Advanced SQL for data science

•••

Week 3

Outline

Joining multiple tables

Grouping data

Common table expression & Window function

Joins

Join is used to combine columns from one (self-join) or more tables

POSTGRESQL JOINS

POSTGRESQL TUTORIAL

```
SELECT * FROM a
INNER JOIN b ON a.key = b.key
```

```
SELECT * FROM a
LEFT JOIN b ON a.key = b.key
```

```
SELECT * FROM a
RIGHT JOIN b ON a.key = b.key
```

```
SELECT * FROM a
LEFT JOIN b ON a.key = b.key
WHERE b.key IS NULL
```

```
SELECT * FROM a
RIGHT JOIN b ON a.key = b.key
WHERE a.key IS NULL
```

```
SELECT * FROM a
FULL JOIN b ON a.key = b.key
```

```
SELECT * FROM a
FULL JOIN b ON a.key = b.key
WHERE a.key IS NULL OR b.key IS NULL
```

Inner join

What are common fruits between the two tables?

The following statement returns data from the `basket_a` table:

	a integer	fruit_a character varying (100)
1	1	Apple
2	2	Orange
3	3	Banana
4	4	Cucumber

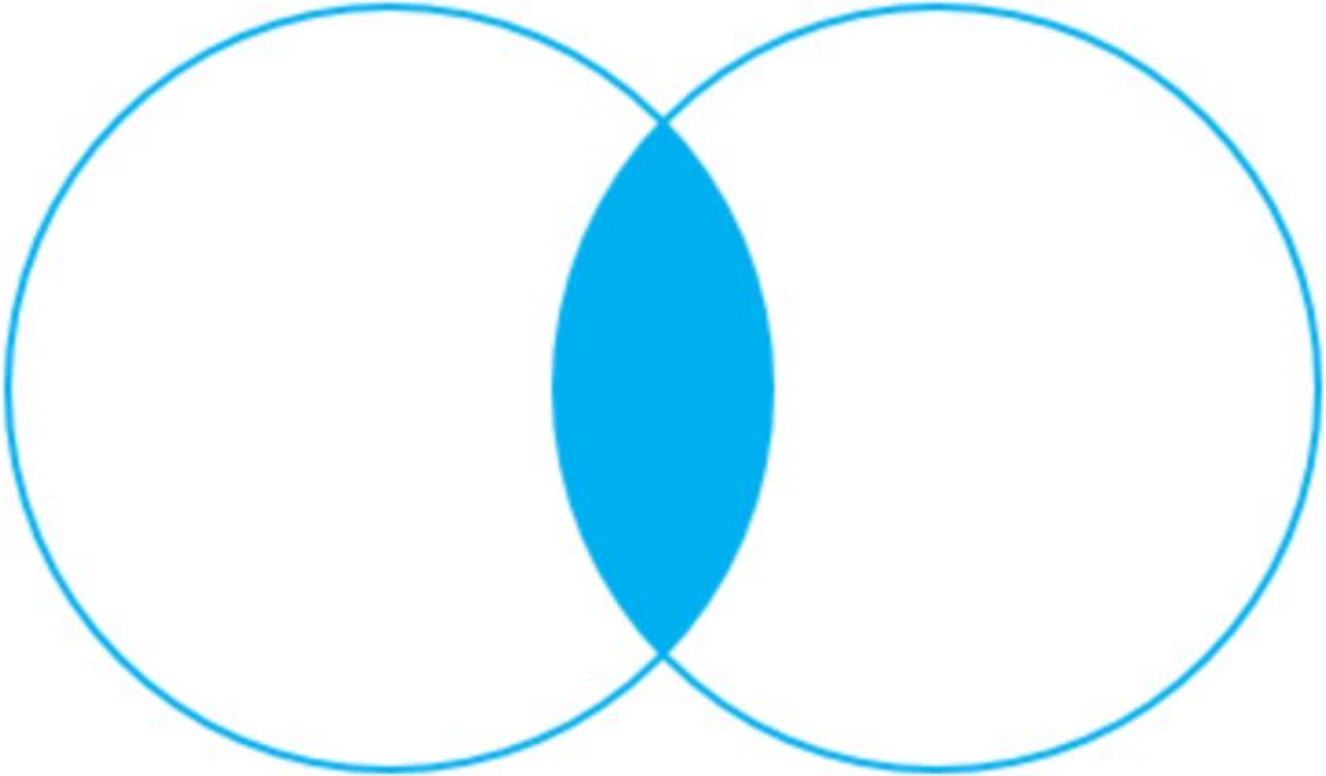
And the following statement returns data from the `basket_b` table:

	b integer	fruit_b character varying (100)
1	1	Orange
2	2	Apple
3	3	Watermelon
4	4	Pear

