SQL

- SQL — Structured Query Language (is a standard language for interacting with RDBMS)
  - Efficient data storage
  - Fast data retrieval
  - Efficient data management
  - Data sharing

✅ 🚨RDBMS (Relational Database Management System) — where the data is in the tabular form including rows and columns (`postgreSQL` , `MySQL` , `MariaDB` , `oracle Database`)

⏳Document Database — specifically NoSQL databases where the data are stored as a documents (MongoDB, Databricks)

- DDL (Data Definition Language) — deals with creation of the database and modifies

- DML (Data Manipulation Language) — mainly deals with querying the data in the database
- DCL (Data Control Language) — It includes commands like GRANT and REVOKE, which primarily deal with rights, permissions and other control-level management tasks for the database system.

RDBMS (Relational Database Management System) —

- Use of SQL (Structured Query Language) for querying and managing data
- Support for ACID transactions (Atomicity, Consistency, Isolation, Durability)
- Enforcement of data integrity through constraints (e.g., primary keys, foreign keys)
- Ability to establish relationships between tables, enabling complex queries and data retrieval
- Scalability and support for multi-user environments

https://www.youtube.com/watch?v=OqjJjpjDRLc

Different types of SQL —

- Relational: Data is stored in a tabular format using rows for individual records and columns to store attributes or data points related to the records.
- Key-Value: Data is stored in a dictionary format with each item having a key and value.
- Document: Data is stored as documents using JSON, XML, or another format for semi-structured data.
- Graph: Data is stored as a knowledge graph using nodes, edges to define the relationship between these nodes, and properties that store individual data points.

https://www.youtube.com/watch?v=_Ss42Vb1SU4

`SELECT` — extracts data from a database

`UPDATE` — updates the data of the database

`DELETE` — delete data from a database

`INSERT INTO` — Inserts new data into a database

`CREATE DATABASE` — creates a new database

`ALTER DATABASE` — modifies a database

`CREATE TABLE` - creates a new table

`ALTER TABLE` - modifies a table

`DROP TABLE` - deletes a table

`CREATE INDEX` - creates an index (search key)

`DROP INDEX` - deletes an index

SELECT

```
#syntax
Select select_list
from table_name
```

```
select expression as column_alias
```

```
select option1,option2,...
from table_name;
```

like from a “employee” table we can have

```
select age
from employee;
```

- Using SELECT FROM statement to query data from multiple columns

```
select age,
lastname,
firstname
from employee
```

- to read every column of the table

```
select * from employee
```

```
select 1+1;
+-------+
| 1 + 1 |
+-------+
|     2 |
+-------+
1 row in set (0.00 sec)
```

```
select now();
+---------------------+
| NOW()               |
+---------------------+
| 2021-07-26 08:08:02 |
+---------------------+
1 row in set (0.00 sec)
```

```
select concat('billo','','bagge');
+-------------------------+
| CONCAT('John',' ','Doe') |
```

```
+-------------------------+
| John Doe                |
+-------------------------+
1 row in set (0.00 sec)
```

```
SELECT CONCAT('Jane',' ','Doe') AS 'Full name';
+-----------+
| Full name |
+-----------+
| John Doe  |
+-----------+
1 row in set (0.00 sec)
```

SELECT DISTINCT

```
select distinct select_list from table_name
```

```
select distinct option1,option2,... from table_name
```

```
select count(distinct select_list) from table_name
# doesn' work in ms access
```

SELECT ORDER BY

```
select select_list from table_name
order by column1 [ASC|DESC],
         column2 [ASC|DESC],
         .....;
```

You use `ASC` to sort the result set in ascending order and `DESC` to sort the result set in descending order.
By default, the `ORDER BY` clause uses `ASC` if you don’t explicitly specify any option

```
ORDER BY
    column1 ASC,
    column2 DESC;
```

In this case, the `ORDER BY` clause:

- First, sort the result set by the values in the `column1` in ascending order.
- Then, sort the sorted result set by the values in the `column2` in descending order. Note that the order of values in the `column1` will not change in this step, only the order of values in the `column2` changes.

Using the ORDER BY clause to sort a result by am expression

```
SELECT
  orderNumber,
  orderlinenumber,
  quantityOrdered * priceEach
FROM
  orderdetails
ORDER BY
  quantityOrdered * priceEach DESC;

  +-------------+----------------+----------------------------+
| orderNumber | orderlinenumber | quantityOrdered * priceEach |
+-------------+----------------+----------------------------+
|       10403 |              9 |                   11503.14 |
|       10405 |              5 |                   11170.52 |
|       10407 |              2 |                   10723.60 |
|       10404 |              3 |                   10460.16 |
|       10312 |              3 |                   10286.40 |
...
```

SQL Comments

```
--select all;
select * from table_name
```

single line comments start with `--`

multi line comments start with `/* */`

```
/*Select all the columns
of all the records
in the Customers table:*/
SELECT * FROM table_name;
```

SQL WHERE

```
select * from table_name
where condition;
```

The operators used in `WHERE` clause

Operator	Description
=	Equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
<>	Not equal. **Note:** In some versions of SQL this operator may be written as !=
BETWEEN	Between a certain range
LIKE	Search for a pattern
IN	To specify multiple possible values for a column

```
-- example between operator
SELECT * FROM Products
```

```
WHERE Price BETWEEN 50 AND 60;
```

```
-- example like operator
SELECT * FROM Customers
WHERE City LIKE 's%'; -- the city names start with 's'
```

```
SELECT
   firstName,
   lastName
FROM
   employees
WHERE
   lastName LIKE '%son'
ORDER BY firstName;

+-----------+-----------+
| firstName | lastName  |
+-----------+-----------+
| Leslie    | Thompson  |
| Mary      | Patterson |
| Steve     | Patterson |
| William   | Patterson |
+-----------+-----------+
4 rows in set (0.00 sec)
```

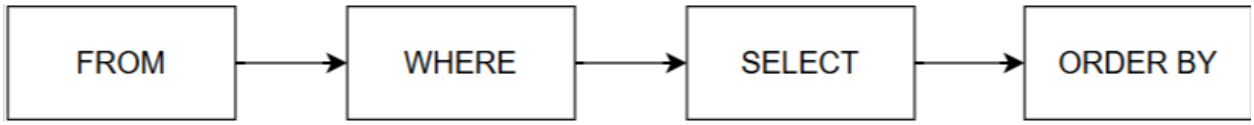
```
-- example in operator
SELECT * FROM Customers
WHERE City IN ('Paris','London');
```

```
SELECT
   lastName,
   firstName,
   reportsTo
```

```
FROM
   employees
WHERE
   reportsTo IS NULL;

+----------+-----------+-----------+
| lastName | firstName | reportsTo |
+----------+-----------+-----------+
| Murphy   | Diane     |      NULL |
+----------+-----------+-----------+
1 row in set (0.01 sec)
```

When executing a `SELECT` statement with a `WHERE` clause, MySQL evaluates the `WHERE` clause after the `FROM` clause and before the `SELECT` and `ORDER BY` clauses:



AND OPERATOR

The `AND` operator is a logical operator that combines two or more Boolean expressions and returns 1, 0, or NULL

```
A AND B
```

```
select A and B;
+---------+
| 1 AND 1 |
+---------+
|       1 |
+---------+
1 row in set (0.00 sec)
```

```
select 1 and 0, 0 and 1, 0 and 0, 0 and null;
+---------+---------+---------+------------+
| 1 AND 0 | 0 AND 1 | 0 AND 0 | 0 AND NULL |
+---------+---------+---------+------------+
|       0 |       0 |       0 |          0 |
+---------+---------+---------+------------+
1 row in set (0.00 sec)
```

	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

```
SELECT column1, column2, ...
FROM table_name
WHERE condition1 AND condition2 AND condition3 ...;
```

```
SELECT
    customername,
    country,
    state,
    creditlimit
FROM
    customers
WHERE
    country = 'USA' AND
    state = 'CA' AND
    creditlimit > 100000;

    +----------------------------+---------+-------+-------------+
| customername               | country | state | creditlimit |
+----------------------------+---------+-------+-------------+
| Mini Gifts Distributors Ltd. | USA     | CA    |   210500.00 |
```

```
| Collectable Mini Designs Co. | USA      | CA    |   105000.00 |
| Corporate Gift Ideas Co.     | USA      | CA    |   105000.00 |
+------------------------------+----------+-------+-------------+
3 rows in set (0.00 sec)
```

The `AND` operator returns true when both expressions are true; otherwise, it returns false

---

OR Operator

```
A or B
```

if both A and B are not NULL, the OR operator returns 1 (true) if either A or B is non-zero.

```
select 1 or 1, 1 or 0, 0 or 1;
+--------+--------+--------+
| 1 OR 1 | 1 OR 0 | 0 OR 1 |
+--------+--------+--------+
|      1 |      1 |      1 |
+--------+--------+--------+
1 row in set (0.00 sec)
```

	TRUE	**FALSE**	**NULL**
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

```
SELECT column1, column2, ...
FROM table_name
WHERE condition1 OR condition2 OR condition3 ...;
```

```
SELECT
  first_name,
```

```
  last_name,
  hire_date,
  department_id
FROM
  employees
WHERE
  department_id = 3
  AND
    EXTRACT(year from hire_date) = 1999 OR
    EXTRACT(year from hire_date) = 2000

ORDER BY
  hire_date;

   first_name | last_name  | hire_date  | department_id
------------+------------+------------+---------------
 Karen      | Colmenares | 1999-08-10 |             3
 Charles    | Johnson    | 2000-01-04 |             8
```

> The `OR` operator displays a record if *any* of the conditions are TRUE.
>
> The `AND` operator displays a record if *all* the conditions are TRUE.

