Unit 6: Querying tables.

2022/2023

Contents

6.1. The select clause.
6.2. The where clause.
6.3. The from clause.
6.4. The rename operation.
6.5. Ordering the display of tuples.
6.6. LIMIT and OFFSET.
6.7. Select clause examples.
6.8. SQL Operators.
6.9. Boolean logic.
6.10. String operations.
6.11. NULL values.
6.12. Joined Relations.
6.13. Set operations.
6.14. Aggregate Functions.
6.15. Window Functions.
6.16. Subqueries.
6.17. Modification of the database with subqueries.

Sample database

```
$ sudo su
$ apt get update
$ apt get install postgresql postgresql-client
$ cp samplecompany_postgresql.sql /var/lib/postgresql/
$ su - postgres
$ cd
$ psql
\i samplecompany_postgresql.sql
```

PostgreSQL

```
$ sudo apt update
$ sudo apt install mariadb-server mariadb-client
$ sudo mariadb < samplecompany_mariadb.sql
```

MariaDB

PostgreSQL

MariaDB

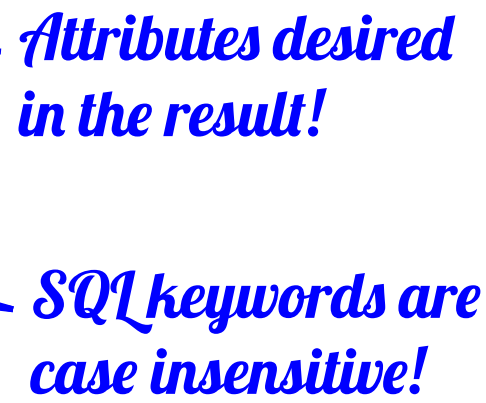
6.1. The select clause.

Structure of a SQL query sentence:

select $A_1, A_2, ..., A_n$
from $r_1, r_2, ..., r_m$
where P;

- A_i represents an attribute
- R_i represents a relation
- P is a predicate.

The **result** of an SQL query is always **relation**.



6.1. The select clause.

Duplicates are allowed in query results, you can avoid them with the keyword **DISTINCT**:

select name, surname
from employees;

name	surname
BRAD	POTTER
SERGIO	SÁNCHEZ
MARTA	ARROYO
REBECA	SALA
JUAN	JIMÉNEZ
MONICA	MARTÍN
BARTOLOME	GOMIS
MARIA	CEREZO
JESUS	GILBERTO
LUIS	TOVAR
FERNANDO	ALONSO
LAURA	ALONSO
XAVIER	GIMENO
ANA	FERNÁNDEZ
ANTONIA	MUÑOZ
ANTONIO	RUIZ
FERNANDA	RUIZ

select surname
from employees;

surname
POTTER
SÁNCHEZ
ARROYO
SALA
JIMÉNEZ
MARTÍN
GOMIS
CEREZO
GILBERTO
TOVAR
ALONSO
ALONSO
GIMENO
FERNÁNDEZ
MUÑOZ
RUIZ
RUIZ

select all surname
from employees;

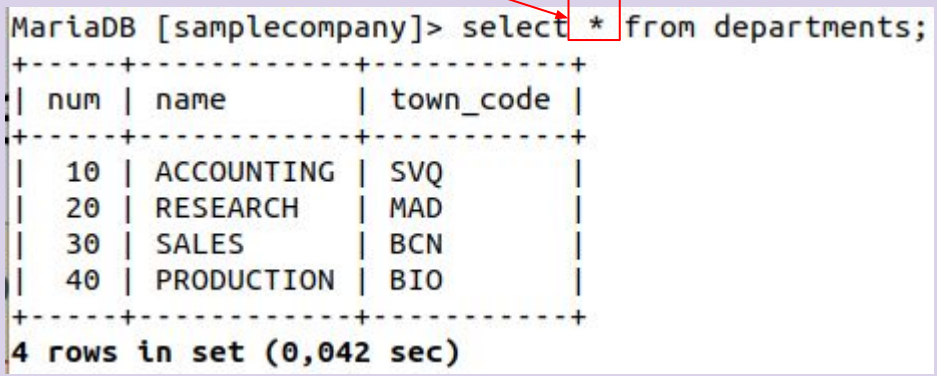
surname
POTTER
SÁNCHEZ
ARROYO
SALA
JIMÉNEZ
MARTÍN
GOMIS
CEREZO
GILBERTO
TOVAR
ALONSO
ALONSO
GIMENO
FERNÁNDEZ
MUÑOZ
RUIZ
RUIZ

select distinct surname
from employees;

surname
POTTER
SÁNCHEZ
ARROYO
SALA
JIMÉNEZ
MARTÍN
GOMIS
CEREZO
GILBERTO
TOVAR
ALONSO
GIMENO
FERNÁNDEZ
MUÑOZ
RUIZ

6.1. The select clause.

Asterisk (*) to denote “all attributes”:



```
MariaDB [samplecompany]> select * from departments;
+-----+------------+-----------+
| num | name       | town_code |
+-----+------------+-----------+
|  10 | ACCOUNTING | SVQ       |
|  20 | RESEARCH   | MAD       |
|  30 | SALES      | BCN       |
|  40 | PRODUCTION | BIO       |
+-----+------------+-----------+
4 rows in set (0,042 sec)
```

6.1. The select clause.

TableName.* (all attributes of a table):

```
MariaDB [samplecompany]> select distinct *
    -> from occupations, departments;
+------+-----------+-----+------------+-----------+
| code | name      | num | name       | town_code |
+------+-----------+-----+------------+-----------+
| ANA  | ANALYST   |  10 | ACCOUNTING | SVQ       |
| ANA  | ANALYST   |  20 | RESEARCH   | MAD       |
| ANA  | ANALYST   |  30 | SALES      | BCN       |
| ANA  | ANALYST   |  40 | PRODUCTION | BIO       |
| EMP  | EMPLOYEE  |  10 | ACCOUNTING | SVQ       |
| EMP  | EMPLOYEE  |  20 | RESEARCH   | MAD       |
| EMP  | EMPLOYEE  |  30 | SALES      | BCN       |
| EMP  | EMPLOYEE  |  40 | PRODUCTION | BIO       |
| MAN  | MANAGER   |  10 | ACCOUNTING | SVQ       |
| MAN  | MANAGER   |  20 | RESEARCH   | MAD       |
| MAN  | MANAGER   |  30 | SALES      | BCN       |
| MAN  | MANAGER   |  40 | PRODUCTION | BIO       |
| PRE  | PRESIDENT |  10 | ACCOUNTING | SVQ       |
| PRE  | PRESIDENT |  20 | RESEARCH   | MAD       |
| PRE  | PRESIDENT |  30 | SALES      | BCN       |
| PRE  | PRESIDENT |  40 | PRODUCTION | BIO       |
| SAL  | SALESMAN  |  10 | ACCOUNTING | SVQ       |
| SAL  | SALESMAN  |  20 | RESEARCH   | MAD       |
| SAL  | SALESMAN  |  30 | SALES      | BCN       |
| SAL  | SALESMAN  |  40 | PRODUCTION | BIO       |
+------+-----------+-----+------------+-----------+
```

```
MariaDB [samplecompany]> select distinct occupations.*
    -> from occupations, departments;
+------+-----------+
| code | name      |
+------+-----------+
| ANA  | ANALYST   |
| EMP  | EMPLOYEE  |
| MAN  | MANAGER   |
| PRE  | PRESIDENT |
| SAL  | SALESMAN  |
+------+-----------+
5 rows in set (0,001 sec)

MariaDB [samplecompany]> select distinct departments.*
    -> from occupations, departments;
+-----+------------+-----------+
| num | name       | town_code |
+-----+------------+-----------+
|  10 | ACCOUNTING | SVQ       |
|  20 | RESEARCH   | MAD       |
|  30 | SALES      | BCN       |
|  40 | PRODUCTION | BIO       |
+-----+------------+-----------+
4 rows in set (0,001 sec)
```

6.1. The select clause.

You can use **literals** like attributes:

```
MariaDB [samplecompany]> select 'DEP.'
    -> from departments;
+------+
| DEP. |
+------+
| DEP. |
| DEP. |
| DEP. |
| DEP. |
+------+
```

```
MariaDB [samplecompany]> select 'DEP.', name
    -> from departments;
+------+------------+
| DEP. | name       |
+------+------------+
| DEP. | ACCOUNTING |
| DEP. | RESEARCH   |
| DEP. | SALES      |
| DEP. | PRODUCTION |
+------+------------+
```

```
MariaDB [samplecompany]> select 'DEP.' as f1, name
    -> from departments;
+------+------------+
| f1   | name       |
+------+------------+
| DEP. | ACCOUNTING |
| DEP. | RESEARCH   |
| DEP. | SALES      |
| DEP. | PRODUCTION |
+------+------------+
```

6.1. The select clause.

You can use **arithmetic operators (+, –, *, and /)**:

```
MariaDB [samplecompany]> select surname, commission, commission + 100
    -> from employees;
+-----------+------------+-----------------+
| surname   | commission | commission + 100 |
+-----------+------------+-----------------+
| POTTER    |       NULL |            NULL |
| SÁNCHEZ   |       NULL |            NULL |
| ARROYO    |        390 |             490 |
| SALA      |        650 |             750 |
| JIMÉNEZ   |       NULL |            NULL |
| MARTÍN    |       1020 |            1120 |
| GOMIS     |       NULL |            NULL |
| CEREZO    |       NULL |            NULL |
| GILBERTO  |       NULL |            NULL |
| TOVAR     |          0 |             100 |
| ALONSO    |       NULL |            NULL |
| ALONSO    |       NULL |            NULL |
| GIMENO    |       NULL |            NULL |
| FERNÁNDEZ |       NULL |            NULL |
| MUÑOZ     |       NULL |            NULL |
| RUIZ      |       NULL |            NULL |
| RUIZ      |       NULL |            NULL |
+-----------+------------+-----------------+
```

```
MariaDB [samplecompany]> select 2 + 3, 8*5;
+-------+-----+
| 2 + 3 | 8*5 |
+-------+-----+
|     5 |  40 |
+-------+-----+
```

```
MariaDB [samplecompany]> select name, surname, commission, salary,
    -> salary / 12 as 'Monthly salary'
    -> from employees;
+-----------+-----------+------------+--------+----------------+
| name      | surname   | commission | salary | Monthly salary |
+-----------+-----------+------------+--------+----------------+
| BRAD      | POTTER    |       NULL |   1040 |        86.6667 |
| SERGIO    | SÁNCHEZ   |       NULL |   1040 |        86.6667 |
| MARTA     | ARROYO    |        390 |   1500 |       125.0000 |
| REBECA    | SALA      |        650 |   1625 |       135.4167 |
| JUAN      | JIMÉNEZ   |       NULL |   2900 |       241.6667 |
| MONICA    | MARTÍN    |       1020 |   1600 |       133.3333 |
| BARTOLOME | GOMIS     |       NULL |   3005 |       250.4167 |
| MARIA     | CEREZO    |       NULL |   2885 |       240.4167 |
| JESUS     | GILBERTO  |       NULL |   3000 |       250.0000 |
| LUIS      | TOVAR     |          0 |   1350 |       112.5000 |
| FERNANDO  | ALONSO    |       NULL |   1430 |       119.1667 |
| LAURA     | ALONSO    |       NULL |   1430 |       119.1667 |
| XAVIER    | GIMENO    |       NULL |   1335 |       111.2500 |
| ANA       | FERNÁNDEZ |       NULL |   3000 |       250.0000 |
| ANTONIA   | MUÑOZ     |       NULL |   1690 |       140.8333 |
| ANTONIO   | RUIZ      |       NULL |   2885 |       240.4167 |
| FERNANDA  | RUIZ      |       NULL |   2885 |       240.4167 |
+-----------+-----------+------------+--------+----------------+
```

6.2. The where clause.

The **where** clause specifies **conditions that the result will satisfy**.

Find all employees with a commission (not NULL):

```
MariaDB [samplecompany]> select name, surname, salary, commission
    -> from employees
    -> where commission IS NOT NULL;
+--------+---------+--------+------------+
| name   | surname | salary | commission |
+--------+---------+--------+------------+
| MARTA  | ARROYO  |   1500 |        390 |
| REBECA | SALA    |   1625 |        650 |
| MONICA | MARTÍN  |   1600 |       1020 |
| LUIS   | TOVAR   |   1350 |          0 |
+--------+---------+--------+------------+
4 rows in set (0,001 sec)
```

6.2. The where clause.

Conditions can be composed using logical operators (**and**, **or**, and **not)**.

Obviously, in conditions, we may use literals or arithmetic operations.

Find all employees with a commission (not NULL) and salary greater than the commission multiplied per 3:

```
MariaDB [samplecompany]> select name, surname, salary, commission
    -> from employees
    -> where commission IS NOT NULL and
    -> salary > commission * 3;
+-------+---------+--------+------------+
| name  | surname | salary | commission |
+-------+---------+--------+------------+
| MARTA | ARROYO  |   1500 |        390 |
| LUIS  | TOVAR   |   1350 |          0 |
+-------+---------+--------+------------+
```

6.3. The from clause.

The **from** clause lists the relations involved in the query.

Corresponds to the **Cartesian product** operation of the relational algebra.

employees X departments generates every possible employee – department pair, with all attributes from both relations:

```
select *
from employees, departments;
```

The Cartesian product is not very useful directly, but useful combined with where-clause condition (**selection operation** in relational algebra).

6.3. The from clause.

```
MariaDB [samplecompany]> select * from employees;
```

num	surname	name	manager	start_date	salary	commission	dept_num	occu_code
1000	POTTER	BRAD	NULL	2004-01-01	1040	NULL	20	NULL
7369	SÁNCHEZ	SERGIO	8001	2010-12-17	1040	NULL	20	EMP
7499	ARROYO	MARTA	7698	2010-02-20	1500	390	30	SAL
7521	SALA	REBECA	7782	2011-02-22	1625	650	30	SAL
7566	JIMÉNEZ	JUAN	1000	2017-04-02	2900	NULL	20	MAN
7654	MARTÍN	MONICA	7698	2017-09-29	1600	1020	30	SAL
7698	GOMIS	BARTOLOME	1000	2017-05-01	3005	NULL	30	MAN
7782	CEREZO	MARIA	1000	2010-06-09	2885	NULL	10	MAN
7788	GILBERTO	JESUS	8000	2010-11-09	3000	NULL	20	NULL
7844	TOVAR	LUIS	7698	2018-09-08	1350	0	30	SAL
7876	ALONSO	FERNANDO	7788	2018-09-23	1430	NULL	20	EMP
7877	ALONSO	LAURA	7788	2020-01-23	1430	NULL	20	EMP
7900	GIMENO	XAVIER	8001	2017-12-03	1335	NULL	30	EMP
7902	FERNÁNDEZ	ANA	8000	2016-12-03	3000	NULL	20	NULL
7934	MUÑOZ	ANTONIA	8001	2016-01-23	1690	NULL	10	EMP
8000	RUIZ	ANTONIO	1000	2017-01-09	2885	NULL	20	MAN
8001	RUIZ	FERNANDA	1000	2018-06-10	2885	NULL	20	MAN

X

```
MariaDB [samplecompany]> select * from departments;
```

num	name	town_code
10	ACCOUNTING	SVQ
20	RESEARCH	MAD
30	SALES	BCN
40	PRODUCTION	BIO