Inner join script

```
SELECT
    a,
    fruit_a,
    b,
    fruit_b
FROM
    basket_a
INNER JOIN basket_b
    ON fruit_a = fruit_b;
```

	a integer	fruit_a character varying (100)	b integer	fruit_b character varying (100)
1	1	Apple	2	Apple
2	2	Orange	1	Orange



INNER JOIN

POSTGRESQL JOINS

POSTGRESQL TUTORIAL

```
SELECT * FROM a
INNER JOIN b ON a.key = b.key
```

```
SELECT * FROM a
LEFT JOIN b ON a.key = b.key
```

```
SELECT * FROM a
RIGHT JOIN b ON a.key = b.key
```

```
SELECT * FROM a
LEFT JOIN b ON a.key = b.key
WHERE b.key IS NULL
```

```
SELECT * FROM a
RIGHT JOIN b ON a.key = b.key
WHERE a.key IS NULL
```

```
SELECT * FROM a
FULL JOIN b ON a.key = b.key
```

```
SELECT * FROM a
FULL JOIN b ON a.key = b.key
WHERE a.key IS NULL OR b.key IS NULL
```

Left join

What are common fruits between the two tables?

The following statement returns data from the `basket_a` table:

	a integer	fruit_a character varying (100)
1	1	Apple
2	2	Orange
3	3	Banana
4	4	Cucumber

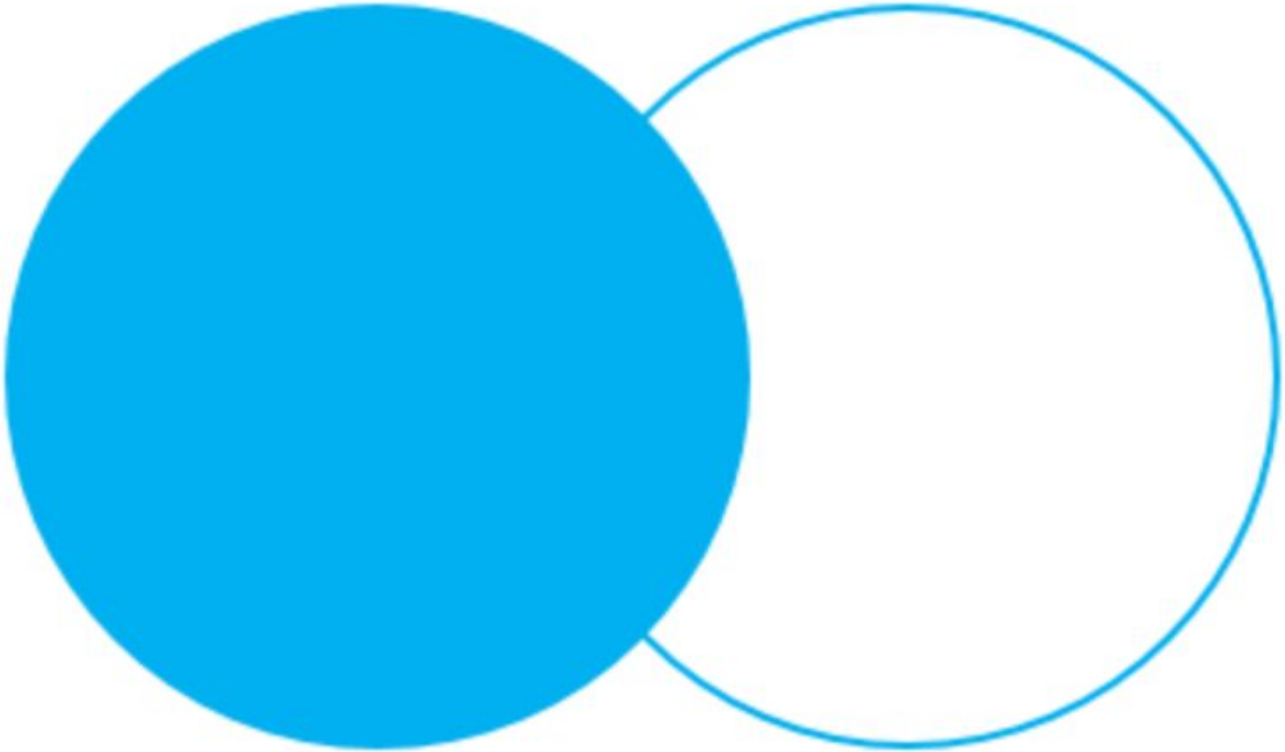
And the following statement returns data from the `basket_b` table:

	b integer	fruit_b character varying (100)
1	1	Orange
2	2	Apple
3	3	Watermelon
4	4	Pear

Left join

```
SELECT
    a,
    fruit_a,
    b,
    fruit_b
FROM
    basket_a
LEFT JOIN basket_b
   ON fruit_a = fruit_b;
```

	a integer	fruit_a character varying (100)	b integer	fruit_b character varying (100)
1	1	Apple	2	Apple
2	2	Orange	1	Orange
3	3	Banana	[null]	[null]
4	4	Cucumber	[null]	[null]



LEFT OUTER JOIN

POSTGRESQL JOINS

POSTGRESQL TUTORIAL

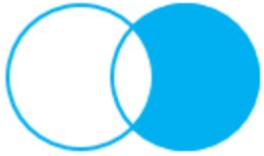
```
SELECT * FROM a
INNER JOIN b ON a.key = b.key
```

```
SELECT * FROM a
LEFT JOIN b ON a.key = b.key
```

```
SELECT * FROM a
RIGHT JOIN b ON a.key = b.key
```

```
SELECT * FROM a
LEFT JOIN b ON a.key = b.key
WHERE b.key IS NULL
```

```
SELECT * FROM a
RIGHT JOIN b ON a.key = b.key
WHERE a.key IS NULL
```



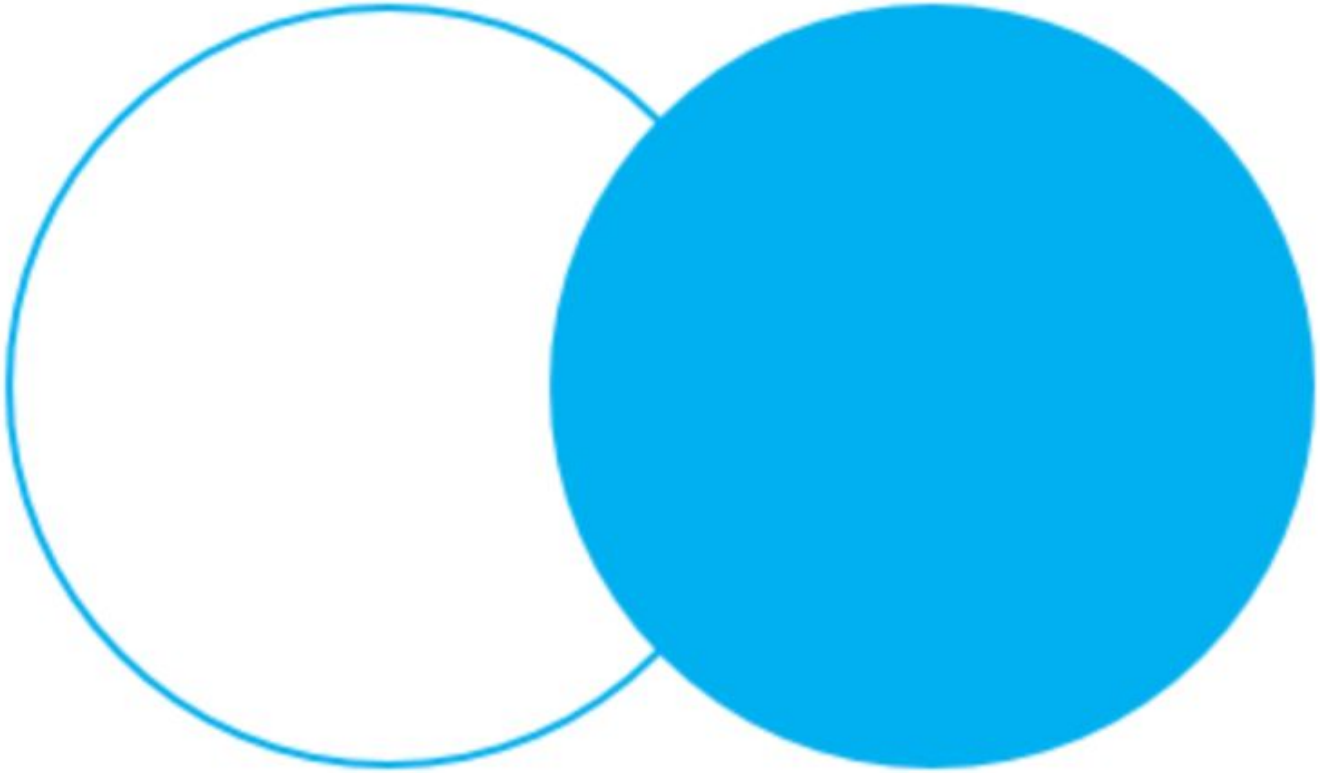
```
SELECT * FROM a
FULL JOIN b ON a.key = b.key
```

```
SELECT * FROM a
FULL JOIN b ON a.key = b.key
WHERE a.key IS NULL OR b.key IS NULL
```