NOT operator

The `NOT` operator is used in combination with other operators to give the opposite result, also called the negative result.

```
SELECT column1, column2, ...
FROM table_name
WHERE NOT condition;
```

If the `condition` is `true`, the `NOT` operator makes it `false` and vice versa. However, if the `condition` is `NULL`, the `NOT` operator returns `NULL`.

```
SELECT
  first_name,
  salary
FROM
  employees
WHERE
  NOT salary >= 3000;

   first_name | salary
------------+---------
 Shelli     | 2900.00
 Sigal      | 2800.00
 Guy        | 2600.00
 Karen      | 2500.00
 Irene      | 2700.00
```

IN Operator

The `IN` operator allows you to determine if a value matches any value in a list of values. Here's the syntax of the `IN` operator:

```
value in (value1, value2, value3, ....)
```

The `IN` operator return 1 (true) if they `value` equals any value in the list (`value1`, `value2`, `value3`, ...) otherwise it returns 0.

The `IN` operator is functionally equivalent to a combination of multiple `OR` operators:

```
value = value1 or value = value2 or value = value3 or ...
```

```
SELECT 1 IN (1,2,3);
+--------------+
| 1 IN (1,2,3) |
+--------------+
```

```
|          1 |
+--------------+
1 row in set (0.00 sec)
```

```
SELECT
   officeCode,
   city,
   phone,
   country
FROM
   offices
WHERE
   country IN ('USA' , 'France');

+------------+---------------+----------------+---------+
| officeCode | city          | phone          | country |
+------------+---------------+----------------+---------+
| 1          | San Francisco | +1 650 219 4782 | USA     |
| 2          | Boston        | +1 215 837 0825 | USA     |
| 3          | NYC           | +1 212 555 3000 | USA     |
| 4          | Paris         | +33 14 723 4404 | France  |
+------------+---------------+----------------+---------+
4 rows in set (0.01 sec)
```

```
SELECT
  first_name,
  last_name,
  job_id
FROM
  employees
WHERE
  job_id IN (8, 9, 10)
ORDER BY
  job_id;
```

```
 first_name | last_name | job_id
------------+-----------+--------
 Susan      | Mavris    |      8
 Bruce      | Ernst     |      9
 David      | Austin    |      9
 Alexander  | Hunold    |      9
 Diana      | Lorentz   |      9
 Valli      | Pataballa |      9
 Michael    | Hartstein |     10
```

NOT IN OPERATOR

```
value NOT IN (value1, value2, value3, ...);
```

The `NOT IN` operator returns one if the value doesn't equal any value in the list. Otherwise, it returns 0.

```
select 1 not in (1,2,3);
+------------------+
| 1 NOT IN (1,2,3) |
+------------------+
|                0 |
+------------------+
1 row in set (0.00 sec)
```

```
select 0 not in (1,2,3);
+------------------+
| 0 NOT IN (1,2,3) |
+------------------+
|                1 |
+------------------+
1 row in set (0.00 sec)
```

```
select null not in (1,2,3)
+--------------------+
| NULL NOT IN (1,2,3) |
+--------------------+
|               NULL |
+--------------------+
1 row in set (0.00 sec)
```

```
select officecode,city,phone from offices;
where country not in ("USA", "FRANCE")
order by city;
+------------+--------+-----------------+
| officeCode | city   | phone           |
+------------+--------+-----------------+
| 7          | London | +44 20 7877 2041 |
| 6          | Sydney | +61 2 9264 2451  |
| 5          | Tokyo  | +81 33 224 5000  |
+------------+--------+-----------------+
3 rows in set (0.02 sec)
```

The following example uses the `NOT IN` operator to find the offices that are not located in `France` and the `USA`

BETWEEN OPERATOR

The `BETWEEN` operator is logical operator that specifies whether a value is in a range or not.

```
value BETWEEN low and high;
```

The `BETWEEN` operator returns 1 if:

```
value >= low and value <= high
```

otherwise it returns 0.

```
select employee_id, first_name, last_name, salary from employees
where salary between 2500 and 2900
order by salary;

 employee_id | first_name |  last_name  | salary
-------------+------------+-------------+---------
         119 | Karen      | Colmenares  | 2500.00
         118 | Guy        | Himuro      | 2600.00
         126 | Irene      | Mikkilineni | 2700.00
         117 | Sigal      | Tobias      | 2800.00
         116 | Shelli     | Baida       | 2900.00
```

NOT BETWEEN — to negate the result of the `BETWEEN` operator you use the `NOT` operator:

```
expression not between low and high
```

The `NOT BETWEEN` returns `true` if the `expression` is less than `low` or greater than `high`; otherwise, it returns `false`.

```
select employee_id, first_name, last_name, salary from employees
where salary not between 2500 and 2900
order by salary;

 employee_id | first_name  | last_name  |  salary
-------------+-------------+------------+----------
         115 | Alexander   | Khoo       |  3100.00
         193 | Britney     | Everett    |  3900.00
         192 | Sarah       | Bell       |  4000.00
         107 | Diana       | Lorentz    |  4200.00
         200 | Jennifer    | Whalen     |  4400.00
...
```

LIKE OPERATOR

The `LIKE` operator is a logical operator that tests whether a string contains a specified pattern or not.

```
expression like pattern escape escape_character
```

in this syntax, if the `expression` matches the `pattern`, the `like` operator returns 1. otherwise it returns 0.

MySQL provides two wildcard characters for constructing patterns: Percentage `%` and underscore `_` .

- The percentage (`%`) wildcard matches any string of zero or more characters.
- The underscore (`_`) wildcard matches any single character.

Expression	Meaning
LIKE `'Kim%'`	match a string that starts with `Kim`
LIKE `'%er'`	match a string that ends with `er`
LIKE `'%ch%'`	match a string that contains `ch`
LIKE `'Le_'`	match a string that starts with `Le` and is followed by one character e.g., `Les` , `Len` ...
LIKE `'_uy'`	match a string that ends with `uy` and is preceded by one character e.g., `guy`
LIKE `'%are_'`	match a string that includes the string `are` and ends with one character.
LIKE `'_are%'`	match a string that includes the string `are` , starts with one character and ends with any number of characters.

```
select first_name, last_name from employees
where first_name like 'Da%'
order by first_name;

 first_name | last_name
------------+-----------
 Daniel     | Faviet
 David      | Austin
```

NOT LIKE

To negate the result of a `LIKE` operator, you use the `NOT` operator:

```
expression not like pattern
```

The `not like` operator returns `true` if the expression doesn't match the pattern or `false` otherwise

```
SELECT first_name, last_name FROM employees
WHERE first_name LIKE 'S%' AND first_name NOT LIKE 'Sh%'
ORDER BY first_name;

 first_name | last_name
------------+-----------
 Sarah      | Bell
 Sigal      | Tobias
 Steven     | King
 Susan      | Mavris
```

IS NULL OPERATOR / NULL OPERATOR

`NULL` is a marker that indicates unknown or missing data in the database. The `NULL` is special because you cannot compare it with any value even with the `NULL` itself.

```
value is NULL
```

```
select null = 5 as result

 result
--------
 NULL
```

```
SELECT customerName, country, salesrepemployeenumber
FROM customers
WHERE salesrepemployeenumber IS NULL
ORDER BY customerName;

+------------------------------+-------------+-----------------------+
| customerName                 | country     | salesrepemployeenumber |
+------------------------------+-------------+-----------------------+
| ANG Resellers                | Spain       |                  NULL |
| Anton Designs, Ltd.          | Spain       |                  NULL |
| Asian Shopping Network, Co   | Singapore   |                  NULL |
| Asian Treasures, Inc.        | Ireland     |                  NULL |
...
```

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

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

- Use the `IS NULL` operator to test if a value is `NULL` or not. The `IS NOT NULL` operator negates the result of the `IS NULL` operator.
- The `value IS NULL` returns true if the value is NULL or false if the value is not NULL.
- The `value IS NOT NULL` returns true if the value is not NULL or false if the value is NULL.

UPDATE STATEMENT

The `UPDATE` statement is used to modify the existing records in a table.

