PW SKILLS

Lesson Plan

Windows Functions

What is Windows Functions?

- Window functions are used to calculate an aggregate value based on a collection of rows and return many rows for each group.
- A window word designates the set of rows on which the function will work.
- The computation carried out by this function is identical to that of the aggregate functions.
- The results cannot be consolidated into a single row when using Window functions, unlike aggregate functions that work on the whole table.
- In other words, window functions operate on a collection of rows and provide a total value for every row. Each row retains its unique individuality.

Types Of Windows Functions:

1. **Aggregate Window Functions:** SUM(), MAX(), MIN(), AVG(), COUNT()
2. **Ranking Window Functions:** RANK(), DENSE_RANK(), ROW_NUMBER(), and NTILE()
3. **Value Window Functions:** LAG(), LEAD(), FIRST_VALUE(), LAST_VALUE()

BASIC SYNTAX:

```
<window_function> (<expression>) OVER (PARTITION BY <partition_clause>
ORDER BY <order_clause>)
```

<window_function>
This is the window function, which runs over a range of rows specified by the window frame to perform a particular computation or operation. The frequently used window functions are ROW_NUMBER, RANK, DENSE_RANK, SUM, and AVG.

(<expression>
The column or value you wish to apply the window function on represents the expression. This expression will have the window function applied to it inside the designated window frame.

OVER
In addition to introducing the window specification, the OVER keyword signals the impending use of the window function. Parentheses () specify the details of the window definition after it.

OVER
In addition to introducing the window specification, the OVER keyword signals the impending use of the window function. Parentheses () specify the details of the window definition after it.

ORDER BY <order_clause>
The optional ORDER sets the order of rows within each partition BY clause. This is employed to specify the order in which the window function is invoked. The columns listed in the <order_clause> are used to determine the order of the rows. Including the ORDER BY clause in cumulative sums, rankings, etc. processes is frequently necessary.

Aggregate Function:

Sum()

The SUM() function is used to compute the total values in a particular column over a specified window of rows. You can use window functions to execute calculations across a collection of rows connected to the current row inside a given window frame. This is very helpful for calculating aggregates without first grouping the complete collection of results.

Syntax

```
SUM(column) OVER (PARTITION BY partition_expression ORDER BY
order_expression ROWS window_frame)
```

Column: The column for which you want to calculate the sum.

PARTITION BY: Optional. It divides the result set into partitions to calculate the sum separately within each partition.

ORDER BY: Specifies the order of rows within each partition.

ROWS: Defines the window frame relative to the current row.

***window_frame:** Specifies the range of rows in the partition to be included in the calculation. It can be UNBOUNDED PRECEDING, n PRECEDING, CURRENT ROW, n FOLLOWING, or UNBOUNDED FOLLOWING.

Example:
Calculate the cumulative sum of revenue for each product over the specified date range

```
CREATE TABLE Sales (
    Product VARCHAR(50),
    Date DATE,
    Revenue DECIMAL(10, 2)
);

INSERT INTO Sales (Product, Date, Revenue)
VALUES
    ('Product A', '2023-08-01', 100.00),
    ('Product A', '2023-08-02', 150.00),
    ('Product A', '2023-08-03', 200.00),
    ('Product B', '2023-08-01', 50.00),
    ('Product B', '2023-08-02', 75.00),
    ('Product B', '2023-08-03', 100.00);
```

Query:

```
SELECT
    Product,
    Date,
    Revenue,
    SUM(Revenue) OVER (PARTITION BY Product ORDER BY Date) AS
CumulativeRevenue
FROM
    Sales;
```

OUTPUT:

	Product	Date	Revenue	CumulativeRevenue
▶	Product A	2023-08-01	100.00	100.00
	Product A	2023-08-02	150.00	250.00
	Product A	2023-08-03	200.00	450.00
	Product B	2023-08-01	50.00	50.00
	Product B	2023-08-02	75.00	125.00
	Product B	2023-08-03	100.00	225.00

MAX()

To calculate a particular window of rows, window functions are frequently used with the aggregate function MAX().