=

```
MariaDB [samplecompany]> select *
    -> from employees, departments;
```

num	surname	name	manager	start_date	salary	commission	dept_num	occu_code	num	name	town_code
1000	POTTER	BRAD	NULL	2004-01-01	1040	NULL	20	NULL	10	ACCOUNTING	SVQ
1000	POTTER	BRAD	NULL	2004-01-01	1040	NULL	20	NULL	20	RESEARCH	MAD
1000	POTTER	BRAD	NULL	2004-01-01	1040	NULL	20	NULL	30	SALES	BCN
1000	POTTER	BRAD	NULL	2004-01-01	1040	NULL	20	NULL	40	PRODUCTION	BIO
7369	SÁNCHEZ	SERGIO	8001	2010-12-17	1040	NULL	20	EMP	10	ACCOUNTING	SVQ
7369	SÁNCHEZ	SERGIO	8001	2010-12-17	1040	NULL	20	EMP	20	RESEARCH	MAD
7369	SÁNCHEZ	SERGIO	8001	2010-12-17	1040	NULL	20	EMP	30	SALES	BCN
7369	SÁNCHEZ	SERGIO	8001	2010-12-17	1040	NULL	20	EMP	40	PRODUCTION	BIO
7499	ARROYO	MARTA	7698	2010-02-20	1500	390	30	SAL	10	ACCOUNTING	SVQ
7499	ARROYO	MARTA	7698	2010-02-20	1500	390	30	SAL	20	RESEARCH	MAD
7499	ARROYO	MARTA	7698	2010-02-20	1500	390	30	SAL	30	SALES	BCN
7499	ARROYO	MARTA	7698	2010-02-20	1500	390	30	SAL	40	PRODUCTION	BIO
7521	SALA	REBECA	7782	2011-02-22	1625	650	30	SAL	10	ACCOUNTING	SVQ
7521	SALA	REBECA	7782	2011-02-22	1625	650	30	SAL	20	RESEARCH	MAD
7521	SALA	REBECA	7782	2011-02-22	1625	650	30	SAL	30	SALES	BCN
7521	SALA	REBECA	7782	2011-02-22	1625	650	30	SAL	40	PRODUCTION	BIO
7566	JIMÉNEZ	JUAN	1000	2017-04-02	2900	NULL	20	MAN	10	ACCOUNTING	SVQ
7566	JIMÉNEZ	JUAN	1000	2017-04-02	2900	NULL	20	MAN	20	RESEARCH	MAD
7566	JIMÉNEZ	JUAN	1000	2017-04-02	2900	NULL	20	MAN	30	SALES	BCN
7566	JIMÉNEZ	JUAN	1000	2017-04-02	2900	NULL	20	MAN	40	PRODUCTION	BIO
7654	MARTÍN	MONICA	7698	2017-09-29	1600	1020	30	SAL	10	ACCOUNTING	SVQ
7654	MARTÍN	MONICA	7698	2017-09-29	1600	1020	30	SAL	20	RESEARCH	MAD
7654	MARTÍN	MONICA	7698	2017-09-29	1600	1020	30	SAL	30	SALES	BCN
7654	MARTÍN	MONICA	7698	2017-09-29	1600	1020	30	SAL	40	PRODUCTION	BIO
7698	GOMIS	BARTOLOME	1000	2017-05-01	3005	NULL	30	MAN	10	ACCOUNTING	SVQ
7698	GOMIS	BARTOLOME	1000	2017-05-01	3005	NULL	30	MAN	20	RESEARCH	MAD
7698	GOMIS	BARTOLOME	1000	2017-05-01	3005	NULL	30	MAN	30	SALES	BCN
7698	GOMIS	BARTOLOME	1000	2017-05-01	3005	NULL	30	MAN	40	PRODUCTION	BIO
7782	CEREZO	MARIA	1000	2010-06-09	2885	NULL	10	MAN	10	ACCOUNTING	SVQ
7782	CEREZO	MARIA	1000	2010-06-09	2885	NULL	10	MAN	20	RESEARCH	MAD
7782	CEREZO	MARIA	1000	2010-06-09	2885	NULL	10	MAN	30	SALES	BCN
7782	CEREZO	MARIA	1000	2010-06-09	2885	NULL	10	MAN	40	PRODUCTION	BIO
7788	GILBERTO	JESUS	8000	2010-11-09	3000	NULL	20	NULL	10	ACCOUNTING	SVQ
7788	GILBERTO	JESUS	8000	2010-11-09	3000	NULL	20	NULL	20	RESEARCH	MAD
7788	GILBERTO	JESUS	8000	2010-11-09	3000	NULL	20	NULL	30	SALES	BCN
7788	GILBERTO	JESUS	8000	2010-11-09	3000	NULL	20	NULL	40	PRODUCTION	BIO
7844	TOVAR	LUIS	7698	2018-09-08	1350	0	30	SAL	10	ACCOUNTING	SVQ
7844	TOVAR	LUIS	7698	2018-09-08	1350	0	30	SAL	20	RESEARCH	MAD
7844	TOVAR	LUIS	7698	2018-09-08	1350	0	30	SAL	30	SALES	BCN
7844	TOVAR	LUIS	7698	2018-09-08	1350	0	30	SAL	40	PRODUCTION	BIO
7876	ALONSO	FERNANDO	7788	2018-09-23	1430	NULL	20	EMP	10	ACCOUNTING	SVQ
7876	ALONSO	FERNANDO	7788	2018-09-23	1430	NULL	20	EMP	20	RESEARCH	MAD
7876	ALONSO	FERNANDO	7788	2018-09-23	1430	NULL	20	EMP	30	SALES	BCN
7876	ALONSO	FERNANDO	7788	2018-09-23	1430	NULL	20	EMP	40	PRODUCTION	BIO
7877	ALONSO	LAURA	7788	2020-01-23	1430	NULL	20	EMP	10	ACCOUNTING	SVQ
7877	ALONSO	LAURA	7788	2020-01-23	1430	NULL	20	EMP	20	RESEARCH	MAD
7877	ALONSO	LAURA	7788	2020-01-23	1430	NULL	20	EMP	30	SALES	BCN
7877	ALONSO	LAURA	7788	2020-01-23	1430	NULL	20	EMP	40	PRODUCTION	BIO
7900	GIMENO	XAVIER	8001	2017-12-03	1335	NULL	30	EMP	10	ACCOUNTING	SVQ
7900	GIMENO	XAVIER	8001	2017-12-03	1335	NULL	30	EMP	20	RESEARCH	MAD
7900	GIMENO	XAVIER	8001	2017-12-03	1335	NULL	30	EMP	30	SALES	BCN
7900	GIMENO	XAVIER	8001	2017-12-03	1335	NULL	30	EMP	40	PRODUCTION	BIO
7902	FERNÁNDEZ	ANA	8000	2016-12-03	3000	NULL	20	NULL	10	ACCOUNTING	SVQ
7902	FERNÁNDEZ	ANA	8000	2016-12-03	3000	NULL	20	NULL	20	RESEARCH	MAD
7902	FERNÁNDEZ	ANA	8000	2016-12-03	3000	NULL	20	NULL	30	SALES	BCN
7902	FERNÁNDEZ	ANA	8000	2016-12-03	3000	NULL	20	NULL	40	PRODUCTION	BIO
7934	MUÑOZ	ANTONIA	8001	2016-01-23	1690	NULL	10	EMP	10	ACCOUNTING	SVQ
7934	MUÑOZ	ANTONIA	8001	2016-01-23	1690	NULL	10	EMP	20	RESEARCH	MAD
7934	MUÑOZ	ANTONIA	8001	2016-01-23	1690	NULL	10	EMP	30	SALES	BCN
7934	MUÑOZ	ANTONIA	8001	2016-01-23	1690	NULL	10	EMP	40	PRODUCTION	BIO
8000	RUIZ	ANTONIO	1000	2017-01-09	2885	NULL	20	MAN	10	ACCOUNTING	SVQ
8000	RUIZ	ANTONIO	1000	2017-01-09	2885	NULL	20	MAN	20	RESEARCH	MAD
8000	RUIZ	ANTONIO	1000	2017-01-09	2885	NULL	20	MAN	30	SALES	BCN
8000	RUIZ	ANTONIO	1000	2017-01-09	2885	NULL	20	MAN	40	PRODUCTION	BIO
8001	RUIZ	FERNANDA	1000	2018-06-10	2885	NULL	20	MAN	10	ACCOUNTING	SVQ
8001	RUIZ	FERNANDA	1000	2018-06-10	2885	NULL	20	MAN	20	RESEARCH	MAD
8001	RUIZ	FERNANDA	1000	2018-06-10	2885	NULL	20	MAN	30	SALES	BCN
8001	RUIZ	FERNANDA	1000	2018-06-10	2885	NULL	20	MAN	40	PRODUCTION	BIO

6.4. The rename operation.

The SQL allows renaming relations and attributes using the as clause:

old-name **as** new-name

Find the name, surname and department name of all the employees:

```
select E.name, E.surname, D.name as dept_name
from employees as E, departments as D
where E.dept_num = D.num;
```

Keyword **as** is optional and may be omitted:

employees **as** E ≡ employees E

6.5. Ordering the display of tuples.

List in alphabetic order the names of all instructors:

```
select distinct name
from employees
order by name;
```

```
MariaDB [samplecompany]> select distinct name
    -> from employees
    -> order by name;
+----------+
| name     |
+----------+
| ANA      |
| ANTONIA  |
| ANTONIO  |
| BARTOLOME |
| BRAD     |
| FERNANDA |
| FERNANDO |
| JESUS    |
| JUAN     |
| LAURA    |
| LUIS     |
| MARIA    |
| MARTA    |
| MONICA   |
| REBECA   |
| SERGIO   |
| XAVIER   |
+----------+
```

We may specify **desc** for descending order or **asc** for ascending order (default), for each attribute:

```
select distinct name
from employees
order by name desc;
```

```
MariaDB [samplecompany]> select distinct name
    -> from employees
    -> order by name desc;
+----------+
| name     |
+----------+
| XAVIER   |
| SERGIO   |
| REBECA   |
| MONICA   |
| MARTA    |
| MARIA    |
| LUIS     |
| LAURA    |
| JUAN     |
| JESUS    |
| FERNANDO |
| FERNANDA |
| BRAD     |
| BARTOLOME |
| ANTONIO  |
| ANTONIA  |
| ANA      |
+----------+
```

Can sort on multiple attributes:

Example: **order by** surname **desc**, name **asc**;

6.6. LIMIT and OFFSET.

PostgreSQL

```
SELECT select_list
    FROM table_expression
    [ ORDER BY ... ]
    [ LIMIT { number | ALL } ] [ OFFSET number ]
```

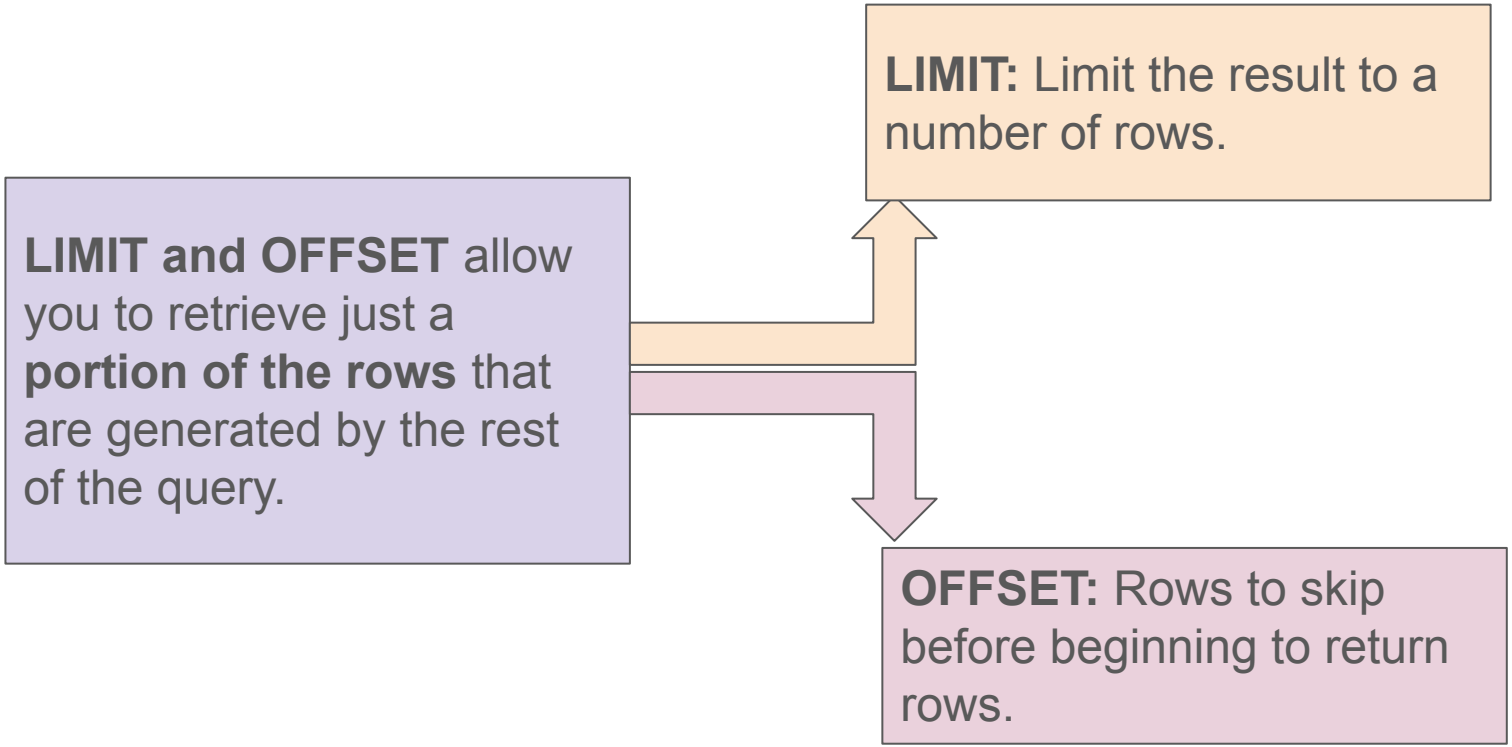
MariaDB

```
LIMIT offset, row_count
LIMIT row_count OFFSET offset
```

```
select name, surname
from employees
order by surname, name
limit 4;
```

```
   name    | surname
-----------+---------
 FERNANDO  | ALONSO
 LAURA     | ALONSO
 MARTA     | ARROYO
 MARIA     | CEREZO
(4 rows)
```

6.6. LIMIT and OFFSET.



6.6. LIMIT and OFFSET.

```
SELECT row_number() OVER (
    ORDER BY start_date
  ),
  name,
  surname
FROM employees
ORDER BY start_date;
```

row_number	name	surname
1	BRAD	POTTER
2	MARTA	ARROYO
3	MARIA	CEREZO
4	JESUS	GILBERTO
5	SERGIO	SÁNCHEZ
6	REBECA	SALA
7	ANTONIA	MUÑOZ
8	ANA	FERNÁNDEZ
9	ANTONIO	RUIZ
10	JUAN	JIMÉNEZ
11	BARTOLOME	GOMIS
12	MONICA	MARTÍN
13	XAVIER	GIMENO
14	FERNANDA	RUIZ
15	LUIS	TOVAR
16	FERNANDO	ALONSO
17	LAURA	ALONSO

(17 rows)

6.6. LIMIT and OFFSET.

```
SELECT row_number() OVER (
    ORDER BY start_date
  ),
  name,
  surname
FROM employees
ORDER BY start_date
LIMIT 5;
```

```
SELECT row_number() OVER (
    ORDER BY start_date
  ),
  name,
  surname
FROM employees
ORDER BY start_date
LIMIT 5 OFFSET 0;
```

```
 row_number |  name   | surname
------------+---------+----------
          1 | BRAD    | POTTER
          2 | MARTA   | ARROYO
          3 | MARIA   | CEREZO
          4 | JESUS   | GILBERTO
          5 | SERGIO  | SÁNCHEZ
(5 rows)
```

6.6. LIMIT and OFFSET.

```
SELECT row_number() OVER (
    ORDER BY start_date
  ),
  name,
  surname
FROM employees
ORDER BY start_date
LIMIT 5 OFFSET 5;
```

```
 row_number |  name    |  surname
------------+----------+-----------
          6 | REBECA   | SALA
          7 | ANTONIA  | MUÑOZ
          8 | ANA      | FERNÁNDEZ
          9 | ANTONIO  | RUIZ
         10 | JUAN     | JIMÉNEZ
(5 rows)
```

6.6. LIMIT and OFFSET.

```
SELECT row_number() OVER (
    ORDER BY start_date
  ),
  name,
  surname
FROM employees
ORDER BY start_date
LIMIT 5 OFFSET 10;
```

```
 row_number |    name     | surname
------------+-------------+---------
         11 | BARTOLOME   | GOMIS
         12 | MONICA      | MARTÍN
         13 | XAVIER      | GIMENO
         14 | FERNANDA    | RUIZ
         15 | LUIS        | TOVAR
(5 rows)
```

6.6. LIMIT and OFFSET.

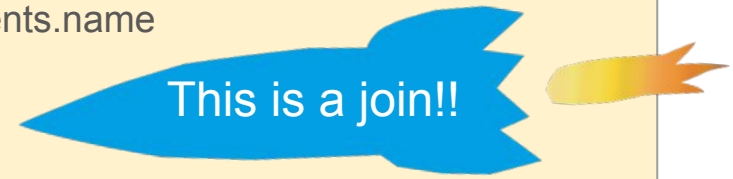
```
SELECT row_number() OVER (
    ORDER BY start_date
  ),
  name,
  surname
FROM employees
ORDER BY start_date
LIMIT 5 OFFSET 15;
```

```
 row_number |    name    | surname
------------+------------+---------
         16 | FERNANDO   | ALONSO
         17 | LAURA      | ALONSO
(2 rows)
```

6.7. Select clause examples.

Find the names, surname and department name of all the employees:

```
select employees.name, employees.surname, departments.name
from employees, departments
where employees.dept_num = departments.num
order by departments.name, employees.surname;
```



Find the names, surname and department name of all the employees who earn more than $2500:

```
select E.name, E.surname, E.salary, D.name
from employees as E, departments as D
where E.dept_num = D.num and
      E.salary > 2500
order by E.salary desc;
```

```
MariaDB [samplecompany]> select E.name, E.surname, E.salary, D.name
    -> from employees as E, departments as D
    -> where E.dept_num = D.num and
    ->   E.salary > 2500
    -> order by E.salary desc;
+-----------+------------+--------+------------+
| name      | surname    | salary | name       |
+-----------+------------+--------+------------+
| BARTOLOME | GOMIS      |   3005 | SALES      |
| JESUS     | GILBERTO   |   3000 | RESEARCH   |
| ANA       | FERNÁNDEZ  |   3000 | RESEARCH   |
| JUAN      | JIMÉNEZ    |   2900 | RESEARCH   |
| MARIA     | CEREZO     |   2885 | ACCOUNTING |
| ANTONIO   | RUIZ       |   2885 | RESEARCH   |
| FERNANDA  | RUIZ       |   2885 | RESEARCH   |
+-----------+------------+--------+------------+
7 rows in set (0,001 sec)
```

6.7. Select clause examples.

INNER JOIN

```
select employees.name, employees.surname, departments.name
from employees, departments
where employees.dept_num = departments.num
order by departments.name, employees.surname;
```

```
select employees.name, employees.surname, departments.name
from employees join departments
on employees.dept_num = departments.num
order by departments.name, employees.surname;
```

```
select employees.name, employees.surname, departments.name
from employees inner join departments
on employees.dept_num = departments.num
order by departments.name, employees.surname;
```

Are the same!!

6.8. SQL Operators.

Source: W3schools

SQL Arithmetic Operators

Operator	Description	Example
+	Add	Try it
-	Subtract	Try it
*	Multiply	Try it
/	Divide	Try it
%	Modulo	Try it

6.8. SQL Operators.

Source: W3 schools

SQL Bitwise Operators

Operator	Description
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR

6.8. SQL Operators.

SQL Comparison Operators

Source:

Operator	Description	Example
=	Equal to	Try it
>	Greater than	Try it
<	Less than	Try it
>=	Greater than or equal to	Try it
<=	Less than or equal to	Try it
<>	Not equal to	Try it

!=

SQL Logical Operators

Operator	Description	Example
ALL	TRUE if all of the subquery values meet the condition	Try it
AND	TRUE if all the conditions separated by AND is TRUE	Try it
ANY	TRUE if any of the subquery values meet the condition	Try it
BETWEEN	TRUE if the operand is within the range of comparisons	Try it
EXISTS	TRUE if the subquery returns one or more records	Try it
IN	TRUE if the operand is equal to one of a list of expressions	Try it
LIKE	TRUE if the operand matches a pattern	Try it
NOT	Displays a record if the condition(s) is NOT TRUE	Try it
OR	TRUE if any of the conditions separated by OR is TRUE	Try it
SOME	TRUE if any of the subquery values meet the condition	Try it

Source:

6.8. SQL Operators.

SQL Logical Operators

ALL

6.15. Subqueries (+ 6.12. Set operations). **all**

Find the names, salary and department number of all employees whose salary is greater than that of all employees in department number 10:

```
select name, surname, salary, dept_num
from employees
where salary > all (select salary
                    from employees
                    where dept_num = 10);
```

```
  name    |  surname  | salary | dept_num
----------+-----------+--------+----------
JUAN      | JIMÉNEZ   |   2900 |       20
BARTOLOME | GOMIS     |   3005 |       30
JESUS     | GILBERTO  |   3000 |       20
ANA       | FERNÁNDEZ |   3000 |       20
(4 rows)
```

6.15. Subqueries (+ 6.12. Set operations). **all**

All the employees but the ones with the highest salary...

