Lecture 7

Aggregate functions.
Window functions.

Aggregate functions. Window functions.

PART I. Aggregate functions.
PART II. Window functions.

PART I. Aggregate functions.

Introduction to PostgreSQL aggregate functions

Aggregate functions perform a calculation on a set of rows and return a single row. PostgreSQL provides all standard SQL's aggregate functions as follows:

AVG() – return the average value.

COUNT() – return the number of values.

MAX() – return the maximum value.

MIN() – return the minimum value.

SUM() – return the sum of all or distinct values.

We often use the aggregate functions with the GROUP BY clause in the SELECT statement. In these cases, the GROUP BY clause divides the result set into groups of rows and the aggregate functions perform a calculation on each group e.g., maximum, minimum, average, etc.

You can use aggregate functions as expressions only in the following clauses:

SELECT clause.

HAVING clause.

PostgreSQL AVG Function

The **AVG()** function is one of the most commonly used aggregate functions in PostgreSQL. The AVG() function allows you to calculate the average value of a set.

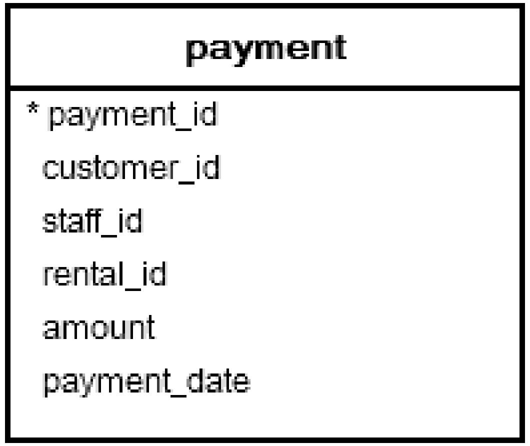
The syntax of the AVG() function is as follows:

AVG(column)

You can use the AVG() function in the SELECT and HAVING clauses.

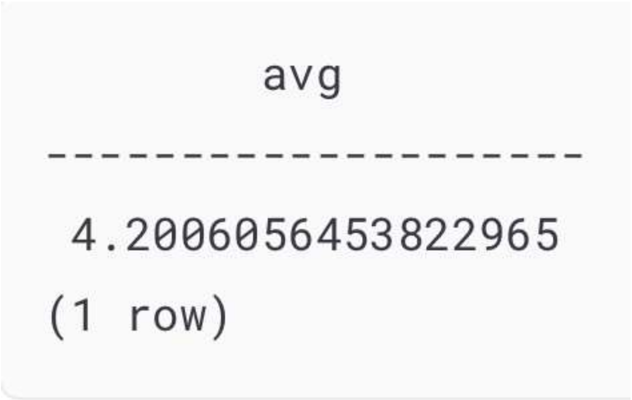
Example

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



payment

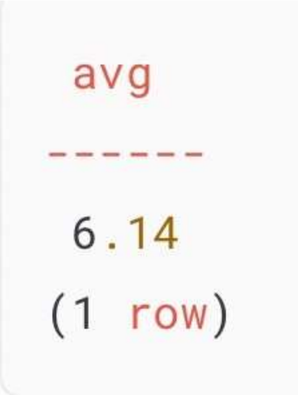
* payment_id
customer_id
staff_id
rental_id
amount
payment_date



```
        avg
--------------------
 4.2006056453822965
(1 row)
```

PostgreSQL AVG() function with DISTINCT operator

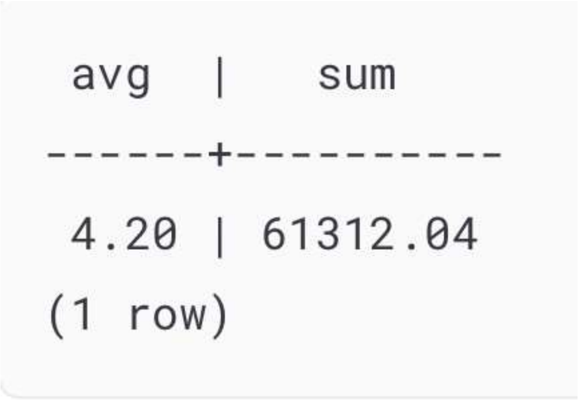
```
SELECT AVG(DISTINCT amount)::numeric(10,2)
FROM payment;
```



```
 avg
------
 6.14
(1 row)
```

PostgreSQL AVG function with SUM function

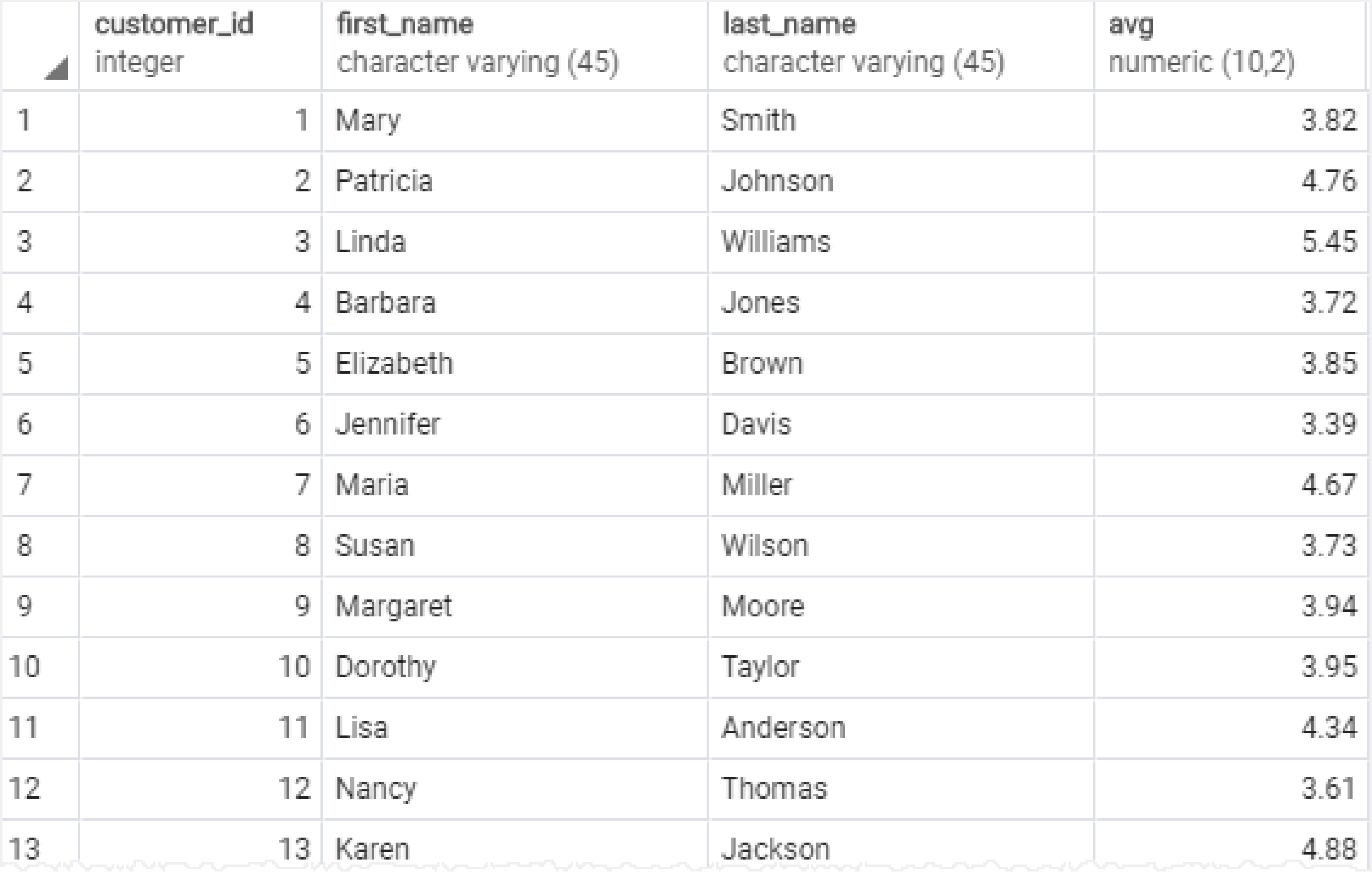
```
SELECT
        AVG(amount)::numeric(10,2),
        SUM(amount)::numeric(10,2)
FROM
        payment;
```



```
 avg  |    sum
------+----------
 4.20 | 61312.04
(1 row)
```