Syntax

```
MAX(expression) OVER (
    [PARTITION BY partition_expression, ... ]
    [ORDER BY sort_expression [ASC | DESC], ... ]
    [ROWS frame_specification]
)
```

- MAX(expression): This is the aggregate function you are using within the window. It will be applied to the set of values within the specified window.
- OVER: This keyword indicates that you are using a window function.
- PARTITION BY partition_expression, This clause is optional. It defines how the result set should be divided into partitions or groups. The window function is applied separately to each partition.

- ORDER BY sort_expression [ASC | DESC], ...: This clause is also optional. It specifies the order in which rows within each partition should be processed by the window function. You can order rows based on one or more columns in ascending or descending order.

- ROWS frame_specification: This specifies the range of rows for the window. It defines the scope of rows over which the window function operates. It can be used to create a sliding or fixed-size window

Example:
Find, for each student, the maximum score they achieved in any exam.

```
CREATE TABLE student_scores (
    student_id INT,
    exam_date DATE,
    score INT

);

INSERT INTO student_scores (student_id, exam_date, score)
VALUES
    (1, '2023-01-15', 85),
    (1, '2023-02-20', 92),
    (1, '2023-03-10', 78),
    (2, '2023-01-15', 70),
    (2, '2023-02-20', 88),
    (2, '2023-03-10', 95);
```

Query:

```
SELECT
    student_id,
    exam_date,
    score,
    MAX(score) OVER (PARTITION BY student_id ORDER BY score DESC) AS
max_score_for_student
FROM
    student_scores;
```

OUTPUT:

```
| student_id | exam_date  | score | max_score_for_student |
|------------|------------|-------|-----------------------|
| 1          | 2023-01-15 | 85    | 92                    |
| 1          | 2023-02-20 | 92    | 92                    |
| 1          | 2023-03-10 | 78    | 92                    |
| 2          | 2023-01-15 | 70    | 95                    |
| 2          | 2023-02-20 | 88    | 95                    |
| 2          | 2023-03-10 | 95    | 95                    |
```

MIN()

With the help of the partition and ordering criteria, the MIN() method may be used to determine the minimum value of a given column inside a window.

Syntax

```
MIN(expression) OVER (
    [PARTITION BY partition_expression, ... ]
    [ORDER BY sort_expression [ASC | DESC], ... ]
    [ROWS frame_specification]
)
```

- MIN (expression): This is the aggregate function that calculates the minimum value within the window.
- OVER: This keyword indicates that you are using a window function.
- PARTITION BY partition_expression, ...: Optional clause that defines how the result set should be partitioned or grouped.
- ORDER BY sort_expression [ASC | DESC], ...: Optional clause that specifies the order in which rows within each partition should be processed.
- ROWS frame_specification: This defines the range of rows for the window.

Example

```
CREATE TABLE student_scores (
    student_id INT,
    exam_date DATE,
    score INT
);

INSERT INTO student_scores (student_id, exam_date, score)
VALUES
    (1, '2023-01-15', 85),
    (1, '2023-02-20', 92),
    (1, '2023-03-10', 78),
    (2, '2023-01-15', 70),
    (2, '2023-02-20', 88),
    (2, '2023-03-10', 95);
```

Query:

```
SELECT student_id, exam_date, score, MIN(score) OVER (PARTITION BY
student_id) AS min_score_for_student FROM student_scores;
```

OUTPUT:

```
| student_id | exam_date  | score | min_score_for_student |
|------------|------------|-------|-----------------------|
| 1          | 2023-01-15 | 85    | 78                    |
| 1          | 2023-02-20 | 92    | 78                    |
| 1          | 2023-03-10 | 78    | 78                    |
| 2          | 2023-01-15 | 70    | 70                    |
| 2          | 2023-02-20 | 88    | 70                    |
| 2          | 2023-03-10 | 95    | 70                    |
```

AVG()

The average (mean) value of a given column inside a timeframe determined by the partition and ordering criteria is computed using the AVG() function.