W3schools:
"The ALL operator:
- returns a boolean value as a result
- returns **TRUE if ALL of the subquery values meet the condition**
- is used with SELECT, WHERE and HAVING statements

ALL means that the condition will be true only if the operation is true for all values in the range".

```
samplecompany=# select num,
  salary
from employees
where salary >= all (
    select distinct salary
    from employees
  )
order by salary;
 num  | salary
------+--------
 7698 |   3005
(1 row)
```

6.15. Subqueries (+ 6.12. Set operations). **all**

(5 < **all** {0, 5, 6}) = false

(5 < **all** {6, 10}) = true

(5 = **all** {4, 5}) = false

(5 ≠ **all** {4, 6}) = true (since 5 ≠ 4 and 5 ≠ 6)

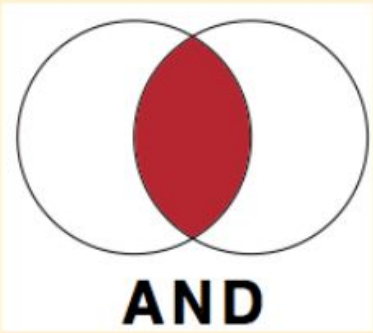
(≠ **all**) ≡ **not in**
However, (= **all**) ≠ **in**

Source of the picture: Abraham Silberschatz, Henry F. Korth and S. Sudarshan. Database System Concepts.

6.8. SQL Operators.

SQL Logical Operators

AND



a	b	a AND b
true	true	true
true	false	false
false	true	false
false	false	false

```
samplecompany=# select num,
  salary,
  commission
from employees
where commission is not null
  and salary > 1500;
 num  | salary | commission
------+--------+------------
 7521 |   1625 |        650
 7654 |   1600 |       1020
(2 rows)
```

PostgreSQL

6.8. SQL Operators.

SQL Logical Operators

ANY

6.16. Subqueries (+ 6.12. Set operations).

any

All the employees but the ones with the lowest salary...

W3schools:
"The ANY operator:

- returns a boolean value as a result
- returns **TRUE if ANY of the subquery values meet the condition**

ANY means that the condition will be true if the operation is true for any of the values in the range".

```
samplecompany=# select num,
  salary
from employees
where salary > any (
    select distinct salary
    from employees
  )
order by salary;
 num  | salary
------+--------
 7900 |   1335
 7844 |   1350
 7877 |   1430
 7876 |   1430
 7499 |   1500
 7654 |   1600
 7521 |   1625
 7934 |   1690
 8001 |   2885
 8000 |   2885
 7782 |   2885
 7566 |   2900
 7902 |   3000
 7788 |   3000
 7698 |   3005
(15 rows)
```

6.8. SQL Operators.

SQL Logical Operators

BETWEEN / NOT BETWEEN

W3schools:

- "The BETWEEN operator selects values within a given range.
- The values can be numbers, text, or dates.
- The BETWEEN operator is inclusive: begin and end values are included".

```
samplecompany=# select num,
  surname
from employees
where surname between 'A' and 'D'
order by surname;
 num  | surname
------+---------
 7876 | ALONSO
 7877 | ALONSO
 7499 | ARROYO
 7782 | CEREZO
(4 rows)
```

PostgreSQL

```
samplecompany=# select num,
  salary
from employees
where salary between 1000 and 1500
order by salary;
 num  | salary
------+--------
 1000 |   1040
 7369 |   1040
 7900 |   1335
 7844 |   1350
 7876 |   1430
 7877 |   1430
 7499 |   1500
(7 rows)
```

PostgreSQL

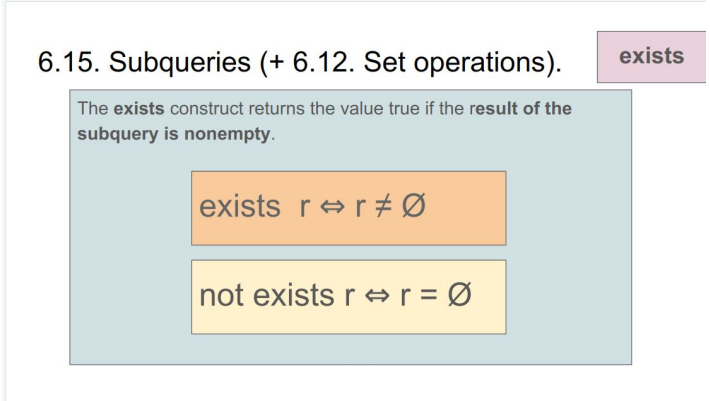
```
samplecompany=# select num,
  start_date
from employees
where start_date between '1/1/2010' and '31/12/2010'
order by start_date;
 num  | start_date
------+------------
 7499 | 2010-02-20
 7782 | 2010-06-09
 7788 | 2010-11-09
 7369 | 2010-12-17
(4 rows)
```

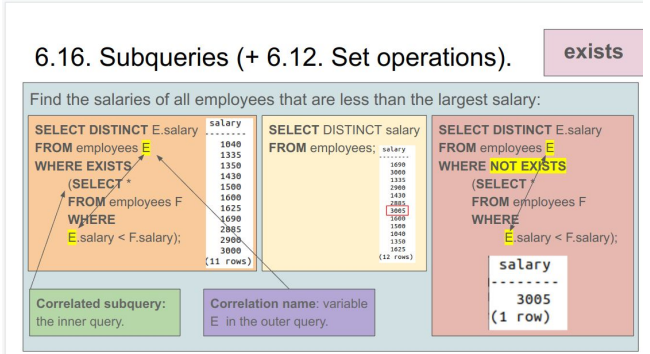
PostgreSQL

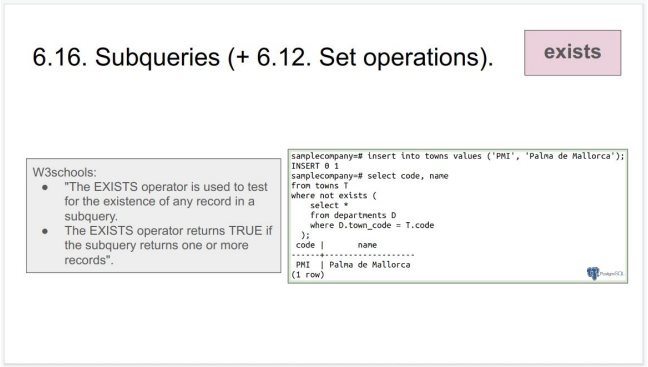
6.8. SQL Operators.

SQL Logical Operators

EXISTS / NOT EXISTS







6.8. SQL Operators.

SQL Logical Operators

IN / NOT IN

W3schools:

- The IN operator allows you to specify multiple values in a WHERE clause.
- The IN operator is a shorthand for multiple OR conditions.

```
samplecompany=# select num,
  name,
  surname
from employees
where num in (
    select distinct manager
    from employees
  );
 num  |   name    | surname
------+-----------+----------
 1000 | BRAD      | POTTER
 7698 | BARTOLOME | GOMIS
 7782 | MARIA     | CEREZO
 7788 | JESUS     | GILBERTO
 8000 | ANTONIO   | RUIZ
 8001 | FERNANDA  | RUIZ
(6 rows)
```

PostgreSQL

```
samplecompany=# select num,
  name
from departments
where num in (10, 20, 30);
 num |    name
-----+------------
  10 | ACCOUNTING
  20 | RESEARCH
  30 | SALES
(3 rows)
```

PostgreSQL

6.8. SQL Operators.

SQL Logical Operators

LIKE / NOT LIKE

W3schools:
"The LIKE operator is used in a WHERE clause to search for a specified pattern in a column. There are two wildcards often used in conjunction with the LIKE operator:

- The percent sign (%) represents zero, one, or multiple characters
- The underscore sign (_) represents one, single character".

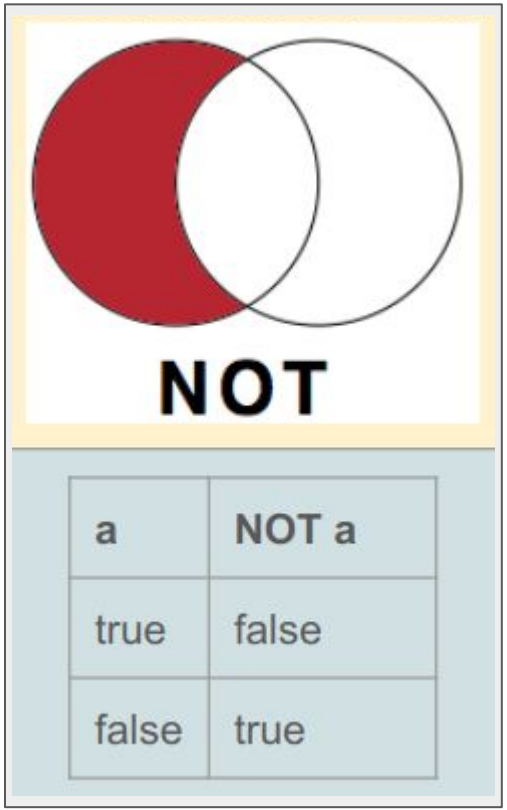
```
samplecompany=# select name, surname
from employees
where name like '%ad%';
 name | surname
------+---------
(0 rows)

samplecompany=# select name, surname
from employees
where name like '%AD%';
 name | surname
------+---------
 BRAD | POTTER
(1 row)
```

PostgreSQL

6.8. SQL Operators.

SQL Logical Operators

NOT




```
samplecompany=# select num,
  dept_num
from employees
where dept_num = 10;
 num  | dept_num
------+----------
 7782 |       10
 7934 |       10
(2 rows)
```

PostgreSQL

```
samplecompany=# select num,
  dept_num
from employees
where not (dept_num = 10);
 num  | dept_num
------+----------
 1000 |       20
 7369 |       20
 7499 |       30
 7521 |       30
 7566 |       20
 7654 |       30
 7698 |       30
 7788 |       20
 7844 |       30
 7876 |       20
 7877 |       20
 7900 |       30
 7902 |       20
 8000 |       20
 8001 |       20
(15 rows)
```

PostgreSQL

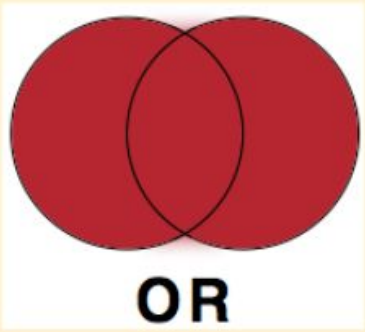
```
samplecompany=# select num,
  dept_num
from employees
where dept_num <> 10;
 num  | dept_num
------+----------
 1000 |       20
 7369 |       20
 7499 |       30
 7521 |       30
 7566 |       20
 7654 |       30
 7698 |       30
 7788 |       20
 7844 |       30
 7876 |       20
 7877 |       20
 7900 |       30
 7902 |       20
 8000 |       20
 8001 |       20
(15 rows)
```

PostgreSQL

6.8. SQL Operators.

SQL Logical Operators

OR



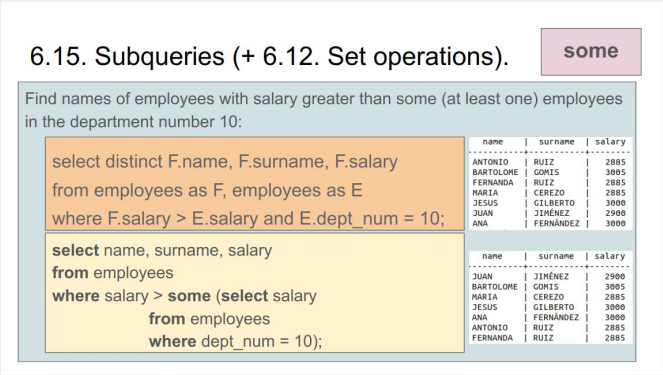
a	b	a OR b
true	true	true
true	false	true
false	true	true
false	false	false

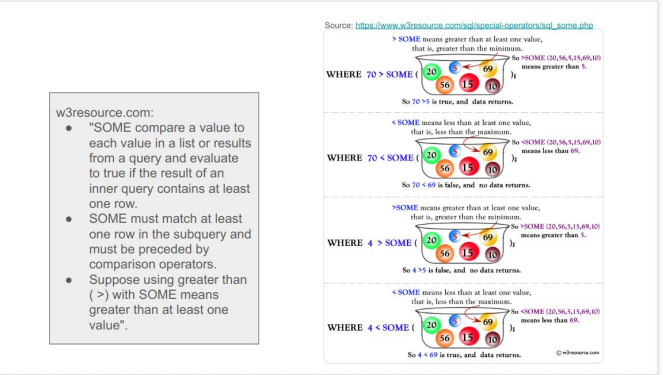
```
samplecompany=# select num,
  salary
from employees
where salary = 1600
  or salary = 1690;
 num  | salary
------+--------
 7654 |   1600
 7934 |   1690
(2 rows)
```

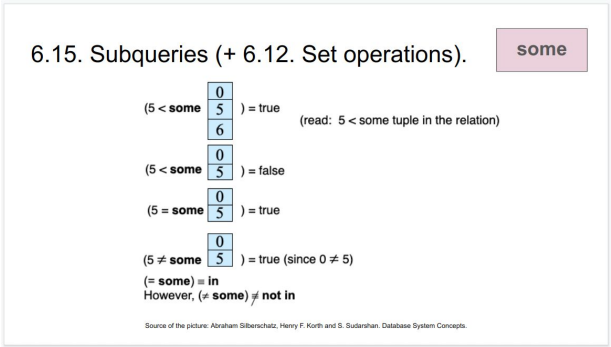
6.8. SQL Operators.

SQL Logical Operators

SOME







6.8. SQL Operators.

Operator precedence

Operator/Element	Associativity	Description
.	left	table/column name separator
::	left	PostgreSQL-style typecast
[ ]	left	array element selection
+ -	right	unary plus, unary minus
^	left	exponentiation
* / %	left	multiplication, division, modulo
+ -	left	addition, subtraction
(any other operator)	left	all other native and user-defined operators
BETWEEN IN LIKE ILIKE SIMILAR		range containment, set membership, string matching
< > = <= >= <>		comparison operators
IS ISNULL NOTNULL		IS TRUE, IS FALSE, IS NULL, IS DISTINCT FROM, etc.
NOT	right	logical negation
AND	left	logical conjunction
OR	left	logical disjunction

PostgreSQL

```
INTERVAL
BINARY, COLLATE
!
- (unary minus), ~ (unary bit inversion)
^
*, /, DIV, %, MOD
-, +
<<, >>
&
|
= (comparison), <=>, >=, >, <=, <, <>, !=, IS, LIKE, REGEXP, IN, MEMBER OF
BETWEEN, CASE, WHEN, THEN, ELSE
NOT
AND, &&
XOR
OR, ||
= (assignment), :=
```

MariaDB

6.9. Boolean logic.

A **condition** will result in **true or false**.

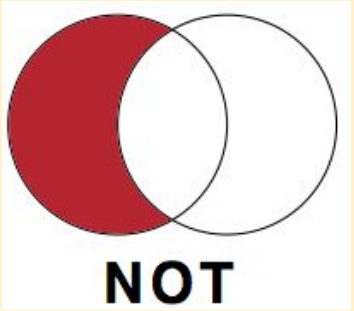
The **where clause** indicates the **rows that must be part of the result** relation = rows where the condition is true!

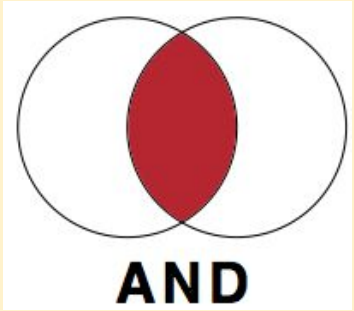
```
MariaDB [samplecompany]> select  *
    -> from departments, towns;
+-----+------------+-----------+------+-----------+
| num | name       | town_code | code | name      |
+-----+------------+-----------+------+-----------+
|  10 | ACCOUNTING | SVQ       | BCN  | BARCELONA |
|  20 | RESEARCH   | MAD       | BCN  | BARCELONA |
|  30 | SALES      | BCN       | BCN  | BARCELONA |
|  40 | PRODUCTION | BIO       | BCN  | BARCELONA |
|  10 | ACCOUNTING | SVQ       | BIO  | BILBAO    |
|  20 | RESEARCH   | MAD       | BIO  | BILBAO    |
|  30 | SALES      | BCN       | BIO  | BILBAO    |
|  40 | PRODUCTION | BIO       | BIO  | BILBAO    |
|  10 | ACCOUNTING | SVQ       | MAD  | MADRID    |
|  20 | RESEARCH   | MAD       | MAD  | MADRID    |
|  30 | SALES      | BCN       | MAD  | MADRID    |
|  40 | PRODUCTION | BIO       | MAD  | MADRID    |
|  10 | ACCOUNTING | SVQ       | SVQ  | SEVILLA   |
|  20 | RESEARCH   | MAD       | SVQ  | SEVILLA   |
|  30 | SALES      | BCN       | SVQ  | SEVILLA   |
|  40 | PRODUCTION | BIO       | SVQ  | SEVILLA   |
+-----+------------+-----------+------+-----------+
```

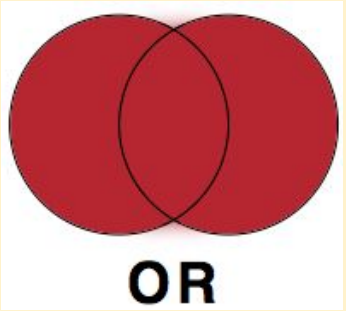
```
MariaDB [samplecompany]> select  *
    -> from departments, towns
    -> where departments.town_code = towns.code;
+-----+------------+-----------+------+-----------+
| num | name       | town_code | code | name      |
+-----+------------+-----------+------+-----------+
|  30 | SALES      | BCN       | BCN  | BARCELONA |
|  40 | PRODUCTION | BIO       | BIO  | BILBAO    |
|  20 | RESEARCH   | MAD       | MAD  | MADRID    |
|  10 | ACCOUNTING | SVQ       | SVQ  | SEVILLA   |
+-----+------------+-----------+------+-----------+
```

6.9. Boolean logic.

BOOLEAN OPERATORS:







TRUTH TABLES:

a	NOT a
true	false
false	true

a	b	a AND b
true	true	true
true	false	false
false	true	false
false	false	false

a	b	a OR b
true	true	true
true	false	true
false	true	true
false	false	false

6.9. Boolean logic.

Use of parentheses: To make sure the computer evaluates the conditions in the expected order.

a	b	NOT a OR b	NOT (a OR b)
true	true	true	false
true	false	false	false
false	true	true	false
false	false	true	true

a	b	c	a OR b AND c	a OR (b AND c)	(a OR b) AND c
false	false	false	false	false	false
false	false	true	false	false	false
false	true	false	false	false	false
false	true	true	true	true	true
true	false	false	true	true	false
true	false	true	true	true	true
true	true	false	true	true	false
true	true	true	true	true	true

6.9. Boolean logic.

De Morgan's laws:

not (A or B) = not A and not B

not (A and B) = not A or not B

Google Classroom

P06_logical_expressions

P06_where_logical_expressions

6.10. String operations.

SQL includes a **string-matching operator** for comparisons on character strings. The operator **like** uses patterns that are described using two special characters:

Percent (%). The % character matches any substring.