Right join

```
SELECT
    a,
    fruit_a,
    b,
    fruit_b
FROM
    basket_a
RIGHT JOIN basket_b ON fruit_a = fruit_b;
```

	a integer	fruit_a character varying (100)	b integer	fruit_b character varying (100)
1	2	Orange	1	Orange
2	1	Apple	2	Apple
3	[null]	[null]	3	Watermelon
4	[null]	[null]	4	Pear



RIGHT OUTER JOIN

SELECT * FROM a
INNER JOIN b ON a.key = b.key

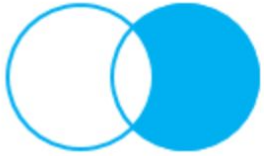
SELECT * FROM a
LEFT JOIN b ON a.key = b.key

SELECT * FROM a
RIGHT JOIN b ON a.key = b.key

POSTGRESQL
JOINS

POSTGRESQL TUTORIAL

SELECT * FROM a
LEFT JOIN b ON a.key = b.key
WHERE b.key IS NULL



SELECT * FROM a
RIGHT JOIN b ON a.key = b.key
WHERE a.key IS NULL

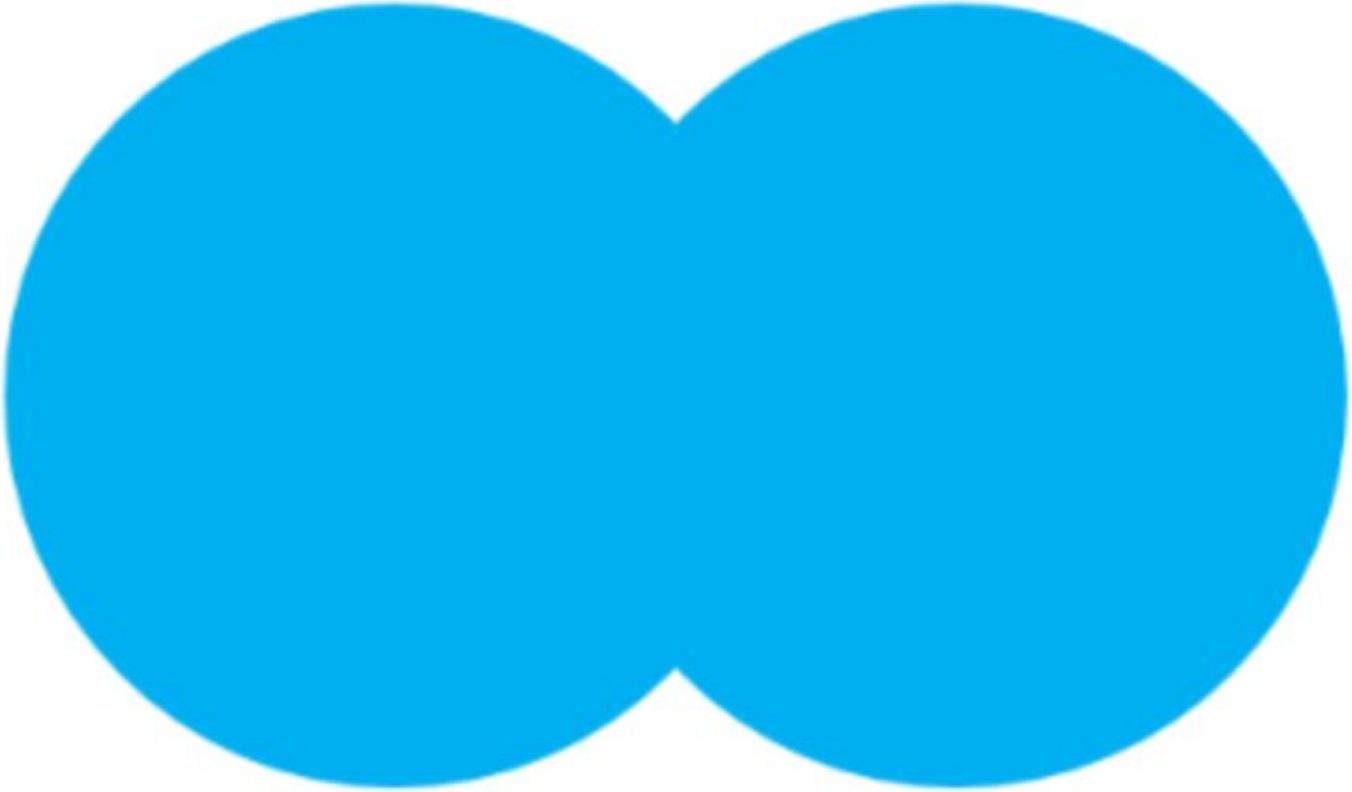
SELECT * FROM a
FULL JOIN b ON a.key = b.key

