WINDOW FUNCTIONS

A window function or analytic function is a function which uses values from one or multiple rows to return a value for each row.

The easiest way to understand the window functions is to start by reviewing the aggregate functions. An aggregate function aggregates data from a set of rows into a single row.

Similar to an aggregate function, a window function operates on a set of rows. However, it does not reduce the number of rows returned by the query.

The term *window* describes the set of rows on which the window function operates. A window function returns values from the rows in a window.

Example:

The following query returns the product name, the price, product group name, along with the average prices of each product group.

```
SELECT
        product_name,
        price,
        group_name,
        AVG (price) OVER (
           PARTITION BY group_name
        )
FROM
        products
        INNER JOIN
                product_groups USING (group_id);
```

Output:

product_name	price	group_name	avg
HP Elite	1200	Laptop	850
Lenovo Thinkpad	700	Laptop	850
Sony VAIO	700	Laptop	850
Dell Vostro	800	Laptop	850
Microsoft Lumia	200	Smartphone	500
HTC One	400	Smartphone	500
Nexus	500	Smartphone	500
iPhone	900	Smartphone	500
iPad	700	Tablet	350
Kindle Fire	150	Tablet	350
Samsung Galaxy Tab	200	Tablet	350

In this query, the AVG() function works as a *window function* that operates on a set of rows specified by the **OVER** clause. Each set of rows is called a window.

The new syntax for this query is the OVER clause

```
AVG(price) OVER (PARTITION BY group_name)
```

In this syntax, the PARTITION BY distributes the rows of the result set into groups and the AVG() function is applied to each group to return the average price for each.

Note that a window function always performs the calculation on the result set after the JOIN, WHERE, GROUP BY and HAVING clause and before the final ORDER BY clause in the evaluation order.

Window Function Call Syntax:

```
window_function(arg1, arg2,..) OVER (
   [PARTITION BY partition_expression]
   [ORDER BY sort_expression [ASC | DESC] [NULLS {FIRST | LAST }])
```

Syntax Explanation:

The **window_function** is the name of the window function. Some window functions do not accept any argument.

The **PARTITION BY** clause divides rows into multiple groups or partitions to which the window function is applied. The PARTITION BY clause is optional. If you skip the PARTITION BY clause, the window function will treat the whole result set as a single partition.

The **ORDER BY** clause specifies the order of rows in each partition to which the window function is applied. It uses the NULLS FIRST or NULLS LAST option to specify whether nullable values should be first or last in the result set. The default is the NULLS LAST option.

The **frame_clause** defines a subset of rows in the current partition to which the window function is applied. This subset of rows is called a frame.

Window Function List

Some aggregate functions such as AVG(), MIN(), MAX(), SUM(), and COUNT() can be also used as window functions.

NAME	DESCRIPTION
CUME_DIST	Return the relative rank of the current row.
DENSE_RANK	Rank the current row within its partition without gaps.
FIRST_VALUE	Return a value evaluated against the first row within its partition.
LAG	Return a value evaluated at the row that is at a specified physical offset row before the current row within the partition.
LAST_VALUE	Return a value evaluated at the row that is at a specified physical offset row before the current row within the partition.
LEAD	Return a value evaluated at the row that is offset rows after the current row within the partition.
NTILE	Divide rows in a partition as equally as possible and assign each row an integer starting from 1 to the argument value.
NTH_VALUE	Return a value evaluated against the nth row in an ordered partition.
PERCENT_RANK	Return the relative rank of the current row (rank-1) / (total rows – 1)
RANK	Rank the current row within its partition with gaps.
ROW_NUMBER	Number the current row within its partition starting from 1.

ROW_NUMBER(), RANK() and DENSE_RANK() FUNCTIONS

The ROW_NUMBER(), RANK(), and DENSE_RANK() functions assign an integer to each row based on its order in its result set.