PostgreSQL AVG() function with GROUP BY clause

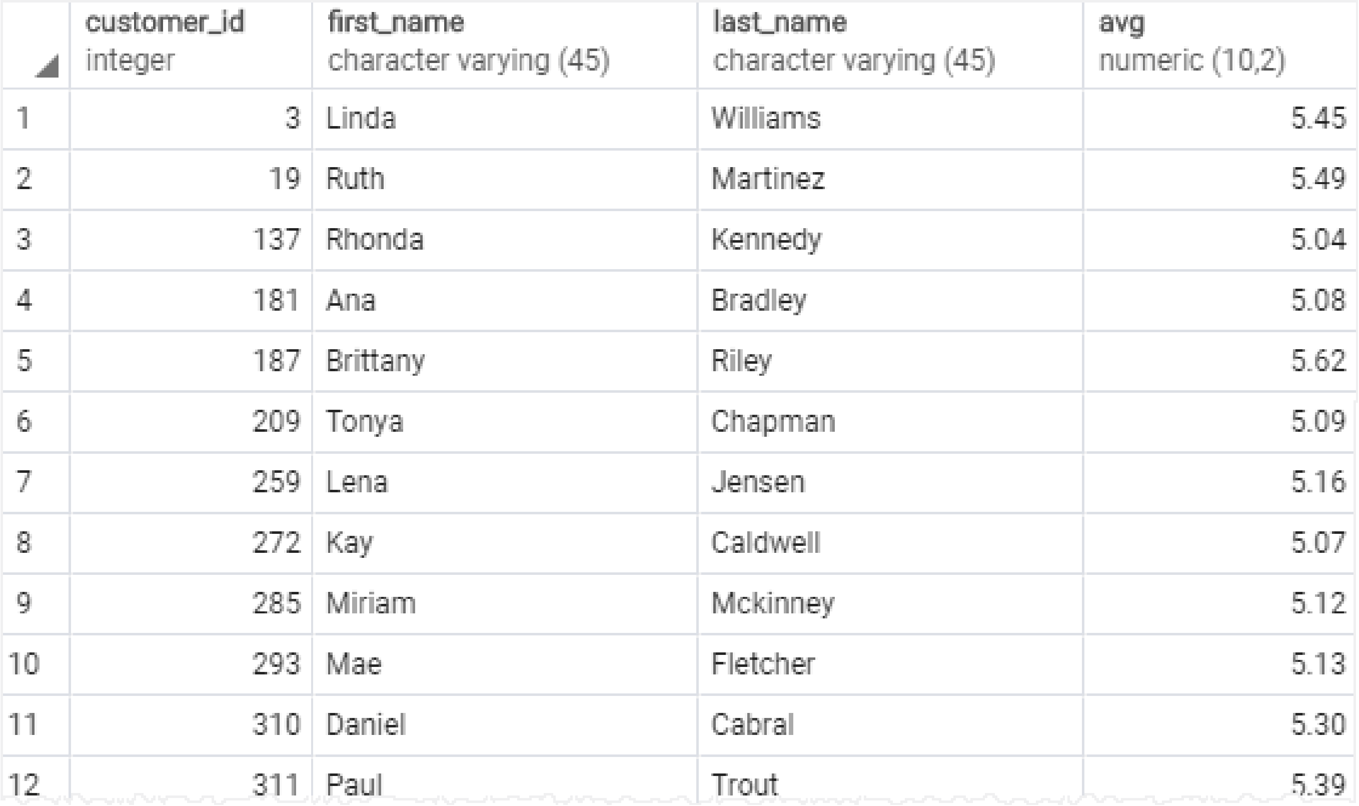
```
SELECT
        customer_id,
        first_name,
        last_name,
        AVG (amount)::NUMERIC(10,2)
FROM
        payment
INNER JOIN customer USING(customer_ic
GROUP BY
        customer_id
ORDER BY
        customer_id;
```



	customer_id integer	first_name character varying (45)	last_name character varying (45)	avg numeric (10,2)
1	1	Mary	Smith	3.82
2	2	Patricia	Johnson	4.76
3	3	Linda	Williams	5.45
4	4	Barbara	Jones	3.72
5	5	Elizabeth	Brown	3.85
6	6	Jennifer	Davis	3.39
7	7	Maria	Miller	4.67
8	8	Susan	Wilson	3.73
9	9	Margaret	Moore	3.94
10	10	Dorothy	Taylor	3.95
11	11	Lisa	Anderson	4.34
12	12	Nancy	Thomas	3.61
13	13	Karen	Jackson	4.88

PostgreSQL AVG() function with HAVING clause

```
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;
```



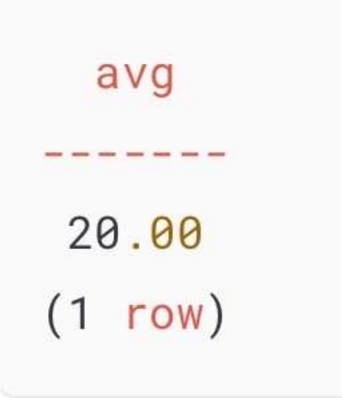
	customer_id	first_name	last_name	avg
	integer	character varying (45)	character varying (45)	numeric (10,2)
1	3	Linda	Williams	5.45
2	19	Ruth	Martinez	5.49
3	137	Rhonda	Kennedy	5.04
4	181	Ana	Bradley	5.08
5	187	Brittany	Riley	5.62
6	209	Tonya	Chapman	5.09
7	259	Lena	Jensen	5.16
8	272	Kay	Caldwell	5.07
9	285	Miriam	Mckinney	5.12
10	293	Mae	Fletcher	5.13
11	310	Daniel	Cabral	5.30
12	311	Paul	Trout	5.39

PostgreSQL AVG() function 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;
```



```
  avg
-------
 20.00
(1 row)
```

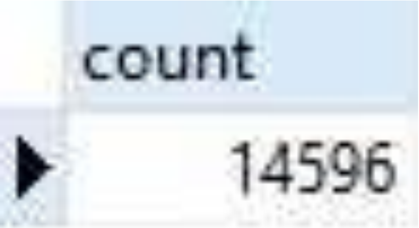
PostgreSQL COUNT Function

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

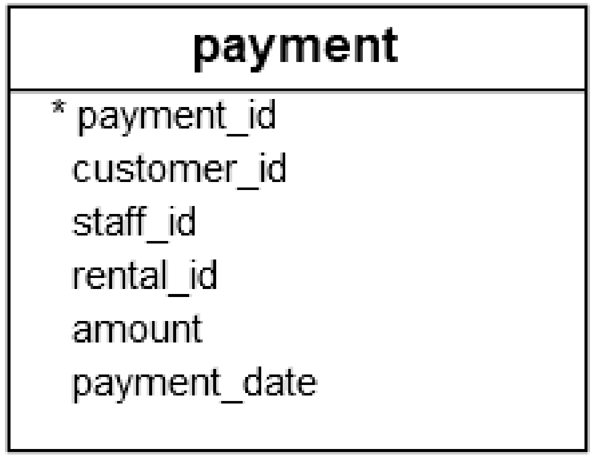
```
SELECT COUNT(column)
FROM table_name
WHERE condition;
```

PostgreSQL COUNT() function examples

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