SELECT * FROM a
FULL JOIN b ON a.key = b.key
WHERE a.key IS NULL OR b.key IS NULL

Outer join

```
SELECT
    a,
    fruit_a,
    b,
    fruit_b
FROM
    basket_a
FULL OUTER JOIN basket_b
    ON fruit_a = fruit_b;
```

	a integer	fruit_a character varying (100)	b integer	fruit_b character varying (100)
1	1	Apple	2	Apple
2	2	Orange	1	Orange
3	3	Banana	[null]	[null]
4	4	Cucumber	[null]	[null]
5	[null]	[null]	3	Watermelon
6	[null]	[null]	4	Pear



FULL OUTER JOIN

Outline

Joining multiple tables

Grouping data

Common table expression & Window function

Grouping data

Group by

Having

Group by

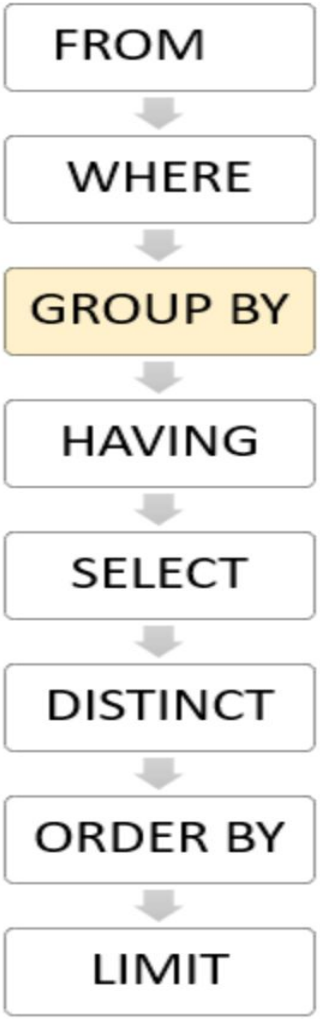
```
SELECT
    column_1,
    column_2,
    ...,
    aggregate_function(column_3)
FROM
    table_name
GROUP BY
    column_1,
    column_2,
    ...;
```

