




Databases



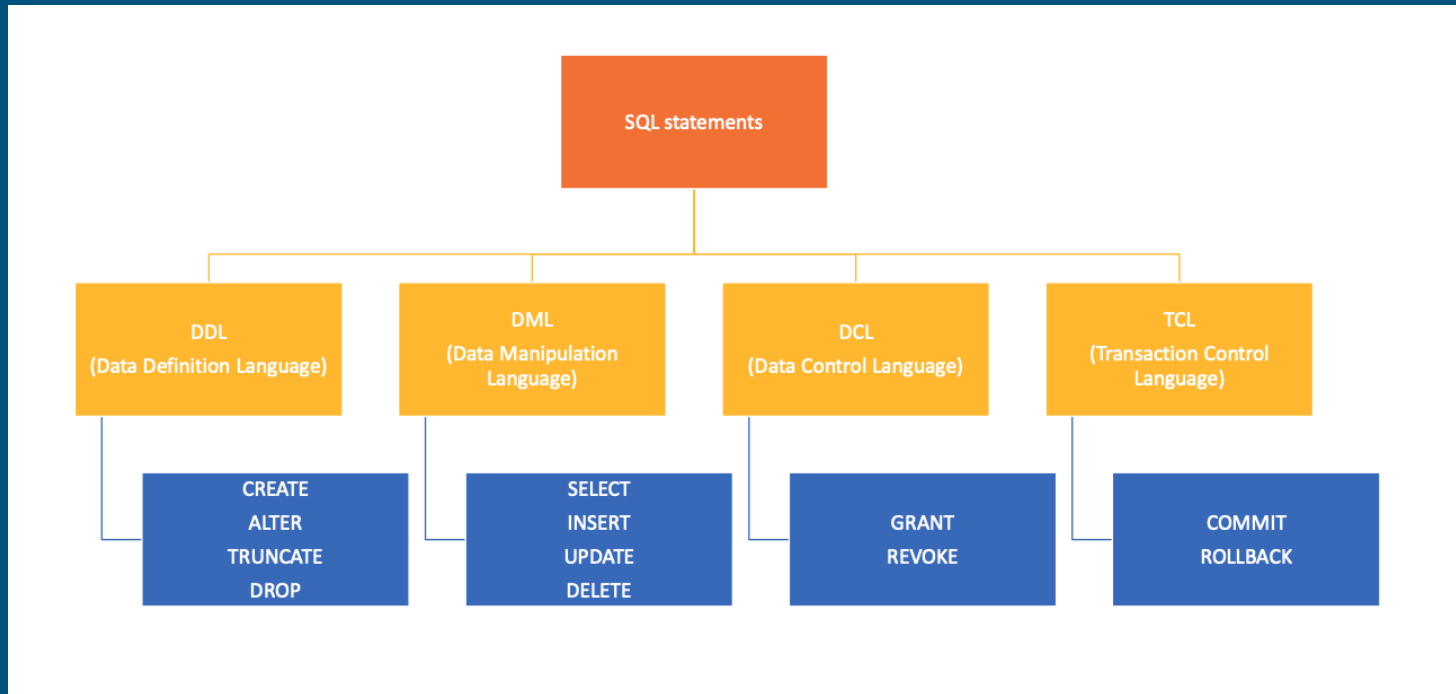
Lecture 3
SQL Basics. DML



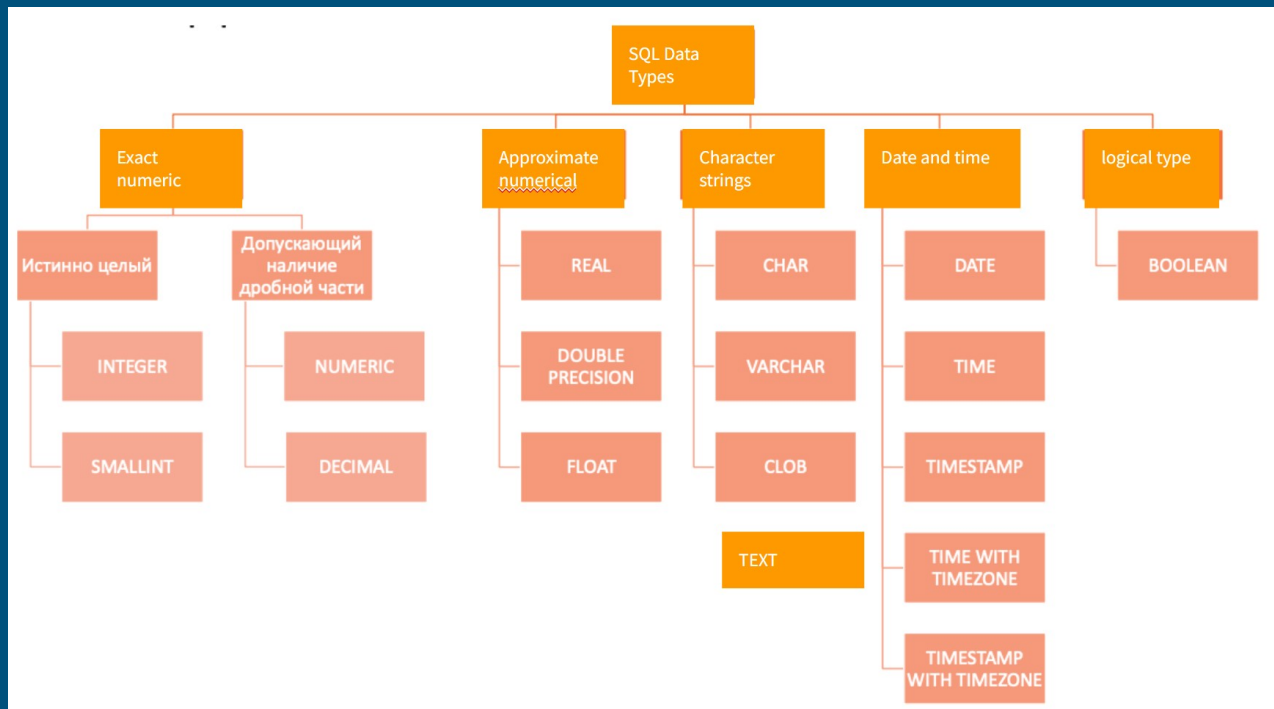
STRUCTURED QUERY LANGUAGE (SQL)

- Domain-specific language (Domain-specific language)
- Used to work with relational databases
- Managing a large amount of information with a single request
- No need to specify how we get the record

SQL Statements



SQL Data Types



Data Definition Language

CREATE

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name(  
    col_name_1  datatype_1,  
    col_name_2  datatype_2,  
    ...  
    col_name_N  datatype_N  
);
```

Create: restrictions

CREATE TABLE students (
 student_id int PRIMARY KEY,
 first_name varchar(50),
 last_name varchar(50),
 age int NOT NULL,
 email varchar(100)
);

Diagram illustrating the components of the CREATE TABLE statement:

- column name: student_id
- column type: int
- restriction: PRIMARY KEY
- restriction: NOT NULL
- restriction: varchar(50)

Restrictions

- UNIQUE
- CHECK
- IS NOT NULL
- PRIMARY/FOREIGN KEY
- DEFAULT
- ...

Restrictions: PRIMARY KEY

PRIMARY KEY (primary key) is a field (or combination fields) that uniquely identifies the record.

PRIMARY KEY should:

- contain unique values
- cannot contain NULL values.

Restrictions: PRIMARY KEY

A table cannot have two records with the same key value.

A table can only have one primary key.