```
count
14596
```

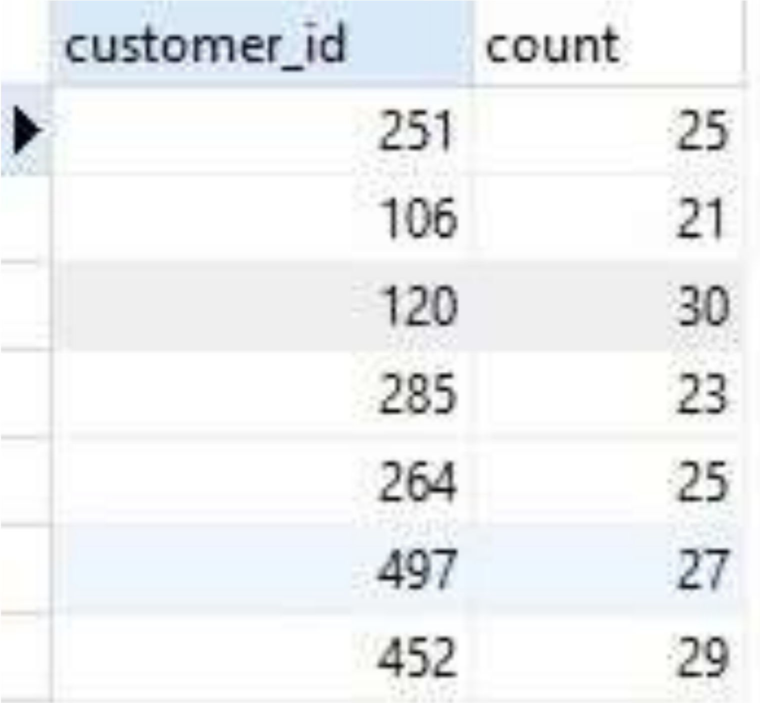


payment

* payment_id
customer_id
staff_id
rental_id
amount
payment_date

PostgreSQL COUNT() with GROUP BY clause

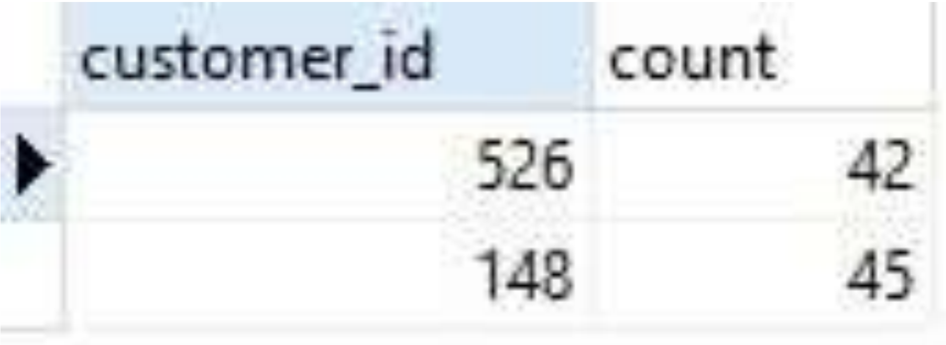
```
SELECT
        customer_id,
        COUNT (customer_id)
FROM
        payment
GROUP BY
        customer_id;
```



customer_id	count
251	25
106	21
120	30
285	23
264	25
497	27
452	29

PostgreSQL COUNT() with HAVING clause

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



customer_id	count
526	42
148	45

PostgreSQL SUM Function

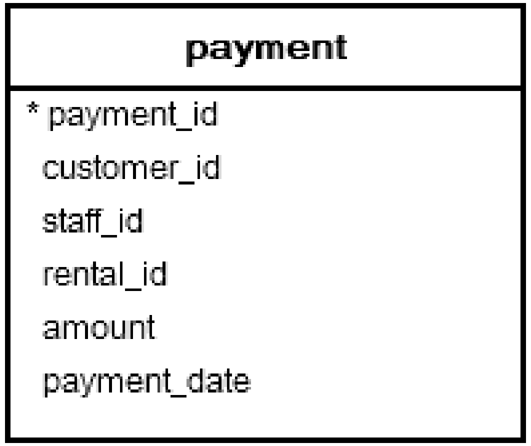
The PostgreSQL **SUM**() is an aggregate function that returns the sum of values or distinct values.

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

```
SUM(DISTINCT expression)
```

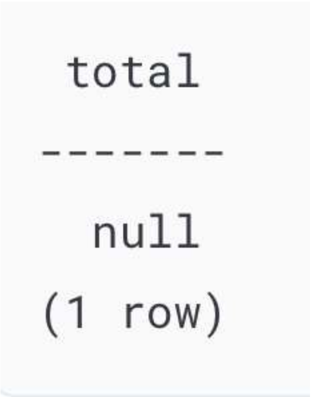
PostgreSQL SUM() function examples

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



payment

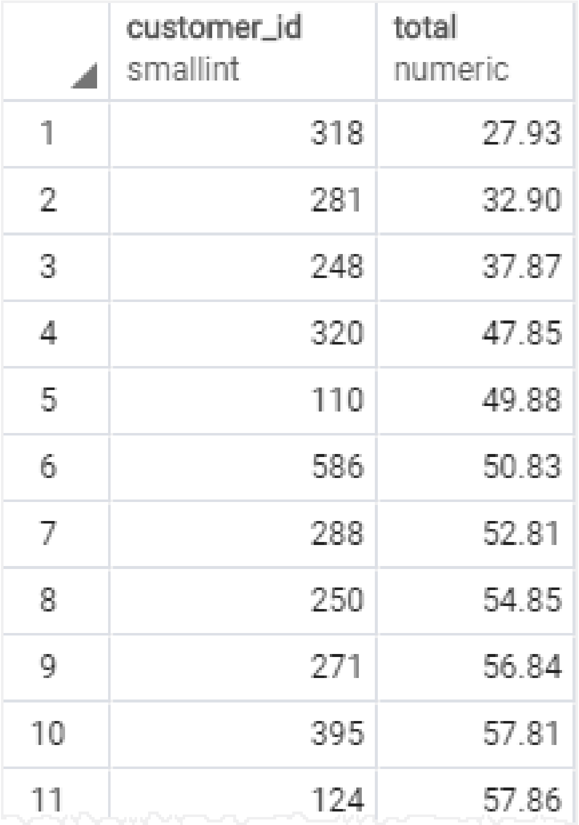
* payment_id
customer_id
staff_id
rental_id
amount
payment_date



```
 total
-------
  null
(1 row)
```

Using PostgreSQL SUM() function with GROUP BY clause

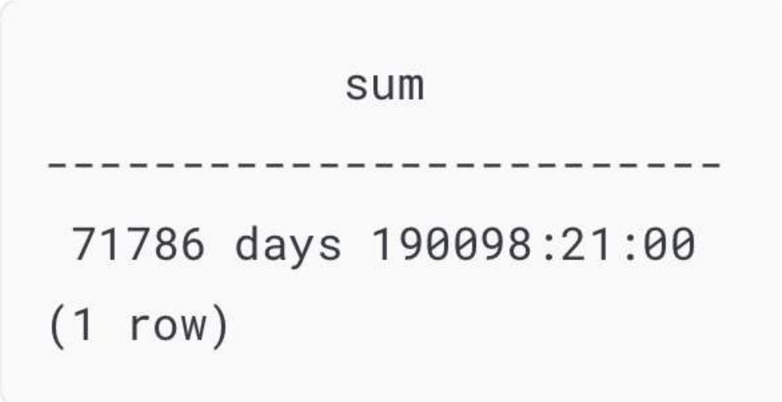
```
SELECT
        customer_id,
        SUM (amount) AS total
FROM
        payment
GROUP BY
        customer_id
ORDER BY total;
```



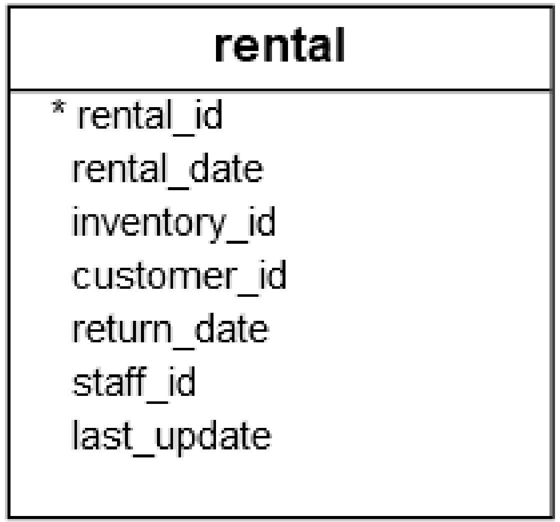
	customer_id smallint	total numeric
1	318	27.93
2	281	32.90
3	248	37.87
4	320	47.85
5	110	49.88
6	586	50.83
7	288	52.81
8	250	54.85
9	271	56.84
10	395	57.81
11	124	57.86

Using PostgreSQL SUM with expression

