

Lecture 7.2. Aggregate functions. Window functions.

What is an aggregate function?

An **aggregate function** takes **many input rows** and returns **one single result value**

Classic examples:

- `COUNT()` – how many rows
- `SUM()` – sum of values
- `AVG()` – average
- `MAX()` – maximum
- `MIN()` – minimum

Where they can be used:

- In the `SELECT` clause
- In the `HAVING` clause
- **Not** in `WHERE` directly (because `WHERE` is applied *before* aggregation).

Often combined with `GROUP BY` :

- `GROUP BY` divides result set into groups
- Aggregate functions calculate per group: count, max, min, avg, sum, etc.

AVG()

Defenition

`AVG()` – returns the **average value** of a set

Syntax:

```
AVG(column)
```

Basic example:

```
SELECT AVG(amount)
FROM payment;
```

```
--AVG with DISTINCT
SELECT AVG(DISTINCT amount)
FROM payment;
```

```
---AVG with SUM together
SELECT AVG(amount),
       SUM(amount)
FROM payment;
```

```
---AVG with GROUP BY
SELECT customer_id,
       first_name,
       last_name,
       AVG(amount)::numeric(10,2)
FROM payment
INNER JOIN customer USING (customer_id)
GROUP BY customer_id
ORDER BY customer_id;
```

```
---AVG with HAVING
SELECT customer_id,
       first_name,
       last_name,
       AVG(amount)::numeric(10,2)
FROM payment
INNER JOIN customer USING (customer_id)
```

```
GROUP BY customer_id  
HAVING AVG(amount) > 5  
ORDER BY customer_id;
```

```
---AVG and NULL  
CREATE TABLE t1 (  
    id    serial PRIMARY KEY,  
    amount integer  
);  
  
INSERT INTO t1 (amount) VALUES (10), (NULL), (30);  
  
SELECT AVG(amount)::numeric(10,2)  
FROM t1;
```

COUNT()

Defenition

The `COUNT()` function is an aggregate function that allows you to **get the number of rows** that match a specific condition of a query

Syntax:

```
SELECT COUNT(column)  
FROM table_name WHERE condition;
```

Basic examples:

```
SELECT COUNT(*)  
FROM payment;
```

```
---COUNT() with GROUP BY
SELECT customer_id,
       COUNT(customer_id)
FROM payment
GROUP BY customer_id;
```

```
---COUNT() with HAVING
SELECT customer_id,
       COUNT(customer_id)
FROM payment
GROUP BY customer_id
HAVING COUNT(customer_id) > 40;
```

SUM()

SUM() is an aggregate function that returns the **sum of values of values or distinct values**

Syntax:

```
SUM(DISTINCT expression)
```

Basic examples:

```
SELECT SUM(amount) AS total
FROM payment
WHERE customer_id = 2000;
```

```
---SUM() with GROUP BY
SELECT customer_id,
```

```
SUM(amount) AS total
FROM payment
GROUP BY customer_id
ORDER BY total;
```

```
---SUM() with an Expression
SELECT SUM(return_date - rental_date)
FROM rental;
```

MAX()

Definition:

MAX function is an aggregate function that returns the **maximum value in a set of values**

Syntax:

```
MAX(expression)
```

You can use the MAX function not only in the SELECT clause but also in the WHERE and HAVING clauses

Basic examples:

```
SELECT MAX(amount)
FROM payment;
```

```
---MAX function in subquery
SELECT * FROM payment
WHERE amount = (
```

```
SELECT MAX (amount) FROM payment  
);
```

MIN()

Definition:

MIN() function an aggregate function that returns the **minimum value in a set of values**.

Syntax:

```
SELECT MIN(expression)  
FROM table_expression  
...;
```

Unlike the AVG(), COUNT() and SUM() functions, the DISTINCT option does not have any effects on the MIN() function.

Basic examples:

```
SELECT MIN (rental_rate)  
FROM film;
```

```
---MIN function with GROUP BY  
SELECT name category,  
       MIN(replacement_cost) replacement_cost  
FROM category  
INNER JOIN film_category USING (category_id)  
INNER JOIN film USING (film_id)
```

```
GROUP BY name  
ORDER BY name;
```

Idea of Window Functions

A window function performs a calculation across a set of table rows that are somehow related to the current row.

Aggregate function vs window function:

- Aggregate function:
 - Takes many rows → returns **one row per group** (or one row total).
 - Example: average price per group_name.
- Window function:
 - Takes many rows → returns a value **for each row**, but this value is calculated using **other rows related to it** (same group, ordered set, etc.).

Syntax:

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

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 against the last row within its 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()

The **ROW_NUMBER()** function assigns a sequential number to each row in each partition.

```
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()

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

DENSE_RANK()

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

FIRST_VALUE()

The **FIRST_VALUE()** function returns a value evaluated against the first row within its partition

```
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()

The **LAST_VALUE()** function returns a value evaluated against the last row in its partition.

```
SELECT product_name, group_name, price,
       LAST_VALUE (price) OVER (
           PARTITION BY group_name ORDER BY
               price
       ) 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

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.

```
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

```

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)