```
CREATE TABLE products (  
  product_no integer UNIQUE NOT NULL,  
  name text,  
  price numeric  
);
```

```
CREATE TABLE products (  
  product_no integer PRIMARY KEY,  
  name text,  
  price numeric  
);
```

```
CREATE TABLE example (  
  a integer,  
  b integer,  
  c integer,  
  PRIMARY KEY (a, c)  
);
```

Restrictions: FOREIGN KEY

Foreign key - an attribute (or group of attributes) whose value can be repeated for several records.

Contains a reference to a primary key field in another table.

The table containing the foreign key is called a child table, containing the primary key - parent table.

Restrictions: FOREIGN KEY

```
CREATE TABLE products (  
  product_no integer PRIMARY KEY,  
  name text,  
  price numeric  
);
```



```
CREATE TABLE orders (  
  order_id integer PRIMARY KEY,  
  product_no integer REFERENCES products (product_no)  
  quantity integer  
);
```

Restrictions: DEFAULT

```
CREATE TABLE order (  
    order_id INTEGER PRIMARY KEY,  
    order_number INTEGER NOT NULL,  
    order_date  DATE DEFAULT now()::date  
);
```

Restrictions syntax

Constraints can be named using the command **CONSTRAINT** (this is useful for error output)

```
CREATE TABLE products (  
  product_no integer,  
  name text,  
  price numeric,  
  discounted_price numeric  
  CHECK (price > discounted_price)  
);
```

```
CREATE TABLE products (  
  product_no integer,  
  name text,  
  price numeric,  
  discounted_price numeric,  
  CONSTRAINT valid_discount CHECK (price >  
  discounted_price)  
);
```

Relations between tables

- one to one
- one to many
- many to many

NULL value

A field with a NULL value is a field with no value.

If a field in a table is not required, then you can insert a new record or update without adding a value to the field; it will be stored as NULL.

A NULL value is different from a null value (0) or a field that contains spaces (" "). It is not possible to check for NULL values with comparison operators such as =, <, or <>.

You need to use IS NULL or IS NOT NULL

Restrictions: IS NULL / IS NOT NULL

IS NULL Syntax

```
SELECT column_names  
FROM table_name  
WHERE column_name IS NULL
```

IS NOT NULL Syntax

```
SELECT column_names  
FROM table_name  
WHERE column_name IS NOT NULL
```


Ternary logic: True, False, Unknown

NOT(A)		AND(A, B)					OR(A, B)					XOR(A, B)				
		$A \wedge B$		B			$A \vee B$		B			$A \oplus B$		B		
A	$\neg A$			F	U	T			F	U	T			F	U	T
F	T		F	F	F	F		F	F	U	T		F	F	U	T
U	U	A	U	F	U	U		U	U	U	T		U	U	U	U
T	F		T	F	U	T		T	T	T	T		T	T	U	F

Data Definition Language

ALTER - modification of objects

```
ALTER TABLE table_name ADD column_name datatype;
```

```
ALTER TABLE table_name DROP column_name;
```

```
ALTER TABLE table_name RENAME column_name TO  
new_column_name;
```

```
ALTER TABLE table_name ALTER column_name TYPE datatype;
```

Data Definition Language

```
TRUNCATE TABLE SUPERHERO;
```

TRUNCATE – deleting the contents of the database object (data is deleted as a whole piece, cannot be deleted by condition)

NAME	BIRTH_DT	ALTER_EGO	RATING
Tony Stark	06-JAN-1966	Iron man	100
Bruce Banner	28-FEB-1969	Hulk	80
Steve Rogers	07-MAR-1921	Captain America	90

TRUNCATE TABLE SUPERHERO;

NAME	BIRTH_DATE	ALTER_EGO	RATING
------	------------	-----------	--------

Drop

```
DROP TABLE SUPERHERO;
```

Deleting an object from the database

NAME	AGE	BIRTH_DT	ALTER_EGO
------	-----	----------	-----------



Data Manipulation Language

SELECT

- Selection of data that meets the specified conditions

INSERT

- Adding new data

UPDATE

- Change existing data

DELETE

- Delete existing data

DML: INSERT



INSERT

```
INTO table_name [(comma_separated_column_names)]  
VALUES (comma_separated_values);
```

INSERT

INSERT

```
    INTO SUPERHERO (NAME, BIRTH_DT, ALTER_EGO, RATING)
VALUES ( 'Natasha Romanoff', '01-AUG-1999', 'Black Widow', 59 );
```

NAME	BIRTH_DT	ALTER_EGO	RATING
Tony Stark	06-JAN-1966	Iron man	100
Bruce Banner	28-FEB-1969	Hulk	80
Steve Rogers	07-MAR-1921	Captain America	90
Natasha Romanoff	01-AUG-1999	Black Widow	59

INSERT: base syntax

```
INSERT INTO table_name [ ( column_name [, ...] ) ]
```

```
    { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) [, ...]  
    | request }
```

```
    [ RETURNING * | result_expression [ [ AS ] result_name ] [, ...] ]
```


INSERT: syntax elements

- `table_name` - the name of an existing table (possibly supplemented with a schema)
- `column_name` - this column name can, if necessary, be supplemented with the name of a nested field or an index in the array
- `expression` - the expression or value that will be assigned to the corresponding column
- `query` - a query (SELECT statement) that will return rows to add to the table
- `result_expression` is an expression that will be evaluated and returned by the INSERT command after each row is added or modified. This expression can use the names of any columns in the `table_name` table. To get all columns, just write *

INSERT: syntax elements

- `result_name` - the name assigned to the returned column
- Column names can be arranged in any order. The names of columns with default values can be omitted when enumerating or the `DEFAULT` parameter can be explicitly specified. If you do not specify a list of columns, but do not provide all the values for the columns in the insertion, then the insertion will occur sequentially in all “cells” of the row, and therefore the system will generate an error either due to a mismatch of data type, or due to insufficient data added to data string

INSERT: examples

```
CREATE TABLE example_table (  
  id SERIAL PRIMARY KEY,  
  name TEXT DEFAULT 'Unknown',  
  age INTEGER DEFAULT 18,  
  email TEXT DEFAULT 'example@example.com'  
);  
SELECT * FROM example_table;
```

id	name	age	email
----	------	-----	-------

INSERT: examples

1) INSERT INTO example_table (name) VALUES ('John');

2) INSERT INTO example_table (name, age) VALUES ('Alice', DEFAULT);

3) INSERT INTO example_table DEFAULT VALUES;

4) INSERT INTO example_table (name, age, email) VALUES

 ('Alice', 25, 'alice@example.com'),

 ('Bob', 30, 'bob@example.com'),

 ('Charlie', DEFAULT, 'charlie@example.com');

5) INSERT INTO example_table (age, name) VALUES (12+7+2, 'Pete');

id	name	age	email
1	John	18	example@example.com
2	Alice	18	example@example.com
3	Unknown	18	example@example.com
4	Alice	25	alice@example.com
5	Bob	30	bob@example.com
6	Charlie	18	charlie@example.com
7	Pete	21	example@example.com

INSERT: examples

1) INSERT INTO example_table
DEFAULT VALUES
RETURNING *;

id	name	age	email
8	Unknown	18	example@example.com

2) INSERT INTO example_table (age)
VALUES (DEFAULT), (DEFAULT),
(DEFAULT)
RETURNING *;

id	name	age	email
9	Unknown	18	example@example.com
10	Unknown	18	example@example.com
11	Unknown	18	example@example.com

INSERT + SELECT