```
SELECT SUM(return_date - rental_date )
FROM rental;
```



```
            sum
---------------------------
 71786 days 190098:21:00
(1 row)
```



PostgreSQL MAX Function

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

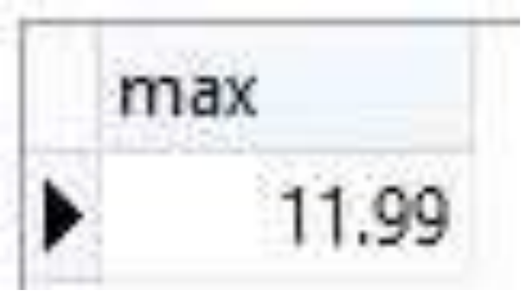
The syntax of the MAX function is as follows:

MAX(expression);

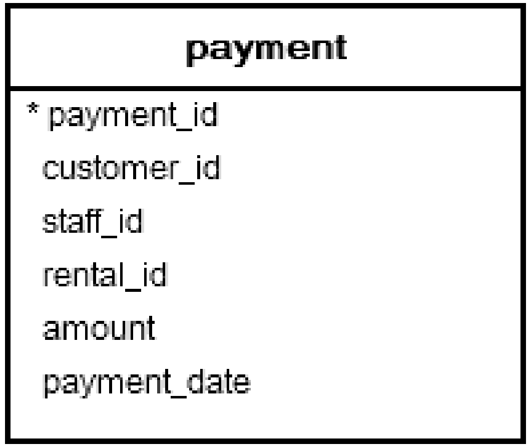
You can use the MAX function not only in the SELECT clause but also in the WHERE and HAVING clauses.

PostgreSQL MAX function examples

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



```
max
11.99
```

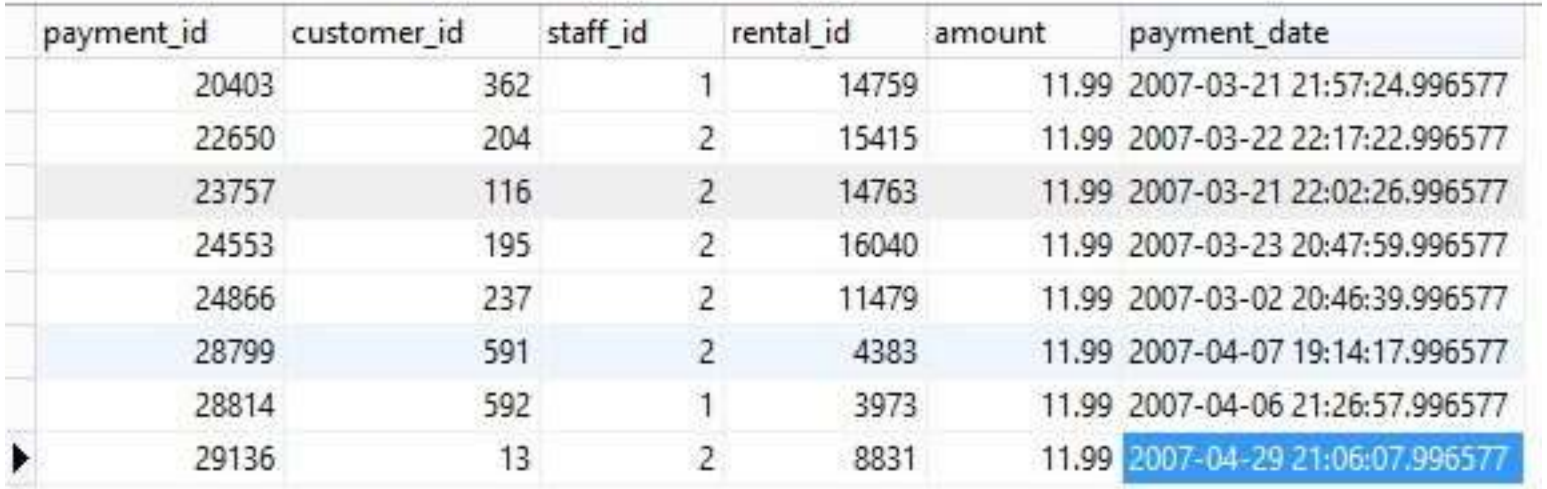


payment

* payment_id
customer_id
staff_id
rental_id
amount
payment_date

PostgreSQL MAX function in subquery