Syntax

```
AVG(expression) OVER (
    [PARTITION BY partition_expression, ... ]
    [ORDER BY sort_expression [ASC | DESC], ... ]
    [ROWS frame_specification]
)
```

- AVG (expression): This is the aggregate function that calculates the average value within the window.
- OVER: This keyword indicates that you are using a window function.
- PARTITION BY partition_expression, ...: Optional clause that defines how the result set should be partitioned or grouped.
- ORDER BY sort_expression [ASC | DESC], ...: Optional clause that specifies the order in which rows within each partition should be processed.
- ROWS frame_specification: This defines the range of rows for the window.

Example

```
CREATE TABLE student_scores (
    student_id INT,
    exam_date DATE,
    score INT
);

INSERT INTO student_scores (student_id, exam_date, score)
VALUES
    (1, '2023-01-15', 85),
    (1, '2023-02-20', 92),
    (1, '2023-03-10', 78),
    (2, '2023-01-15', 70),
    (2, '2023-02-20', 88),
    (2, '2023-03-10', 95);
```

Query:

```
SELECT
    student_id,
    exam_date,
    score,
    AVG(score) OVER (PARTITION BY student_id) AS avg_score_for_student
FROM
    student_scores;
```

OUTPUT:

student_id	exam_date	score	avg_score_for_student
1	2023-01-15	85	85
1	2023-02-20	92	85
1	2023-03-10	78	85
2	2023-01-15	70	84.33333333333333
2	2023-02-20	88	84.33333333333333
2	2023-03-10	95	84.33333333333333

COUNT():

The count function will return the total number of rows or records in the table or group. The standard aggregate procedure uses the DISTINCT keyword to avoid counting duplicate data.

Syntax

```
COUNT(expression) OVER (
    [PARTITION BY partition_expression, ... ]
    [ORDER BY sort_expression [ASC | DESC], ... ]
    [ROWS frame_specification]
)
```

- COUNT(expression): This is the aggregate function that calculates the count of rows within the window. Note that the expression can be any column or expression you want to count. Using COUNT(*) counts all rows.
- OVER: This keyword indicates that you are using a window function
- PARTITION BY partition_expression, ...: Optional clause that defines how the result set should be partitioned or grouped.
- ORDER BY sort_expression [ASC | DESC], ...: Optional clause that specifies the order in which rows within each partition should be processed
- ROWS frame_specification: This defines the range of rows for the window.

Example:

Calculates the count of exams for each student

```
CREATE TABLE student_scores (
    student_id INT,
    exam_date DATE,
    score INT
);

INSERT INTO student_scores (student_id, exam_date, score)
VALUES
    (1, '2023-01-15', 85),
    (1, '2023-02-20', 92),
    (1, '2023-03-10', 78),
    (2, '2023-01-15', 70),
    (2, '2023-02-20', 88),
    (2, '2023-03-10', 95);
```

Query:

```
SELECT
    student_id,
    exam_date,
    score,
    COUNT(*) OVER (PARTITION BY student_id) AS exam_count_for_student
FROM
    student_scores;
```

OUTPUT:

student_id	exam_date	score	exam_count_for_student
1	2023-01-15	85	3
1	2023-02-20	92	3
1	2023-03-10	78	3
2	2023-01-15	70	3
2	2023-02-20	88	3
2	2023-03-10	95	3

Ranking Window Functions

RANK()

The RANK() method determines a unique rank for each row inside a result set based on the provided column values. It's frequently used to rank based on particular criteria, allowing you to ascertain the relative position of each row inside a result set.

Syntax

```
RANK() OVER (
    [PARTITION BY partition_expression, ... ]
    ORDER BY sort_expression [ASC | DESC], ...
)
```

- RANK(): This is the window function that calculates the rank for each row within the window.
- OVER: This keyword indicates that you are using a window function.
- PARTITION BY partition_expression, ...: Optional clause that defines how the result set should be partitioned or grouped.
- ORDER BY sort_expression [ASC | DESC], ...: Clause that specifies the order in which rows should be ranked within each partition. You can order rows based on one or more columns in ascending or descending order.