Aggregate functions example

SUM (): To calculate the sum of items

COUNT (): To get the number of items in the groups

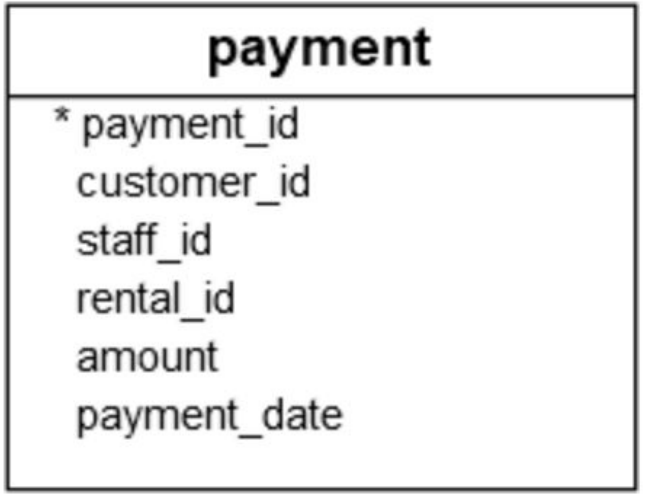
Order of evaluation of group by



Group by

```
SELECT
    customer_id
FROM
    payment
GROUP BY
    customer_id;
```

	customer_id smallint
1	184
2	87
3	477
4	273
5	550
6	51
7	394
8	272
9	70



Group by with SUM ()

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

	customer_id smallint	sum numeric
1	184	80.80
2	87	137.72
3	477	106.79
4	273	130.72
5	550	151.69
6	51	123.70
7	394	77.80
8	272	65.87
9	70	75.83
10	190	102.75
11	350	63.79

payment

* payment_id
customer_id
staff_id
rental_id
amount
payment_date

Group by with Multiple columns

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

payment

- * payment_id
- customer_id
- staff_id
- rental_id
- amount
- payment_date

	customer_id smallint	staff_id smallint	sum numeric
1	1	2	53.85
2	1	1	60.85
3	2	2	67.88
4	2	1	55.86
5	3	1	59.88
6	3	2	70.88
7	4	2	31.90
8	4	1	49.88
9	5	1	63.86
10	5	2	70.79

Group by with Date column

This can be used a lot because companies wants to know the trends of important key performance indicator (KPI) **by time**

```
SELECT
        DATE(payment_date) paid_date,
        SUM(amount) sum
FROM
        payment
GROUP BY
        DATE(payment_date);
```

	paid_date date	sum numeric
1	2007-02-14	116.73
2	2007-02-19	1290.90
3	2007-02-20	1219.09
4	2007-03-19	2617.69
5	2007-04-26	347.21
6	2007-04-08	2227.84
7	2007-02-15	1188.92
8	2007-04-28	2622.73
9	2007-03-17	2442.16
10	2007-03-20	2669.89
11	2007-03-23	2342.43
12	2007-03-21	2868.27

payment

* payment_id
customer_id
staff_id
rental_id
amount
payment_date

Grouping data

Group by

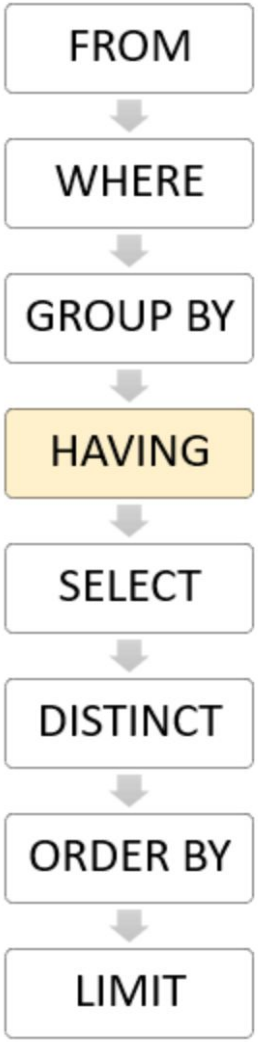
Having

Having

The HAVING clause specifies a search condition for a group or an aggregate.

```
SELECT
        column1,
        aggregate_function (column2)
FROM
        table_name
GROUP BY
        column1
HAVING
        condition;
```

Order of evaluation of HAVING



HAVING VS. WHERE

The **WHERE** clause allows you to **filter rows based on a specified condition**.