```
INSERT INTO products (product_no, name, price)
    SELECT product_no, name, price FROM new_products
    WHERE release_date = 'today';
```

INSERT for table copying

```
CREATE TABLE example_table_2 (  
  id SERIAL PRIMARY KEY,  
  name TEXT DEFAULT 'Unknown',  
  age INTEGER DEFAULT 18,  
  email TEXT DEFAULT  
'example@example.com'  
);
```

```
INSERT INTO example_table_2 SELECT  
* FROM example_table;
```

id	name	age	email
1	John	18	example@example.com
2	Alice	18	example@example.com
3	Unknown	18	example@example.com
4	Alice	25	alice@example.com
5	Bob	30	bob@example.com
6	Charlie	18	charlie@example.com
7	Pete	21	example@example.com
8	Unknown	18	example@example.com
9	Unknown	18	example@example.com
10	Unknown	18	example@example.com
11	Unknown	18	example@example.com

DML: UPDATE



```
UPDATE table_name  
    SET update_assignment_comma_list  
[WHERE conditional_expression];
```


Update

```
UPDATE SUPERHERO
  SET BIRTH_DT = '01-AUG-1940'
 WHERE NAME = 'Natasha Romanoff';
```

NAME	BIRTH_DATE	ALTER_EGO	RATING
Tony Stark	06-JAN-1966	Iron man	100
Bruce Banner	28-FEB-1969	Hulk	80
Steve Rogers	07-MAR-1921	Captain America	90
Natasha Romanoff	01-AUG-1940	Black Widow	59

UPDATE: syntax

```
UPDATE table_name [ * ]  
SET { column_name = { expression | DEFAULT } |  
    ( column_name [, ...] ) = ( nested_SELECT )  
    } [, ...]  
[ FROM element_FROM [, ...] ]  
[ WHERE condition ]  
[ RETURNING * | result_expression [ [ AS ] result_name ] [, ...] ]
```

UPDATE: syntax

- `nested_SELECT` is a subSELECT query that returns as many output columns as are listed in the parenthetical list of columns that precede it. This subquery should return a maximum of one row. If it produces a single row, the column values in it are assigned to the target columns; if it does not return a row, the target columns are assigned NULL. This subquery can access the previous values of the currently modified row in the table
- `element_FROM` is a table expression that allows access to columns of other tables in the WHERE clause and new data expressions. It uses the same syntax as the FROM clause of the SELECT statement
- `condition` - an expression that returns a boolean value. Only those stocks for which this expression returns true will be changed.

UPDATE: examples

UPDATE

example_table

SET age = 25

WHERE id = 1;

id	name	age	email
1	John	18	example@example.com
2	Alice	18	example@example.com
3	Unknown	18	example@example.com
4	Alice	25	alice@example.com
5	Bob	30	bob@example.com
6	Charlie	18	charlie@example.com
7	Pete	21	example@example.com
8	Unknown	18	example@example.com
9	Unknown	18	example@example.com
10	Unknown	18	example@example.com
11	Unknown	18	example@example.com

id	name	age	email
1	John	25	example@example.com
2	Alice	18	example@example.com
3	Unknown	18	example@example.com
4	Alice	25	alice@example.com
5	Bob	30	bob@example.com
6	Charlie	18	charlie@example.com
7	Pete	21	example@example.com
8	Unknown	18	example@example.com
9	Unknown	18	example@example.com
10	Unknown	18	example@example.com
11	Unknown	18	example@example.com



UPDATE: examples

```
UPDATE example_table  
  SET (age, name, email)  
    = (20, 'Will',  
      'will@mail.com')  
  WHERE id = 8;
```

id	name	age	email
1	John	18	example@example.com
2	Alice	18	example@example.com
3	Unknown	18	example@example.com
4	Alice	25	alice@example.com
5	Bob	30	bob@example.com
6	Charlie	18	charlie@example.com
7	Pete	21	example@example.com
8	Unknown	18	example@example.com
9	Unknown	18	example@example.com
10	Unknown	18	example@example.com
11	Unknown	18	example@example.com

id	name	age	email
1	John	25	example@example.com
2	Alice	18	example@example.com
3	Unknown	18	example@example.com
4	Alice	25	alice@example.com
5	Bob	30	bob@example.com
6	Charlie	18	charlie@example.com
7	Pete	21	example@example.com
8	Will	20	will@mail.com
9	Unknown	18	example@example.com
10	Unknown	18	example@example.com
11	Unknown	18	example@example.com



UPDATE: examples

```
UPDATE example_table
  SET (age, name, email) = (
    SELECT age+3,
    name, email
      FROM
    example_table
    WHERE id = 4
  )
  WHERE id = 11;
```

id	name	age	email
1	John	18	example@example.com
2	Alice	18	example@example.com
3	Unknown	18	example@example.com
4	Alice	25	alice@example.com
5	Bob	30	bob@example.com
6	Charlie	18	charlie@example.com
7	Pete	21	example@example.com
8	Unknown	18	example@example.com
9	Unknown	18	example@example.com
10	Unknown	18	example@example.com
11	Unknown	18	example@example.com

id	name	age	email
1	John	25	example@example.com
2	Alice	18	example@example.com
3	Unknown	18	example@example.com
4	Alice	25	alice@example.com
5	Bob	30	bob@example.com
6	Charlie	18	charlie@example.com
7	Pete	21	example@example.com
8	Will	20	will@mail.com
9	Unknown	18	example@example.com
10	Unknown	18	example@example.com
11	Alice	28	alice@example.com



UPDATE: Example with FROM... WHERE

Changing the contact name in the account table (this should be the name of the assigned sales manager):

```
UPDATE accounts SET
    contact_first_name = first_name,
    contact_last_name = last_name
FROM employees
WHERE employees.id = accounts.sales_person;
```

DML: DELETE

DELETE

FROM table_name

[**WHERE** conditional_expression];



Delete

DELETE

FROM SUPERHERO

WHERE NAME = 'Bruce Banner';

NAME	BIRTH_DT	ALTER_EGO	RATING
Tony Stark	06-JAN-1966	Iron man	100
Bruce Banner	28-FEB-1969	Hulk	80
Steve Rogers	07-MAR-1921	Captain America	90
Natasha Romanoff	01-AUG-1940	Black Widow	59

DELETE: example

DELETE FROM example_table_2
WHERE id = 11
RETURNING *



id	name	age	email
11	Alice	28	alice@example.com

id	name	age	email
1	John	25	example@example.com
2	Alice	18	example@example.com
3	Unknown	18	example@example.com
4	Alice	25	alice@example.com
5	Bob	30	bob@example.com
6	Charlie	18	charlie@example.com
7	Pete	21	example@example.com
8	Will	20	will@mail.com
9	Unknown	18	example@example.com
10	Unknown	18	example@example.com
11	Alice	28	alice@example.com

id	name	age	email
1	John	25	example@example.com
2	Alice	18	example@example.com
3	Unknown	18	example@example.com
4	Alice	25	alice@example.com
5	Bob	30	bob@example.com
6	Charlie	18	charlie@example.com
7	Pete	21	example@example.com
8	Will	20	will@mail.com
9	Unknown	18	example@example.com
10	Unknown	18	example@example.com



DELETE vs TRUNCATE vs DROP

DELETE	TRUNCATE	DROP
"Delete" line by line	Deleting the entire block at once	Deleting the entire table
You can set conditions for deletion	Conditions for deletion cannot be set	Conditions for deletion cannot be set
Ability to roll back changes	There is no possibility of rolling back changes	There is no possibility of rolling back changes
Physically, the lines are not deleted, only marked "invisible" from a certain moment VACUUM is needed for removal	Deleting data and freeing up disk space occurs immediately	Deleting data and freeing up disk space occurs immediately

DML: SELECT