Underscore (_). The _ character matches any character.

6.10. String operations.

Find the names of all instructors whose name includes the substring “ad”:

```
select name, surname
from employees
where name like '%ad%';
```

```
samplecompany=# select name, surname
from employees
where name like '%ad%';
 name | surname
------+---------
(0 rows)

samplecompany=# select name, surname
from employees
where name like '%AD%';
 name | surname
------+---------
 BRAD | POTTER
(1 row)
```

PostgreSQL

```
MariaDB [samplecompany]> select name, surname
    -> from employees
    -> where name like '%ad%';
+------+----------+
| name | surname |
+------+----------+
| BRAD | POTTER  |
+------+----------+
1 row in set (0,000 sec)
```

MariaDB

Match the string “10%”:

Backslash (\) is the escape character!

```
select name, surname
from employees
where name like '%\%%';
```

```
samplecompany=# insert into employees
(num, name, surname) values
(1, 'asdads%adasdasd', 'Gual');
INSERT 0 1
samplecompany=# select name, surname
from employees
where name like '%\%%';
       name        | surname
-----------------+---------
 asdads%adasdasd | Gual
(1 row)
```

PostgreSQL

```
MariaDB [samplecompany]> select name, surname
    -> from employees
    -> where name like '%\%%';
+-----------------+---------+
| name            | surname |
+-----------------+---------+
| asdads%adasdasd | Gual    |
+-----------------+---------+
1 row in set (0,000 sec)
```

MariaDB

6.10. String operations.

Backslash Escape Sequence	Interpretation
\b	backspace
\f	form feed
\n	newline
\r	carriage return
\t	tab
\o, \oo, \ooo (o = 0–7)	octal byte value
\xh, \xhh (h = 0–9, A–F)	hexadecimal byte value
\uxxxx, \Uxxxxxxxx (x = 0–9, A–F)	16 or 32-bit hexadecimal Unicode character value

```
samplecompany=# insert into departments values
samplecompany-# (99, 'Sergi''s department', NULL);
INSERT 0 1
samplecompany=# insert into departments values
(100, 'Sergi\'s department 2', NULL);
samplecompany'# ';
samplecompany(# );
ERROR:  syntax error at or near "s"
LINE 2: (100, 'Sergi\'s department 2', NULL);
```

Any other character following a backslash is taken literally. Thus, to include a backslash character, write two backslashes (\\). Also, a single quote can be included in an escape string by writing \', in addition to the normal way of ''.

PostgreSQL

6.10. String operations.

Patterns use to be **case-sensitive**.

```
samplecompany=# select surname from employees where surname like 'GO%';
 surname
---------
 GOMIS
(1 row)

samplecompany=# select surname from employees where surname like 'go%';
 surname
---------
(0 rows)
```

PostgreSQL

YES!

NO!

```
MariaDB [samplecompany]> select surname from employees where surname like 'GO%';
+---------+
| surname |
+---------+
| GOMIS   |
+---------+
1 row in set (0,000 sec)

MariaDB [samplecompany]> select surname from employees where surname like 'go%';
+---------+
| surname |
+---------+
| GOMIS   |
+---------+
1 row in set (0,000 sec)
```

MariaDB

6.10. String operations.

SQL supports a variety of string operations such as:

Converting from **upper** to **lower** case (and vice versa):

```
select lower(name)
from employees;
```

Concatenation using CONCAT(string1, string2, ...):

```
select concat(name, ' ', surname)
from employees;
```

Getting **string length**, **extracting substrings**, etc.

```
samplecompany=# select concat_ws (
    '',
    concat(
      upper (left (E.name, 1)),
      lower (substring(E.name, 2))
    ),
    concat(
      upper (left (E.surname, 1)),
      lower (substring (E.surname, 2))
    )
  ) as FullName
from employees E;
    fullname
----------------
 BradPotter
 SergioSánchez
 MartaArroyo
 RebecaSala
 JuanJiménez
 MonicaMartin
 BartolomeGomis
 MariaCerezo
 JesusGilberto
 LuisTovar
 FernandoAlonso
 LauraAlonso
 XavierGimeno
 AnaFernández
 AntoniaMuñoz
 AntonioRuiz
 FernandaRuiz
(17 rows)
```

initcap

```
samplecompany=# select concat_ws (' ', initcap(E.name), initcap(E.surname)) as FullName
from employees E;
    fullname
----------------
 Brad Potter
 Sergio Sánchez
 Marta Arroyo
 Rebeca Sala
 Juan Jiménez
 Monica Martin
 Bartolome Gomis
 Maria Cerezo
 Jesus Gilberto
 Luis Tovar
 Fernando Alonso
 Laura Alonso
 Xavier Gimeno
 Ana Fernández
 Antonia Muñoz
 Antonio Ruiz
 Fernanda Ruiz
(17 rows)
```

PostgreSQL

6.10. String operations.

SQL supports a variety of **date functions** to transform them to strings, such as:

It also works in MariaDB...

```
samplecompany=#  select to_char(start_date, 'Month'),
to_char(start_date, 'YYYY')
from employees;
  to_char  | to_char
-----------+---------
 January   | 2004
 December  | 2010
 February  | 2010
 February  | 2011
 April     | 2017
 September | 2017
 May       | 2017
 June      | 2010
 November  | 2010
 September | 2018
 September | 2018
 January   | 2020
 December  | 2017
 December  | 2016
 January   | 2016
 January   | 2017
 June      | 2018
```

PostgreSQL

```
MariaDB [samplecompany]> select monthname(start_date),
    -> year(start_date)
    -> from employees;
+-----------------------+------------------+
| monthname(start_date) | year(start_date) |
+-----------------------+------------------+
| January               |             2004 |
| December              |             2010 |
| February              |             2010 |
| February              |             2011 |
| April                 |             2017 |
| September             |             2017 |
| May                   |             2017 |
| June                  |             2010 |
| November              |             2010 |
| September             |             2018 |
| September             |             2018 |
| January               |             2020 |
| December              |             2017 |
| December              |             2016 |
| January               |             2016 |
| January               |             2017 |
| June                  |             2018 |
+-----------------------+------------------+
```

MariaDB

6.11. NULL values.

It is possible for tuples to have a null value for some of their attributes.

NULL may be:

- unknown value
- value does not exist
- value not applicable

The result of any arithmetic expression involving null is null.

Example:

```
5 + NULL is  NULL
```

The predicate **is null** can be used to check for null values. Example:

```
select name, surname
from employees
where commission is null;
```

6.11. NULL values.

Evaluating a condition, we may have 3 results: **true** **false** **unknown**

Any comparison with a null value returns unknown. Examples:

5 < null

null <> null

null = null

is unknown → **true**

```
samplecompany=# select null = null is unknown;
 ?column?
----------
 t
(1 row)
```

PostgreSQL

```
MariaDB [samplecompany]> select null = null is unknown;
+-------------------------+
| null = null is unknown |
+-------------------------+
|                       1 |
+-------------------------+
1 row in set (0,000 sec)
```

MariaDB

Three-valued logic using the value unknown:

NOT: (not unknown) = unknown

OR: (unknown or true) = true,
(unknown or false) = unknown
(unknown or unknown) = unknown

AND: (true and unknown) = unknown,
(false and unknown) = false,
(unknown and unknown) = unknown

"P **is unknown**" evaluates to true if predicate P evaluates to unknown

The result of where clause predicate is treated as false if it evaluates to unknown.

6.11. NULL values.

COALESCE

w3resource.com:

- "Return the first non-null value in a list:
- SELECT COALESCE(NULL, NULL, NULL, 'W3Schools.com', NULL, 'Example.com');".

```
samplecompany=# select num,
  salary,
  coalesce(commission, 0)
from employees;
 num  | salary | coalesce
------+--------+----------
 1000 |   1040 |        0
 7369 |   1040 |        0
 7499 |   1500 |      390
 7521 |   1625 |      650
 7566 |   2900 |        0
 7654 |   1600 |     1020
 7698 |   3005 |        0
 7782 |   2885 |        0
 7788 |   3000 |        0
 7844 |   1350 |        0
 7876 |   1430 |        0
 7877 |   1430 |        0
 7900 |   1335 |        0
 7902 |   3000 |        0
 7934 |   1690 |        0
 8000 |   2885 |        0
 8001 |   2885 |        0
(17 rows)
```

PostgreSQL

Google Classroom

P06_queries01

6.12. Joined Relations.

Join operations take **two relations** and return as a **result another relation**.

ANSI-standard SQL specifies five **types of JOIN**:

INNER | LEFT OUTER | FULL OUTER | RIGHT OUTER | CROSS

6.12. Joined Relations.

A join operation is a Cartesian product which requires that tuples in the two relations match under some conditions.

```
select E.name, E.surname, E.dept_num, D.num, D.name as deptname
from employees as E, departments as D;
```

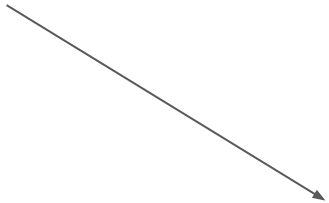
```
select E.name, E.surname, E.dept_num, D.num, D.name as deptname
from EMPLOYEES as E, DEPARTMENTS as D
where E.dept_num = D.num;
```

name	surname	dept_num	num	deptname
BRAD	POTTER	20	10	ACCOUNTING
BRAD	POTTER	20	20	RESEARCH
BRAD	POTTER	20	30	SALES
BRAD	POTTER	20	40	PRODUCTION
SERGIO	SÁNCHEZ	20	10	ACCOUNTING
SERGIO	SÁNCHEZ	20	20	RESEARCH
SERGIO	SÁNCHEZ	20	30	SALES
SERGIO	SÁNCHEZ	20	40	PRODUCTION
MARTA	ARROYO	30	10	ACCOUNTING
MARTA	ARROYO	30	20	RESEARCH
MARTA	ARROYO	30	30	SALES
MARTA	ARROYO	30	40	PRODUCTION
REBECA	SALA	30	10	ACCOUNTING
REBECA	SALA	30	20	RESEARCH
REBECA	SALA	30	30	SALES
REBECA	SALA	30	40	PRODUCTION
JUAN	JIMÉNEZ	20	10	ACCOUNTING
JUAN	JIMÉNEZ	20	20	RESEARCH
JUAN	JIMÉNEZ	20	30	SALES
JUAN	JIMÉNEZ	20	40	PRODUCTION
MONICA	MARTÍN	30	10	ACCOUNTING
MONICA	MARTÍN	30	20	RESEARCH
MONICA	MARTÍN	30	30	SALES
MONICA	MARTÍN	30	40	PRODUCTION
BARTOLOME	GOMIS	30	10	ACCOUNTING
BARTOLOME	GOMIS	30	20	RESEARCH
BARTOLOME	GOMIS	30	30	SALES
BARTOLOME	GOMIS	30	40	PRODUCTION
MARIA	CEREZO	10	10	ACCOUNTING
MARIA	CEREZO	10	20	RESEARCH
MARIA	CEREZO	10	30	SALES
MARIA	CEREZO	10	40	PRODUCTION
JESUS	GILBERTO	20	10	ACCOUNTING
JESUS	GILBERTO	20	20	RESEARCH
JESUS	GILBERTO	20	30	SALES
JESUS	GILBERTO	20	40	PRODUCTION
LUIS	TOVAR	30	10	ACCOUNTING
LUIS	TOVAR	30	20	RESEARCH
LUIS	TOVAR	30	30	SALES
LUIS	TOVAR	30	40	PRODUCTION
FERNANDO	ALONSO	20	10	ACCOUNTING
FERNANDO	ALONSO	20	20	RESEARCH
FERNANDO	ALONSO	20	30	SALES
FERNANDO	ALONSO	20	40	PRODUCTION
LAURA	ALONSO	20	10	ACCOUNTING
LAURA	ALONSO	20	20	RESEARCH
LAURA	ALONSO	20	30	SALES
LAURA	ALONSO	20	40	PRODUCTION
XAVIER	GIMENO	30	10	ACCOUNTING
XAVIER	GIMENO	30	20	RESEARCH
XAVIER	GIMENO	30	30	SALES
XAVIER	GIMENO	30	40	PRODUCTION
ANA	FERNÁNDEZ	20	10	ACCOUNTING
ANA	FERNÁNDEZ	20	20	RESEARCH
ANA	FERNÁNDEZ	20	30	SALES
ANA	FERNÁNDEZ	20	40	PRODUCTION
ANTONIA	MUÑOZ	10	10	ACCOUNTING
ANTONIA	MUÑOZ	10	20	RESEARCH
ANTONIA	MUÑOZ	10	30	SALES
ANTONIA	MUÑOZ	10	40	PRODUCTION
ANTONIO	RUIZ	20	10	ACCOUNTING
ANTONIO	RUIZ	20	20	RESEARCH
ANTONIO	RUIZ	20	30	SALES
ANTONIO	RUIZ	20	40	PRODUCTION
FERNANDA	RUIZ	20	10	ACCOUNTING
FERNANDA	RUIZ	20	20	RESEARCH
FERNANDA	RUIZ	20	30	SALES
FERNANDA	RUIZ	20	40	PRODUCTION

6.12. Joined Relations.

A table (base table, view, or joined table) can JOIN to itself in a self-join.

```
select E.name as employee_name, E.surname as employee_surname,
M.name as manager_name, M.surname as manager_surname
from employees as E, employees as M
where E.manager = M.num;
```



employee_name	employee_surname	manager_name	manager_surname
SERGIO	SÁNCHEZ	FERNANDA	RUIZ
MARTA	ARROYO	BARTOLOME	GOMIS
REBECA	SALA	MARIA	CEREZO
JUAN	JIMÉNEZ	BRAD	POTTER
MONICA	MARTÍN	BARTOLOME	GOMIS
BARTOLOME	GOMIS	BRAD	POTTER
MARIA	CEREZO	BRAD	POTTER
JESUS	GILBERTO	ANTONIO	RUIZ
LUIS	TOVAR	BARTOLOME	GOMIS
FERNANDO	ALONSO	JESUS	GILBERTO
LAURA	ALONSO	JESUS	GILBERTO
XAVIER	GIMENO	FERNANDA	RUIZ
ANA	FERNÁNDEZ	ANTONIO	RUIZ
ANTONIA	MUÑOZ	FERNANDA	RUIZ
ANTONIO	RUIZ	BRAD	POTTER
FERNANDA	RUIZ	BRAD	POTTER

(16 rows)

6.12. Joined Relations.

We will see the **types of joins** in **Unit 7**.

ANSI-standard SQL specifies five **types of JOIN**:

INNER | LEFT OUTER | FULL OUTER | RIGHT OUTER | CROSS

6.13. Set operations.

UNION

Employees who start working in 2004 or 2010:

```
(select name, surname, start_date
from employees
where year(start_date) = 2004)
UNION
(select name, surname, start_date
from employees
where year(start_date) = 2010);
```

```
+--------+----------+------------+
| name   | surname  | start_date |
+--------+----------+------------+
| BRAD   | POTTER   | 2004-01-01 |
| SERGIO | SÁNCHEZ  | 2010-12-17 |
| MARTA  | ARROYO   | 2010-02-20 |
| MARIA  | CEREZO   | 2010-06-09 |
| JESUS  | GILBERTO | 2010-11-09 |
+--------+----------+------------+
5 rows in set (0,000 sec)
```

MariaDB

Is there another way to do the same query?

6.13. Set operations.

UNION

It also works in MariaDB...

```
(select name, surname, start_date
from employees
where cast(to_char(start_date, 'YYYY') as integer) = 2004)
UNION
(select name, surname, start_date
from employees
where cast(to_char(start_date, 'YYYY') as integer) = 2010);
```

```
 name   | surname   | start_date
--------+-----------+------------
 BRAD   | POTTER    | 2004-01-01
 JESUS  | GILBERTO  | 2010-11-09
 MARIA  | CEREZO    | 2010-06-09
 MARTA  | ARROYO    | 2010-02-20
 SERGIO | SÁNCHEZ   | 2010-12-17
(5 rows)
```

PostgreSQL

5.6. Data types in SQL.

- In MySQL/MariaDB **cast between data types** is automatic. In Postgresql is not automatic…
- **CAST function**:
  - CAST (expression AS target_type);

Data type: money

```
bookings=# SELECT * FROM facilities WHERE cust_cost = 0;
ERROR:  operator does not exist: money = integer
LINE 1: SELECT * FROM facilities WHERE cust_cost = 0;
                                       ^
HINT:  No operator matches the given name and argument types. You might need to add explicit type casts.
bookings=# SELECT * FROM facilities WHERE CAST(cust_cost AS numeric) = 0;
 id |       name       | cust_cost | guest_cost | purchase_cost | maintenance_cost
----+------------------+-----------+------------+---------------+------------------
  2 | Badminton Court  |     $0.00 |    $155.00 |     $4,000.00 |           $50.00
  3 | Table Tennis     |     $0.00 |      $5.00 |       $320.00 |           $10.00
  7 | Snooker Table    |     $0.00 |      $5.00 |       $450.00 |           $15.00
  8 | Pool Table       |     $0.00 |      $5.00 |       $400.00 |           $15.00
(4 rows)
```

6.13. Set operations.

INTERSECT

Employees who start working in 2010 and who are managers of other employees:

```
(select num from employees where year(start_date) = 2010)
INTERSECT
(select distinct manager from employees);
```

```
+------+
| num  |
+------+
| 7782 |
| 7788 |
+------+
2 rows in set (0,000 sec)
```

MariaDB

```
select num
from employees
where year(start_date) = 2010
  and num in (
    select distinct manager
    from employees
  );
```

MariaDB

Is there another way to do the same query?