However, the **HAVING** clause allows you to **filter groups of rows according to a specified condition.**

Having example

```
SELECT
        customer_id,
        SUM (amount)
FROM
        payment
GROUP BY
        customer_id
HAVING
        SUM (amount) > 200;
```

payment
* payment_id
customer_id
staff_id
rental_id
amount
payment_date

	customer_id smallint	sum numeric
1	526	208.58
2	148	211.55

Outline

Joining multiple tables (inner join, left join, right join, outer join)

Grouping data (group by, having)

Common table expression & Window function

Common table expression

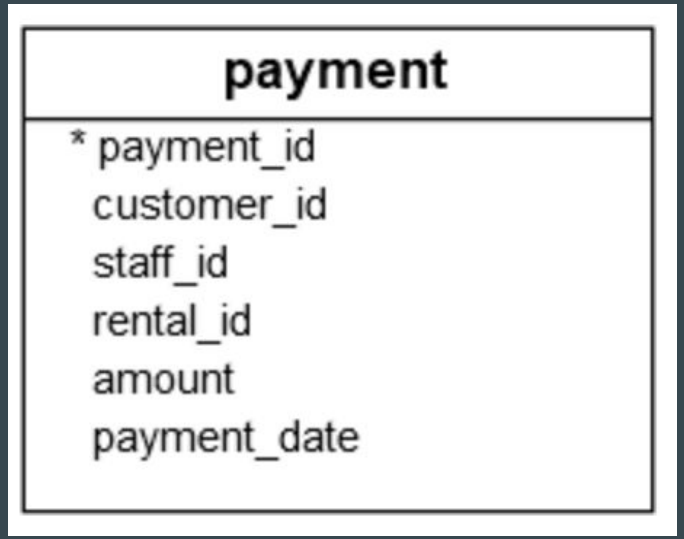
A common table expression is a temporary result set which you can reference within another SQL statement

Basic syntax for creating a CTE

```
WITH cte_name (column_list) AS (
    CTE_query_definition
)
statement;
```

CTE example (Let's make a temporary table with ids only)

```
WITH id_only AS (
SELECT      payment_id,
            Customer_id,
            Staff_id,
            Rental_id
From payment
)
SELECT *
FROM id_only
ORDER BY payment_id;
```



Another example of CTE

The first part defines the name of the CTE which is cte_film.

The second part defines a SELECT statement that populates the expression with rows.

film_id	title	length
4	Affair Prejudice	Long
5	African Egg	Long
6	Agent Truman	Long
9	Alabama Devil	Long
11	Alamo Videotape	Long
12	Alaska Phantom	Long
13	Ali Forever	Long

```
WITH cte_film AS (
    SELECT
        film_id,
        title,
        (CASE
            WHEN length < 30 THEN 'Short'
            WHEN length < 90 THEN 'Medium'
            ELSE 'Long'
        END) length
    FROM
        film
)
SELECT
    film_id,
    title,
    length
FROM
    cte_film
WHERE
    length = 'Long'
ORDER BY
    title;
```

Advantages of using CTE

Improve the readability of complex queries.

Use in conjunction with window functions.

Window function

Let's start with reviewing the aggregate function!

Aggregate function aggregates data from a set of rows into a single row.

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

avg
586.363636363636

Window function

Let's start with reviewing the aggregate function!

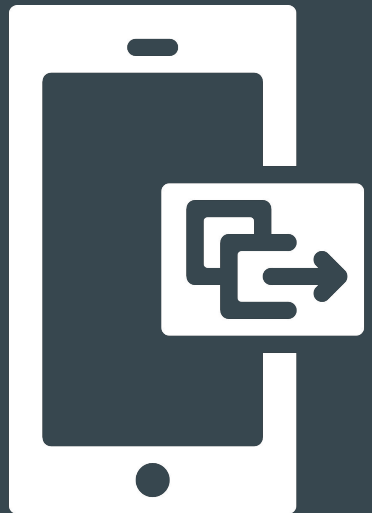
Aggregate function aggregates data from a set of rows into **a single row.**

Window function operates on a set of rows. However, it does not reduce the number of rows returned by the query.

Window function

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.



Window function 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

Window function 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

Window function syntax

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

The **window_function** is the name of the window function.

There are many window function provided by PostgreSQL

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.

Window function syntax

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

The window_function is the name of the window function.

The **PARTITION BY** clause divides rows into multiple groups or partitions to which the window function is applied.

Window function syntax

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

The window_function is the name of the window function.

The PARTITION BY clause divides rows into multiple groups or partitions to which the window function is applied.

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

There are many window function provided by PostgreSQL

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.

Window - RANK () function

```
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

Window - DENSE_RANK () function

```
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

THANK YOU :)