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

> Be careful when updating records in a table! Notice the `WHERE` clause in the `UPDATE` statement. The `WHERE` clause specifies which record(s) that should be updated. If you omit the `WHERE` clause, all records in the table will be updated!

```
UPDATE Customers
SET ContactName='Juan'
WHERE Country='Mexico';

-- here the rows which contains mexico will set their contact_name to juan
```

```
UPDATE Customers
SET ContactName='Juan';

-- here contact_name for every rows will get updated to Juan
```

INSERT STATEMENT

The `INSERT INTO` statement is used to insert new records in a table

```
-- specify both the column names and the values to be inserted
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

```
/*  If you are adding values for all the columns of the table, you do not need to specify the column names in the SQL query. However, make sure the order of
```

```
the values is in the same order as the columns in the table. */
INSERT INTO table_name
VALUES (value1, value2, value3, ...);
```

```
INSERT INTO Customers (CustomerName, ContactName, Address, City, Posta
lCode, Country)
VALUES ('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006', 'Norw
ay');
```

```
-- Insert data only in specified columns
INSERT INTO Customers (CustomerName, City, Country)
VALUES ('Cardinal', 'Stavanger', 'Norway');
-- the other column for this newly added row will be null
```

```
-- insert multiple rows
INSERT INTO Customers (CustomerName, ContactName, Address, City, Posta
lCode, Country)
VALUES
('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006', 'Norway'),
('Greasy Burger', 'Per Olsen', 'Gateveien 15', 'Sandnes', '4306', 'Norway'),
('Tasty Tee', 'Finn Egan', 'Streetroad 19B', 'Liverpool', 'L1 0AA', 'UK');
```

DELETE STATEMENT

The `DELETE` statement is used to delete existing records in a table.

```
delete from table_name where condition;
```

> Be careful when deleting records in a table! Notice the `WHERE` clause in the `DELETE` statement. The `WHERE` clause specifies which record(s) should be deleted. If you omit the `WHERE` clause, all records in the table will be deleted!

```
delete from coutomers where coutomername = "Alfreds Futterkiste";

--it will delete the whole row which contains the coustomer_name "Alfreds Fut
terkiste"
```

Delete all records — here all the data means rows gets deleted while keeping the table intact with their attributes and indexes

```
delete from table_name;
/*delete from coustomers
this will deletes all rows in the coustomers without deleting table */
```

Delete a Table — it's deletes the whole tables

```
drop table table_name;
/* drop table coustomer
it will deletes the whole table */
```

SQL TOP, LIMIT, FETCH FIRST or ROWNUM Clause

- SQL SELECT TOP Clause

  The `SELECT TOP` clause is used to specify the number of records to return.

  The `SELECT TOP` clause is useful on large tables with thousands of records. Returning a large number of records can impact performance.

  ```
  SELECT TOP 3 * FROM Customers;
  ```

  > Not all database systems support the `SELECT TOP` clause. MySQL supports the `LIMIT` clause to select a limited number of records, while Oracle uses `FETCH FIRST` *n* `ROWS ONLY` and `ROWNUM`.

  ```
  -- SQL server / MS Access
  SELECT TOP number|percent column_name(s)
  ```

```
FROM table_name
WHERE condition;
```

```
-- MySQL Syntax
SELECT column_name(s)
FROM table_name
WHERE condition
LIMIT number;
```

- LIMIT

```
SELECT * FROM Customers
LIMIT 3;
-- Restricts the output to only the first 3 rows from the coustomers
```

- FETCH FIRST

```
SELECT * FROM Customers
FETCH FIRST 3 ROWS ONLY;
```

- SELECT TOP PERCENT

```
SELECT TOP 50 PERCENT * FROM Customers;
```

SQL Aggregate Functions

An aggregate function is a function that performs a calculation on a set of values, and returns a single value.

- `MIN()` - returns the smallest value within the selected column
- `MAX()` - returns the largest value within the selected column
- `COUNT()` - returns the number of rows in a set
- `SUM()` - returns the total sum of a numerical column
- `AVG()` - returns the average value of a numerical column

SQL AGGREGATE Functions —

An **aggregate function** in SQL performs a **calculation on a set of values** and returns a **single summary value**.

Function	Description	Example
`COUNT()`	Counts the number of values (or rows)	`COUNT(*)`, `COUNT(column)`
`SUM()`	Adds up numeric values	`SUM(salary)`
`AVG()`	Calculates the average (mean) value	`AVG(score)`
`MIN()`	Finds the smallest value	`MIN(price)`
`MAX()`	Finds the largest value	`MAX(age)`

SQL MIN() and MAX() Functions

The `MIN()` function returns the smallest value of the selected column.

The `MAX()` function returns the largest value of the selected column.

```
-- MIN()
select MIN(column_name)
from table_name
where condition;
```

```
select MAX(column_name)
from table_name
where condition;
```

When you use `MIN()` or `MAX()`, the returned column will not have a descriptive name. To give the column a descriptive name, use the `AS` keyword:

```
SELECT MIN(Price) AS SmallestPrice
FROM Products;
/* Here it will show the minimum value of the Price in the
new column SmallestPrice  */
```

SQL COUNT() Function

The `COUNT()` function returns the number of rows that matches a specified criterion.

```
select count(column_name) from table_name
where condition;
```

```
SELECT COUNT(*) FROM Products;
/* SELECT COUNT(*): This counts all rows in the table,
including rows with NULL values. */
```

You can specify a column name instead of the asterix symbol `(*)`.

If you specify a column name instead of `(*)`, NULL values will not be counted.

```
SELECT COUNT(ProductName) FROM Products;
```

```
-- Find the number of products where Price is higher than 20:
select count(ProductID) from Products
where Price > 20
```

```
-- to ignore duplicates
select count(distinct column_name)
from table_name;
```

```
-- use an alice
select count (column_name) as [number of records]
from tavble_name
```

SQL SUM() Function

The `SUM()` function returns the total sum of a numeric column

```
select sum(column_name)
from table_name
where condition;
```

```
-- Return sum of all Quantity fields in OrderDetails table
select sum(Quantity)
from OrderDetails
```

```
-- with where clause
-- Return the sum of the Quantity field for the product with ProductID 11:
select sum(Quantity) from OrderDetails
where ProductID = 11;
```

```
-- with alias
-- Name the column "total":
select sum (Quantity) as total
from OrderDetails
```

SQL AVG() Function

The `AVG()` function returns the average value of a numeric column.

```
select avg(column_name)
from table_name
where condition;
```

```
-- Find the average price of all products:
select avg(Price)
from Products
```

```
-- add WHERE clause
select avg(Price)
from Products
where CategoryID = 1;
```

```
-- use an alias
-- Name the column "average price":
select avg(Price) as [average price]
from Products;
```

```
-- Return all products with a higher price than the average price:
select * from Products
where price > (select avg(price) from Products);
```

SQL Wildcards

Symbol	Description
%	Represents zero or more characters
_	Represents a single character
[]	Represents any single character within the brackets *
^	Represents any character not in the brackets *
-	Represents any single character within the specified range *
{}	Represents any escaped character **

```
--  using the % wildcard
-- Return all customers that ends with the pattern 'es':
SELECT * FROM Customers
WHERE CustomerName LIKE '%es';
```

```
-- Using the _ Wildcard (It can be any character or number, but each _ represe
nts one, and only one, character.)
-- Return all customers with a City starting with any character,
--followed by "ondon":
SELECT * FROM Customers
WHERE City LIKE '_ondon';
```

```
-- Using the [] Wildcard (The [] wildcard returns a result if any of the characte
rs inside gets a match.)
-- Return all customers starting with either "b", "s", or "p":
SELECT * FROM Customers
WHERE CustomerName LIKE '[bsp]%';
```

```
-- Using the - Wildcard
-- The - wildcard allows you to specify a range of characters inside the [] wild
card
-- Return all customers starting with "a", "b", "c", "d", "e" or "f":
SELECT * FROM Customers
WHERE CustomerName LIKE '[a-f]%';
```

SQL Aliases

SQL aliases are used to give a table, or a column in a table, a temporary name.

Aliases are often used to make column names more readable.

```
SELECT column_name AS alias_name
FROM table_name;
```

```
-- When alias is used on table:
SELECT column_name(s)
FROM table_name AS alias_name;
```

```
SELECT CustomerID AS ID
FROM Customers;
```

If you want your alias to contain one or more spaces, like "`My Great Products`", surround your alias with square brackets or double quotes.