6.13. Set operations.

EXCEPT

Employees who start working in 2010 and who are not managers of other employees:

```
(select num from employees where YEAR(start_date) = 2010)
EXCEPT
(select distinct manager from employees);
```

```
+------+
| num  |
+------+
| 7369 |
| 7782 |
+------+
2 rows in set (0,000 sec)
```

MariaDB

Operation in relational algebra?

```
select num
from employees
where year(start_date) = 2010
  and num not in (
    select distinct coalesce(manager, 0)
    from employees
  );
```

MariaDB

6.13. Set operations.

```
MariaDB [samplecompany]> select distinct salary
    -> from employees;
+--------+
| salary |
+--------+
|   1040 |
|   1500 |
|   1625 |
|   2900 |
|   1600 |
|   3005 |
|   2885 |
|   3000 |
|   1350 |
|   1430 |
|   1335 |
|   1690 |
+--------+
12 rows in set (0,000 sec)
```

EXCEPT

```
MariaDB [samplecompany]> select distinct E.salary
    -> from employees E, employees F
    -> where E.salary < F.salary;
+--------+
| salary |
+--------+
|   1040 |
|   1350 |
|   1430 |
|   1335 |
|   1500 |
|   1600 |
|   1625 |
|   2885 |
|   1690 |
|   2900 |
|   3000 |
+--------+
11 rows in set (0,000 sec)
```

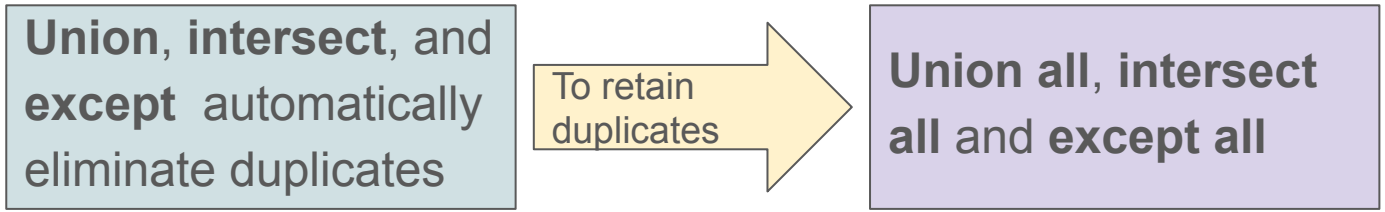
MAX SALARY

```
MariaDB [samplecompany]> select max(salary)
    -> from employees;
+-------------+
| max(salary) |
+-------------+
|        3005 |
+-------------+
1 row in set (0,000 sec)
```

We will see this later...

```
MariaDB [samplecompany]> select distinct salary
    -> from employees
    -> except
    -> select distinct E.salary
    -> from employees E, employees F
    -> where E.salary < F.salary;
+--------+
| salary |
+--------+
|   3005 |
+--------+
1 row in set (0,001 sec)
```

6.13. Set operations.



6.14. Aggregate Functions.

These functions operate on the multiset of values of a column of a relation, and return a value.

avg:
average value.

min:
minimum value.

max:
maximum value.

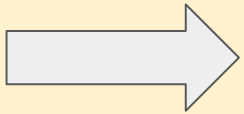
sum:
sum of values.

count:
number of values.

6.14. Aggregate Functions.

Find the average salary of employees in department with num 10:

```
select avg (salary)
from employees
where dept_num = 10;
```

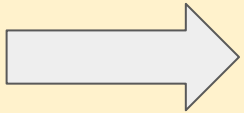


```
+---------------+
| avg (salary) |
+---------------+
|    2287.5000 |
+---------------+
1 row in set (0,005 sec)
```

6.14. Aggregate Functions.

Find the total number of departments with employees working in it:

```
select count(distinct dept_num)
from employees;
```



```
+--------------------------+
| count(distinct dept_num) |
+--------------------------+
|                        3 |
+--------------------------+
1 row in set (0,000 sec)
```

```
select count(dept_num)
from employees;
```

```
+-----------------+
| count(dept_num) |
+-----------------+
|              17 |
+-----------------+
```

6.14. Aggregate Functions.

Find the number of tuples in the employees relation:

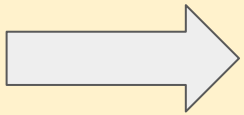
```
select count(*)
from employees;
```

```
+----------+
| count(*) |
+----------+
|       17 |
+----------+
```

6.14. Aggregate Functions.

Find the minimum and maximum salary of employees in department with num 10:

```
select min(salary), max(salary)
from employees
where dept_num = 10;
```

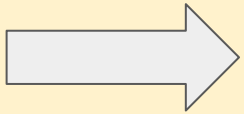


```
+-------------+-------------+
| min(salary) | max(salary) |
+-------------+-------------+
|        1690 |        2885 |
+-------------+-------------+
1 row in set (0,000 sec)
```

6.14. Aggregate Functions.

Find the minimum and maximum salary of employees (all employees):

```
select min(salary), max(salary)
from employees;
```



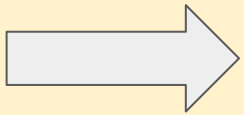
```
+-------------+-------------+
| min(salary) | max(salary) |
+-------------+-------------+
|        1040 |        3005 |
+-------------+-------------+
1 row in set (0,000 sec)
```

6.14. Aggregate Functions.

GROUP BY

Find the average salary of employees in each department:

```
select D.name, avg(E.salary)
from employees E, departments D
where E.dept_num = D.num
group by D.name;
```



```
+------------+---------------+
| name       | avg(E.salary) |
+------------+---------------+
| ACCOUNTING |     2287.5000 |
| RESEARCH   |     2178.8889 |
| SALES      |     1735.8333 |
+------------+---------------+
3 rows in set (0,001 sec)
```

6.14. Aggregate Functions.

GROUP BY

Find the average salary of employees in each department:

```
MariaDB [samplecompany]> select *
    -> from departments;
+-----+------------+-----------+
| num | name       | town_code |
+-----+------------+-----------+
|  10 | ACCOUNTING | SVQ       |
|  20 | RESEARCH   | MAD       |
|  30 | SALES      | BCN       |
|  40 | PRODUCTION | BIO       |
+-----+------------+-----------+
4 rows in set (0,000 sec)
```

```
MariaDB [samplecompany]> select E.num, E.name,
    -> E.surname, E.salary, D.name
    -> from employees E, departments D
    -> where E.dept_num = D.num
    -> order by D.name;
+------+----------+-----------+--------+------------+
| num  | name     | surname   | salary | name       |
+------+----------+-----------+--------+------------+
| 7782 | MARIA    | CEREZO    |   2885 | ACCOUNTING |
| 7934 | ANTONIA  | MUÑOZ     |   1690 | ACCOUNTING |
| 1000 | BRAD     | POTTER    |   1040 | RESEARCH   |
| 7369 | SERGIO   | SÁNCHEZ   |   1040 | RESEARCH   |
| 7566 | JUAN     | JIMÉNEZ   |   2900 | RESEARCH   |
| 7788 | JESUS    | GILBERTO  |   3000 | RESEARCH   |
| 7876 | FERNANDO | ALONSO    |   1430 | RESEARCH   |
| 7877 | LAURA    | ALONSO    |   1430 | RESEARCH   |
| 7902 | ANA      | FERNÁNDEZ |   3000 | RESEARCH   |
| 8000 | ANTONIO  | RUIZ      |   2885 | RESEARCH   |
| 8001 | FERNANDA | RUIZ      |   2885 | RESEARCH   |
| 7499 | MARTA    | ARROYO    |   1500 | SALES      |
| 7521 | REBECA   | SALA      |   1625 | SALES      |
| 7654 | MONICA   | MARTÍN    |   1600 | SALES      |
| 7698 | BARTOLOME | GOMIS    |   3005 | SALES      |
| 7844 | LUIS     | TOVAR     |   1350 | SALES      |
| 7900 | XAVIER   | GIMENO    |   1335 | SALES      |
+------+----------+-----------+--------+------------+
17 rows in set (0,001 sec)
```

```
MariaDB [samplecompany]> select D.name, avg(E.salary)
    -> from employees E, departments D
    -> where E.dept_num = D.num
    -> group by D.name;
+------------+---------------+
| name       | avg(E.salary) |
+------------+---------------+
| ACCOUNTING |     2287.5000 |
| RESEARCH   |     2178.8889 |
| SALES      |     1735.8333 |
+------------+---------------+
3 rows in set (0,001 sec)
```

6.14. Aggregate Functions.

GROUP BY

Attributes in **select** clause outside of aggregate functions must appear in **group by** list:

Upside aggregate function → Inside Group By

```
/* wrong query */
select D.name, avg(E.salary)
from employees E, departments D
where E.dept_num = D.num;
```

```
ERROR:  column "d.name" must appear in the GROUP BY clause
or be used in an aggregate function
LINE 2: select D.name, avg(E.salary)
```

PostgreSQL

```
+------------+---------------+
| name       | avg(E.salary) |
+------------+---------------+
| ACCOUNTING |     2035.2941 |
+------------+---------------+
1 row in set (0,001 sec)
```

MariaDB

```
/* right query */
select D.name, avg(E.salary)
from employees E, departments D
where E.dept_num = D.num
group by D.name;
```

```
+------------+---------------+
| name       | avg(E.salary) |
+------------+---------------+
| ACCOUNTING |     2287.5000 |
| RESEARCH   |     2178.8889 |
| SALES      |     1735.8333 |
+------------+---------------+
3 rows in set (0,001 sec)
```

MariaDB

```
    name    |          avg
------------+-----------------------
 ACCOUNTING | 2287.5000000000000000
 RESEARCH   | 2178.8888888888888889
 SALES      | 1735.8333333333333333
(3 rows)
```

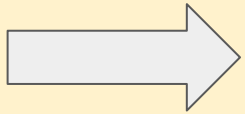
PostgreSQL

6.14. Aggregate Functions.

HAVING

Find the names and average salaries of all departments whose average salary is greater than 2100:

```
select D.name, avg(E.salary)
from employees E, departments D
where E.dept_num = D.num
group by D.name
having avg(salary) > 2100;
```

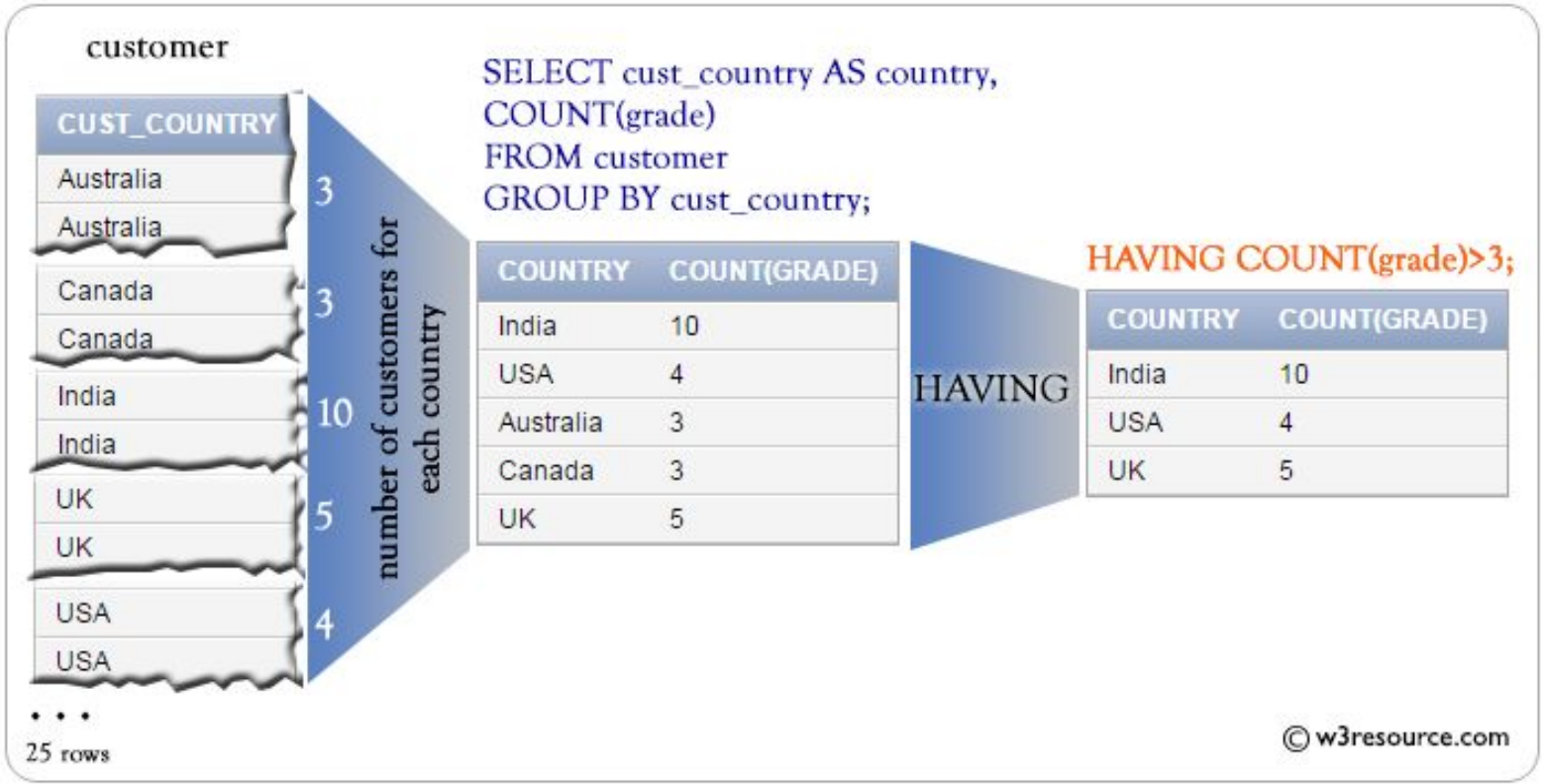


```
+------------+---------------+
| name       | avg(E.salary) |
+------------+---------------+
| ACCOUNTING |     2287.5000 |
| RESEARCH   |     2178.8889 |
+------------+---------------+
```

Predicates in the **having** clause are applied after the formation of groups whereas predicates in the where clause are applied before forming groups.

6.14. Aggregate Functions.

HAVING



6.14. Aggregate Functions.

NULL values

Sum of all salaries:
select sum(salary)
from EMPLOYEES;

This statement ignores null salaries.

The result is NULL if there are only NULL values.

All aggregate functions ignore tuples with null values on the aggregated attributes. Count have a slightly different behavior.

6.14. Aggregate Functions.

```sql
create table r (
    id int primary key,
    value int
);

insert into r (id) values
(1), (2), (3), (4), (5);

select avg(value)
from r;

select min(value)
from r;

select max(value)
from r;

select sum(value)
from r;

select count(value)
from r;
```

```
+------------+
| avg(value) |
+------------+
|       NULL |
+------------+
1 row in set (0,000 sec)
```

```
+------------+
| min(value) |
+------------+
|       NULL |
+------------+
1 row in set (0,000 sec)
```

```
+------------+
| max(value) |
+------------+
|       NULL |
+------------+
1 row in set (0,000 sec)
```

```
+------------+
| sum(value) |
+------------+
|       NULL |
+------------+
1 row in set (0,000 sec)
```

```
+--------------+
| count(value) |
+--------------+
|            0 |
+--------------+
1 row in set (0,000 sec)
```

NULL values

6.15. Window Functions.

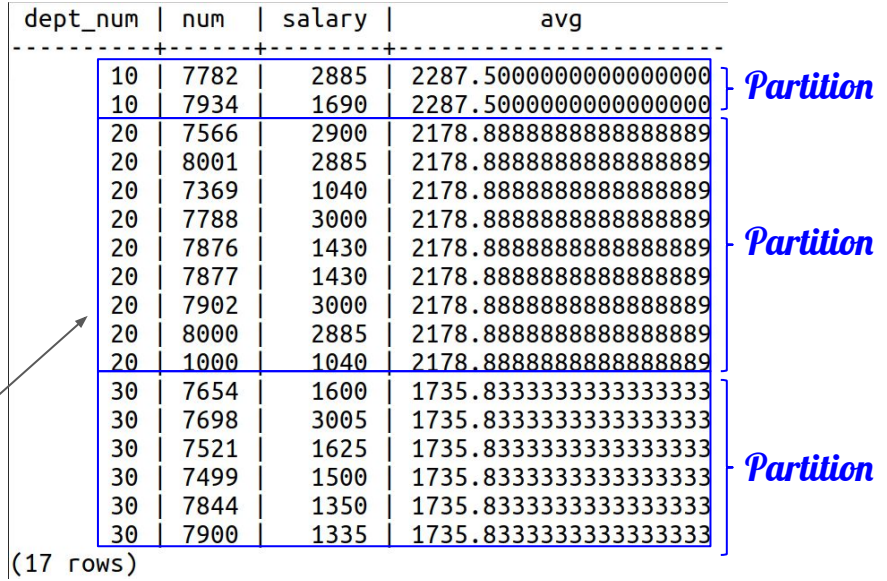
MariaDB starting with 10.2.

This is **comparable to** the type of calculation that can be done with an **aggregate function** (but it's not the same!)