```
SELECT * FROM payment
WHERE amount = (
   SELECT MAX (amount)
   FROM payment
);
```



payment_id	customer_id	staff_id	rental_id	amount	payment_date
20403	362	1	14759	11.99	2007-03-21 21:57:24.996577
22650	204	2	15415	11.99	2007-03-22 22:17:22.996577
23757	116	2	14763	11.99	2007-03-21 22:02:26.996577
24553	195	2	16040	11.99	2007-03-23 20:47:59.996577
24866	237	2	11479	11.99	2007-03-02 20:46:39.996577
28799	591	2	4383	11.99	2007-04-07 19:14:17.996577
28814	592	1	3973	11.99	2007-04-06 21:26:57.996577
29136	13	2	8831	11.99	2007-04-29 21:06:07.996577

PostgreSQL MIN Function

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

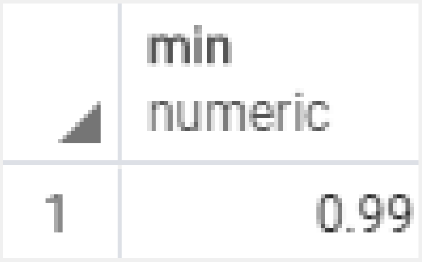
The syntax of the MIN() function is as follows:

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

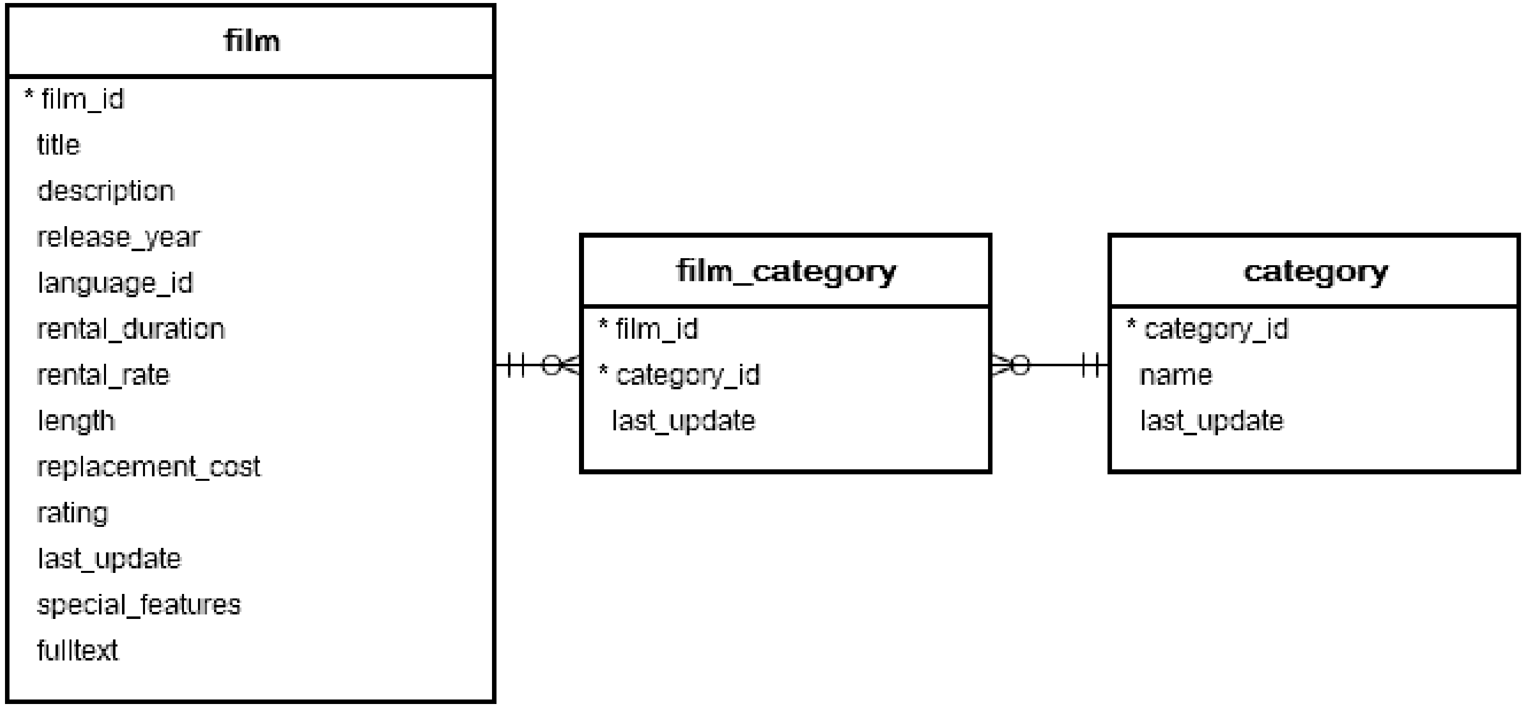
Unlike the AVG(), COUNT() and SUM() functions, the DISTINCT option does not have any effects on the MIN() function.

PostgreSQL MIN() function examples

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



	min numeric
1	0.99



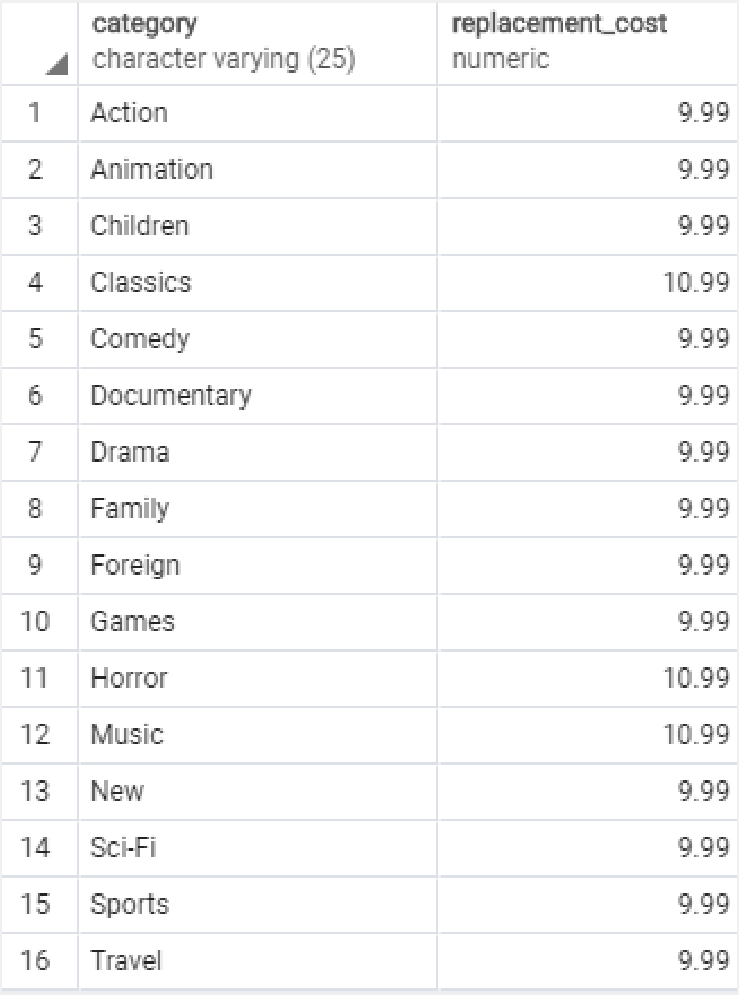
film
* film_id
title
description
release_year
language_id
rental_duration
rental_rate
length
replacement_cost
rating
last_update
special_features
fulltext

film_category
* film_id
* category_id
last_update

category
* category_id
name
last_update

Using PostgreSQL MIN function with GROUP BY clause

```
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;
```



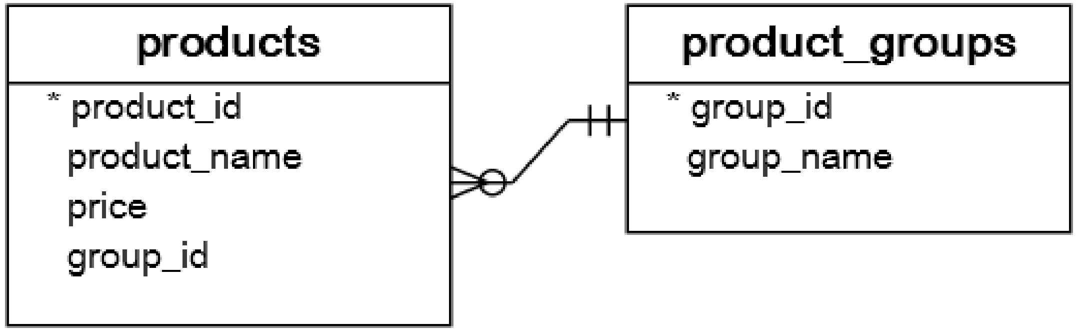
	category	replacement_cost
	character varying (25)	numeric
1	Action	9.99
2	Animation	9.99
3	Children	9.99
4	Classics	10.99
5	Comedy	9.99
6	Documentary	9.99
7	Drama	9.99
8	Family	9.99
9	Foreign	9.99
10	Games	9.99
11	Horror	10.99
12	Music	10.99
13	New	9.99
14	Sci-Fi	9.99
15	Sports	9.99
16	Travel	9.99

PART II. Window functions.

PostgreSQL Window Functions

```
CREATE TABLE product_groups (
        group_id serial PRIMARY KEY,
        group_name VARCHAR (255) NOT NULL
);

CREATE TABLE products (
        product_id serial PRIMARY KEY,
        product_name VARCHAR (255) NOT NULL,
        price DECIMAL (11, 2),
        group_id INT NOT NULL,
        FOREIGN KEY (group_id) REFERENCES product_groups (group_id)
);
```



products
* product_id
product_name
price
group_id

product_groups
* group_id
group_name

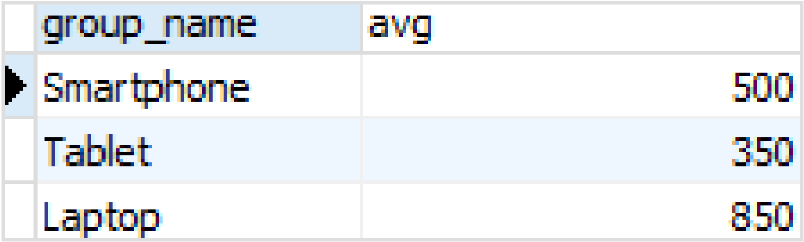
PostgreSQL Window Functions