The ROW_NUMBER() function assigns a sequential number to each row in each partition. See the following query:

```
SELECT
        product_name,
        group_name,
        price,
        ROW_NUMBER () OVER (
                PARTITION BY group_name
                ORDER BY
                        price
        )
FROM
        products
INNER JOIN product_groups USING (group_id);
```

product_name	group_name	price	row_number
Sony VAIO	Laptop	700	1
Lenovo Thinkpad	Laptop	700	2
Dell Vostro	Laptop	800	3
HP Elite	Laptop	1200	4
Microsoft Lumia	Smartphone	200	1
HTC One	Smartphone	400	2
Nexus	Smartphone	500	3
iPhone	Smartphone	900	4
Kindle Fire	Tablet	150	1
Samsung Galaxy Tab	Tablet	200	2
iPad	Tablet	700	3

The RANK() function assigns ranking within an ordered partition. If rows have the same values, the RANK() function assigns the same rank, with the next ranking(s) skipped.

```
SELECT
        product_name,
        group_name,
  price,
        RANK () OVER (
                PARTITION BY group_name
                ORDER BY
                        price
        )
FROM
        products
INNER JOIN product_groups USING (group_id);
```

product_name	group_name	price	rank
Sony VAIO	Laptop	700	1
Lenovo Thinkpad	Laptop	700	1
Dell Vostro	Laptop	800	3
HP Elite	Laptop	1200	4
Microsoft Lumia	Smartphone	200	1
HTC One	Smartphone	400	2
Nexus	Smartphone	500	3
iPhone	Smartphone	900	4
Kindle Fire	Tablet	150	1
Samsung Galaxy Tab	Tablet	200	2
iPad	Tablet	700	3

In the laptop product group, both Dell Vostro and Sony VAIO products have the same price, therefore, they receive the same rank 1. The next row in the group is HP Elite that receives the rank 3 because the rank 2 is skipped.

Similar to the RANK() function, the DENSE_RANK() function assigns a rank to each row within an ordered partition, but the ranks have no gap. In other words, the same ranks are assigned to multiple rows and no ranks are skipped.

```
SELECT
        product_name,
        group_name,
        price,
        DENSE_RANK () OVER (
                PARTITION BY group_name
                ORDER BY
                        price
        )
FROM
        products
INNER JOIN product_groups USING (group_id);
```

product_name	group_name	price	dense_rank
Sony VAIO	Laptop	700	1
Lenovo Thinkpad	Laptop	700	1
Dell Vostro	Laptop	800	2
HP Elite	Laptop	1200	3
Microsoft Lumia	Smartphone	200	1
HTC One	Smartphone	400	2
Nexus	Smartphone	500	3
iPhone	Smartphone	900	4
Kindle Fire	Tablet	150	1
Samsung Galaxy Tab	Tablet	200	2
iPad	Tablet	700	3

Within the laptop product group, rank 1 is assigned twice to Dell Vostro and Sony VAIO. The next rank is 2 assigned to HP Elite.

THE FIRST_VALUE() AND LAST_VALUE() FUNCTIONS

The FIRST_VALUE() function returns a value evaluated against the first row within its partition, whereas the LAST_VALUE() function returns a value evaluated against the last row in its partition.

The following statement uses the FIRST_VALUE() to return the lowest price for every product group.

```
SELECT
        product_name,
        group_name,
        price,
        FIRST_VALUE (price) OVER (
                PARTITION BY group_name
                ORDER BY
                        price
        ) AS lowest_price_per_group
FROM
        products
INNER JOIN product_groups USING (group_id);
```

product_name	group_name	price	lowest_price_per_group
Sony VAIO	Laptop	700	700
Lenovo Thinkpad	Laptop	700	700
Dell Vostro	Laptop	800	700
HP Elite	Laptop	1200	700
Microsoft Lumia	Smartphone	200	200
HTC One	Smartphone	400	200
Nexus	Smartphone	500	200
iPhone	Smartphone	900	200
Kindle Fire	Tablet	150	150
Samsung Galaxy Tab	Tablet	200	150
iPad	Tablet	700	150

The following statement uses the LAST_VALUE() function to return the highest price for every product group.

```
SELECT
        product_name,
        group_name,
        price,
        LAST_VALUE (price) OVER (
                PARTITION BY group_name
                ORDER BY
                        price RANGE BETWEEN UNBOUNDED PRECEDING
                AND UNBOUNDED FOLLOWING
        ) AS highest_price_per_group
FROM
        products
INNER JOIN product_groups USING (group_id);
```

product_name	group_name	price	highest_price_per_group
Sony VAIO	Laptop	700	1200
Lenovo Thinkpad	Laptop	700	1200
Dell Vostro	Laptop	800	1200
HP Elite	Laptop	1200	1200
Microsoft Lumia	Smartphone	200	900
HTC One	Smartphone	400	900
Nexus	Smartphone	500	900
iPhone	Smartphone	900	900
Kindle Fire	Tablet	150	700
Samsung Galaxy Tab	Tablet	200	700
iPad	Tablet	700	700

Notice that we added the frame clause RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING because by default the frame clause is RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.