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

PostgreSQL

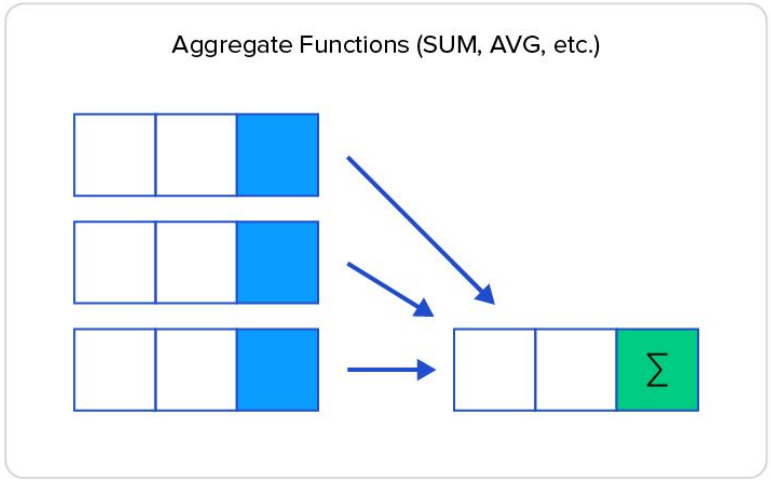
```
select dept_num,
  num,
  salary,
  avg(salary) OVER (PARTITION BY dept_num)
from employees;
```

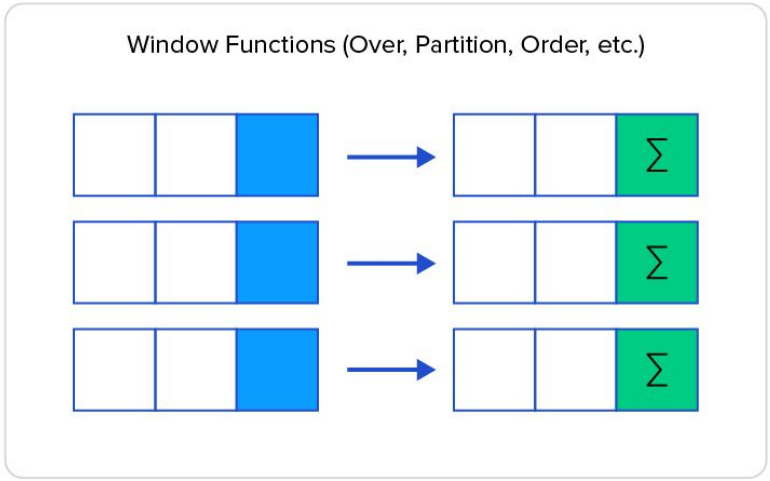


dept_num	num	salary	avg	
10	7782	2885	2287.5000000000000000	Partition
10	7934	1690	2287.5000000000000000	
20	7566	2900	2178.8888888888888889	Partition
20	8001	2885	2178.8888888888888889	
20	7369	1040	2178.8888888888888889	
20	7788	3000	2178.8888888888888889	
20	7876	1430	2178.8888888888888889	
20	7877	1430	2178.8888888888888889	
20	7902	3000	2178.8888888888888889	
20	8000	2885	2178.8888888888888889	
20	1000	1040	2178.8888888888888889	
30	7654	1600	1735.8333333333333333	Partition
30	7698	3005	1735.8333333333333333	
30	7521	1625	1735.8333333333333333	
30	7499	1500	1735.8333333333333333	
30	7844	1350	1735.8333333333333333	
30	7900	1335	1735.8333333333333333	

(17 rows)

6.15. Window Functions.





Source: https://www.toptal.com/sql/intro-to-sql-windows-functions

6.15. Window Functions.

It provides a way to perform advanced calculations without having to use subqueries or join operations.

PROBLEM: I want calculations ant the rows of the table.

```
select E1.dept_num,
  E1.num,
  E1.salary,
  T.avg_salary
from employees E1,
  (
    select E.dept_num,
      avg(E.salary) avg_salary
    from employees E
    group by E.dept_num
  ) as T,
  departments D
where D.num = E1.dept_num
  and E1.dept_num = T.dept_num;
```

```
 dept_num | num  | salary |      avg_salary
----------+------+--------+----------------------
       10 | 7782 |   2885 | 2287.5000000000000000
       10 | 7934 |   1690 | 2287.5000000000000000
       20 | 7566 |   2900 | 2178.8888888888888889
       20 | 8001 |   2885 | 2178.8888888888888889
       20 | 7369 |   1040 | 2178.8888888888888889
       20 | 7788 |   3000 | 2178.8888888888888889
       20 | 7876 |   1430 | 2178.8888888888888889
       20 | 7877 |   1430 | 2178.8888888888888889
       20 | 7902 |   3000 | 2178.8888888888888889
       20 | 8000 |   2885 | 2178.8888888888888889
       20 | 1000 |   1040 | 2178.8888888888888889
       30 | 7654 |   1600 | 1735.8333333333333333
       30 | 7698 |   3005 | 1735.8333333333333333
       30 | 7521 |   1625 | 1735.8333333333333333
       30 | 7499 |   1500 | 1735.8333333333333333
       30 | 7844 |   1350 | 1735.8333333333333333
       30 | 7900 |   1335 | 1735.8333333333333333
(17 rows)
```

6.15. Window Functions.

OVER() is the entire rowset.

```
SELECT COUNT(*) OVER() As
NumEmployees,
    name,
    surname,
    start_date
FROM employees
ORDER BY start_date;
```

```
SELECT (
      SELECT COUNT(*)
      FROM employees
   ) as NumEmployees,
   name,
   surname,
   start_date
FROM employees
ORDER BY start_date;
```

Without window function.

```
 numemployees |   name    |  surname   | start_date
--------------+-----------+------------+------------
           17 | BRAD      | POTTER     | 2004-01-01
           17 | MARTA     | ARROYO     | 2010-02-20
           17 | MARIA     | CEREZO     | 2010-06-09
           17 | JESUS     | GILBERTO   | 2010-11-09
           17 | SERGIO    | SÁNCHEZ    | 2010-12-17
           17 | REBECA    | SALA       | 2011-02-22
           17 | ANTONIA   | MUÑOZ      | 2016-01-23
           17 | ANA       | FERNÁNDEZ  | 2016-12-03
           17 | ANTONIO   | RUIZ       | 2017-01-09
           17 | JUAN      | JIMÉNEZ    | 2017-04-02
           17 | BARTOLOME | GOMIS      | 2017-05-01
           17 | MONICA    | MARTÍN     | 2017-09-29
           17 | XAVIER    | GIMENO     | 2017-12-03
           17 | FERNANDA  | RUIZ       | 2018-06-10
           17 | LUIS      | TOVAR      | 2018-09-08
           17 | FERNANDO  | ALONSO     | 2018-09-23
           17 | LAURA     | ALONSO     | 2020-01-23
(17 rows)
```

6.15. Window Functions.

Number of employees who started the **same month of the same year**.

```
SELECT COUNT(*) OVER (
    PARTITION BY to_char(start_date, 'Month'),
    to_char(start_date, 'YYYY')
  ) As NumPerMonth,
  concat(
    to_char(start_date, 'Month'),
    ' ',
    to_char(start_date, 'YYYY')
  ) As TheMonth,
  name,
  surname
FROM employees
ORDER BY start_date;
```

PostgreSQL

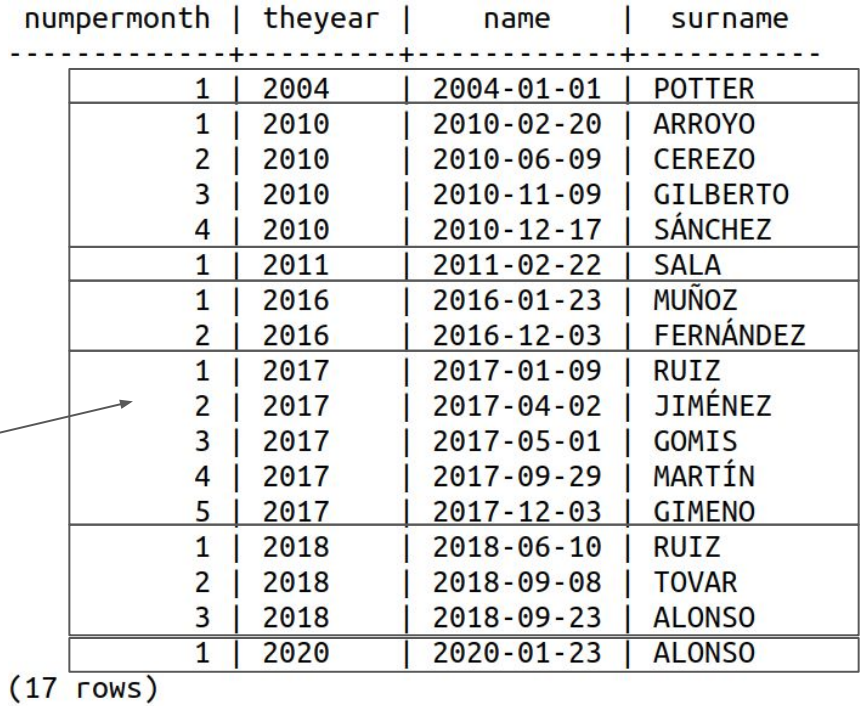
numpermonth	themonth	name	surname
1	January 2004	BRAD	POTTER
1	February 2010	MARTA	ARROYO
1	June 2010	MARIA	CEREZO
1	November 2010	JESUS	GILBERTO
1	December 2010	SERGIO	SÁNCHEZ
1	February 2011	REBECA	SALA
1	January 2016	ANTONIA	MUÑOZ
1	December 2016	ANA	FERNÁNDEZ
1	January 2017	ANTONIO	RUIZ
1	April 2017	JUAN	JIMÉNEZ
1	May 2017	BARTOLOME	GOMIS
1	September 2017	MONICA	MARTÍN
1	December 2017	XAVIER	GIMENO
1	June 2018	FERNANDA	RUIZ
2	September 2018	LUIS	TOVAR
2	September 2018	FERNANDO	ALONSO
1	January 2020	LAURA	ALONSO

(17 rows)

6.15. Window Functions.

Number of employees who started the **same year and the order when they started**.

```
SELECT COUNT(*) OVER (
       PARTITION BY to_char(start_date, 'YYYY')
       ORDER BY start_date
    ) As NumPerMonth,
    to_char(start_date, 'YYYY') As TheYear,
    start_date name,
    surname
FROM employees
ORDER BY start_date;
```



numpermonth	theyear	name	surname
1	2004	2004-01-01	POTTER
1	2010	2010-02-20	ARROYO
2	2010	2010-06-09	CEREZO
3	2010	2010-11-09	GILBERTO
4	2010	2010-12-17	SÁNCHEZ
1	2011	2011-02-22	SALA
1	2016	2016-01-23	MUÑOZ
2	2016	2016-12-03	FERNÁNDEZ
1	2017	2017-01-09	RUIZ
2	2017	2017-04-02	JIMÉNEZ
3	2017	2017-05-01	GOMIS
4	2017	2017-09-29	MARTÍN
5	2017	2017-12-03	GIMENO
1	2018	2018-06-10	RUIZ
2	2018	2018-09-08	TOVAR
3	2018	2018-09-23	ALONSO
1	2020	2020-01-23	ALONSO

(17 rows)

6.15. Window Functions.

```
select dept_num,
  num,
  salary,
  avg(salary) OVER (PARTITION BY dept_num)
from employees;
```

dept_num	num	salary	avg
10	7782	2885	2287.5000000000000000
10	7934	1690	2287.5000000000000000
20	7566	2900	2178.8888888888888889
20	8001	2885	2178.8888888888888889
20	7369	1040	2178.8888888888888889
20	7788	3000	2178.8888888888888889
20	7876	1430	2178.8888888888888889
20	7877	1430	2178.8888888888888889
20	7902	3000	2178.8888888888888889
20	8000	2885	2178.8888888888888889
20	1000	1040	2178.8888888888888889
30	7654	1600	1735.8333333333333333
30	7698	3005	1735.8333333333333333
30	7521	1625	1735.8333333333333333
30	7499	1500	1735.8333333333333333
30	7844	1350	1735.8333333333333333
30	7900	1335	1735.8333333333333333

(17 rows)

```
select dept_num,
  avg(salary)
from employees
group by dept_num;
```

dept_num	avg
30	1735.8333333333333333
10	2287.5000000000000000
20	2178.8888888888888889

(3 rows)

6.15. Window Functions.

*Columns come directly from the **employees table***

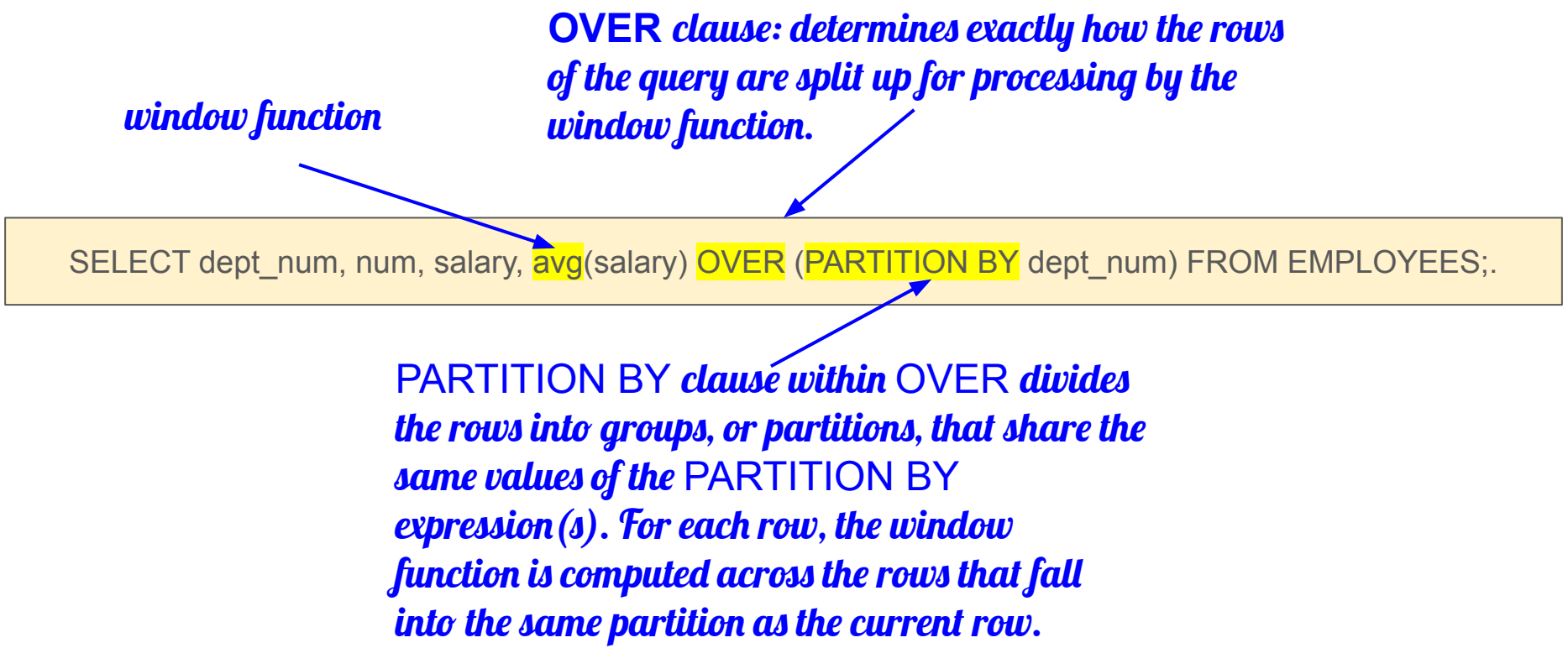
dept_num	num	salary	avg
10	7782	2885	2287.5000000000000000
10	7934	1690	2287.5000000000000000
20	7566	2900	2178.8888888888888889
20	8001	2885	2178.8888888888888889
20	7369	1040	2178.8888888888888889
20	7788	3000	2178.8888888888888889
20	7876	1430	2178.8888888888888889
20	7877	1430	2178.8888888888888889
20	7902	3000	2178.8888888888888889
20	8000	2885	2178.8888888888888889
20	1000	1040	2178.8888888888888889
30	7654	1600	1735.8333333333333333
30	7698	3005	1735.8333333333333333
30	7521	1625	1735.8333333333333333
30	7499	1500	1735.8333333333333333
30	7844	1350	1735.8333333333333333
30	7900	1335	1735.8333333333333333

(17 rows)

*There is one output row for each row in the table employees (there is **no where condition**)*

*Represents an **average** taken across all the table **rows** that have the **same dept_num** value as the current row.*

6.15. Window Functions.



6.15. Window Functions.

You can also **control the order in which rows are processed by window functions** using **ORDER BY within OVER**. (The window ORDER BY does not even have to match the order in which the rows are output).

```
SELECT dept_num, num, salary,
        rank() OVER (PARTITION BY dept_num ORDER BY salary DESC)
FROM EMPLOYEES;
```

rank ***function produces a numerical rank for each distinct*** ORDER BY ***value in the current row's partition, using the order defined by the*** ORDER BY ***clause.***

dept_num	num	salary	rank
10	7782	2885	1
10	7934	1690	2
20	7902	3000	1
20	7788	3000	1
20	7566	2900	3
20	8001	2885	4
20	8000	2885	4
20	7876	1430	6
20	7877	1430	6
20	7369	1040	8
20	1000	1040	8
30	7698	3005	1
30	7521	1625	2
30	7654	1600	3
30	7499	1500	4
30	7844	1350	5
30	7900	1335	6

(17 rows)

6.15. Window Functions.

The **window frame** is a set of rows related to the current row where the window function is used for calculation.

The window frame can be a different set of rows for the next row in the query result, since it depends on the current row being processed.

By default, if ORDER BY is supplied then the frame consists of all rows from the start of the partition up through the current row, plus any following rows that are equal to the current row according to the ORDER BY clause.

When ORDER BY is omitted, the default frame consists of all rows in the partition.

```
SELECT salary,
    sum(salary) OVER ()
FROM employees;
```

```
SELECT salary,
    sum(salary) OVER (
        ORDER BY salary
    )
FROM employees;
```

salary	sum
1040	34600
1040	34600
1500	34600
1625	34600
2900	34600
1600	34600
3005	34600
2885	34600
3000	34600
1350	34600
1430	34600
1430	34600
1335	34600
3000	34600
1690	34600
2885	34600
2885	34600

(17 rows)

salary	sum
1040	2080
1040	2080
1335	3415
1350	4765
1430	7625
1430	7625
1500	9125
1600	10725
1625	12350
1690	14040
2885	22695
2885	22695
2885	22695
2900	25595
3000	31595
3000	31595
3005	34600

(17 rows)

```
select sum(salary)
from employees;
```

```
+-------------+
| sum(salary) |
+-------------+
|       34600 |
+-------------+
1 row in set (0,000 sec)
```

6.15. Window Functions.

```
SELECT salary,
   sum(salary) OVER ()
FROM employees;
```

salary	sum
1040	34600
1040	34600
1500	34600
1625	34600
2900	34600
1600	34600
3005	34600
2885	34600
3000	34600
1350	34600
1430	34600
1430	34600
1335	34600
3000	34600
1690	34600
2885	34600
2885	34600

(17 rows)

Since there is no ORDER BY *in the* OVER *clause, the window frame is the same as the partition, which for lack of* PARTITION BY *is the whole table.*

```
SELECT salary,
   sum(salary) OVER (
      ORDER BY salary
   )
FROM employees;
```

salary	sum
1040	2080
1040	2080
1335	3415
1350	4765
1430	7625
1430	7625
1500	9125
1600	10725
1625	12350
1690	14040
2885	22695
2885	22695
2885	22695
2900	25595
3000	31595
3000	31595
3005	34600

(17 rows)

The sum is taken from the first (lowest) salary up through the current one, including any duplicates of the current one (notice the results for the duplicated salaries).