```
-- Using Aliases With a Space Character
-- Using [square brackets] for aliases with space characters:
SELECT ProductName AS [My Great Products]
FROM Products;
```

```
-- Using "double quotes" for aliases with space characters:
SELECT ProductName AS "My Great Products"
FROM Products;
```

Concatenate Columns

```
SELECT CustomerName, CONCAT(Address, ', ', PostalCode, ', ', City, ', ', Coun
try) AS Address
FROM Customers;
```

Alias for Tables

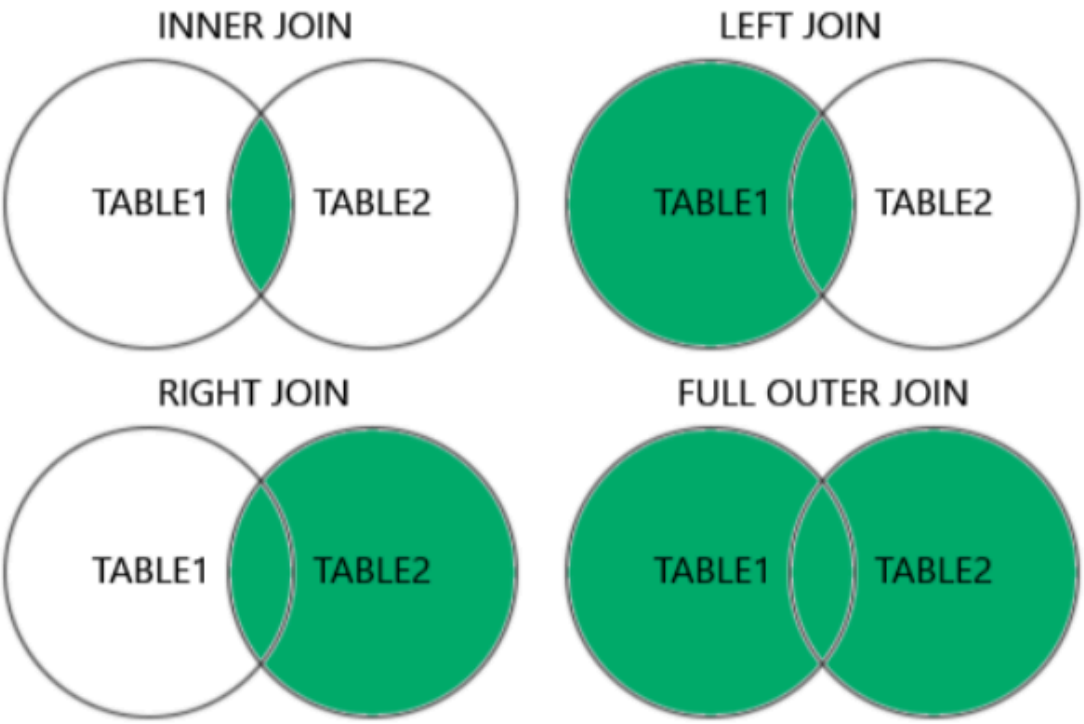
```
select column_nmae()
from table_name as alias_name;
```

SQL JOINS

A join clause is used to combine rows from two or more tables, based on a related column between them

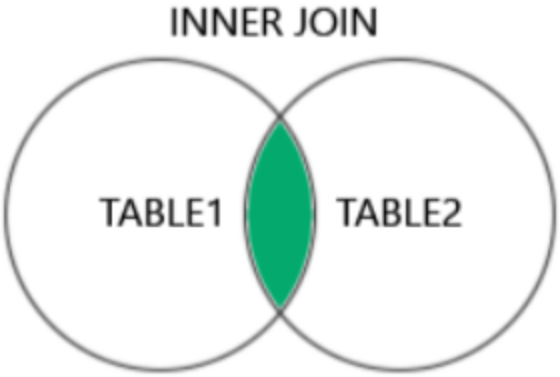
- INNER JOIN — Return records that have matching values in both tables
- LEFT (OUTER) JOIN — Returns all records from the left table, and the matched records from the right table

- RIGHT (OUTER) JOIN — Returns all records from the right table and the matched records from the left table
- Full (OUTER) JOIN — Returns all records when there is a match in either left or right table



SQL INNER JOIN

The INNER JOIN keyword selects records that have matching values in both tables




```
select column_name from table1
inner join table2
on table1.column_name = table2.column_name;
```

The example works without specifying table names, because none of the specified column names are present in both tables. If you try to include `CategoryID` in the `SELECT` statement, you will get an error if you do not specify the table name (because `CategoryID` is present in both tables).

```
select Products.ProductID, Products.ProductName, Categories.CategoryName
from Products
inner join Categories on Products.CategoryID = Categories.CategoryID;
```

`JOIN` and `INNER JOIN` will return the same result.

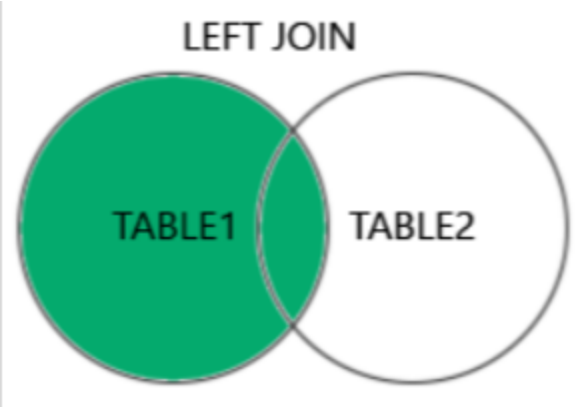
`INNER` is the default join type for `JOIN`, so when you write `JOIN` the parser actually writes `INNER JOIN`.

SQL LEFT (OUTER) JOIN

The `LEFT JOIN` keyword returns all records from the left table (table1), and the matching records from the right table (table2). The result is 0 records from the right side, if there is no match.

```
select column_name
from table1
```

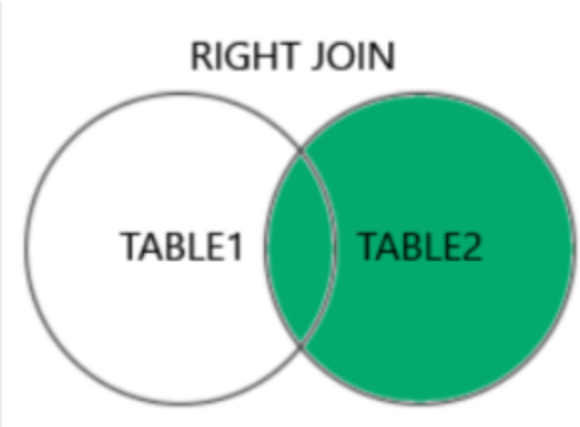
```
left join table2
on table1.column_name = table2.column_name;
```



SQL RIGHT (OUTER) JOIN

The `RIGHT JOIN` keyword returns all records from the right table (table2), and the matching records from the left table (table1). The result is 0 records from the left side, if there is no match

```
select column_name
from table1
right join table2
on table1.column_name = table2.column_name
```

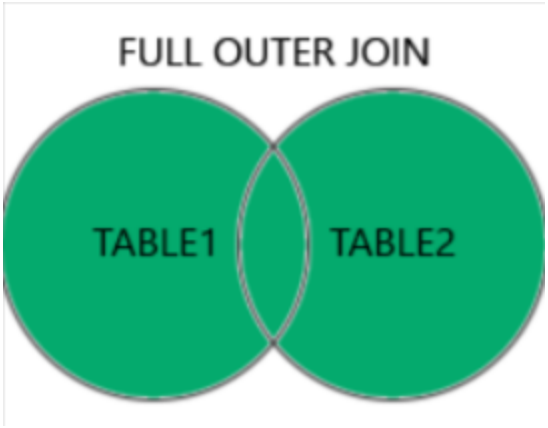


SQL FULL (OUTER) JOIN

The `FULL OUTER JOIN` keyword returns all records when there is a match in left (table1) or right (table2) table records.

Tip: `FULL OUTER JOIN` and `FULL JOIN` are the same.

```
select column_name(s) from table1
full outer join on table2
on table1.column_name = table2.column_name
where condition;
```



SQL SELF JOIN

A self join is a regular join, but the table is joined with itself

```
select column_name(s) from table1 T1, table2 T2
-- T1 and T2 are different table aliases for the same table.
where condition
```

```
select A.customerName as CustomerName1, B.CustomerName as CustomerN
ame2, A.City
from Customers A, Customer B
where A.CustomerID <> B.Customer.ID
```

```
and A.City = B.City
order by A.city;
```

SQL UNION OPERATOR

The `UNION` operator is used to combine the result-set of two or more `SELECT` statements

- Every `SELECT` statement within `UNION` must have the same number of columns
- The columns must also have similar data types
- The columns in every in `SELECT` statement must also be in the same order

```
--The UNION operator selects only distinct values by default.
--To allow duplicate values, use UNION ALL
select column_name(s) from table_1
union
select column_name(s) from table_2
```

```
--The following SQL statement returns the cities (only distinct values) from
--both the "Customers" and the "Suppliers" table:
select city from customers
union
select city from suppliers
order by city
```

If some customers or suppliers have the same city, each city will only be listed once, `because UNION selects only distinct values`

```
-- The following SQL statement returns the cities (duplicate values also) from
-- both the "Customers" and the "Suppliers" table:
select city from customers
union all
select city from suppliers
```

SQL UNION with WHERE

```
SELECT City, Country FROM Customers
WHERE Country='Germany'
UNION
SELECT City, Country FROM Suppliers
WHERE Country='Germany'
ORDER BY City;
/* for union all
 SELECT City, Country FROM Customers
WHERE Country='Germany'
UNION ALL
SELECT City, Country FROM Suppliers
WHERE Country='Germany'
ORDER BY City;*/
```

SQL GROUP BY STATEMENT

The `GROUP BY` statement groups rows that have the same values into summary rows, like “find the number of customers in each country”

The `GROUP BY` statement is often used with aggregate functions (`COUNT()` , `MAX()` , `MIN()` , `SUM()` , `AVG()`) to group the result-set by one or more columns

```
select column_name(s)
from table_name
group by column_name(s)
order by column_name(s)
```

SQL GROUP BY EXAMPLES

```
--The following SQL statement lists the number of customers in each country:
select count(customerID), country
from customers
group by country
```

```
--The following SQL statement lists the number of customers in each country,
--sorted high to low:
select count(customerID), country
from customers
group by country
order by count(customerID) DESC
```

SQL HAVING Clause

The `HAVING` clause was added to SQL because the `WHERE` keyword cannot be used with aggregate functions.

> 💡 Is where clause can only filter out rows but not the groups?
> Yes — **the `WHERE` clause filters rows before grouping happens**, so it **can only filter individual rows**, **not groups**.