Example:
Determine the ranking of student exam scores for each student based on their scores in descending order

```
CREATE TABLE student_scores (
    student_id INT,
    exam_date DATE,
    score INT
);

INSERT INTO student_scores (student_id, exam_date, score)
VALUES
    (1, '2023-01-15', 85),
    (1, '2023-02-20', 92),
    (1, '2023-03-10', 78),
    (2, '2023-01-15', 70),
    (2, '2023-02-20', 88),
    (2, '2023-03-10', 95);
```

Query:

```
SELECT
    student_id,
    exam_date,
    score,
    RANK() OVER (PARTITION BY student_id ORDER BY score DESC) AS score_rank
FROM
    student_scores;
```

OUTPUT:

	student_id	exam_date	score	score_rank
▶	1	2023-02-20	92	1
	1	2023-01-15	85	2
	1	2023-03-10	78	3
	2	2023-03-10	95	1
	2	2023-02-20	88	2
	2	2023-01-15	70	3

DENSE_RANK()

Like the RANK() method, the DENSE_RANK() function gives each row in a result set an independent rank. With DENSE_RANK(), the subsequent level is not skipped even if many rows have the same values and, consequently, the same position. This implies that the ranks are assigned consecutively, even in ties.

Syntax

```
DENSE_RANK() OVER (
    [PARTITION BY partition_expression, ... ]
    ORDER BY sort_expression [ASC | DESC], ...
)
```

- DENSE_RANK(): This is the window function that calculates the dense rank for each row within the window.
- OVER: This keyword indicates that you are using a window function.
- PARTITION BY partition_expression, ...: Optional clause that defines how the result set should be partitioned or grouped.
- ORDER BY sort_expression [ASC | DESC], ...: Clause that specifies the order in which rows should be ranked within each partition. You can order rows based on one or more columns in ascending or descending order.

Example

```
CREATE TABLE employees (
    employee_id INT,
    department_id INT,
    salary DECIMAL(10, 2)
);

-- Insert sample values
INSERT INTO employees (employee_id, department_id, salary)
VALUES
    (1, 101, 55000.00),
    (2, 101, 60000.00),
    (3, 102, 65000.00),
    (4, 102, 58000.00),
    (5, 101, 62000.00),
    (6, 102, 59000.00);
```

Query:

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

OUTPUT:

employee_id	department_id	salary	dense_rank_within_department
2	101	60000.00	1
5	101	62000.00	2
1	101	55000.00	3
3	102	65000.00	1
6	102	59000.00	2
4	102	58000.00	3

DIFFERENCE B/W RANK AND DENSE RANK:

RANK()

The RANK() function assigns the same rank to all linked rows when there are ties (many rows with the same values), skipping the next level in the process. In other words, the next row will increase its rank by 2 (the missed class) if two rows have the same position. When ties occur, the RANK() function may produce a unique level for each value but also create gaps in the ranking order.

DENSE_RANK() :

The DENSE_RANK() function assigns the same rank to all tied rows in the event of a tie without skipping any levels. In other words, the subsequent row will still have the next sequential class if two have the same position. The DENSE_RANK() function produces a unique rank for each distinct value without gaps in the ranking sequence, even when ties occur.

Example:
Consider te following Data

```
|| Score |
|-------|
| 85    |
| 92    |
| 78    |
| 70    |
| 88    |
| 95    |
| 78    |
```

RANK()

```
| Score | RANK() |
|-------|--------|
| 95    | 1      |
| 92    | 2      |
| 88    | 3      |
| 85    | 4      |
| 78    | 5      |
| 78    | 5      |
| 70    | 7      |
```

NOTE: RANK() WILL ALWAYS BE IN SYNC WITH ROW NUMBER

DENSE_RANK() :

```
| Score | DENSE_RANK() |
|-------|--------------|
| 95    | 1            |
| 92    | 2            |
| 88    | 3            |
| 85    | 4            |
| 78    | 5            |
| 78    | 5            |
| 70    | 6            |
```

ROW_NUMBER()

Each row in a result set is given a distinct sequential integer by the ROW_NUMBER() function. This frequently arranges rows in a random sequence, offering a quick way to recognize specific rows.

