214	34.99
	214	34.99
	214	34.99
	214	34.99
	214	34.99
	214	34.99
	214	34.99
	214	34.99
	214	34.99

Result 18 x

IMPORTANT:

Using ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING ensures that:

- **LAST_VALUE():** Considers all rows in the partition, giving you the true last value.
- **NTH_VALUE():** Considers all rows in the partition to correctly identify the nth value.
- **Without this window frame, LAST_VALUE() and NTH_VALUE() might only consider rows up to the current row, potentially leading to incorrect results if you are not aware of the default behavior.**

Why Use It:

- Ensures the function considers all rows in the partition, not just those up to the current row.
- Provides accurate results for functions like LAST_VALUE() and NTH_VALUE() where considering the entire partition is crucial.
- Useful in scenarios where you need to compare each row to a specific value in the entire partition.

QUESTIONS:

1. Identify the customers who have ordered products from the same subcategory and calculate the average number of days between orders.

```
WITH subcategory_orders AS (
    SELECT s.customerkey, p.ProductSubcategoryKey, s.orderdate,
           LAG(s.orderdate) OVER (PARTITION BY s.customerkey,
                                   p.ProductSubcategoryKey ORDER BY s.orderdate) AS prev_orderdate
    FROM sales_2017 s
    JOIN products p ON s.productkey = p.productkey
```

```

)
SELECT customerkey, ProductSubcategoryKey,
       round(AVG(DATEDIFF(orderdate, prev_orderdate)), 0) AS
avg_days_between_orders
FROM subcategory_orders
WHERE prev_orderdate IS NOT NULL
GROUP BY customerkey, ProductSubcategoryKey
HAVING COUNT(*) > 1;

```

Output:

Result Grid	Filter Rows:	Export:	Wrap Cell Content:
customerkey	ProductSubcategoryKey	avg_days_between_orders	
11019	37	13	
11027	37	0	
11032	37	0	
11039	37	26	
11078	37	22	
11086	37	86	
11091	31	43	
11091	37	8	
11125	37	0	
11126	37	0	
11131	37	9	

Result 43 x

- Find the product that has been out of stock the longest and calculate the number of days it has been unavailable.

```

WITH stock_changes AS (
    SELECT productkey, stockdate,
           LEAD(stockdate) OVER (PARTITION BY productkey ORDER BY
stockdate) AS next_stockdate
    FROM sales_2017
)
SELECT productkey,
       MAX(DATEDIFF(next_stockdate, stockdate)) AS days_out_of_stock
FROM stock_changes
WHERE next_stockdate IS NOT NULL
GROUP BY productkey;

```

Output:

Result Grid			Filter Rows:
	productkey	days_out_of_stock	
▶	214	3	
	215	6	
	220	6	
	223	4	
	226	10	
	229	10	
	232	9	
	235	7	
	352	8	
	354	12	
	356	15	

Result 46 x

- Write a query to calculate the yearly sales trend, showing the total sales amount and the percentage change from the previous year.

```

WITH yearly_sales AS (
    SELECT YEAR(orderdate) AS year,
           round(SUM(orderquantity * ProductPrice), 2) AS total_sales
    FROM
        (SELECT * FROM sales_2015 UNION ALL SELECT * FROM sales_2016
        UNION ALL SELECT * FROM sales_2017) s
    JOIN products p ON s.productkey = p.productkey
    GROUP BY YEAR(orderdate)
),
sales_trend AS (
    SELECT year, total_sales,
           LAG(total_sales) OVER (ORDER BY year) AS prev_year_sales
    FROM yearly_sales
)
SELECT year, total_sales,
       (total_sales - prev_year_sales) / prev_year_sales * 100.00 AS
sales_change_percentage
FROM sales_trend
WHERE prev_year_sales IS NOT NULL;

```

Output:

Result Grid				Filter Rows:	Export:
	year	total_sales	sales_change_percentage		
▶	2016	9324203.79	45.57846187688334		
	2017	9185449.45	-1.4881092597827044		

Example:

```
SELECT
sale_id,
sale_date,
product_id,
amount,
FIRST_VALUE(amount) OVER (PARTITION BY product_id ORDER BY sale_date) AS
first_sale_amount,
LAST_VALUE(amount) OVER (PARTITION BY product_id ORDER BY sale_date ROWS
BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS
last_sale_amount,
NTH_VALUE(amount, 3) OVER (PARTITION BY product_id ORDER BY sale_date
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS
third_sale_amount
FROM sales;
```

Note:

- **ROWS:** Specifies that the frame should be based on the actual physical rows.
- **BETWEEN:** Indicates the range for the frame.
- **UNBOUNDED PRECEDING:** Refers to all rows from the start of the partition up to the current row.
- **UNBOUNDED FOLLOWING:** Refers to all rows from the current row to the end of the partition.