```
select column_name(s)
from table_name
where condition
group by column_name(s)
having condition
order by column_name(s)
```

SQL HAVING examples

```
--The following SQL statement lists the number of customers
--in each country. Only include countries with more than 5
--customers:
select count(CustomerID),Country
from customers
group by country
having count(CustomerID) > 5;
```

```
--The following SQL statement lists the number of customers
--in each country, sorted high to low (Only include countries
--with more than 5 customers):
select count(CustomerID),country
from customers
group by country
having count(CustomerID) > 5
order by count(CustomerID) DESC
```

SQL EXISTS Operator

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.

```
select column_name(s)
from table_name
where exists
(select column_name from table_name where condition);
```

Examples —

```
--The following SQL statement returns TRUE and lists the
--suppliers with a product price less than 20:
select suppliername
from suppliers
where exists (select productname from products where products.supplierID =
suppliers.supplierID and price < 20)
```

```
--The following SQL statement returns TRUE and lists the
--suppliers with a product price equal to 22:
select suppliername
from suppliers
```

```
where exists (select productname from products where product.supplierID = s
uppliers.supplierID and price = 22)
```

SQL ANY and ALL Operators

The `ANY` and `ALL` operators allow you to perform a comparison between a single column value and a range of other values.

- 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.

```
select column_name(s)
from table_name
where column_name operator ANY
(select column_name
from table_name
where condition)
```

`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.

```
select all column_name(s)
from table_name
where condition
```

SQL ANY Examples

> The following SQL statement lists the `ProductName` if it finds ANY records in `OrderDetails` table has Quantity equal to 10 (this will return TRUE because the Quantity column has some values of 10):

```
select ProductName
fro Products
where ProductID = ANY
(select ProductID
from OrderDetails
where quantity = 10)
```

> The following SQL statement lists the `ProductName` if it finds ANY records in the `OrderDetails` table has Quantity larger than 1000 (this will return FALSE because the Quantity column has no values larger than 1000):

```
select ProductName
from Products
where ProductID = any
(select ProductID
from OrderDetails
where quantity > 1000)
```

SQL ALL Examples

> The following SQL statement lists the ProductName if ALL the records in the OrderDetails table has Quantity equal to 10. This will of course return FALSE because the Quantity column has many different values (not only the value of 10):

```
SELECT ProductName
FROM Products
WHERE ProductID = ALL
  (SELECT ProductID
```

```
FROM OrderDetails
WHERE Quantity = 10);
```

SQL SELECT INTO

The `select into` statement copies copies data from one table into a new table

```
select * into newtable [in externalab]
from oldtable
where condition;
```

Copy only some columns into a new table

```
select column1, column2, column3, ...
into newtable [in externalab]
from oldtable
where condition;
```

the new table will be created with the column-names and types as defined in the old table. You can create new column names using the `AS` clause

```
-- The following SQL statement creates a backup copy of Customers:
select * into customersbackup2017
from customers;
```

```
-- The following SQL statement uses the IN clause to copy the table into
--a new table in another database:
select * into customersbackup2017 in 'Backup.mdb'
from customers;
```

```
-- The following sql statement copies only a few columns into a new table
select customername, contactname into customersbackup2017
from customers;
```

```
-- the following sql statement copies only the german customers into a new ta
ble
select * into customergermany
from customers
where country = 'Germany'
```

```
-- the following sql statement copies data from more than one table into a
-- new table:
select customers.customername, orders.orderID
into ustomersorderbackup2017
from customers
left join orders on customers.customerID = orders.customerID;
```

Tip: `SELECT INTO` can also be used to create a new, empty table using the schema of another. Just add a `WHERE` clause that causes the query to return no data:

```
select * into newtable
from oldtable
where 1 = 0 ;
```

SQL INSERT INTO SELECT

The `INSERT INTO SELECT` copies data from one table and inserts it into another table

The `INSERT INTO SELECT` requires that the data types in source and target tables match

```
-- copy all columns from one table to another table
insert into table2
select * from table 1
where conditionif the column anme
```

```
-- copy only some columns from one table to another table
insert into table2(col1,col2,col3,....)
```

```
select col1, col2, col3, ....
from table1
    where condition
```

```
-- copy all rows
INSERT INTO employees_backup (id, name, department)
SELECT id, name, department
FROM employees;
```

here columns of both tables that are employees_backup and employees are same so can you give this one example with any data you want?

Initial table — employee

id	name	department
1	Alice	HR
2	Bob	Sales
3	Charlie	Engineering

Initial table — employees_backup

id	name	department
5	David	Finance

```
INSERT INTO employees_backup (id, name, department)
SELECT id, name, department
FROM employees;

output --
| id | name    | department  |
| -- | ------- | ----------- |
| 5  | David   | Finance     |
| 1  | Alice   | HR          |
| 2  | Bob     | Sales       |
| 3  | Charlie | Engineering |
```

```
-- copy all columns
insert into supplier
select * from customer
```

> To copy all columns from
>
> `insert into supplier`
>
> `select * from customer`
>
> is it correct?

Yes, but requirements are that both tables should have

- **Same number of columns**
- **Same column order**
- **Compatible data types** (e.g., both `name` columns are strings, etc.)

> my question is can i copy a whole table and it's all columns without putting any condition like this
> "insert into table2
> select * from table 1
> where condition"?

Yes, you **absolutely can** copy an entire table's data — all columns, all rows — without any `WHERE` clause.

Just make sure that table1 and table2 have —

- the **same number of columns**
- in the **same order**
- with **compatible data types**

If `employees_backup` has different columns or order, it will **fail**. Then you'd need to explicitly list columns:

> If the column names differ?

If the column names differ, but you're using SELECT *, then your query will fail unless the column order and data types match exactly.

SQL CASE Expression

The `CASE` expression goes through condition and returns a value when the first condition is met(like an if-then-else statement). So, once a condition is true, it will stop reading and return the result. If no conditions are true, it returns the value in the `ELSE` clause.

If there is no `ELSE` part and no conditions are true it returns null.

```
case
    when condition1 then result1
    when condition2 then result2
    when conditionN then resultN
    else result
end;
```

```
select OrderID, Quantity,
case
    when Quantity > 30 then 'The quantity is greater than 30'
    when Quantity = 30 then 'The quantity is 30'
    else 'The quantity is under 30'
end as QuantityText
from OrderDetails
```

```
-- The following SQL will order the customer by city , however if city is null,
-- then order by country
select CustomerName, City,Country
from customers
order by
(case
    when city is null then country
    else city
end)
```

SQL NULL Function (SQL IFNULL(), ISNULL(), COALESCE(), and NVL() Functions)

Product table

P_Id	productname	unitprice	unitsinstock	unitsonorder
1	Jarlsberg	10.45	16	15
2	Mascarpone	32.56	23	
3	Gorgonzola	15.67	9	20

```
select productname, unitprice * (unitsinstock + unitsonorder)
from products
```

If here any of the “unitsonorder” values are null, the result will be null

For MySQL

- The `IFNULL()` function lets you return an alternative value if an expression is null

  ```
  select productname, unitprice * (unitsinstock + ifnull(unitsonorder,0))
  from products;
  ```

- or we can use the coalesce() function

  ```
  select productname, unitprice * (unitsinstock + coalesce(unitsonoreder,0))
  from products;
  ```

For SQL Server

- `ISNULL()` function lets you return an alternative value when an expression is null:

  ```
  select productname, unitprice * (unitsinstock + isnull(unitsonorder,0))
  from products;
  ```

- or we can use the coalesce() function, like this:

```
select productname, unitprice * (unitsinstock + coalesce(unitsonorder,0))
from products;
```

For Oracle

- NVL() function achieves the same result

```
select productname, unitprice * (unitsinstock + NVL(unitsonorder,0))
from products;
```

- or we can use the coalesce() function like this

```
select productname, unitprice * (unitsinstock + coalesce(unitsonorder,
0))
from products;
```

SQL Stored Procedures for SQL Server

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

So if you have an SQL query that you write over and over again, save it as a stored procedure, and then just call it to execute it.

You can also pass parameters to a stored procedure, so that the stored procedure can act based on the parameter value(s) that is passed.

