

# Database Management Systems.

Lecture 7

# Content:

- **Window Functions**
- **Aggregate Functions as Window Functions**

# PostgreSQL Window Functions

- A *window function* performs a calculation across a set of table rows that are somehow related to the current row. This is comparable to the type of calculation that can be done with an aggregate function. But unlike regular aggregate functions, use of a window function does not cause rows to become grouped into a single output row — the rows retain their separate identities. Behind the scenes, the window function is able to access more than just the current row of the query result.

Lets take an example table:

employees

| last_name | salary | department |
|-----------|--------|------------|
| Jones     | 45000  | Accounting |
| Adams     | 50000  | Sales      |
| Johnson   | 40000  | Marketing  |
| Williams  | 37000  | Accounting |
| Smith     | 55000  | Sales      |

# Window Functions in Action

- Lets assume that you wanted to find the highest paid person in each department. There's a chance you could do this by creating a complicated stored procedure, or maybe even some very complex SQL. Most developers would even opt for pulling the data back into their preferred language and then looping over results. With window functions this gets much easier.

First we can rank each individual over a certain grouping:

```
SELECT last_name,  
       salary,  
       department,  
       rank() OVER (  
         PARTITION BY department  
         ORDER BY salary  
         DESC  
       )  
FROM employees;
```

| last_name | salary | department | rank |
|-----------|--------|------------|------|
| Jones     | 45000  | Accounting | 1    |
| Williams  | 37000  | Accounting | 2    |
| Smith     | 55000  | Sales      | 1    |
| Adams     | 50000  | Sales      | 2    |
| Johnson   | 40000  | Marketing  | 1    |

Hopefully its clear from here how we can filter and find only the top paid employee in each department:

(cont.)

```
SELECT *
FROM (
  SELECT
    last_name,
    salary,
    department,
    rank() OVER (
      PARTITION BY department
      ORDER BY salary
      DESC
    )
  FROM employees) sub_query
WHERE rank = 1;
```


| last_name | salary | department | rank |
|-----------|--------|------------|------|
| Jones     | 45000  | Accounting | 1    |
| Smith     | 55000  | Sales      | 1    |
| Johnson   | 40000  | Marketing  | 1    |

The best part of this is Postgres will optimize the query for you versus parsing over the entire result set if you were to do this your self in plpgsql or in your applications code.

# Window Functions

- 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.
- The following example uses the `AVG()` aggregate function to calculate the average price of all products in the products table:


```
SELECT
    AVG (price)
FROM
    products;
```



| avg              |
|------------------|
| 586.363636363636 |

- To apply the aggregate function to subsets of rows, you use the `GROUP BY` clause. The following example returns the average price for every product group:

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



| group_name | avg |
|------------|-----|
| Smartphone | 500 |
| Tablet     | 350 |
| Laptop     | 850 |

# Window Functions

- As you see clearly from the output, the `AVG()` function reduces the number of rows returned by the queries in both examples.
- 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.
- For instance, 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);
```

| 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.

# Analytic (Window) Functions

- Window functions applies aggregate and ranking functions over a particular window (set of rows).
- Unlike aggregate functions, they can return **multiple rows for each group**
- OVER clause is used with window functions to define that window.
- OVER clause does two things :
  - Partitions rows into form set of rows. (PARTITION BY clause is used)
  - Orders rows within those partitions into a particular order. (ORDER BY clause is used)
- **Note** – If partitions aren't done, then ORDER BY orders all rows of table



# Basic Syntax:

- Syntax:

```
window_function(arg1, arg2,..)
```

```
OVER ( [PARTITION BY partition_expression]
```

```
[ORDER BY expression] );
```

## Syntax descr.:

### **In this syntax:**

**window\_function(arg1,arg2,...)**

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

### **PARTITION BY clause**

The PARTITION BY clause divides rows into multiple groups or partitions to which the window function is applied. Like the example above, we used the product group to divide the products into groups (or partitions).

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.

### **ORDER BY clause**

The ORDER BY clause specifies the order of rows in each partition to which the window function is applied.

The ORDER BY clause 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 NULLS LAST option.

### **frame\_clause**

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 clause

If you use multiple window functions in a query:

```
SELECT
    wf1() OVER(PARTITION BY c1 ORDER BY c2),
    wf2() OVER(PARTITION BY c1 ORDER BY c2)
FROM table_name;
```

you can use the WINDOW clause to shorten the query as shown in the following query:

```
SELECT
    wf1() OVER w,
    wf2() OVER w,
FROM table_name
WINDOW w AS (PARTITION BY c1 ORDER BY c2);
```

It is also possible to use the WINDOW clause even though you call one window function in a query:

```
SELECT wf1() OVER w
FROM table_name
WINDOW w AS (PARTITION BY c1 ORDER BY c2);
```

# Aggregate functions as window functions