```
SELECT [DISTINCT] select_item_comma_list  
  FROM table_reference_comma_list  
[WHERE conditional_expression]  
[GROUP BY column_name_comma_list]  
[HAVING conditional_expression]  
[ORDER BY order_item_comma_list];
```

SELECT: syntax

SELECT [ALL | DISTINCT [ON (expression [, ...])]]
[* | expression [[AS] result_name] [, ...]]
[FROM element_FROM [, ...]]
[WHERE condition]
[GROUP BY group_item [, ...]]
[HAVING condition]
[{ UNION | INTERSECT | EXCEPT } selection]
[ORDER BY expression [ASC | [, ...]]]
[LIMIT { number | ALL }]
[OFFSET start [ROW | ROWS]]

list of samples

table expression

sorting

SELECT syntax

SELECT returns a set consisting of at least one value. For simplicity at this point, consider that SELECT always returns a table that has at least one row with at least one cell. In the latter case, SELECT is also said to return a scalar value.

The select list specifies the columns that will appear in the resulting relation. It is also possible to specify an operation on each column value.

The syntax for column references is variable:

```
SELECT a, b, c FROM ...
```

```
SELECT tbl1.a, tbl2.a, tbl1.b FROM ...
```

```
SELECT tbl1.*, tbl2.a FROM ...
```

SELECT syntax

- In the selection list, you can assign “aliases” to the columns of the resulting table:

`SELECT a AS value, b + c AS sum FROM ...`

- The word AS can usually be omitted, but in some cases, when the desired column name is the same as a PostgreSQL keyword, you need to write AS or enclose the column name in quotes to avoid ambiguity.
- A similar rule applies to tables listed after the FROM clause, but:
 - the alias becomes the new table name within the current query, i.e., after assigning an alias, the original table name cannot be used elsewhere in the query
 - they are necessary when a table is joined to itself

SELECT syntax

- The result of evaluating a table expression is a (virtual) table, which can also consist of just one cell (scalar)
- You can omit the table expression and use SELECT to return the result of the expression evaluation: `SELECT 3*4`; `SELECT 5!=4`;
- The FROM clause creates a table from one or more table references, separated by commas:

FROM table1 [, table_2, ..., table_N]

- A table reference (table_N in the example) could be:
- table name (possibly with schema name),
- derived table, for example:
 - subquery
 - joining tables
 - a complex combination of these options.
- If multiple references are listed in the FROM clause, a cross join is applied to them as the result of the Cartesian product of the relations

SELECT syntax

- The basic syntax for joining tables is as follows:

T1 connection_type T2 [connection_condition]

- Junctions of any type can be nested or combined: both T1 and T2 can be the results of a join
- Connection types in PostgreSQL:
 - CROSS JOIN
 - String matching joins:
 - INNER JOIN
 - LEFT OUTER JOIN
 - RIGHT OUTER JOIN
 - FULL OUTER JOIN

SELECT: JOINs

T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
ON boolean_expression

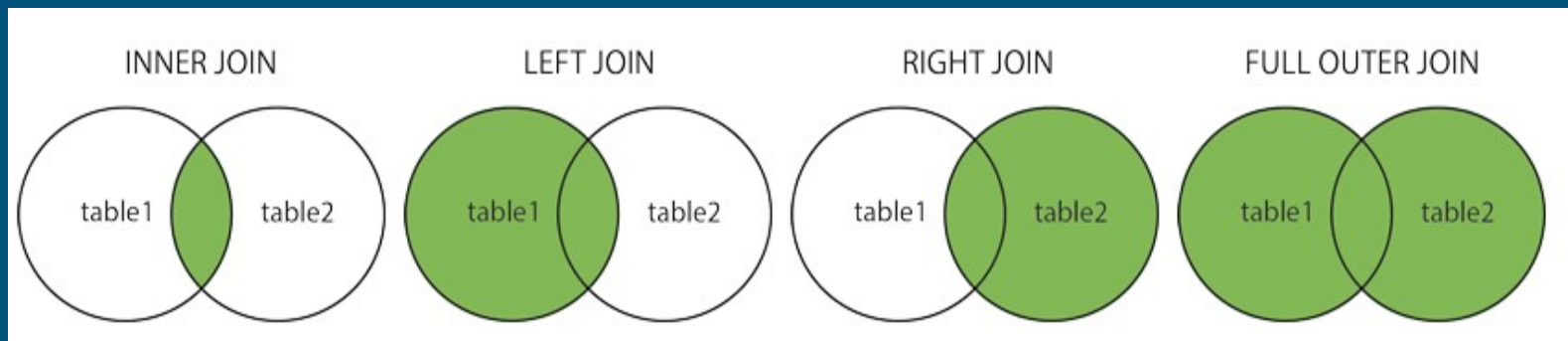
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
USING (join column list)

T1 NATURAL { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2

SELECT: JOINS

- The words INNER and OUTER are optional in all forms. By default, an INNER is assumed, and when LEFT, RIGHT and FULL are specified, an outer join is assumed.
- The join condition is specified in the ON or USING clause, or is implicitly specified by the NATURAL keyword. This condition determines which rows of the two source tables are considered to "match" each other (this is discussed in detail below).

JOIN



CROSS JOIN

CROSS JOIN = Cartesian product of two tables

```
SELECT column_name(s)  
FROM table1  
CROSS JOIN table2
```

CROSS JOIN

SELECT * FROM employees CROSS JOIN departments;

employee_id	employee_name	department_id
1	John Doe	1
2	Jane Smith	2
3	Alice Johnson	1

department_id	department_name
1	Engineering
2	Sales



employee_id	employee_name	department_id	department_id-2	department_name
1	John Doe	1	1	Engineering
1	John Doe	1	2	Sales
2	Jane Smith	2	1	Engineering
2	Jane Smith	2	2	Sales
3	Alice Johnson	1	1	Engineering
3	Alice Johnson	1	2	Sales

CROSS JOIN

SELECT * FROM employees CROSS JOIN departments;
SELECT * FROM employees INNER JOIN departments ON TRUE;

employee_id	employee_name	department_id
1	John Doe	1
2	Jane Smith	2
3	Alice Johnson	1



department_id	department_name
1	Engineering
2	Sales

employee_id	employee_name	department_id	department_id-2	department_name
1	John Doe	1	1	Engineering
1	John Doe	1	2	Sales
2	Jane Smith	2	1	Engineering
2	Jane Smith	2	2	Sales
3	Alice Johnson	1	1	Engineering
3	Alice Johnson	1	2	Sales

INNER JOIN

INNER JOIN: Returns records that have matching values in both tables

INNER JOIN Syntax