Syntax

```
ROW_NUMBER() OVER (
    [PARTITION BY partition_expression, ... ]
    ORDER BY sort_expression [ASC | DESC], ...
)
```

- PARTITION BY: Optional clause to partition the data into groups based on specified columns.
- ORDER BY: Specifies the column(s) used to order the rows within each partition.
- ASC | DESC: Specifies whether the ordering is in ascending or descending order.

Example

```
CREATE TABLE orders (
    order_id INT,
    customer_id VARCHAR(10),
    order_date DATE
);

-- Inserting sample data into "orders" table
INSERT INTO orders (order_id, customer_id, order_date)
VALUES
    (1, 'A', '2023-01-15'),
    (2, 'B', '2023-01-16'),
    (3, 'A', '2023-01-17'),
    (4, 'C', '2023-01-18');
```

Query:

```
SELECT
    order_id,
    customer_id,
    order_date,
    ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY order_date) AS
row_num
FROM
    orders;
```

OUTPUT:

```
| order_id | customer_id | order_date  | row_num |
|----------|-------------|-------------|---------|
| 1        | A           | 2023-01-15  | 1       |
| 3        | A           | 2023-01-17  | 2       |
| 2        | B           | 2023-01-16  | 1       |
| 4        | C           | 2023-01-18  | 1       |
```

NTILE()

The NTILE() function separates the result set into a specified number of groups or "tiles" that are roughly equal in size. It gives each row an integer number that represents the tile to assist spread data uniformly.

Syntax

```
NTILE(n) OVER (
    [PARTITION BY partition_expression, ... ]
    ORDER BY sort_expression [ASC | DESC], ...
)
```

- n: The number of tiles to divide the data into.
- PARTITION BY: Optional clause to partition the data into groups based on specified columns
- ORDER BY: Specifies the column(s) used to order the rows within each partition.

```
CREATE TABLE sales (
    salesperson_id INT,
    sale_amount DECIMAL(10, 2)
);

-- Inserting sample data into "sales" table
INSERT INTO sales (salesperson_id, sale_amount)
VALUES
    (1, 500.00),
    (2, 750.00),
    (1, 600.00),
    (3, 800.00),
    (2, 550.00),
    (3, 700.00),
    (1, 450.00);
```

Query:

```
SELECT
    salesperson_id,
    SUM(sale_amount) AS total_sale_amount,
    NTILE(3) OVER (ORDER BY SUM(sale_amount) DESC) AS salesperson_group
FROM
    sales
GROUP BY
    salesperson_id;
```

Value Window Functions

LAG()

The NTILE() function separates the result set into a specified number of groups or "tiles" that are roughly equal Based on a given offset, the LAG() function fetches the value from a preceding row in the result set. Comparing the importance of the current row with those of the rows before is helpful.

Syntax

```
LAG(column_expression, offset, default_value) OVER (
    [PARTITION BY partition_expression, ... ]
    ORDER BY sort_expression [ASC | DESC], ...
)
```

- column_expression: Specifies the column whose value you want to retrieve from a previous row.
- offset: The number of rows back from the current row to retrieve the value. The default is 1.
- default_value: Optional default value to return if there's no preceding row.

```
CREATE TABLE sales (
    sale_date DATE,
    sale_amount DECIMAL(10, 2)
);

-- Insert sample data into the "sales" table
INSERT INTO sales (sale_date, sale_amount)
VALUES
    ('2023-01-15', 500.00),
    ('2023-01-16', 750.00),
    ('2023-01-17', 600.00),
    ('2023-01-18', 800.00),
    ('2023-01-19', 550.00),
    ('2023-01-20', 700.00),
    ('2023-01-21', 450.00);
```

Query:

```
-- Query to calculate the previous day's sale amount using LAG() function
SELECT
    sale_date,
    sale_amount,
    LAG(sale_amount, 1, 0) OVER (ORDER BY sale_date) AS
prev_day_sale_amount
FROM
    sales;
```