```
-- Average grade in every section:  
select section_id, round(avg(final_grade))  
from enrollment  
group by section_id  
order by section_id;
```

Output Result 295

64 rows

|   | section_id | round  |
|---|------------|--------|
| 1 | 80         | 91     |
| 2 | 81         | 89     |
| 3 | 82         | <null> |
| 4 | 83         | 95     |

```
select student_id, section_id, avg(final_grade) over (partition by section_id)  
from enrollment  
order by section_id;
```

Output Result 296

226 rows

|   | student_id | section_id | avg                |
|---|------------|------------|--------------------|
| 1 | 128        | 80         | 91                 |
| 2 | 103        | 81         | 88.666666666666667 |
| 3 | 104        | 81         | 88.666666666666667 |
| 4 | 240        | 81         | 88.666666666666667 |

# ROW\_NUMBER Function

- Assigns numbering to the rows of the result set data.
- The set of rows on which the ROW\_NUMBER() function operates is called a window.

- Syntax:

```
ROW_NUMBER() OVER( [PARTITION BY column_1,  
column_2,...] [ORDER BY column_3,column_4,...] )
```

## Row\_number() Example:

```
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          |

# RANK() Function

- The RANK() function assigns a ranking within an ordered partition.
- The rank of the first row is 1.
- The RANK() function adds the number of tied rows to the tied rank to calculate the rank of the next row, so the ranks may not be sequential. In addition, rows with the same values will get the same rank.

- Syntax:

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

# RANK() Example:

```
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.



## DENSE\_RANK( ) Function

- The DENSE\_Rank() function assigns ranking within an ordered partition BUT the ranks are consecutive.
- For each partition, the DENSE\_RANK() function returns the same rank for the rows which have the same values.

- Syntax:

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

## DENSE\_RANK() Example:

```
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.

# FIRST\_VALUE() Function

- The function returns the first value from the first row of the ordered set.

- Syntax:

```
FIRST_VALUE ( expression ) OVER ( [PARTITION  
BY partition_expression, ... ] ORDER BY  
sort_expression)
```

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

## FIRST\_VALUE() Example:

```
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                    |

## LAST\_VALUE() Function

- The function returns the last value in an ordered partition of a result set.

- Syntax:

```
LAST_VALUE ( expression ) OVER ( [PARTITION  
BY partition_expression, ... ] ORDER BY  
sort_expression )
```

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

## LAST\_VALUE() Example:

```
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                     |

## LEAD() Function

- The LEAD() function has the ability to access data from the next row.
- The LEAD() function is very useful for comparing the value of the current row with the value of the row that following the current row.

- **Syntax:**

- **LEAD**(expression [,offset [,default\_value]]) **OVER** ( [PARTITION BY partition\_expression, ... ] **ORDER BY** sort\_expression [ASC | DESC], ... )

# LEAD() Example:

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);
```

| product_name       | group_name | price | next_price | cur_next_diff |
|--------------------|------------|-------|------------|---------------|
| ▶ Sony VAIO        | Laptop     | 700   | 700        | 0             |
| Lenovo Thinkpad    | Laptop     | 700   | 800        | -100          |
| Dell Vostro        | Laptop     | 800   | 1200       | -400          |
| HP Elite           | Laptop     | 1200  | (Null)     | (Null)        |
| Microsoft Lumia    | Smartphone | 200   | 400        | -200          |
| HTC One            | Smartphone | 400   | 500        | -100          |
| Nexus              | Smartphone | 500   | 900        | -400          |
| iPhone             | Smartphone | 900   | (Null)     | (Null)        |
| Kindle Fire        | Tablet     | 150   | 200        | -50           |
| Samsung Galaxy Tab | Tablet     | 200   | 700        | -500          |
| iPad               | Tablet     | 700   | (Null)     | (Null)        |



## LAG() Function

- The LAG() function has the ability to access data from the previous row.
- The LAG() function will be very useful for comparing the values of the current and the previous row.

- Syntax:

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

# LAG() Example:

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);
```

| product_name       | group_name | price | prev_price | cur_prev_diff |
|--------------------|------------|-------|------------|---------------|
| ▶ Sony VAIO        | Laptop     | 700   | (Null)     | (Null)        |
| Lenovo Thinkpad    | Laptop     | 700   | 700        | 0             |
| Dell Vostro        | Laptop     | 800   | 700        | 100           |
| HP Elite           | Laptop     | 1200  | 800        | 400           |
| Microsoft Lumia    | Smartphone | 200   | (Null)     | (Null)        |
| HTC One            | Smartphone | 400   | 200        | 200           |
| Nexus              | Smartphone | 500   | 400        | 100           |
| iPhone             | Smartphone | 900   | 500        | 400           |
| Kindle Fire        | Tablet     | 150   | (Null)     | (Null)        |
| Samsung Galaxy Tab | Tablet     | 200   | 150        | 50            |
| iPad               | Tablet     | 700   | 200        | 500           |