THE LAG() AND LEAD() FUNCTIONS

The LAG() function has the ability to access data from the previous row, while the LEAD() function can access data from the next row.

Both LAG() and LEAD() functions have the same syntax as follows:

```
LAG  (expression [,offset] [,default]) over_clause;
LEAD (expression [,offset] [,default]) over_clause;
```

The following statement uses the LAG() function to return the prices from the previous row and calculates the difference between the price of the current row and the previous row:

```
SELECT
        product_name,
        group_name,
        price,
        LAG (price, 1) OVER (
                PARTITION BY group_name
                ORDER BY
                        price
        ) AS prev_price,
        price - LAG (price, 1) OVER (
                PARTITION BY group_name
                ORDER BY
                        price
        ) AS cur_prev_diff
FROM
        products
INNER JOIN product_groups USING (group_id);
```

The following statement uses the LEAD() function to return the prices from the next row and calculates the difference between the price of the current row and the next row.

```
SELECT
        product_name,
        group_name,
        price,
        LEAD (price, 1) OVER (
                PARTITION BY group_name
                ORDER BY
                        price
        ) AS next_price,
        price - LEAD (price, 1) OVER (
                PARTITION BY group_name
                ORDER BY
                        price
        ) AS cur_next_diff
FROM
        products
INNER JOIN product_groups USING (group_id);
```

NTILE() FUNCTION

NTILE() function allows you to divide ordered rows in the partition into a specified number of ranked groups as equal size as possible. These ranked groups are called buckets.

The NTILE() function assigns each group a bucket number starting from 1. For each row in a group, the NTILE() function assigns a bucket number representing the group to which the row belongs.

The syntax of the NTILE() function is as follows:

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

EXAMPLES:

```
SELECT
        year,
        name,
        amount
FROM
        actual_sales
ORDER BY
        year, name;
```

	year smallint	name character varying (100)	amount numeric (10,2)
1	2018	Jack Daniel	150000.00
2	2018	Jane Johnson	110000.00
3	2018	John Doe	120000.00
4	2018	Stephane Heady	200000.00
5	2018	Yin Yang	30000.00
6	2019	Jack Daniel	180000.00
7	2019	Jane Johnson	130000.00
8	2019	John Doe	150000.00
9	2019	Stephane Heady	270000.00
10	2019	Yin Yang	25000.00

USING NTILE() FUNCTION OVER A RESULT SET

This example uses the NTILE() function to distribute rows into 3 buckets:

```
SELECT
        name,
        amount,
        NTILE(3) OVER(
                ORDER BY amount
        )
FROM
        sales_stats
WHERE
        year = 2019;
```

Output:

	name character varying (100)	amount numeric (10,2)	ntile integer
1	Yin Yang	25000.00	1
2	Jane Johnson	130000.00	1
3	John Doe	150000.00	2
4	Jack Daniel	180000.00	2
5	Stephane Heady	270000.00	3

USING NTILE() FUNCTION OVER A PARTITION

This example uses the NTILE() function to divide rows in the sales_stats table into two partitions and 3 buckets for each:

```
SELECT
        name,
        amount,
        NTILE(3) OVER(
                PARTITION BY year
                ORDER BY amount
        )
FROM
        sales_stats;
```

Output:

	year smallint	name character varying (100)	amount numeric (10,2)	ntile integer
1	2018	Yin Yang	30000.00	1
2	2018	Jane Johnson	110000.00	1
3	2018	John Doe	120000.00	2
4	2018	Jack Daniel	150000.00	2
5	2018	Stephane Heady	200000.00	3
6	2019	Yin Yang	25000.00	1
7	2019	Jane Johnson	130000.00	1
8	2019	John Doe	150000.00	2
9	2019	Jack Daniel	180000.00	2
10	2019	Stephane Heady	270000.00	3