**Learning SQL Fundamentals**

# Understanding SQL

**Data** is everywhere. To create, describe, structure and manipulate our data, we use a software application called a Database Management System or DBMS.

**S**tructured **Q**uery **L**anguage or **SQL** is a standard Database language which is used to communicate with relational databases. It is a tool used by data professionals for handling structured data (data which is stored in the form of tables).

SQL is not a database system, but it is a query language.

**FEATURES OF SQL:**

* SQL is simple and easy to learn as it contains English-like sentences, have a clear meaning, allowing them to be interpreted as natural sentences.
* SQL supports client/server architecture. A SQL database serves as the glue between “front-end” computer systems focussed toward user interaction and “back-end” systems that focus on database management, giving each system the ability to do what it does best.
* SQL is dynamic. SQL provides maximum flexibility, even while users are accessing database content, it is possible to change and expand a database’s structure dynamically.
* Portability across different machines.
* SQL includes several security-enabling features, including encrypted communication over SSL/TLS,
* SQL database is vertically scalable, which means that you can increase the load on a single server. No horizontal scaling - horizontal scaling is only possible for NoSQL databases.
* A large amount of data is retrieved quickly and efficiently.
* SQL is mainly used for relational databases. SQL has become the database language for relational databases.

**WHY SQL?**

* SQL is used to communicate with popular relational database systems. It is used in data systems such as MySQL, PostgreSQL, Oracle and many others.
* SQL is widely used in data science and data analytics, it helps them to describe, manipulate the structured data.

# Basic SQL Commands

The SQL commands help in creating and managing the database. The most common SQL commands which are highly used are mentioned below:

## **Create** Command

This command helps in creating the new database, new table, table view, and other objects of the database.

Let’s create databases and table –

* **Create Database**
* SQL command creates a database named *testdb.*

*# CREATE DATABASE testdb;*

* SQL command creates a database named *testdb* only if there is no existing database with the same name.

*# CREATE DATABASE IF NOT EXISTS testdb;*

* SQL command lists all the available databases in the DBMS.

*# SHOW DATABASES;*

* There can be multiple databases, to connect to the particular database, we use the statement.

*# USE testdb*

Now, we can access the tables, and perform operations inside the database.

* **Create Tables**
* A database table is used to store records (data). To create a database table, we use the SQL *CREATE TABLE* statement.

*# CREATE TABLE employee (id SERIAL PRIMARY KEY, first\_name VARCHAR, last\_name VARCHAR, role VARCHAR);*

Here, the SQL command creates a table *employee* in database *testdb.* The table contains column (field) *id, first\_name, last\_name* and *role*.

.

There are different data types available for each RDBMS, follow the appendix 1.1 to see the datatypes in postgresql.

* SQL command will only create a table if there is not one with a similar name.

*# CREATE TABLE IF NOT EXISTS employee (id SERIAL PRIMARY KEY, first\_name VARCHAR, last\_name VARCHAR, role VARCHAR);*

## **INSERT** Command

This command helps in inserting the data or records into the database tables. We can easily insert the records in single as well as multiple rows of the table.

Let’s see some commands –

* SQL command inserts a new row in the *employee* table with the given values.

*# INSERT INTO employee (first\_name, last\_name, role) VALUES ('John', 'Snow', 'Manager');*

* It is also possible to insert data to certain columns. For the other columns, it will take default values, or the constraints that are there.

*# INSERT INTO employee (first\_name, role) VALUES ('John', 'Manager');*

Here, it will take *last\_name* column as NULL as there is no value.

* You can also insert multiple values at once to the table –
* *# INSERT INTO employee (first\_name, last\_name, role) VALUES ('John', 'Snow', 'Manager'),*

*('Sheldon', 'Cooper', 'CEO’);*

There are other methods, available to insert values into tables, you can export the data via csv, excels, etc.

## **SELECT** Command

This command helps in accessing the single or multiple rows from one or multiple tables of the database. We can also use this command with the WHERE clause.

The SQL command to access the table –

# *SELECT \* from employee;*

This statement will return all the rows from the table.

We will learn about WHERE clause in upcoming sections, the where clause is usually is to filter out the data, based on conditions. For eg – we are only interested in the employee whose role is Manager, then we can use `where` clauses.

## **UPDATE** Command

To change existing data in a table, you use the UPDATE statement. The UPDATE statement affects one or more rows in a table based on the condition in the WHERE clause.

* SQL command to update a specific row.

*# UPDATE employee SET first\_name = ‘Rom’ where id =1;*

Here this statement will update the *first\_name* of id 1.

*# UPDATE employee SET first\_name = ‘Rom’, role = ‘Manager’ where id =2;*

Here this statement will update the *first\_name* and *role* of id 2.

* SQL command to update multiple rows.