```
INSERT INTO product_groups (group_name)
VALUES
        ('Smartphone'),
        ('Laptop'),
        ('Tablet');

INSERT INTO products (product_name, group_id,price)
VALUES
        ('Microsoft Lumia', 1, 200),
        ('HTC One', 1, 400),
        ('Nexus', 1, 500),
        ('iPhone', 1, 900),
        ('HP Elite', 2, 1200),
        ('Lenovo Thinkpad', 2, 700),
        ('Sony VAIO', 2, 700),
        ('Dell Vostro', 2, 800),
        ('iPad', 3, 700),
        ('Kindle Fire', 3, 150),
        ('Samsung Galaxy Tab', 3, 200);
```

Introduction to PostgreSQL window functions

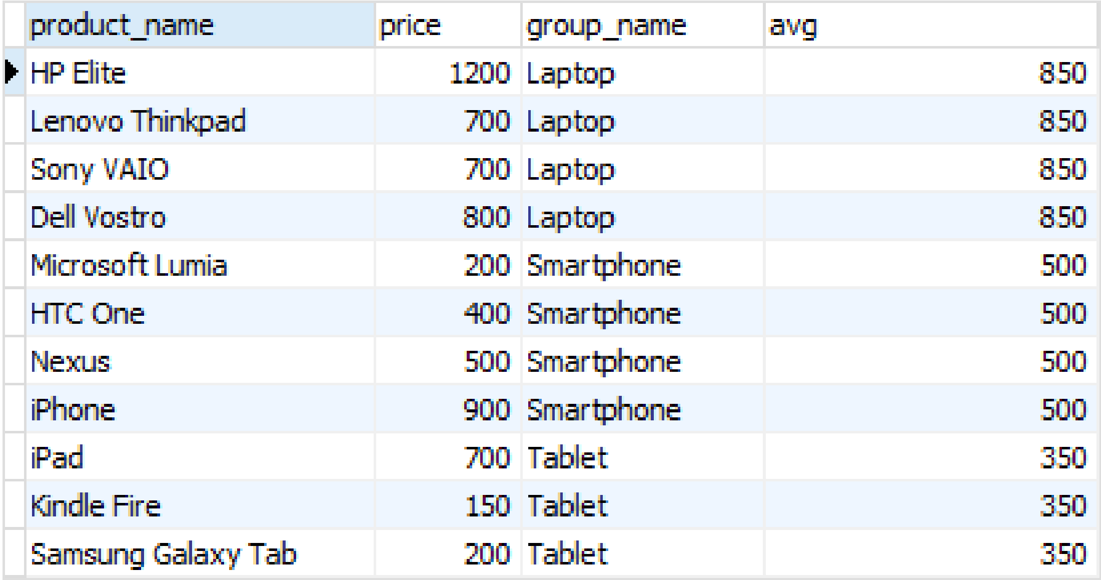
```
SELECT
        group_name,
        AVG (price)
FROM
        products
INNER JOIN product_groups USING (group_id)
GROUP BY
        group_name;
```



group_name	avg
Smartphone	500
Tablet	350
Laptop	850

Example

```
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

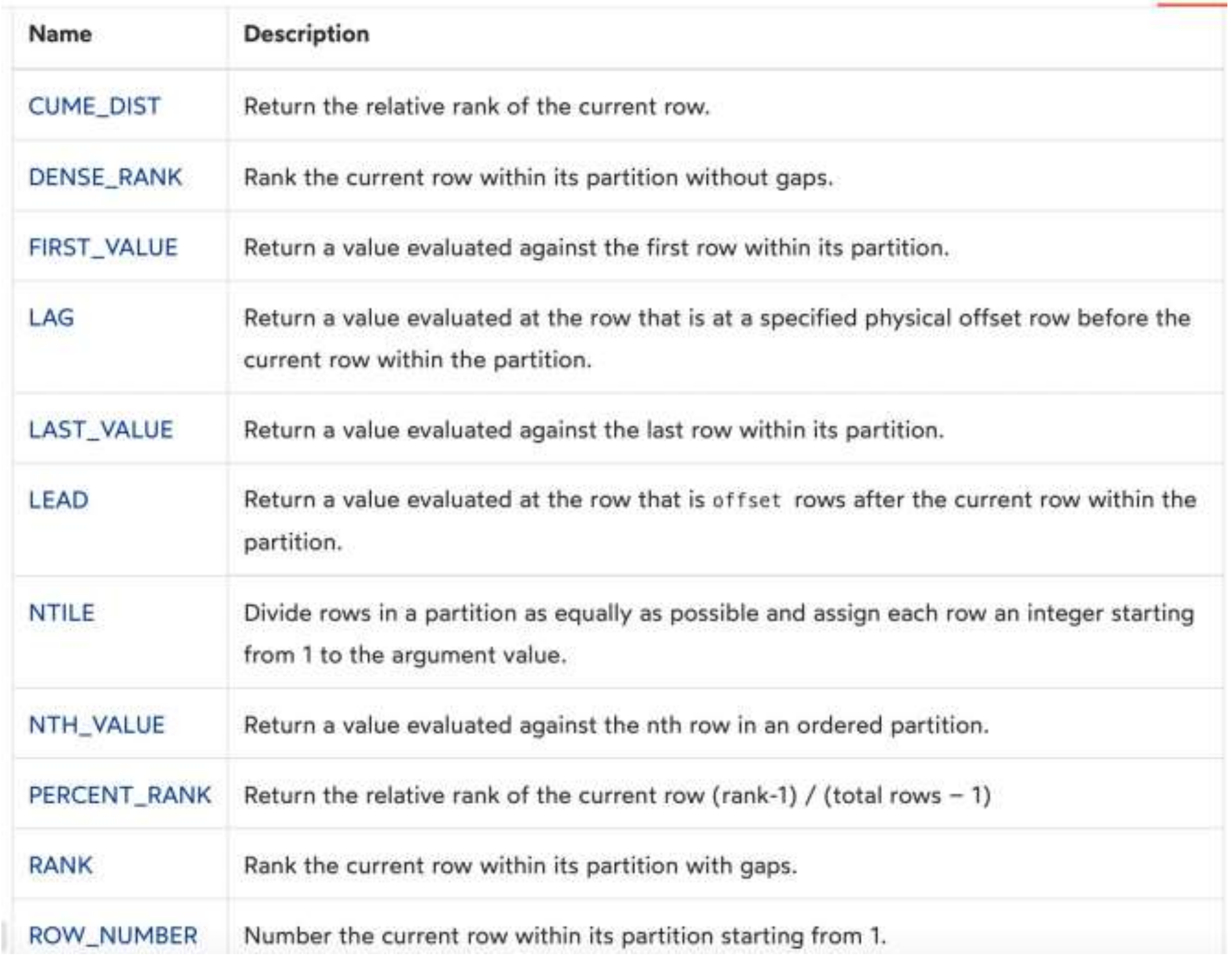
PostgreSQL Window Function

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

PostgreSQL Window Function Syntax:

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

Window functions

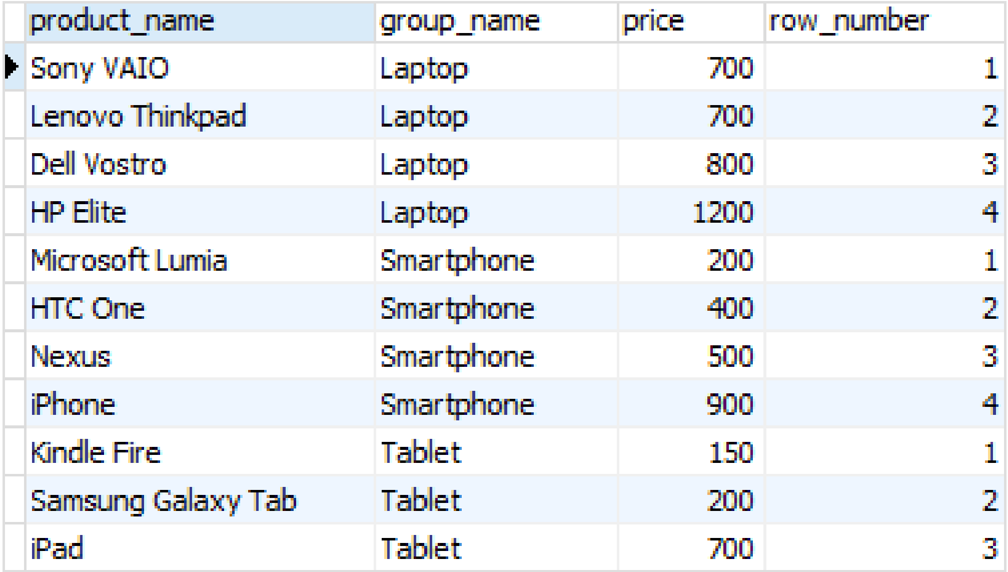


Name	Description
CUME_DIST	Return the relative rank of the current row.
DENSE_RANK	Rank the current row within its partition without gaps.
FIRST_VALUE	Return a value evaluated against the first row within its partition.
LAG	Return a value evaluated at the row that is at a specified physical offset row before the current row within the partition.
LAST_VALUE	Return a value evaluated 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.

The ROW_NUMBER() function

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);
```

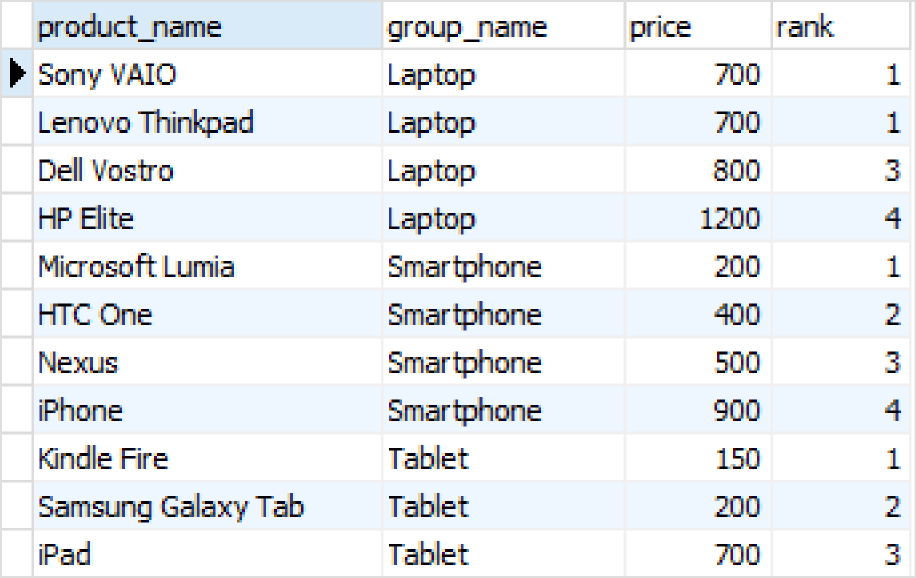


product_name	group_name	price	row_number
Sony VAIO	Laptop	700	1
Lenovo Thinkpad	Laptop	700	2
Dell Vostro	Laptop	800	3
HP Elite	Laptop	1200	4
Microsoft Lumia	Smartphone	200	1
HTC One	Smartphone	400	2
Nexus	Smartphone	500	3
iPhone	Smartphone	900	4
Kindle Fire	Tablet	150	1
Samsung Galaxy Tab	Tablet	200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

The Rank() function

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);
```

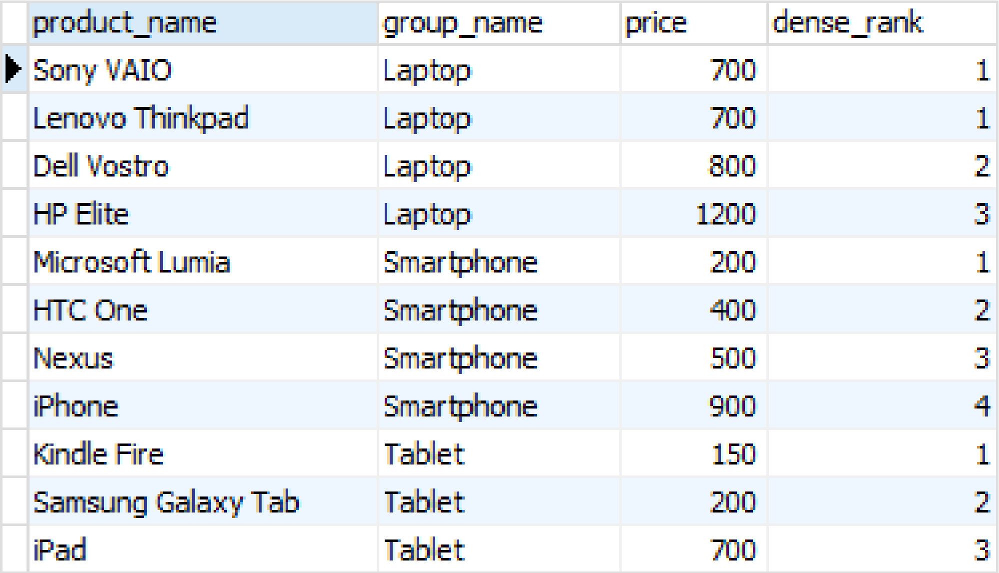


product_name	group_name	price	rank
Sony VAIO	Laptop	700	1
Lenovo Thinkpad	Laptop	700	1
Dell Vostro	Laptop	800	3
HP Elite	Laptop	1200	4
Microsoft Lumia	Smartphone	200	1
HTC One	Smartphone	400	2
Nexus	Smartphone	500	3
iPhone	Smartphone	900	4
Kindle Fire	Tablet	150	1
Samsung Galaxy Tab	Tablet	200	2
iPad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

The Dense_rank() function

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);
```

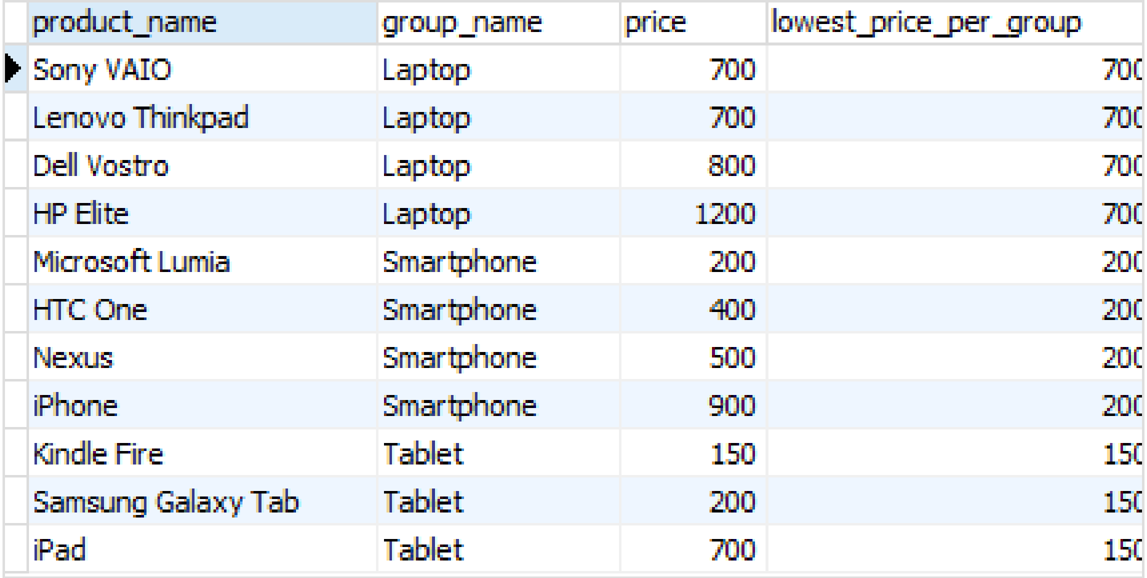


product_name	group_name	price	dense_rank
Sony VAIO	Laptop	700	1
Lenovo Thinkpad	Laptop	700	1
Dell Vostro	Laptop	800	2
HP Elite	Laptop	1200	3
Microsoft Lumia	Smartphone	200	1
HTC One	Smartphone	400	2
Nexus	Smartphone	500	3
iPhone	Smartphone	900	4
Kindle Fire	Tablet	150	1
Samsung Galaxy Tab	Tablet	200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

The FIRST_VALUE() function

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);
```