```
create procedure procedure_name
as
sql_statement
go;
```

Execute a stored produce

```
exec procedure_name
```

Stored Procedure with One Parameter

```
--The following SQL statement creates a stored procedure that
--selects Customers from a particular City from Customers table
CREATE PROCEDURE SelectAllCustomers @City nvarchar(30)
AS
SELECT * FROM Customers WHERE City = @City
GO;
```

Execute the stored procedure above as follows

```
exec selectallcustomers @city = 'London';
```

Stored Procedure with Multiple Parameters

The following SQL statement creates a stored procedure that selects Customers from a particular City with a particular postalcode from the "Customers" table:

```
CREATE PROCEDURE SelectAllCustomers @City nvarchar(30), @PostalCode
nvarchar(10)
AS
SELECT * FROM Customers WHERE City = @City AND PostalCode = @Postal
Code
GO;
```

Execute the stored procedure above as follows

```
exec selectallcustomer @city = 'London', @postalcode = 'WA1 1DP'
```

Create Database Example

The `CREATE DATABASE` statement is used to create a new sql databse

```
create database database_name
```

```
-- the following SQL statement creates database called "testDB"
create database testDB
```

SQL DROP DATABASE

The `DROP DATABASE` statement is used to drop an existing SQL database

```
drop database database_name;
```

```
-- following sql statement drops the existing database "testDB"
drop database testDB;
```

SQL BACKUP DATABASE Statement

The `BACKUP DATABASE` statement is used in SQL Server to create a full back up of an existing SQL database.

```
BACKUP DATABASE databasename
TO DISK = 'filepath';
```

```
BACKUP DATABASE testDB
TO DISK = 'D:\backups\testDB.bak';
```

SQL BACKUP WITH DIFFERENTIAL STATEMENT

A differential back up only backs up the parts of the database that have changed since the last full database backup.

```
BACKUP DATABASE databasename
TO DISK = 'filepath'
WITH DIFFERENTIAL;
```

```
BACKUP DATABASE testDB
TO DISK = 'D:\backups\testDB.bak'
WITH DIFFERENTIAL;
```

SQL CREATE TABLE

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
   ....
);
```

The following example creates a table called "Persons" that contains five columns: PersonID, LastName, FirstName, Address, and City:

```
CREATE TABLE Persons (
    PersonID int,
    LastName varchar(255),
    FirstName varchar(255),
    Address varchar(255),
    City varchar(255)
);
```

Create Table Using Another Table

```
CREATE TABLE new_table_name AS
    SELECT column1, column2,...
    FROM existing_table_name
    WHERE ....;
```

```
CREATE TABLE TestTable AS
SELECT customername, contactname
```

```
FROM customers;
```

SQL DROP TABLE

The `DROP TABLE` statement is used to drop an existing table in a database.

```
DROP TABLE table_name;
```

`TRUNCATE TABLE` used to delete the data inside a table, but not the table itself

```
truncate table table_name
```

SQL ALTER TABLE

The `alter table` statement is used to add, delete, or modify columns in an existing table

The `alter table` statement is also used to add and drop various constraints on an table

- Add Column

  ```
  alter table table_name
  add column_name datatype
  ```

  ```
  -- SQL adds an "Email" column to the
  alter table customers
  add email varchar(255);
  ```

- Drop Column

  ```
  -- deletes a column in a table
  alter table table_name
  drop column column_name
  ```

```
-- The following SQL deletes the "Email" column from the "Customers" ta
ble:
alter table customer
drop column email;
```

- Rename Column

```
-- to rename a column in a table
alter table table_name
rename column old_name to new_name
```

```
-- to rename a column in a table in SQL Server
exec sp_rename 'table_name.old_name', 'new_name', 'Column';
```

- Alter/Modify datatype

```
-- to change the datatype of a column in a table
alter table table_name
alter column column_name datatype
```

MySQL

```
alter table table_name
modify column column_name datatype
```

SQL Create Constraints

Constraints can be specified when the table is created with the `create table` statement or after the table is created with the `alter table` statement

```
CREATE TABLE table_name (
    column1 datatype constraint,
    column2 datatype constraint,
```

```
    column3 datatype constraint,
    ....
);
```

`not null` — ensures that a column can't have a null value

`unique` — ensures that all values in a column are different

`primary key` — a combination of a `not null` and `unique` uniquely identifies each row in a table

`foreign key` — prevents actions that would destroy links between tables

`check` — ensures that the value in a column satisfies a specific condition

`default` — sets a default value for a column if no value is specified

`create index` — used to create and retrieve data from the database very quickly

---

SQL NOT NULL Constraints

NOT NULL constraint enforces a column to not accept null values

This enforces a field to always contain a value, which means that you cannot insert a new record, or update a record without adding a value to this field.

```
-- The following SQL ensures that the "ID", "LastName", and
-- "FirstName" columns will NOT accept NULL values when the
-- "Persons" table is created:
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255) NOT NULL,
    Age int
);
```

```
alter table persons
alter column Age int not null;
```

```
-- MySQL
alter table persons
modify column age int not null;
```

SQL UNIQUE Constraints

The `UNIQUE` constraint ensures that all values in a column are different.

Both the `UNIQUE` and `PRIMARY KEY` constraints provide a guarantee for uniqueness for a column or set of columns.

A `PRIMARY KEY` constraint automatically has a `UNIQUE` constraint.

However, you can have many `UNIQUE` constraints per table, but only one `PRIMARY KEY` constraint per table.

```
-- The following SQL creates a UNIQUE constraint on the "ID"
-- column when the "Persons" table is created:
CREATE TABLE Persons (
    ID int NOT NULL UNIQUE,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int
);
```

```
-- MySQL
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    UNIQUE (ID)
);
```

```
-- To name a UNIQUE constraint, and to define a UNIQUE
--constraint on multiple columns, use the following SQL syntax:
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    CONSTRAINT UC_Person UNIQUE (ID,LastName)
);
```

```
ALTER TABLE Persons
ADD UNIQUE (ID);
```

Primary Key

The `PRIMARY KEY` constraint uniquely identifies each record in a table which contain UNIQUE values and cannot contain NULL values.

A table can have only ONE primary key; and in the table, this primary key can consist of single or multiple columns (fields).

```
-- The following SQL creates a PRIMARY KEY on the "ID" column
-- when the "Persons" table is created:
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    PRIMARY KEY (ID)
);
```

```


-- To allow naming of a PRIMARY KEY constraint, and for
-- defining a PRIMARY KEY constraint on multiple columns,
```

```
--use the following SQL syntax:
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    CONSTRAINT PK_Person PRIMARY KEY (ID,LastName)
);
```

```
-- Alter table
alter table persons
add primary key (id)
```

FOREIGN KEY

The `FOREIGN KEY` constraint is used to prevent actions that would destroy links between tables.

A `FOREIGN KEY` is a field (or collection of fields) in one table, that refers to the `PRIMARY KEY` in another table.

The table with the foreign key is called the child table, and the table with the primary key is called the referenced or parent table.

```
-- The following SQL creates a FOREIGN KEY on the "PersonID"
-- column when the "Orders" table is created:
CREATE TABLE Orders (
    OrderID int NOT NULL,
    OrderNumber int NOT NULL,
    PersonID int,
    PRIMARY KEY (OrderID),
    FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)
);
```

```
-- to create a foreign key consists on the "PersonID" column when the
-- "Orders" table is already created
ALTER TABLE Orders
ADD FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);
```

CHECK

If you define a `CHECK` constraint on a column it will allow only certain values for this column.

If you define a `CHECK` constraint on a table it can limit the values in certain columns based on values in other columns in the row.

```
-- The following SQL creates a CHECK constraint on the "Age"
-- column when the "Persons" table is created. The CHECK
-- constraint ensures that the age of a person must be 18, or
-- older:
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    CHECK (Age>=18)
);
```

```
-- To allow naming of a CHECK constraint, and for defining a
-- CHECK constraint on multiple columns, use the following
-- SQL syntax:
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    City varchar(255),
```

```
    CONSTRAINT CHK_Person CHECK (Age>=18 AND City='Sandnes')
);
```

```
-- To create a CHECK constraint on the "Age" column when the
-- table is already created, use the following SQL:
ALTER TABLE Persons
ADD CHECK (Age>=18);
```

DEFAULT Constraint

The `DEFAULT` constraint is used to set a default value for a column.

The default value will be added to all new records, if no other value is specified.

```
-- The following SQL sets a DEFAULT value for the "City"
-- column when the "Persons" table is created:
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    City varchar(255) DEFAULT 'Sandnes'
);
```

```
-- To create a DEFAULT constraint on the "City" column when
-- the table is already created, use the following SQL:
ALTER TABLE Persons
ALTER City SET DEFAULT 'Sandnes';
```

INDEX Statement

The `CREATE INDEX` statement is used to create indexes in tables.