OUTPUT:

	sale_date	sale_amount	prev_day_sale_amount
▶	2023-01-15	500.00	0.00
	2023-01-16	750.00	500.00
	2023-01-17	600.00	750.00
	2023-01-18	800.00	600.00
	2023-01-19	550.00	800.00
	2023-01-20	700.00	550.00
	2023-01-21	450.00	700.00

LEAD():

While LAG() retrieves values from the first row of the result set, the LEAD() function retrieves values from subsequent rows.

Syntax

```
LEAD(column_expression, offset, default_value) OVER (
    [PARTITION BY partition_expression, ... ]
    ORDER BY sort_expression [ASC | DESC], ...
)
```

- column_expression: Specifies the column whose value you want to retrieve from a subsequent row.
- offset: The number of rows forward from the current row to retrieve the value. Default is 1.
- default_value: Optional default value to return if there's no subsequent row.

Example

```
CREATE TABLE sales (
    sale_date DATE,
    sale_amount DECIMAL(10, 2)
);

-- Insert sample data into the "sales" table
INSERT INTO sales (sale_date, sale_amount)
VALUES
    ('2023-01-15', 500.00),
    ('2023-01-16', 750.00),
    ('2023-01-17', 600.00),
    ('2023-01-18', 800.00),
    ('2023-01-19', 550.00),
    ('2023-01-20', 700.00),
    ('2023-01-21', 450.00);
```

Query:

```
SELECT
    sale_date,
    sale_amount,
    LEAD(sale_amount, 1, 0) OVER (ORDER BY sale_date) AS
next_day_sale_amount
FROM
    sales;
```

FIRST_VALUE()

The FIRST_VALUE() function retrieves the value from the first row of a window frame.

Syntax

```
FIRST_VALUE(column_expression) OVER (
    [PARTITION BY partition_expression, ... ]
    ORDER BY sort_expression [ASC | DESC], ...
)
```

Example

```
CREATE TABLE groceries (
    product_id INT,
    product_name VARCHAR(50),
    purchase_date DATE,
    price DECIMAL(10, 2)
);

-- Insert sample data into the "groceries" table
INSERT INTO groceries (product_id, product_name, purchase_date, price)
VALUES
    (1, 'Apples', '2023-01-15', 2.99),
    (1, 'Apples', '2023-01-17', 2.75),
    (2, 'Bananas', '2023-01-16', 1.25),
    (2, 'Bananas', '2023-01-18', 1.20),
    (3, 'Milk', '2023-01-17', 3.49),
    (4, 'Bread', '2023-01-18', 2.00),
    (4, 'Bread', '2023-01-19', 2.20);
```

Query:

```
SELECT
    product_id,
    product_name,
    purchase_date,
    price,
    FIRST_VALUE(price) OVER (PARTITION BY product_id ORDER BY
purchase_date) AS first_purchase_price
FROM
    groceries;
```

OUTPUT:

```
| product_id | product_name | purchase_date | price | first_purchase_price |
|------------|--------------|---------------|-------|----------------------|
| 1          | Apples       | 2023-01-15    | 2.99  | 2.99                 |
| 1          | Apples       | 2023-01-17    | 2.75  | 2.99                 |
| 2          | Bananas      | 2023-01-16    | 1.25  | 1.25                 |
| 2          | Bananas      | 2023-01-18    | 1.20  | 1.20                 |
| 3          | Milk         | 2023-01-17    | 3.49  | 3.49                 |
| 4          | Bread        | 2023-01-18    | 2.00  | 2.00                 |
| 4          | Bread        | 2023-01-19    | 2.20  | 2.00                 |
```

So from above, we can observe that the first price for apples is 2.99

LAST_VALUE()

The LAST_VALUE() function retrieves the value from the last row of a window frame.