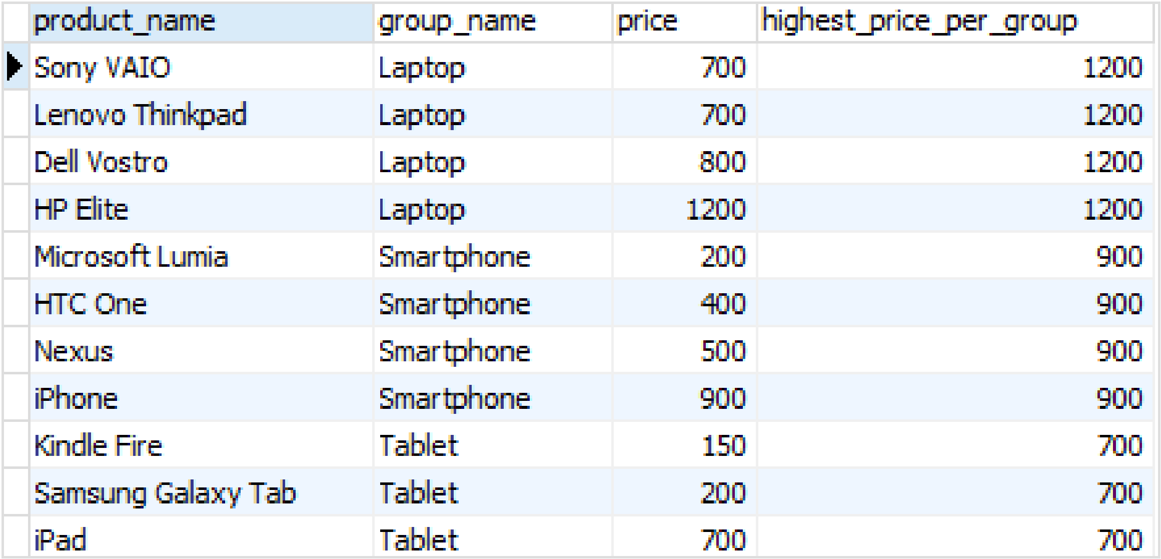


product_name	group_name	price	lowest_price_per_group
Sony VAIO	Laptop	700	700
Lenovo Thinkpad	Laptop	700	700
Dell Vostro	Laptop	800	700
HP Elite	Laptop	1200	700
Microsoft Lumia	Smartphone	200	200
HTC One	Smartphone	400	200
Nexus	Smartphone	500	200
iPhone	Smartphone	900	200
Kindle Fire	Tablet	150	150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

The LAST_VALUE() function

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

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.

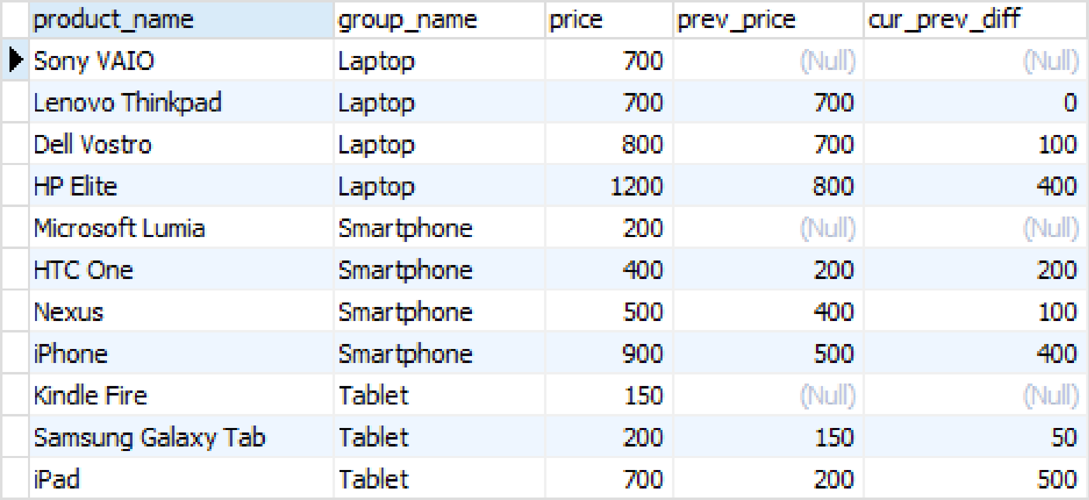
Both LAG() and LEAD() functions have the same syntax as follows:

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

In this syntax:
- expression – a column or expression to compute the returned value.
-offset – the number of rows preceding (LAG)/ following (LEAD) the current row. It defaults to 1.
- default – the default returned value if the offset goes beyond the scope of the window. The default is NULL if you skip it.

The LAG() function example

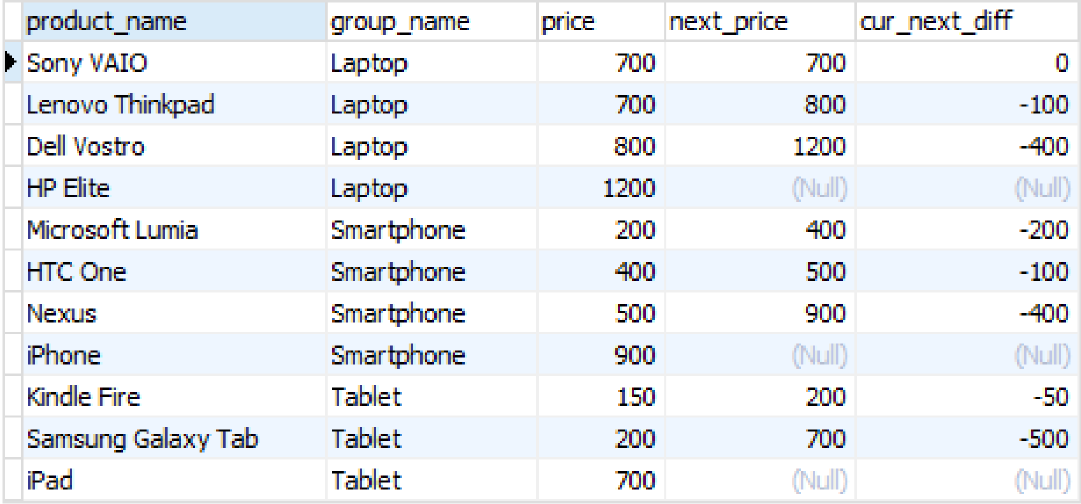
```
SELECT
        product_name,
        group_name,
        price,
        LAG (price, 1) OVER (
                PARTITION BY group_name
                ORDER BY
                        price
        ) AS prev_price,
        price - LAG (price, 1) OVER (
                PARTITION BY group_name
                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

The LEAD() function example

```
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)