6.15. Window Functions.

Some common uses of window functions include **ranking**, **cumulative sum**, and **moving average** calculations.

Ranking: You can use the ROW_NUMBER() function to assign a unique rank to each row in a result set based on a specific order.

```
SELECT num,
    salary,
    ROW_NUMBER() OVER (
        ORDER BY sales DESC
    ) AS rank
FROM employees;
```

PostgreSQL

```
 num  | salary | rank
------+--------+------
 7698 |   3005 |    1
 7902 |   3000 |    2
 7788 |   3000 |    3
 7566 |   2900 |    4
 8001 |   2885 |    5
 7782 |   2885 |    6
 8000 |   2885 |    7
 7934 |   1690 |    8
 7521 |   1625 |    9
 7654 |   1600 |   10
 7499 |   1500 |   11
 7876 |   1430 |   12
 7877 |   1430 |   13
 7844 |   1350 |   14
 7900 |   1335 |   15
 7369 |   1040 |   16
 1000 |   1040 |   17
(17 rows)
```

6.15. Window Functions.

Some common uses of window functions include **ranking**, **cumulative sum**, and **moving average** calculations.

Cumulative Sum: You can use the SUM() function along with the OVER() clause to calculate a cumulative sum of a specific column.

```
SELECT num,
    salary,
    SUM(salary) OVER (
        ORDER BY num ROWS BETWEEN
UNBOUNDED PRECEDING AND CURRENT ROW
    ) AS cumulative_salaries
FROM employees;
```

PostgreSQL

```
 num  | salary | cumulative_salaries
------+--------+---------------------
 1000 |   1040 |                1040
 7369 |   1040 |                2080
 7499 |   1500 |                3580
 7521 |   1625 |                5205
 7566 |   2900 |                8105
 7654 |   1600 |                9705
 7698 |   3005 |               12710
 7782 |   2885 |               15595
 7788 |   3000 |               18595
 7844 |   1350 |               19945
 7876 |   1430 |               21375
 7877 |   1430 |               22805
 7900 |   1335 |               24140
 7902 |   3000 |               27140
 7934 |   1690 |               28830
 8000 |   2885 |               31715
 8001 |   2885 |               34600
(17 rows)
```

6.15. Window Functions.

Some common uses of window functions include **ranking**, **cumulative sum**, and **moving average** calculations.

Moving Average: You can use the AVG() function along with the OVER() clause to calculate a moving average of a specific column.

```
SELECT num,
    salary,
    AVG(salary) OVER (
        ORDER BY num ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
    ) AS moving_salary
FROM employees;
```

PostgreSQL

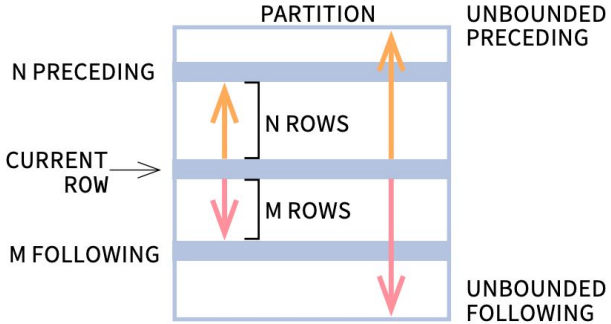
```
 num  | salary |     moving_salary
------+--------+-----------------------
 1000 |   1040 | 1040.0000000000000000
 7369 |   1040 | 1040.0000000000000000
 7499 |   1500 | 1193.3333333333333333
 7521 |   1625 | 1388.3333333333333333
 7566 |   2900 | 2008.3333333333333333
 7654 |   1600 | 2041.6666666666666667
 7698 |   3005 | 2501.6666666666666667
 7782 |   2885 | 2496.6666666666666667
 7788 |   3000 | 2963.3333333333333333
 7844 |   1350 | 2411.6666666666666667
 7876 |   1430 | 1926.6666666666666667
 7877 |   1430 | 1403.3333333333333333
 7900 |   1335 | 1398.3333333333333333
 7902 |   3000 | 1921.6666666666666667
 7934 |   1690 | 2008.3333333333333333
 8000 |   2885 | 2525.0000000000000000
 8001 |   2885 | 2486.6666666666666667
(17 rows)
```

6.15. Window Functions.

The purpose of the **ROWS** clause is to specify the window frame in relation to the current row. The syntax is:
ROWS BETWEEN lower_bound AND upper_bound

The bounds can be any of these five options:

- UNBOUNDED PRECEDING – All rows before the current row.
- n PRECEDING – n rows before the current row.
- CURRENT ROW – Just the current row.
- n FOLLOWING – n rows after the current row.
- UNBOUNDED FOLLOWING – All rows after the current row.



Source:
https://learnsql.com/blog/sql-window-functions-rows-clause/

6.15. Window Functions.

Function	Return Type	Description
row_number()	**bigint**	**number of the current row within its partition, counting from 1**
rank()	**bigint**	**rank of the current row with gaps; same as** row_number **of its first peer**
dense_rank()	**bigint**	**rank of the current row without gaps; this function counts peer groups**
percent_rank()	**double precision**	**relative rank of the current row: (**rank **- 1) / (total rows - 1)**
cume_dist()	**double precision**	**relative rank of the current row: (number of rows preceding or peer with current row) / (total rows)**
ntile(***num_buckets*** integer)	**integer**	**integer ranging from 1 to the argument value, dividing the partition as equally as possible**
lag(***value***anyelement [,***offset*** integer [,***default***anyelement]])	**same type as *value***	**returns *value* evaluated at the row that is *offset* rows before the current row within the partition; if there is no such row, instead return *default* (which must be of the same type as *value*). Both *offset* and *default* are evaluated with respect to the current row. If omitted, *offset* defaults to 1 and *default* to null**

6.15. Window Functions.

Function	Return Type	Description
lead(***value*** anyelement [,***offset*** integer [,***default*** anyelement]])	same type as ***value***	returns ***value*** evaluated at the row that is ***offset*** rows after the current row within the partition; if there is no such row, instead return ***default*** (which must be of the same type as ***value***). Both ***offset*** and ***default*** are evaluated with respect to the current row. If omitted, ***offset*** defaults to 1 and ***default*** to null
first_value(***value*** any)	same type as ***value***	returns ***value*** evaluated at the row that is the first row of the window frame
last_value(***value*** any)	same type as ***value***	returns ***value*** evaluated at the row that is the last row of the window frame
nth_value(***value*** any, ***nth*** integer)	same type as ***value***	returns ***value*** evaluated at the row that is the ***nth*** row of the window frame (counting from 1); null if no such row

- Full list of window functions here: https://www.postgresql.org/docs/current/functions-window.html

6.15. Window Functions.

Window Functions Overview
Window functions perform calculations across a set of rows related to the current row.

AVG
Returns the average value.

BIT_AND
Bitwise AND.

BIT_OR
Bitwise OR.

BIT_XOR
Bitwise XOR.

COUNT
Returns count of non-null values.

CUME_DIST
Window function that returns the cumulative distribution of a given row.

DENSE_RANK
Rank of a given row with identical values receiving the same result, no skipping.

FIRST_VALUE
Returns the first result from an ordered set.

JSON_ARRAYAGG
Returns a JSON array containing an element for each value in a given set of JSON or SQL values.

JSON_ARRAYAGG
Returns a JSON array containing an element for each value in a given set of JSON or SQL values.

JSON_OBJECTAGG
Returns a JSON object containing key-value pairs.

LAG
Accesses data from a previous row in the same result set without the need for a self-join.

LAST_VALUE
Returns the last value in a list or set of values.

LEAD
Accesses data from a following row in the same result set without the need for a self-join.

MAX
Returns the maximum value.

MEDIAN
Window function that returns the median value of a range of values.

MIN
Returns the minimum value.

NTH_VALUE
Returns the value evaluated at the specified row number of the window frame.

NTILE
Returns an integer indicating which group a given row falls into.

6.15. Window Functions.

PERCENTILE_CONT
Continuous percentile.

PERCENTILE_DISC
Discrete percentile.

RANK
Rank of a given row with identical values receiving the same result.

ROW_NUMBER
Row number of a given row with identical values receiving a different result.

STD
Population standard deviation.

STDDEV
Population standard deviation.

STDDEV_POP
Returns the population standard deviation.

STDDEV_SAMP
Standard deviation.

SUM
Sum total.

VAR_POP
Population standard variance.

VAR_SAMP
Returns the sample variance.

VARIANCE
Population standard variance.

Aggregate Functions as Window Functions
It is possible to use aggregate functions as window functions.

ColumnStore Window Functions
Summary of window function use with the ColumnStore engine

Window Frames
Some window functions operate on window frames.

- Full list of window functions here: https://www.postgresql.org/docs/current/tutorial-window.html

6.15. Window Functions.

They are forbidden elsewhere, such as in GROUP BY, HAVING and WHERE clauses.

Window functions are permitted only in the SELECT list and the ORDER BY clause of the query.

We saw it very quickly... Check this link:

https://www.toptal.com/sql/intro-to-sql-windows-functions

https://learnsql.com/blog/sql-window-functions-rows-clause/

6.16. Subqueries.

SQL provides a mechanism for the nesting of subqueries. A **subquery** is a select-from-where expression that is nested within another query.

The nesting can be done in the following SQL query

select $A_1, A_2, ..., A_n$
from $r_1, r_2, ..., r_m$
where P
as follows:

CASE 1:

A_i can be replaced be a subquery that generates a single value.

CASE 2

r_i can be replaced by any valid subquery

CASE 3:

P can be replaced with an expression of the form:

B <operation> (subquery)

Where B is an attribute and <operation> to be defined later.

6.16. Subqueries.

CASE 1:

Show name and surname of employees with their department name:

```
select E.name, E.surname, E.dept_num,
            (select count(*)
            from employees E1
            where E1.dept_num = E.dept_num) as n_dept
      from employees E
      order by E.dept_num;
```

name	surname	dept_num	n_dept
MARIA	CEREZO	10	2
ANTONIA	MUÑOZ	10	2
JUAN	JIMÉNEZ	20	9
FERNANDA	RUIZ	20	9
SERGIO	SÁNCHEZ	20	9
JESUS	GILBERTO	20	9
FERNANDO	ALONSO	20	9
LAURA	ALONSO	20	9
ANA	FERNÁNDEZ	20	9
ANTONIO	RUIZ	20	9
BRAD	POTTER	20	9
MONICA	MARTÍN	30	6
BARTOLOME	GOMIS	30	6
REBECA	SALA	30	6
MARTA	ARROYO	30	6
LUIS	TOVAR	30	6
XAVIER	GIMENO	30	6

(17 rows)

6.16. Subqueries.

CASE 2:

Show name, surname, salary and department number from employees but also the average salary in their department:

```
select E.name, E.surname, E.salary,
            E.dept_num, S.avgsalary
from employees as E,
            (select dept_num,
                  AVG(salary) as avgsalary
            from employees
            group by dept_num) as S
where E.dept_num = S.dept_num;
```

name	surname	salary	dept_num	avgsalary
BRAD	POTTER	1040	20	2178.8888888888888889
SERGIO	SÁNCHEZ	1040	20	2178.8888888888888889
MARTA	ARROYO	1500	30	1735.8333333333333333
REBECA	SALA	1625	30	1735.8333333333333333
JUAN	JIMÉNEZ	2900	20	2178.8888888888888889
MONICA	MARTÍN	1600	30	1735.8333333333333333
BARTOLOME	GOMIS	3005	30	1735.8333333333333333
MARIA	CEREZO	2885	10	2287.5000000000000000
JESUS	GILBERTO	3000	20	2178.8888888888888889
LUIS	TOVAR	1350	30	1735.8333333333333333
FERNANDO	ALONSO	1430	20	2178.8888888888888889
LAURA	ALONSO	1430	20	2178.8888888888888889
XAVIER	GIMENO	1335	30	1735.8333333333333333
ANA	FERNÁNDEZ	3000	20	2178.8888888888888889
ANTONIA	MUÑOZ	1690	10	2287.5000000000000000
ANTONIO	RUIZ	2885	20	2178.8888888888888889
FERNANDA	RUIZ	2885	20	2178.8888888888888889

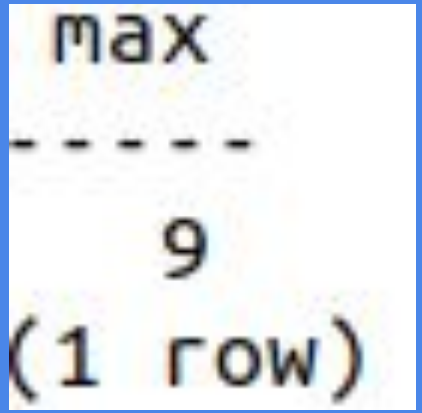
(17 rows)

6.16. Subqueries.

CASE 2:

Find the the maximum number of employees of all the departments:

```
select MAX(T.num_employees)
from (
        select  E.dept_num, count(*) as num_employees
                from employees E
                group by E.dept_num
        ) as T;
```



6.16. Subqueries.

CASE 3:

Show the departments whose average salary is greater than the average of salaries of all employees:

```
 dept_num |          avg
----------+-----------------------
       10 | 2287.5000000000000000
       20 | 2178.8888888888888889
(2 rows)
```

```
select dept_num, avg(salary)
    from employees
    group by dept_num
    having avg(salary) > (select avg(salary) from employees);
```

6.16. Subqueries.

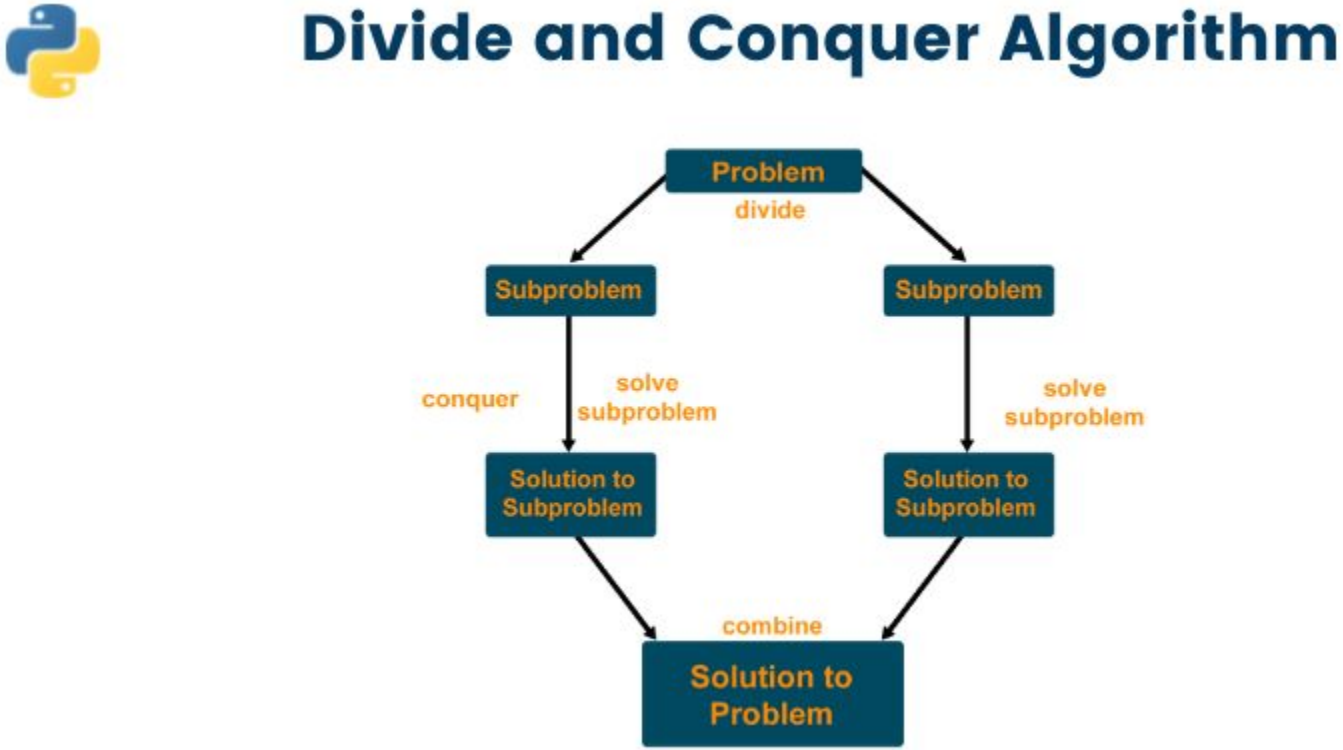
CASE 3:

List all employees who are managers:

```
select name, surname
      from employees
      where num IN
            (select distinct manager
            from employees);
```

```
   name     | surname
------------+----------
BRAD        | POTTER
BARTOLOME   | GOMIS
MARIA       | CEREZO
JESUS       | GILBERTO
ANTONIO     | RUIZ
FERNANDA    | RUIZ
(6 rows)
```

6.16. Subqueries.



Source: https://favtutor.com/blogs/divide-and-conquer-algorithm

6.16. Subqueries (+ 6.13. Set operations).

some

Find names of employees with salary greater than some (at least one) employees in the department number 10:

```
select distinct F.name, F.surname, F.salary
from employees as F, employees as E
where F.salary > E.salary and E.dept_num = 10;
```

name	surname	salary
ANTONIO	RUIZ	2885
BARTOLOME	GOMIS	3005
FERNANDA	RUIZ	2885
MARIA	CEREZO	2885
JESUS	GILBERTO	3000
JUAN	JIMÉNEZ	2900
ANA	FERNÁNDEZ	3000