Indexes are used to retrieve data from the database more quickly than otherwise. The users cannot see the indexes, they are just used to speed up searches/queries.

```
-- Creates an index on a table. Duplicate values are allowed:
CREATE INDEX index_name
ON table_name (column1, column2, ...);
```

```
-- Creates a unique index on a table. Duplicate values are not
-- allowed
CREATE UNIQUE INDEX index_name
ON table_name (column1, column2, ...);
```

```
-- The SQL statement below creates an index named
-- "idx_lastname" on the "LastName" column in the "Persons"
-- table:
CREATE INDEX idx_lastname
ON Persons (LastName);
```

```
-- The DROP INDEX statement is used to delete an index in a table.
DROP INDEX index_name ON table_name;
```

AUTO INCREMENT Field —

Auto - increment allows a unique number to be generated automatically when a new record is inserted into a table

The following SQL statement defines the “`Personid`” column to be auto-increment primary key field in the “Persons” table:

```
create table Persons(
    Personid int not null auto_increment,
    LastName varchar(255) not null,
    FirstName varchar(255),
```

```
    Age int,
    primary key (Personid)
);
```

By default, the starting value for `AUTO_INCREMENT` is 1, and it will increment by 1 for each new record.

To let the `AUTO_INCREMENT` sequence start with another value, use the following SQL statement:

```
alter table Persons AUTO_INCREMENT = 100;
```

To insert a new record into the "`Persons`" table, we will not have to specify a value for the "`Personid`" column (a unique value will be added automatically)

```
insert into Persons (FirstName,LastName)
values ('Lars','Monsen');
```

The SQL statement above would insert a new record into the "`Persons`" table. The "`Personid`" column would be assigned a unique value. The "FirstName" column would be set to "`Lars`" and the "`LastName`" column would be set to "`Monsen`"

```
-- Syntax for SQL Server
CREATE TABLE Persons (
   Personid int IDENTITY(1,1) PRIMARY KEY,
   LastName varchar(255) NOT NULL,
   FirstName varchar(255),
   Age int
);
--In the example, the starting value for IDENTITY is 1, and it will
--increment by 1 for each new record.
```

```
--Syntax for ACCESS
CREATE TABLE Persons (
   Personid AUTOINCREMENT PRIMARY KEY,
   LastName varchar(255) NOT NULL,
```

```
    FirstName varchar(255),
    Age int
);
-- The MS Access uses the AUTOINCREMENT keyword to perform an
-- auto-increment feature.
-- the starting value for AUTOINCREMENT is 1, and it will increment
-- by 1 for each new record.
```

SQL Working With Dates

SQL Data Types

MySQL comes with the following data types for storing a date or a date/time value in the database:

- `DATE` - format YYYY--MM--DD
- `DATETIME` - format: YYYY-MM-DD HH:MI:SS
- `TIMESTAMP` - format: YYYY-MM-DD HH:MI:SS
- `YEAR` - format YYYY or YY

SQL Server comes with the following data types for storing a date or a date/time value in the database:

- `DATE` - format YYYY--MM--DD
- `DATETIME` - format: YYYY-MM-DD HH:MI:SS
- `SMALLDATETIME` format: YYYY-MM-DD HH:MI:SS
- `TIMESTAMP` format: a unique number

OrderId	ProductName	OrderDate
1	Geitost	2008-11-11
2	Camembert Pierrot	2008-11-09
3	Mozzarella di Giovanni	2008-11-11

We use the following `select` statement

```
SELECT * FROM Orders WHERE OrderDate='2008-11-11'
```

the result will be

OrderId	ProductName	OrderDate
1	Geitost	2008-11-11
3	Mozzarella di Giovanni	2008-11-11

OrderId	ProductName	OrderDate
1	Geitost	2008-11-11 13:23:44
2	Camembert Pierrot	2008-11-09 15:45:21
3	Mozzarella di Giovanni	2008-11-11 11:12:01
4	Mascarpone Fabioli	2008-10-29 14:56:59

If we use the same `SELECT` statement as above:

```
SELECT * FROM Orders WHERE OrderDate='2008-11-11'
```

we will get no result! This is because the query is looking only for dates with no time portion.

VIEWS

A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

You can add SQL statements and functions to a view and present the data as if the data were coming from one single table.

A view is created with the `CREATE VIEW` statement.

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
```

```
FROM table_name
WHERE condition;
```

The following SQL creates a view that shows all customers from Brazil:

```
create view [Brazil Customers] AS
Select CustomerName, ContactName
From Customers
Where Country = 'Brazil';
```

We can query the view above as follows:

```
select * from [Brazil Customer];
```

The following SQL creates a view that selects every product in the "Products" table with a price higher than the average price:

```
create view [Products Above Averge Price] as
select ProductName, Price
From Products
where Price > (Select AVG(Price) from Products);
```

We can query the view above as follows

```
select * from [Products Above Average Price];
```

SQL Updating a View

A view can be updated with the `create or replace view` statement

```
-- SQL Create or Replace View Syntax
CREATE OR REPLACE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

```
-- following SQL adds  "City" column to the "Brazil Customers" view:
CREATE OR REPLACE VIEW [Brazil Customers] AS
SELECT CustomerName, ContactName, City
FROM Customers
WHERE Country = 'Brazil';
```

SQL Dropping a View

A view is deleted with the `DROP VIEW` statement

```
drop view view_name
```

```
-- the following SQL drops the "Brazil Customers" view:
drop view [Brazil Customers];
```

SQL Injection

- SQL injection is a code injection technique that might destroy your database.
- SQL injection is the placement of malicious code in SQL statements, via web page input.

> SQL injection usually occurs when you ask a user for input, like their username/userid, and instead of a name/id, the user gives you an SQL statement that you will **unknowingly** run on your database.

SQL Injection Based on 1=1 is Always True

If there is nothing to prevent a user from entering "wrong" input, the user can enter some "smart" input like this:

`UserId`: `105 or 1=1`

Then, the SQL statement will look like this:

```
SELECT * FROM Users WHERE UserId = 105 OR 1=1;
```

The SQL above is valid and will return ALL rows from the "Users" table, since **OR 1=1** is always TRUE.

Does the example above look dangerous? What if the "Users" table contains names and passwords?

The SQL statement above is much the same as this:

```
SELECT UserId, Name, Password FROM Users WHERE UserId = 105 or 1=1;
```

A hacker might get access to all the user names and passwords in a database, by simply inserting 105 OR 1=1 into the input field.

SQL Injection Based on ""="" is Always True

Here is an example of a user login on a web site:

Username: `John Doe`

Password: `myPass`

Example

```
uName = getRequestString("username");
uPass = getRequestString("userpassword");
sql = 'SELECT * FROM Users WHERE Name ="' + uName + '" AND Pass ="' +
uPass + '"'
```

Result

```
SELECT * FROM Users WHERE Name ="John Doe" AND Pass ="myPass"
```

A hacker might get access to user names and passwords in a database by simply inserting " OR ""=" into the user name or password text box:

User Name: `" or ""="`

Password: `" or ""="`

The code at the server will create a valid SQL statement like this:

Result

```
SELECT * FROM Users WHERE Name ="" or ""="" AND Pass ="" or ""=""
```

The SQL above is valid and will return all rows from the "Users" table, since **OR ""=""** is always TRUE.

SQL Injection Based on Batched SQL Statements

Most databases support batched SQL statement.

A batch of SQL statements is a group of two or more SQL statements, separated by semicolons.

The SQL statement below will return all rows from the "Users" table, then delete the "Suppliers" table.

Example

```
SELECT * FROM Users; DROP TABLE Suppliers
```

Look at the following example:

Example

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

And the following input:

User id: `105; DROP TABLE Suppliers`

The valid SQL statement would look like this:

Result

```
SELECT * FROM Users WHERE UserId = 105; DROP TABLE Suppliers;
```

SQL Hosting

The most common SQL hosting databases are MS SQL Server, Oracle, MySQL, and MS Access

- **MS SQL Server** is a popular database software for database-driven web sites with high traffic

  SQL Server is a very powerful, robust and full featured SQL database system
- **Oracle** is also a database software for database-driven web sites with high traffic
- MySQL is an inexpensive alternative to the expensive Microsoft and Oracle solution
- MS Access a web site requires only a simple database, Microsoft Access can be a solution

  MS Access is not well suited for very high-traffic, and not as powerful as MySQL, SQL Server, or Oracle.