*# UPDATE employee SET first\_name = ‘Rom’ where role =’Manager’;*

This statement will update the *first\_name* of all the *employees* which have the *role* ‘Manager’

* SQL command to update all rows.

*# UPDATE employee SET role =’Manager’;*

This statement will update the role to ‘Manager’ to all the *employees*.

## **DELETE** Command

This command helps in removing or erasing the saved records from the database tables. It erases single or multiple rows from the tables of the database.

* SQL command to delete a specific record

*# DELETE FROM employee WHERE id = 1*

This statement will delete the row with *id* 1.

* SQL command to delete multiple rows

*# DELETE FROM employee WHERE role = ‘MANAGER’*

This statement will delete all the *employees* which have the *role* ‘Manager’

* SQL command to delete all rows.

*# DELETE FROM employee;*

This statement will delete all the rows of *employees*.

We can also use TRUNCATE statement to delete all rows from the table.

*# TRUNCATE FROM employee;*

This statement will delete all the rows of *employees*.

You should always use the DELETE statement with caution.

## **DROP** Command

This command helps in deleting the entire databases, table, table view, and other objects from the database.

* **DROP Database**

SQL command will delete a database named testdb.

*# DROP DATABASE testdb;*

This will delete the database from the DBMS.

* **Drop Table**

SQL command will delete a table named *employee*.

*# DROP TABLE employee;*

This will delete the table from the database.

# SQL Filtering

One of the most powerful features of a SQL DBMS is the ability to [filter](https://swcarpentry.github.io/sql-novice-survey/reference.html#filter) data, i.e., to select only those records that match certain criteria.

Let’s discuss some popular filtering clauses in SQL –

## **WHERE** Clause

The WHERE clause allows us to fetch records from a database table that matches specified condition(s).

# *SELECT \* from employee WHERE id = 1;*

This statement will return the employee whose id is 1.

 You can use various operators to form the row selection criteria used in the WHERE clause.

There are different types of operators available –

### a. **Comparison Operator**

The SQL comparison operators allow you to test if two expressions are the same.

These are the comparison operators -

|  |  |  |
| --- | --- | --- |
| **Operator** | **Meaning** | **Example** |
| = | Equal | ***SELECT \* from employee WHERE id = 1;***  This will return employee whose id is 1 |
| <> | Not equal to | ***SELECT \* from employee WHERE id = 1;***  This will return all employee whose id is not 1 |
| > | Greater than | ***SELECT \* from employee WHERE id > 4;***  This will return employees whose id is greater than 4 |
| < | Less than | ***SELECT \* from employee WHERE id < 4;***  This will return employees whose id is less than 4 |
| >= | Greater than or equal to | ***SELECT \* from employee WHERE id >= 4;***  This will return employees whose id is >= 4 |
| <= | Less than or equal to | ***SELECT \* from employee WHERE id <= 4;***  This will return employees whose id is <= 4 |

### b. **Logical Operator**

A logical operator allows you to test for the truth of a condition. Similar to a [comparison operator](https://www.sqltutorial.org/sql-comparison-operators/), a logical operator returns a value of true, false, or unknown.

The operators are –

* **AND** Operator

The SQL AND operator select data if all conditions are TRUE

# *SELECT \* from employee WHERE first\_name = ‘John’ and last\_name = ‘Snow’;*

This statement will return all the rows, which have *first\_name* as ‘John’ and *last\_name* as ‘Snow’.

* **OR** Operator

The SQL OR operator select data if any one condition are TRUE

# *SELECT \* from employee WHERE first\_name = ‘John’ OR last\_name = ‘Snow’;*

This statement will return all the rows, which have *first\_name* as ‘John’ or *last\_name* as ‘Snow’.

* **NOT** Operator

The SQL NOT operator selects data if the given condition is FALSE.

# *SELECT \* from employee WHERE NOT first\_name = ‘John’;*

This statement will return all the rows, where *first\_name* is not ‘John’.

* **IS NULL** Operator

The SQL `IS NULL` operator, compares a value with a null value and returns true if the compared value is null; otherwise, it returns false.

# *SELECT \* from employee WHERE first\_name IS NULL;*

This statement will return all the rows, where *first\_name* is empty.

* **IS NOT NULL** Operator

The SQL `IS NOT NULL` operator, condition is used to select rows if the specified field is NOT NULL.

# *SELECT \* from employee WHERE first\_name IS NOT NULL;*

This statement will return all the rows, where *first\_name* is not empty.

* **IN** Operator

The SQL IN operator is used with the [WHERE](https://www.programiz.com/sql/select) clause to match values in a list.

# *SELECT \* from employee WHERE role IN (‘Manager’, ‘CEO’);*

This statement will return all the rows, where *role* are either ‘Manager’ or ‘CEO’.