```
SELECT column_name(s)  
FROM table1  
INNER JOIN table2 ON table1.column_name = table2.column_name
```


INNER JOIN

SELECT *

FROM employees e

INNER JOIN departments
d

ON e.department_id
=d.department_id;

employee_id	employee_name	department_id
1	John Doe	1
2	Jane Smith	2
3	Alice Johnson	1

department_id	department_name
1	Engineering
2	Sales



employee_id	employee_name	department_id	department_id-2	department_name
1	John Doe	1	1	Engineering
2	Jane Smith	2	2	Sales
3	Alice Johnson	1	1	Engineering

INNER JOIN

SELECT *

FROM employees e

INNER JOIN departments
d

ON e.department_id = 1;

employee_id	employee_name	department_id
1	John Doe	1
2	Jane Smith	2
3	Alice Johnson	1

department_id	department_name
1	Engineering
2	Sales



employee_id	employee_name	department_id	department_id-2	department_name
1	John Doe	1	1	Engineering
3	Alice Johnson	1	1	Engineering
1	John Doe	1	2	Sales
3	Alice Johnson	1	2	Sales

INNER JOIN

SELECT *

FROM employees e

INNER JOIN departments
d

ON e.department_id =
d.department_id

AND e.department_id =
1;

employee_id	employee_name	department_id
1	John Doe	1
2	Jane Smith	2
3	Alice Johnson	1

department_id	department_name
1	Engineering
2	Sales



employee_id	employee_name	department_id	department_id-2	department_name
1	John Doe	1	1	Engineering
3	Alice Johnson	1	1	Engineering

LEFT JOIN

LEFT (OUTER) JOIN: Returns all records from the left table and matching records from the table on the right

LEFT JOIN Syntax

```
SELECT column_name(s)  
FROM table1  
LEFT JOIN table2 ON table1.column_name = table2.column_name;
```

LEFT JOIN

```
SELECT * FROM employees  
LEFT JOIN departments  
ON  
employees.department_id =  
departments.department_id;
```

employee_id	employee_name	department_id
1	John Doe	1
2	Jane Smith	1
3	Alice Johnson	2
4	Emily Brown	2
5	David Wilson	4

department_id	department_name
1	Engineering
2	Sales
3	Marketing



employee_id	employee_name	department_id	department_id-2	department_name
1	John Doe	1	1	Engineering
2	Jane Smith	1	1	Engineering
3	Alice Johnson	2	2	Sales
4	Emily Brown	2	2	Sales
5	David Wilson	4		

RIGHT JOIN

RIGHT (OUTER) JOIN: Returns all records from the table on the right, and matching records from the left table

RIGHT JOIN Syntax

```
SELECT column_name(s)  
FROM table1  
RIGHT JOIN table2 ON table1.column_name = table2.column_name;
```

RIGHT JOIN

```
SELECT * FROM employees  
RIGHT JOIN departments  
ON employees.department_id  
=  
departments.department_id;
```

employee_id	employee_name	department_id
1	John Doe	1
2	Jane Smith	1
3	Alice Johnson	2
4	Emily Brown	2
5	David Wilson	4



department_id	department_name
1	Engineering
2	Sales
3	Marketing

employee_id	employee_name	department_id	department_id-2	department_name
2	Jane Smith	1	1	Engineering
1	John Doe	1	1	Engineering
4	Emily Brown	2	2	Sales
3	Alice Johnson	2	2	Sales
			3	Marketing

FULL JOIN

FULL (OUTER) JOIN: Returns all records when there is a match in the left or right table

FULL OUTER JOIN Syntax

```
SELECT column_name(s)  
FROM table1  
FULL OUTER JOIN table2 ON table1.column_name = table2.column_name;
```


FULL JOIN

SELECT * FROM

employees

FULL JOIN departments

ON employees.department_id

=

departments.department_id

employee_id	employee_name	department_id
1	John Doe	1
2	Jane Smith	1
3	Alice Johnson	2
4	Emily Brown	2
5	David Wilson	4

department_id	department_name
1	Engineering
2	Sales
3	Marketing



employee_id	employee_name	department_id	department_id-2	department_name
1	John Doe	1	1	Engineering
2	Jane Smith	1	1	Engineering
3	Alice Johnson	2	2	Sales
4	Emily Brown	2	2	Sales
5	David Wilson	4		
			3	Marketing

SELECT syntax: WHERE

WHERE `constraint_condition`

`constraint_condition` - any value expression that produces a boolean result

```
SELECT ... FROM fdt WHERE c1 > 5
```

```
SELECT ... FROM fdt WHERE c1 IN (1, 2, 3)
```

```
SELECT ... FROM fdt WHERE c1 IN (SELECT c1 FROM t2)
```

SELECT syntax: GROUP BY + HAVING

```
SELECT selection_list  
  FROM...  
  [WHERE...]  
  GROUP BY group_item [, group_item ]...  
    HAVING boolean_expression
```

SELECT syntax: GROUP BY + HAVING

- In the SQL standard, GROUP BY can only group by columns of the source table, but the PostgreSQL extension allows you to use GROUP BY more flexibly. The GROUP BY clause collects into one row all selected rows that produce the same values for the grouping expressions. The expression inside the group_element can be the name of the input column, or the name or sequence number of the output column (from the list of SELECT elements), or an arbitrary value calculated from the values of the input columns. In case of ambiguity, the name in GROUP BY will be treated as the name of the input column, not the output column

SELECT syntax: GROUP BY + HAVING

- Aggregate functions, when used, are evaluated over all rows that make up each group and ultimately produce a separate value for each group
- When a GROUP BY clause or any aggregate function is present in a query, expressions in the SELECT list generally cannot access non-groupable columns because otherwise the non-groupable column would have to return more than one possible value
- If the table has been grouped using GROUP BY, but only some of the groups are of interest, you can filter them using the HAVING clause, which acts similar to WHERE

SELECT syntax: GROUP BY + HAVING

SELECT department FROM
new_employees GROUP BY
department;

employee_id	name	age	date_of_employment	salary	department	position
1	Иван Иванов	35	15.01.2020	50000.00	ИТ	программист
2	Екатерина Смирнова	28	20.05.2018	60000.00	Маркетинг	ведущий маркетолог
3	Михаил Иванов	42	10.11.2015	70000.00	Финансы	руководитель отдела
4	Анна Сергеева	30	03.09.2019	50000.00	Кадры	специалист
5	Дмитрий Петров	45	22.07.2012	70000.00	ИТ	руководитель отдела
6	Ольга Кузнецова	38	12.03.2017	60000.00	Маркетинг	руководитель отдела
7	Александр Сидоров	33	18.08.2021	50000.00	Финансы	бухгалтер
8	Наталья Иванова	31	25.04.2016	50000.00	Кадры	специалист
9	Павел Морозов	40	30.10.2014	50000.00	ИТ	программист
10	Анастасия Лебедева	29	01.02.2022	50000.00	Маркетинг	маркетолог


SELECT salary FROM
new_employees GROUP BY
salary;

SELECT syntax: GROUP BY + HAVING

SELECT department FROM
new_employees GROUP BY
department;

SELECT salary FROM
new_employees GROUP BY
salary;

employee_id	name	age	date_of_employment	salary	department	position
1	Иван Иванов	35	15.01.2020	50000.00	ИТ	программист
2	Екатерина Смирнова	28	20.05.2018	60000.00	Маркетинг	ведущий маркетолог
3	Михаил Иванов	42	10.11.2015	70000.00	Финансы	руководитель отдела
4	Анна Сергеева	30	03.09.2019	50000.00	Кадры	специалист
5	Дмитрий Петров	45	22.07.2012	70000.00	ИТ	руководитель отдела
6	Ольга Кузнецова	38	12.03.2017	60000.00	Маркетинг	руководитель отдела
7	Александр Сидоров	33	18.08.2021	50000.00	Финансы	бухгалтер
8	Наталья Иванова	31	25.04.2016	50000.00	Кадры	специалист
9	Павел Морозов	40	30.10.2014	50000.00	ИТ	программист
10	Анастасия Лебедева	29	01.02.2022	50000.00	Маркетинг	маркетолог



salary
60000.00
50000.00
70000.00



department
Кадры
Маркетинг
Финансы
ИТ

SELECT syntax: GROUP BY + HAVING