Syntax

```
LAST_VALUE(column_expression) OVER (
    [PARTITION BY partition_expression, ... ]
    ORDER BY sort_expression [ASC | DESC], ...
)
```

Example:
Find the last purchased price for each product

```
CREATE TABLE groceries (
    product_id INT,
    product_name VARCHAR(50),
    purchase_date DATE,
    price DECIMAL(10, 2)
);

-- Insert sample data into the "groceries" table
INSERT INTO groceries (product_id, product_name, purchase_date, price)
VALUES
    (1, 'Apples', '2023-01-15', 2.99),
    (1, 'Apples', '2023-01-17', 2.75),
    (2, 'Bananas', '2023-01-16', 1.25),
    (2, 'Bananas', '2023-01-18', 1.20),
    (3, 'Milk', '2023-01-17', 3.49),
    (4, 'Bread', '2023-01-18', 2.00),
    (4, 'Bread', '2023-01-19', 2.20);
```

Query:

```
SELECT
    product_id,
    product_name,
    purchase_date,
    price,
    LAST_VALUE(price) OVER (PARTITION BY product_id ORDER BY purchase_date)
AS last_purchase_price
FROM
    groceries;
```

OUTPUT:

product_id	product_name	purchase_date	price	last_purchase_price
1	Apples	2023-01-15	2.99	2.75
1	Apples	2023-01-17	2.75	2.75
2	Bananas	2023-01-16	1.25	1.20
2	Bananas	2023-01-18	1.20	1.20
3	Milk	2023-01-17	3.49	3.49
4	Bread	2023-01-18	2.00	2.20
4	Bread	2023-01-19	2.20	2.20

So now we have seen the windows function in the above documentation, there is something which helps to specify the rang e of rows to be considered for Windows function like sum,avg, rank

ROWS BEETWEEN:

```
<window function> OVER (
    PARTITION BY partition_expression
    ORDER BY sort_expression
    ROWS BETWEEN start PRECEDING AND end FOLLOWING
)
```

- start and end define the number of rows before and after the current row, respectively.
- PRECEDING indicates rows before the current row.
- FOLLOWING indicates rows after the current row.

Commonly used cases include:

- ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING: Includes the current row and the immediately preceding and following rows.
- ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW: Includes all rows from the beginning of the partition up to and including the current row.
- ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING: Includes the current row and all rows following it within the partition

Example

```
CREATE TABLE groceries (
    product_id INT,
    product_name VARCHAR(50),
    purchase_date DATE,
    price DECIMAL(10, 2)
);
-- Insert sample data into the "groceries" table
INSERT INTO groceries (product_id, product_name, purchase_date, price)
VALUES
    (1, 'Apples', '2023-01-15', 2.99),
    (1, 'Apples', '2023-01-17', 2.75),
    (2, 'Bananas', '2023-01-16', 1.25),
    (2, 'Bananas', '2023-01-18', 1.20),
    (3, 'Milk', '2023-01-17', 3.49),
    (4, 'Bread', '2023-01-18', 2.00),
    (4, 'Bread', '2023-01-19', 2.20);
```

Query:

```
SELECT
product_id,
product_name,
purchase_date,
price,
SUM(price) OVER (
        PARTITION BY product_id
        ORDER BY purchase_date
        ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
) AS price_sum_around
FROM
    groceries;
```

OUTPUT:

product_id	product_name	purchase_date	price	price_sum_around
1	Apples	2023-01-15	2.99	5.74
1	Apples	2023-01-17	2.75	8.74
2	Bananas	2023-01-16	1.25	2.45
2	Bananas	2023-01-18	1.20	2.45
3	Milk	2023-01-17	3.49	3.49
4	Bread	2023-01-18	2.00	4.20
4	Bread	2023-01-19	2.20	4.20

- For the first row with product_id 1, the sum includes its price (2.99) and the price of the next row (2.75), resulting in 5.74.
- For the second row with product_id 1, the sum includes its price (2.75), the price of the previous row (2.99), and the price of the next row (1.25), resulting in 8.74 and so on for the remaining rows.
- The price_sum_around column represents the sum of prices for the current row and the surrounding rows with the same product_id.