SQL Keywords

Keyword	Description
ADD	Adds a column in an existing table
ADD CONSTRAINT	Adds a constraint after a table is already created
ALL	Returns true if all of the subquery values meet the condition
ALTER	Adds, deletes, or modifies columns in a table, or changes the data type of a column in a table
ALTER COLUMN	Changes the data type of a column in a table
ALTER TABLE	Adds, deletes, or modifies columns in a table
AND	Only includes rows where both conditions is true
ANY	Returns true if any of the subquery values meet the condition
AS	Renames a column or table with an alias
ASC	Sorts the result set in ascending order
BACKUP DATABASE	Creates a back up of an existing database
BETWEEN	Selects values within a given range
CASE	Creates different outputs based on conditions
CHECK	A constraint that limits the value that can be placed in a column

Keyword	Description
COLUMN	Changes the data type of a column or deletes a column in a table
CONSTRAINT	Adds or deletes a constraint
CREATE	Creates a database, index, view, table, or procedure
CREATE DATABASE	Creates a new SQL database
CREATE INDEX	Creates an index on a table (allows duplicate values)
CREATE OR REPLACE VIEW	Updates a view
CREATE TABLE	Creates a new table in the database
CREATE PROCEDURE	Creates a stored procedure
CREATE UNIQUE INDEX	Creates a unique index on a table (no duplicate values)
CREATE VIEW	Creates a view based on the result set of a SELECT statement
DATABASE	Creates or deletes an SQL database
DEFAULT	A constraint that provides a default value for a column
DELETE	Deletes rows from a table
DESC	Sorts the result set in descending order
DISTINCT	Selects only distinct (different) values
DROP	Deletes a column, constraint, database, index, table, or view
DROP COLUMN	Deletes a column in a table
DROP CONSTRAINT	Deletes a UNIQUE, PRIMARY KEY, FOREIGN KEY, or CHECK constraint
DROP DATABASE	Deletes an existing SQL database
DROP DEFAULT	Deletes a DEFAULT constraint
DROP INDEX	Deletes an index in a table
DROP TABLE	Deletes an existing table in the database
DROP VIEW	Deletes a view
EXEC	Executes a stored procedure
EXISTS	Tests for the existence of any record in a subquery

Keyword	Description
FOREIGN KEY	A constraint that is a key used to link two tables together
FROM	Specifies which table to select or delete data from
FULL OUTER JOIN	Returns all rows when there is a match in either left table or right table
GROUP BY	Groups the result set (used with aggregate functions: COUNT, MAX, MIN, SUM, AVG)
HAVING	Used instead of WHERE with aggregate functions
IN	Allows you to specify multiple values in a WHERE clause
INDEX	Creates or deletes an index in a table
INNER JOIN	Returns rows that have matching values in both tables
INSERT INTO	Inserts new rows in a table
INSERT INTO SELECT	Copies data from one table into another table
IS NULL	Tests for empty values
IS NOT NULL	Tests for non-empty values
JOIN	Joins tables
LEFT JOIN	Returns all rows from the left table, and the matching rows from the right table
LIKE	Searches for a specified pattern in a column
LIMIT	Specifies the number of records to return in the result set
NOT	Only includes rows where a condition is not true
NOT NULL	A constraint that enforces a column to not accept NULL values
OR	Includes rows where either condition is true
ORDER BY	Sorts the result set in ascending or descending order
OUTER JOIN	Returns all rows when there is a match in either left table or right table
PRIMARY KEY	A constraint that uniquely identifies each record in a database table
PROCEDURE	A stored procedure
RIGHT JOIN	Returns all rows from the right table, and the matching rows from the left table

Keyword	Description
ROWNUM	Specifies the number of records to return in the result set
SELECT	Selects data from a database
SELECT DISTINCT	Selects only distinct (different) values
SELECT INTO	Copies data from one table into a new table
SELECT TOP	Specifies the number of records to return in the result set
SET	Specifies which columns and values that should be updated in a table
TABLE	Creates a table, or adds, deletes, or modifies columns in a table, or deletes a table or data inside a table
TOP	Specifies the number of records to return in the result set
TRUNCATE TABLE	Deletes the data inside a table, but not the table itself
UNION	Combines the result set of two or more SELECT statements (only distinct values)
UNION ALL	Combines the result set of two or more SELECT statements (allows duplicate values)
UNIQUE	A constraint that ensures that all values in a column are unique
UPDATE	Updates existing rows in a table
VALUES	Specifies the values of an INSERT INTO statement
VIEW	Creates, updates, or deletes a view
WHERE	Filters a result set to include only records that fulfill a specified condition

Python-PostgreSQL Question

8. For the database connection, use the following connection string variables:

database = sys.argv[1] //name of the database is obtained from the command line argument
user = os.environ.get('PGUSER')
password = os.environ.get('PGPASSWORD')
host = os.environ.get('PGHOST')
port = os.environ.get('PGPORT')

Problem Statement:

Users

user_id	varchar(20)
name	varchar(30)
dob	date
email	varchar(50)
phone_num	int

Write a Python program to print the email id of the user.
The user id of the user is given in a file named 'user.txt' resides in the same folder as the python program file.
The output of the python program is only email id.
For example, if the user_id of the student is 'CLAUS'.
Then output must be **hohoho@santa.coming** only.
Note: No spaces.

```
import psycopg2
import sys
```

```
import os
# Opening the given file to read the data inside
file = open("user.txt", "r")
uid = file.read().strip()
# Closing the file
file.close() # Not a necessary step
# Using try-except block to handle errors
try:
# Creating the connection
connection = psycopg2.connect(
database = sys.argv[1],
user = os.environ.get(“PGUSER”),
password = os.environ.get(“PGPASSWORD”),
host = os.environ.get(“PGHOST”),
port = os.environ.get(“PGPORT”))
# Creating the cursor
cursor = connection.cursor()
# executing the query
query = f"select email from users where user_id = '{uid}'"
cursor.execute(query)
# Obtaining the result
result = cursor.fetchall() # or fetchmany() or fetchone()
for r in result:
print(r[0]) # Extracting the output we require
# Closing the cursor
cursor.close()
# This except block will catch both the general Python exception and
PostgreSQL(Database) related errors and print them to the console
except(Exception, psycopg2.DatabaseError) as error:
print(error)
# Closing the connection
finally:
connection.close()
```

Steps by steps explanation

1. Imports

```
import sys,os,psycopg2
```

   - `psycopg2` → used to connect and query PostgreSQL.
   - `sys` → to read command-line arguments (`sys.argv`).
   - `os` → to read environment variables (`PGUSER`, `PGPASSWORD`, etc.).

2. Reading `user.txt`

```
file = open("user.txt", "r")
uid = file.read()
file.close()
```

- Opens the file `user.txt` in **read mode**.
- Reads its full content into variable `uid`.
- Closes the file (good practice, but here not strictly needed).

  ⚠️ **Note**: This reads exactly what's in the file — including newlines (`\n`) if present.

  Example: If `user.txt` has `CLAUS\n`, then `uid = "CLAUS\n"`.

  This can cause problems unless `.strip()` is used.

3. Try-except-finally block

```
try:
    # connection + query
except(Exception, psycopg2.DatabaseError) as error:
    print(error)
finally:
    connection.close()
```

- **try** → Runs the database operations.

- **except** → If something goes wrong (connection issue, query issue), prints the error.
- **finally** → Ensures connection is closed regardless of success/failure.

4. Connecting to PostgreSQL

```
connection = psycopg2.connect(
    database = sys.argv[1],
    user = os.environ.get("PGUSER"),
    password = os.environ.get("PGPASSWORD"),
    host = os.environ.get("PGHOST"),
    port = os.environ.get("PGPORT"))
```

- Establishes a connection to the database.
- The **database name** comes from command-line argument `sys.argv[1]`.

  Example:

  ```
  python get_email.py studentdb
  ```

  → `studentdb` is the database name.
- Credentials (username, password, host, port) are read from **environment variables**.

5. Cursor creation

```
cursor = connection.cursor()
```

- A **cursor** is like a handle to send SQL commands to the database.

6. Executing the SQL query

```
query = f"select email from users where user_id = '{uid}'"
cursor.execute(query)
```

- Dynamically builds a SQL string with the user ID.

Example: if `uid = "CLAUS"`, query becomes:

```
select email from users where user_id = 'CLAUS';
```

- Sends query to PostgreSQL.

  ⚠️ **Risk**: This method can be unsafe (SQL injection) if `uid` contains malicious input. A safer way is parameterized queries (`WHERE user_id = %s`).

7. Fetching results

```
result = cursor.fetchall()
for r in result:
    print(r[0])
```

- `fetchall()` returns all rows from the query.
- Each row is a tuple → `r[0]` is the first column (`email`).
- Prints the email(s).

Example output:

```
hohoho@santa.coming
```

8. Closing resources

```
cursor.close()
connection.close()
```

- Closes the cursor and the connection → good practice to free resources.

⚙️ Example Run

Suppose:

- `user.txt` contains:

  ```
  CLAUS
  ```

- Table `users` has:

```
user_id | email
--------+---------------------
CLAUS   | hohoho@santa.coming
```

Run:

```
python get_email.py studentdb
```

Output:

```
hohoho@santa.coming
```

✅ **In summary:**

1. Reads `user_id` from `user.txt`.
2. Connects to PostgreSQL using env vars + command-line db name.
3. Runs query `SELECT email ...` for that `user_id`.
4. Prints only the email(s).
5. Closes cursor + connection.