```
SELECT department,  
salary  
FROM new_employees  
GROUP BY  
    salary  
    department;
```

employee_id	name	age	date_of_employment	salary	department	position
1	Иван Иванов	35	15.01.2020	50000.00	ИТ	программист
2	Екатерина Смирнова	28	20.05.2018	60000.00	Маркетинг	ведущий маркетолог
3	Михаил Иванов	42	10.11.2015	70000.00	Финансы	руководитель отдела
4	Анна Сергеева	30	03.09.2019	50000.00	Кадры	специалист
5	Дмитрий Петров	45	22.07.2012	70000.00	ИТ	руководитель отдела
6	Ольга Кузнецова	38	12.03.2017	60000.00	Маркетинг	руководитель отдела
7	Александр Сидоров	33	18.08.2021	50000.00	Финансы	бухгалтер
8	Наталья Иванова	31	25.04.2016	50000.00	Кадры	специалист
9	Павел Морозов	40	30.10.2014	50000.00	ИТ	программист
10	Анастасия Лебедева	29	01.02.2022	50000.00	Маркетинг	маркетолог



department	salary
Финансы	70000.00
Маркетинг	60000.00
Финансы	50000.00
ИТ	70000.00
ИТ	50000.00
Кадры	50000.00
Маркетинг	50000.00

The order in which the columns are listed does not affect the result!

SELECT syntax: GROUP BY + HAVING

```
SELECT department,  
salary  
FROM new_employees  
GROUP BY  
salary,  
department  
HAVING salary >  
60000;
```

employee_id	name	age	date_of_employment	salary	department	position
1	Иван Иванов	35	15.01.2020	50000.00	ИТ	программист
2	Екатерина Смирнова	28	20.05.2018	60000.00	Маркетинг	ведущий маркетолог
3	Михаил Иванов	42	10.11.2015	70000.00	Финансы	руководитель отдела
4	Анна Сергеева	30	03.09.2019	50000.00	Кадры	специалист
5	Дмитрий Петров	45	22.07.2012	70000.00	ИТ	руководитель отдела
6	Ольга Кузнецова	38	12.03.2017	60000.00	Маркетинг	руководитель отдела
7	Александр Сидоров	33	18.08.2021	50000.00	Финансы	бухгалтер
8	Наталья Иванова	31	25.04.2016	50000.00	Кадры	специалист
9	Павел Морозов	40	30.10.2014	50000.00	ИТ	программист
10	Анастасия Лебедева	29	01.02.2022	50000.00	Маркетинг	маркетолог



department	salary
ИТ	70000.00
Финансы	70000.00

SELECT syntax: SORT

The syntax for a sort expression is as follows:

```
SELECT selection_list
```

```
    FROM table_expression
```

```
    ORDER BY sort_expression1 [ASC | DESC] [NULLS { FIRST | LAST }]
```

```
           [, sort_expression2 [ASC | DESC] [NULLS { FIRST | LAST }]] ...]
```

Sort expressions can be any expressions allowed in the query's select list

SELECT syntax: SORT

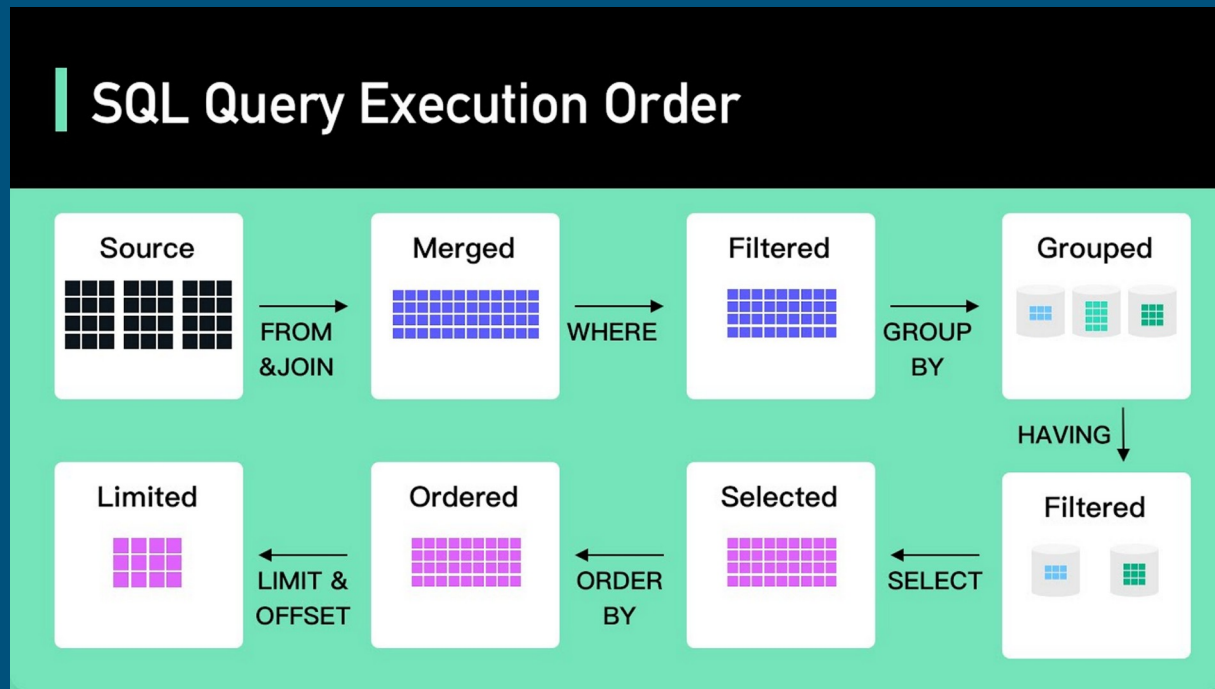
- When multiple expressions are specified, subsequent values allow you to sort rows that match all previous values. Each expression can be supplemented with the keywords ASC or DESC, which select sorting in ascending or descending order, respectively. The default is ascending order (ASC)
- To determine the location of NULL values, you can use the NULLS FIRST and NULLS LAST hints, which place NULL values before or after non-NULL values, respectively. By default, NULL values are considered to be greater than any other, that is, NULLS FIRST is assumed for DESC order and NULLS LAST otherwise

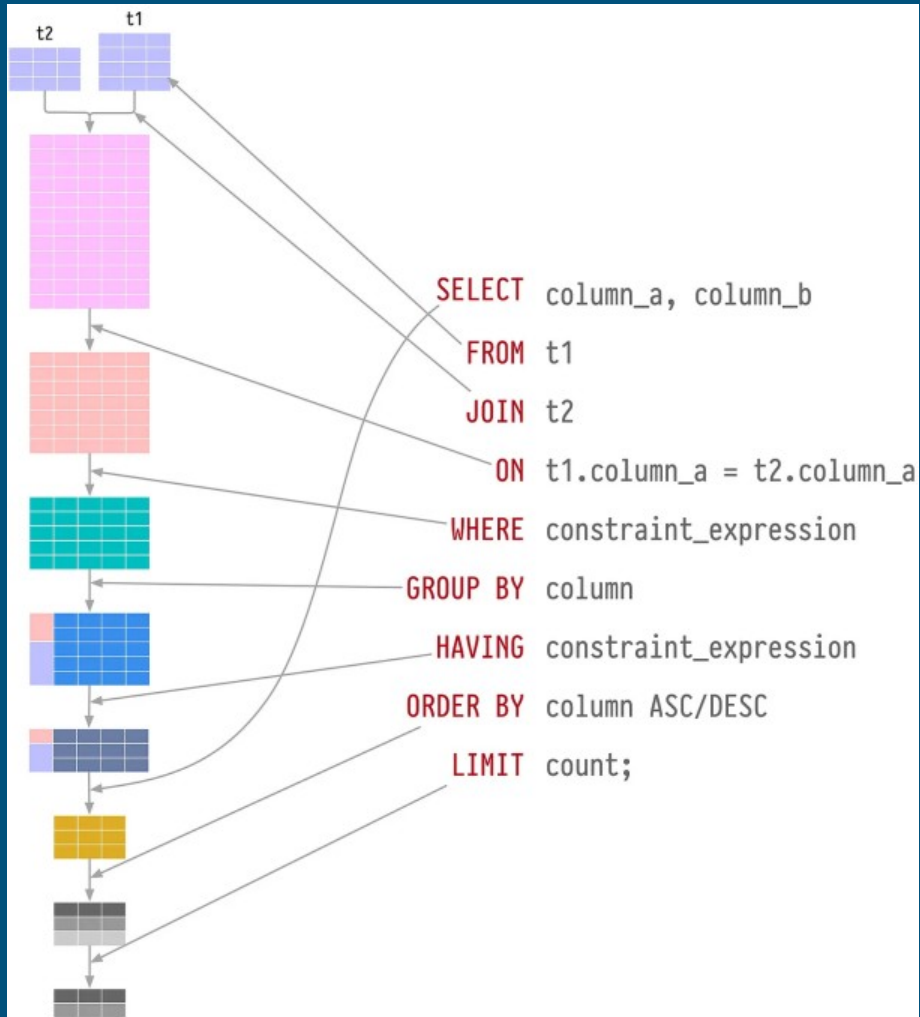
SQL: query order

- SQL is a declarative language, in which you describe what you want to achieve, not how to do it
- SQL is a semantically redundant language!
- Although the language developers tried to reduce the entry threshold for beginners working with SQL as much as possible, SQL queries are not trivial
- The process or command of obtaining data from a database is called a query. In SQL, queries are formulated using the SELECT command
- Let's understand how a SELECT query works

The order of execution of the request

1. FROM & JOIN
2. WHERE
3. GROUP BY
4. HAVING
5. SELECT
6. ORDER BY
7. LIMIT & OFFSET





SELECT example

We want to know the name of only two cities other than San Bruno that have two or more residents.
We also want to get the result sorted alphabetically



CITIZEN	
Name	City_id
Andre	3
Rachel	2
Jenny	1
Dough	3
Kevin	1
Sarah	2
Trevor	3
Al	1
Yung	1

CITY	
City_id	City_name
1	Palo Alto
2	Sunnyvale
3	San Bruno

SELECT example

We want to know the name of only two cities other than San Bruno that have two or more residents.
We also want to get the result sorted alphabetically



CITIZEN	
Name	City_id
Andre	3
Rachel	2
Jenny	1
Dough	3
Kevin	1
Sarah	2
Trevor	3
Al	1
Yung	1

CITY	
City_id	City_name
1	Palo Alto
2	Sunnyvale
3	San Bruno

SELECT example

```
SELECT city.city_name AS "City"  
FROM citizen  
JOIN city  
ON citizen.city_id = city.city_id  
WHERE city.city_name != 'San Bruno'  
GROUP BY city.city_name  
HAVING COUNT(*) >= 2  
ORDER BY city.city_name ASC  
LIMIT 2
```

CITIZEN	
Name	City_id
Andre	3
Rachel	2
Jenny	1
Dough	3
Kevin	1
Sarah	2
Trevor	3
Al	1
Yung	1

CITY	
City_id	City_name
1	Palo Alto
2	Sunnyvale
3	San Bruno

SELECT example

```
SELECT city.city_name AS "City"  
FROM citizen  
JOIN city  
ON citizen.city_id = city.city_id  
WHERE city.city_name != 'San Bruno'  
GROUP BY city.city_name  
HAVING COUNT(*) >= 2  
ORDER BY city.city_name ASC  
LIMIT 2
```

Name	City_id	City_id	City_name
Andre	3	1	Palo Alto
Andre	3	2	Sunnyvale
Andre	3	3	San Bruno
Rachel	2	1	Palo Alto
Rachel	2	2	Sunnyvale
Rachel	2	3	San Bruno
Jenny	1	1	Palo Alto
Jenny	1	2	Sunnyvale
Jenny	1	3	San Bruno
Dough	3	1	Palo Alto
Dough	3	2	Sunnyvale
Dough	3	3	San Bruno
Kevin	1	1	Palo Alto
Kevin	1	2	Sunnyvale
Kevin	1	3	San Bruno
Sarah	2	1	Palo Alto
Sarah	2	2	Sunnyvale
Sarah	2	3	San Bruno
Trevor	3	1	Palo Alto
Trevor	3	2	Sunnyvale
Trevor	3	3	San Bruno
Al	1	1	Palo Alto
Al	1	2	Sunnyvale
Al	1	3	San Bruno
Yung	1	1	Palo Alto
Yung	1	2	Sunnyvale
Yung	1	3	San Bruno

SELECT example

```
SELECT city.city_name AS "City"  
FROM citizen  
JOIN city  
ON citizen.city_id = city.city_id  
WHERE city.city_name != 'San Bruno'  
GROUP BY city.city_name  
HAVING COUNT(*) >= 2  
ORDER BY city.city_name ASC  
LIMIT 2
```

Name	City_id	City_name
Andre	3	San Bruno
Rachel	2	Sunnyvale
Jenny	1	Palo Alto
Dough	3	San Bruno
Kevin	1	Palo Alto
Sarah	2	Sunnyvale
Trevor	3	San Bruno
Al	1	Palo Alto
Yung	1	Palo Alto

SELECT example

```
SELECT city.city_name AS "City"  
FROM citizen  
JOIN city  
ON citizen.city_id = city.city_id  
WHERE city.city_name != 'San Bruno'  
GROUP BY city.city_name  
HAVING COUNT(*) >= 2  
ORDER BY city.city_name ASC  
LIMIT 2
```

Name	City_id	City_name
Rachel	2	Sunnyvale
Jenny	1	Palo Alto
Kevin	1	Palo Alto
Sarah	2	Sunnyvale
Al	1	Palo Alto
Yung	1	Palo Alto

SELECT example