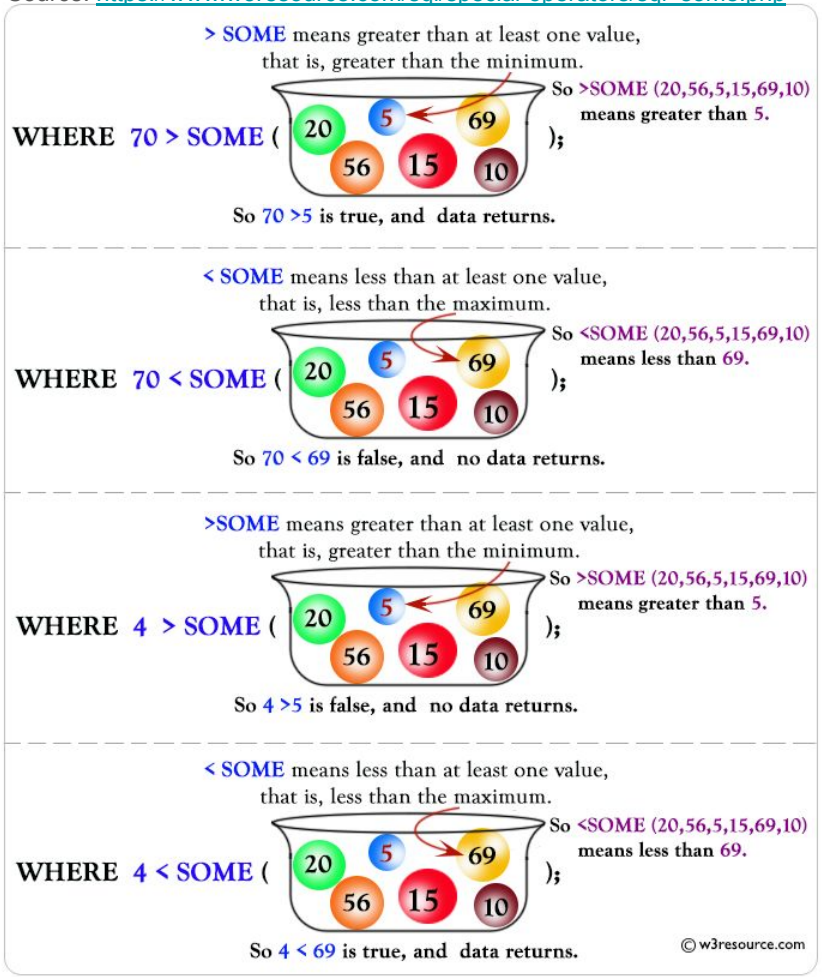
```
select name, surname, salary
from employees
where salary > some (select salary
                     from employees
                     where dept_num = 10);
```

name	surname	salary
JUAN	JIMÉNEZ	2900
BARTOLOME	GOMIS	3005
MARIA	CEREZO	2885
JESUS	GILBERTO	3000
ANA	FERNÁNDEZ	3000
ANTONIO	RUIZ	2885
FERNANDA	RUIZ	2885

w3resource.com:

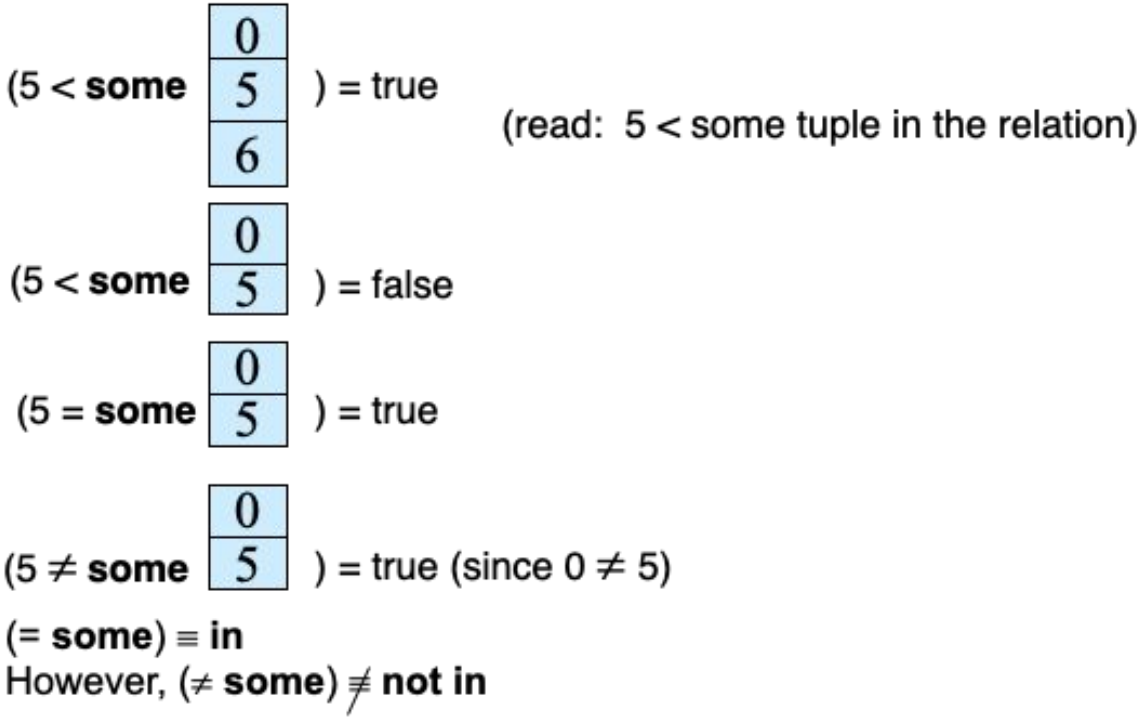
- "SOME compare a value to each value in a list or results from a query and evaluate to true if the result of an inner query contains at least one row.
- SOME must match at least one row in the subquery and must be preceded by comparison operators.
- Suppose using greater than (>) with SOME means greater than at least one value".

Source: https://www.w3resource.com/sql/special-operators/sql_some.php



6.16. Subqueries (+ 6.13. Set operations).

some



Source of the picture: Abraham Silberschatz, Henry F. Korth and S. Sudarshan. Database System Concepts.

6.16. Subqueries (+ 6.13. Set operations).

all

Find the names, salary and department number of all employees whose salary is greater than that of all employees in department number 10:

```
select name, surname, salary, dept_num
from employees
where salary > all (select salary
                    from employees
                    where dept_num = 10);
```

name	surname	salary	dept_num
JUAN	JIMÉNEZ	2900	20
BARTOLOME	GOMIS	3005	30
JESUS	GILBERTO	3000	20
ANA	FERNÁNDEZ	3000	20

(4 rows)

6.16. Subqueries (+ 6.13. Set operations).

all

All the employees but the ones with the highest salary...

W3schools:
"The ALL operator:

- returns a boolean value as a result
- returns **TRUE if ALL of the subquery values meet the condition**
- is used with SELECT, WHERE and HAVING statements

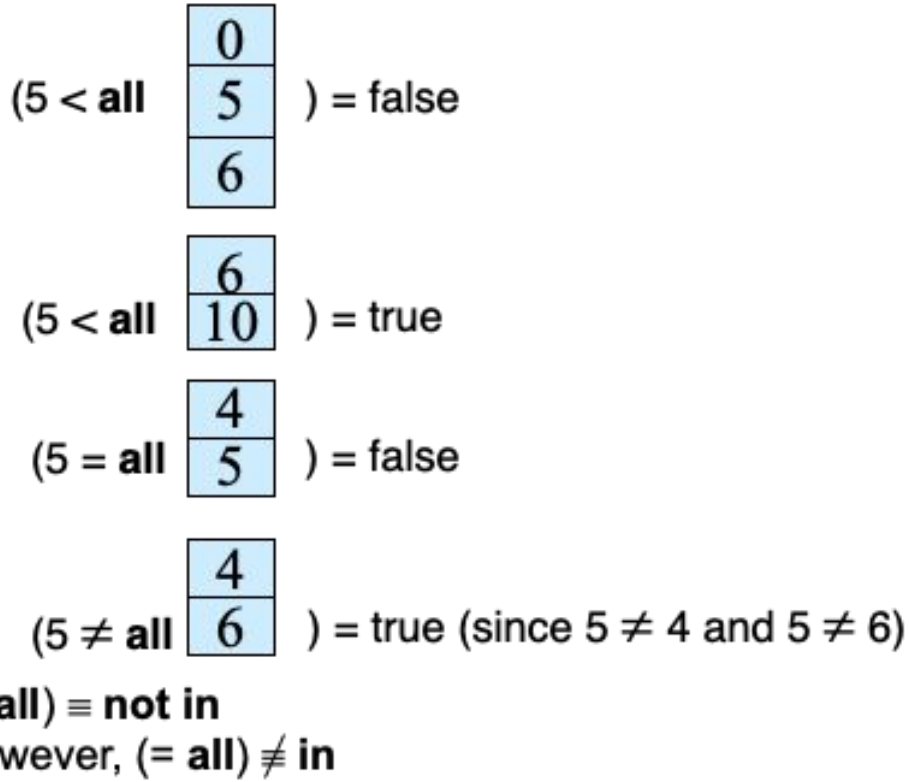
ALL means that the condition will be true only if the operation is true for all values in the range".

```
samplecompany=# select num,
  salary
from employees
where salary >= all (
    select distinct salary
    from employees
  )
order by salary;
 num  | salary
------+--------
 7698 |   3005
(1 row)
```

PostgreSQL

6.16. Subqueries (+ 6.13. Set operations).

all



Source of the picture: Abraham Silberschatz, Henry F. Korth and S. Sudarshan. Database System Concepts.

6.16. Subqueries (+ 6.13. Set operations).

exists

The **exists** construct returns the value true if the r**esult of the subquery is nonempty**.

$$\text{exists } r \Leftrightarrow r \neq \emptyset$$

$$\text{not exists } r \Leftrightarrow r = \emptyset$$

6.16. Subqueries (+ 6.12. Set operations).

exists

Find the salaries of all employees that are less than the largest salary:

```
SELECT DISTINCT E.salary
FROM employees E
WHERE EXISTS
        (SELECT *
        FROM employees F
        WHERE
        E.salary < F.salary);
```

```
 salary
--------
   1040
   1335
   1350
   1430
   1500
   1600
   1625
   1690
   2885
   2900
   3000
(11 rows)
```

```
SELECT DISTINCT salary
FROM employees;
```

```
 salary
--------
   1690
   3000
   1335
   2900
   1430
   2885
   3005
   1600
   1500
   1040
   1350
   1625
(12 rows)
```

```
SELECT DISTINCT E.salary
FROM employees E
WHERE NOT EXISTS
        (SELECT *
        FROM employees F
        WHERE
                E.salary < F.salary);
```

```
 salary
--------
   3005
(1 row)
```

Correlated subquery: the inner query.

Correlation name: variable E in the outer query.

6.16. Subqueries (+ 6.12. Set operations).

exists

W3schools:

- "The EXISTS operator is used to test for the existence of any record in a subquery.
- The EXISTS operator returns TRUE if the subquery returns one or more records".

```
samplecompany=# insert into towns values ('PMI', 'Palma de Mallorca');
INSERT 0 1
samplecompany=# select code, name
from towns T
where not exists (
    select *
    from departments D
    where D.town_code = T.code
  );
 code |       name
------+------------------
 PMI  | Palma de Mallorca
(1 row)
```

PostgreSQL

6.16. Subqueries (+ 6.12. Set operations).

any

All the employees but the ones with the lowest salary...

W3schools:
"The ANY operator:

- returns a boolean value as a result
- returns **TRUE if ANY of the subquery values meet the condition**

ANY means that the condition will be true if the operation is true for any of the values in the range".

```
samplecompany=# select num,
  salary
from employees
where salary > any (
    select distinct salary
    from employees
  )
order by salary;
 num  | salary
------+--------
 7900 |   1335
 7844 |   1350
 7877 |   1430
 7876 |   1430
 7499 |   1500
 7654 |   1600
 7521 |   1625
 7934 |   1690
 8001 |   2885
 8000 |   2885
 7782 |   2885
 7566 |   2900
 7902 |   3000
 7788 |   3000
 7698 |   3005
(15 rows)
```

PostgreSQL

6.17. Modification of the database with subqueries.

Deletion of tuples from a given relation.

Insertion of new tuples into a given relation.

Updating values in some tuples of a given relation.

6.17. Modification of the database with subqueries.

Deletion with subqueries

Deletion of tuples from a given relation.

Delete all employees:
delete from employees;

Delete all employees from the Sales department:

```
delete from employees
where dept_num = 30;

delete from employees
where dept_num = (select num
                  from departments
                  where name='SALES');
```

Delete all tuples in the "employees" relation for those instructors associated with a department located in MADRID:

```
delete from employees
where dept_num IN (select num
                   from departments D, towns T
                   where T.code = D.town_code and
                   T.name='MADRID');
```

6.17. Modification of the database with subqueries.

Deletion with subqueries

Delete all employees whose salary is less than the average salary:

```
delete from employees
where salary < (select avg(salary)
                from employees);
```

```
+-------------+
| avg(salary) |
+-------------+
|  2035.2941  |
+-------------+
```

```
        avg
---------------------
2035.2941176470588235
```

```
samplecompany=# delete from employees
where salary < (select avg (salary)
                from employees);
DELETE 10
```

```
MariaDB [samplecompany]> delete from employees
    -> where salary < (select avg (salary)
    ->                 from employees);
Query OK, 10 rows affected (0,007 sec)
```

Problem: As we delete tuples from deposit, the average salary changes.

The DBMS first solves the subquery and, with the result of the subquery, executes the main query.

6.17. Modification of the database with subqueries.

Insertion with subqueries

Let's imagine that we split the "employees" relation into another table called "managers" and we want to insert the managers there:

```
CREATE TABLE managers (
  num int NOT NULL primary key,
  surname varchar(50) NOT NULL,
  name varchar(50) NOT NULL
  start_date date DEFAULT NULL,
  salary smallint DEFAULT NULL,
  commission smallint DEFAULT NULL,
  dept_num smallint DEFAULT NULL references departments (num)
);
CREATE INDEX idx_managers_fk ON managers (dept_num);
```

```
insert into managers
        select num, surname, name, start_date,
                salary, commission, dept_num
        from employees
        where occu_code = 'MAN';
```

```
samplecompany=# insert into managers
select num, surname, name, start_date, salary, commission, dept_num
from employees
where occu_code = 'MAN';
INSERT 0 5
samplecompany=# select *
samplecompany-# from managers;
 num  | surname |   name    | start_date | salary | commission | dept_num
------+---------+-----------+------------+--------+------------+----------
 7566 | JIMÉNEZ | JUAN      | 2017-04-02 |   2900 |            |       20
 7698 | GOMIS   | BARTOLOME | 2017-05-01 |   3005 |            |       30
 7782 | CEREZO  | MARIA     | 2010-06-09 |   2885 |            |       10
 8000 | RUIZ    | ANTONIO   | 2017-01-09 |   2885 |            |       20
 8001 | RUIZ    | FERNANDA  | 2018-06-10 |   2885 |            |       20
(5 rows)
```

6.17. Modification of the database with subqueries.

Updating with subqueries

Increase salaries and commissions of employees whose salary is over $2,500 by 3%, and all others by a 5%:

Two statements:

```
update EMPLOYEES
set salary = salary * 1.03,
        commission = commission * 1.03
where salary > 2500;

update EMPLOYEES
set salary = salary * 1.05,
        commission = commission * 1.05
where salary <= 2500;
```

Order matters, can you see why?

It is best implemented using a case statement (**next slide**).

6.17. Modification of the database with subqueries.

Updating with subqueries

Increase salaries and commissions of employees whose salary is over $2,500 by 3%, and all others by a 5%:

```
update employees
set commission = case
    when salary <= 2500 then commission * 1.05
    else commission * 1.03
  end,
  salary = case
    when salary <= 2500 then salary * 1.05
    else salary * 1.03
  end;
```

https://www.postgresql.org/docs/current/functions-conditional.html

PostgreSQL

https://mariadb.com/kb/en/case-operator/

MariaDB

6.17. Modification of the database with subqueries.

Updating with subqueries

Update salaries for the employees with salary smaller than 1200 to the average salary of the employees with salary smaller or equal to 1800:

```
update employees E
set E.salary = (
    select avg(salary)
    from employees F
    where E.dept_num = F.dept_num
      and F.salary <= 1800
  )
where E.salary < 1200;
```

```
MariaDB [samplecompany]> select num, salary
    -> from employees;
+------+--------+
| num  | salary |
+------+--------+
| 1000 |   1040 |
| 7369 |   1040 |
| 7499 |   1500 |
| 7521 |   1625 |
| 7566 |   2900 |
| 7654 |   1600 |
| 7698 |   3005 |
| 7782 |   2885 |
| 7788 |   3000 |
| 7844 |   1350 |
| 7876 |   1430 |
| 7877 |   1430 |
| 7900 |   1335 |
| 7902 |   3000 |
| 7934 |   1690 |
| 8000 |   2885 |
| 8001 |   2885 |
+------+--------+
17 rows in set (0,000 sec)
```

```
MariaDB [samplecompany]> select F.dept_num, avg(F.salary)
    -> from employees F
    -> where F.salary <= 1800
    -> group by dept_num;
+----------+---------------+
| dept_num | avg(F.salary) |
+----------+---------------+
|       10 |     1690.0000 |
|       20 |     1235.0000 |
|       30 |     1482.0000 |
+----------+---------------+
3 rows in set (0,000 sec)
```

Sources.

- **https://www.w3schools.com/sql/default.asp**
- **M. J. Ramos, A. Ramos and F. Montero.** Sistemas gestores de bases de datos (Chapter 1, pag. 7-15). McGrawHill: 1th Edition, 2006.
- **Abraham Silberschatz, Henry F. Korth and S. Sudarshan**. *Database System Concepts (Chapter 3)*. McGrawHill: 6th Edition, 2010.<http://codex.cs.yale.edu/avi/db-book/db6/slide-dir/index.html>
- Apunts de la UIB del professor Miquel Manresa (1996).
- https://www.studytonight.com/dbms/
- https://www.toptal.com/sql/intro-to-sql-windows-functions
- https://learnsql.com/blog/sql-window-functions-rows-clause/