```
SELECT city.city_name AS "City"  
FROM citizen  
JOIN city  
ON citizen.city_id = city.city_id  
WHERE city.city_name != 'San Bruno'  
GROUP BY city.city_name  
HAVING COUNT(*) >= 2  
ORDER BY city.city_name ASC  
LIMIT 2
```

Name	City_id	City_name
Rachel	2	Sunnyvale
Sarah	2	
Jenny	1	Palo Alto
Kevin	1	
Al	1	
Yung	1	

SELECT example

```
SELECT city.city_name AS "City"  
FROM citizen  
JOIN city  
ON citizen.city_id = city.city_id  
WHERE city.city_name != 'San Bruno'  
GROUP BY city.city_name  
HAVING COUNT(*) >= 2  
ORDER BY city.city_name ASC  
LIMIT 2
```

Name	City_id	City_name
Rachel	2	Sunnyvale
Sarah	2	
Jenny	1	Palo Alto
Kevin	1	
Al	1	
Yung	1	

SELECT example

```
SELECT city.city_name AS "City"  
FROM citizen  
JOIN city  
ON citizen.city_id = city.city_id  
WHERE city.city_name != 'San Bruno'  
GROUP BY city.city_name  
HAVING COUNT(*) >= 2  
ORDER BY city.city_name ASC  
LIMIT 2
```

City
Sunnyvale
Palo Alto

SELECT example

```
SELECT city.city_name AS "City"  
FROM citizen  
JOIN city  
ON citizen.city_id = city.city_id  
WHERE city.city_name != 'San Bruno'  
GROUP BY city.city_name  
HAVING COUNT(*) >= 2  
ORDER BY city.city_name ASC  
LIMIT 2
```

City
Palo Alto
Sunnyvale

Aggregating functions

- `count()` - the number of records with a known value. If you need to count the number of unique values, you can use `count(DISTINCT field_nm)`
- `max()` - the largest of all selected field values (`!= NULL`)
- `min()` - the smallest of all selected field values (`!= NULL`)
- `sum()` - sum of all selected field values (`!= NULL`)
- `avg()` - average of all selected field values (`!= NULL`)

Count

SELECT count(*) as
total_rows
FROM new_employees;

total_rows

10



employee_id	name	age	date_of_employment	salary	department	position
1	Иван Иванов	35	15.01.2020	50000.00	ИТ	программист
2	Екатерина Смирнова	28	20.05.2018	60000.00	Маркетинг	ведущий маркетолог
3	Михаил Иванов	42	10.11.2015	70000.00	Финансы	руководитель отдела
4	Анна Сергеева	30	03.09.2019	50000.00	Кадры	специалист
5	Дмитрий Петров	45	22.07.2012	70000.00	ИТ	руководитель отдела
6	Ольга Кузнецова	38	12.03.2017	60000.00	Маркетинг	руководитель отдела
7	Александр Сидоров	33	18.08.2021	50000.00	Финансы	бухгалтер
8	Наталья Иванова	31	25.04.2016	50000.00	Кадры	специалист
9	Павел Морозов	40	30.10.2014	50000.00	ИТ	программист
10	Анастасия Лебедева	29	01.02.2022	50000.00	Маркетинг	маркетолог

Sum

```
SELECT sum(salary) as  
total_payments  
FROM new_employees;
```

total_payments
560000.00



employee_id	name	age	date_of_employment	salary	department	position
1	Иван Иванов	35	15.01.2020	50000.00	ИТ	программист
2	Екатерина Смирнова	28	20.05.2018	60000.00	Маркетинг	ведущий маркетолог
3	Михаил Иванов	42	10.11.2015	70000.00	Финансы	руководитель отдела
4	Анна Сергеева	30	03.09.2019	50000.00	Кадры	специалист
5	Дмитрий Петров	45	22.07.2012	70000.00	ИТ	руководитель отдела
6	Ольга Кузнецова	38	12.03.2017	60000.00	Маркетинг	руководитель отдела
7	Александр Сидоров	33	18.08.2021	50000.00	Финансы	бухгалтер
8	Наталья Иванова	31	25.04.2016	50000.00	Кадры	специалист
9	Павел Морозов	40	30.10.2014	50000.00	ИТ	программист
10	Анастасия Лебедева	29	01.02.2022	50000.00	Маркетинг	маркетолог

Max

```
SELECT  
max(date_of_employment)  
AS last_employee  
FROM new_employees;
```

last_employee
01.02.2022



employee_id	name	age	date_of_employment	salary	department	position
1	Иван Иванов	35	15.01.2020	50000.00	ИТ	программист
2	Екатерина Смирнова	28	20.05.2018	60000.00	Маркетинг	ведущий маркетолог
3	Михаил Иванов	42	10.11.2015	70000.00	Финансы	руководитель отдела
4	Анна Сергеева	30	03.09.2019	50000.00	Кадры	специалист
5	Дмитрий Петров	45	22.07.2012	70000.00	ИТ	руководитель отдела
6	Ольга Кузнецова	38	12.03.2017	60000.00	Маркетинг	руководитель отдела
7	Александр Сидоров	33	18.08.2021	50000.00	Финансы	бухгалтер
8	Наталья Иванова	31	25.04.2016	50000.00	Кадры	специалист
9	Павел Морозов	40	30.10.2014	50000.00	ИТ	программист
10	Анастасия Лебедева	29	01.02.2022	50000.00	Маркетинг	маркетолог

Min

```
SELECT  
min(date_of_employment)  
AS first_employee  
FROM new_employees;
```

first_employee

22.07.2012



employee_id	name	age	date_of_employment	salary	department	position
1	Иван Иванов	35	15.01.2020	50000.00	ИТ	программист
2	Екатерина Смирнова	28	20.05.2018	60000.00	Маркетинг	ведущий маркетолог
3	Михаил Иванов	42	10.11.2015	70000.00	Финансы	руководитель отдела
4	Анна Сергеева	30	03.09.2019	50000.00	Кадры	специалист
5	Дмитрий Петров	45	22.07.2012	70000.00	ИТ	руководитель отдела
6	Ольга Кузнецова	38	12.03.2017	60000.00	Маркетинг	руководитель отдела
7	Александр Сидоров	33	18.08.2021	50000.00	Финансы	бухгалтер
8	Наталья Иванова	31	25.04.2016	50000.00	Кадры	специалист
9	Павел Морозов	40	30.10.2014	50000.00	ИТ	программист
10	Анастасия Лебедева	29	01.02.2022	50000.00	Маркетинг	маркетолог

Avg

```
SELECT avg(salary) as  
salary_average  
FROM new_employees;
```

salary_average
56000.0000000000



employee_id	name	age	date_of_employment	salary	department	position
1	Иван Иванов	35	15.01.2020	50000.00	ИТ	программист
2	Екатерина Смирнова	28	20.05.2018	60000.00	Маркетинг	ведущий маркетолог
3	Михаил Иванов	42	10.11.2015	70000.00	Финансы	руководитель отдела
4	Анна Сергеева	30	03.09.2019	50000.00	Кадры	специалист
5	Дмитрий Петров	45	22.07.2012	70000.00	ИТ	руководитель отдела
6	Ольга Кузнецова	38	12.03.2017	60000.00	Маркетинг	руководитель отдела
7	Александр Сидоров	33	18.08.2021	50000.00	Финансы	бухгалтер
8	Наталья Иванова	31	25.04.2016	50000.00	Кадры	специалист
9	Павел Морозов	40	30.10.2014	50000.00	ИТ	программист
10	Анастасия Лебедева	29	01.02.2022	50000.00	Маркетинг	маркетолог