* **NOT IN** Operator

The SQL `NOT IN` operator is used with the [WHERE](https://www.programiz.com/sql/select) clause to does not match values in a list.

# *SELECT \* from employee WHERE role NOT IN (‘Manager’, ‘CEO’);*

This statement will return all the rows, where *role* are not ‘Manager’ and ‘CEO’.

* **BETWEEN** Operator

The SQL BETWEEN operator is used with the [WHERE](https://www.programiz.com/sql/select) clause to match values in a range.

# *SELECT \* from employee WHERE id BETWEEN 4 AND 10;*

This statement will return all the rows, whose *id* are between 4 and 10, including 4 and 10.

* **NOT BETWEEN** Operator

The SQL NOT BETWEEN operator is used with the [WHERE](https://www.programiz.com/sql/select) clause to match values not in a range.

# *SELECT \* from employee WHERE id NOT BETWEEN 4 AND 10;*

This statement will return all the rows, whose *id* are not between 4 and 10, including 4 and 10.

* **LIKE** Operator

The LIKE operator in SQL is used with the [WHERE clause](https://www.programiz.com/sql/select#sql-where) to get a result set that matches the given string pattern.

SQL provides two wildcards used in conjunction with the LIKE operator:

* The percent sign ( %) represents zero, one, or multiple characters.
* The underscore sign ( \_) represents a single character.

# *SELECT \* from employee WHERE f\_name LIKE ‘%a’;*

This statement will return the *employee* which have last character of *first\_name* as ‘a’.

# *SELECT \* from employee WHERE f\_name LIKE ‘\_a’;*

This statement will return the *employee* which have any first character and second character as ‘a’

## **DISTINCT** Clause

The SQL SELECT DISTINCT statement selects unique rows from a database table.

# *SELECT DISTINCT role from employee;*

This statement will return the distinct role from the table employee.

## **LIMIT** Clause

The LIMIT command is used to select a fixed number of rows from a database.

# *SELECT \* from employee LIMIT 2;*

This statement will return the first 2 rows from the *employee* table.

# Aggregate Functions in SQL

An SQL aggregate function calculates on a set of values and returns a single value.

There are 5 types of aggregate functions in SQL.

## **COUNT** Function

The COUNT function returns the number of rows in the result set.

*# SELECT COUNT(\*) from employee;*

This statement will return the number of rows in the table *employee*.

*# SELECT COUNT(\*) from employee where role = ‘Manager’;*

This statement will return the number of employees with *role* ‘Manager’ in the table *employee*.

## **MIN** Function

The [MIN](https://www.sqltutorial.org/sql-aggregate-functions/sql-min/) function returns the minimum value of a set.

*# SELECT MIN(salary) from employee;*

This statement will return the minimum salary from the *salary* in the table *employee*.

*# SELECT MIN(salary) from employee where role = ‘Manager’;*

This statement will return the minimum salary from the *salary* with *role* ‘Manager’ in the table *employee*.

## **MAX** Function

The [MAX](https://www.sqltutorial.org/sql-aggregate-functions/sql-min/) function returns the maximum value of a set.

*# SELECT MAX(salary) from employee;*

This statement will return the maximum salary from the *salary* in the table *employee*.

*# SELECT MAX(salary) from employee where role = ‘Manager’;*

This statement will return the maximum salary from the *salary* with *role* ‘Manager’ in the table *employee*.

## **AVG** Function

The [AVG](https://www.sqltutorial.org/sql-aggregate-functions/sql-min/) function returns the average value of a set.

*# SELECT AVG(salary) from employee;*

This statement will return the average salary from the *salary* in the table *employee*.

*# SELECT AVG(salary) from employee where role = ‘Manager’;*

This statement will return the average salary from the *salary* with *role* ‘Manager’ in the table *employee*.

## **SUM** Function

The [SUM](https://www.sqltutorial.org/sql-aggregate-functions/sql-min/) function returns the sum value of a set.

*# SELECT SUM(salary) from employee;*

This statement will return the sum of all salary from the *salary* in the table *employee*.

*# SELECT SUM(salary) from employee where role = ‘Manager’;*

This statement will return the sum of salary from the *salary* with *role* ‘Manager’ in the table *employee*.

# Sorting Data in SQL

Sorting in SQL is done by ‘ORDER BY’ clause.

The ORDER BY is an optional clause of the [SELECT](https://www.sqltutorial.org/sql-select/) statement. The ORDER BY clause allows you to sort the rows returned by the SELECT clause by one or more sort expressions in ascending or descending order.

*# SELECT \* from employee ORDER BY first\_name;*

the SQL command selects all employees and then sorts them in ascending order by *first\_name*.

*# SELECT \* from employee ORDER BY first\_name desc;*

the SQL command selects all employees and then sorts them in descending order by *first\_name*.

By default, ORDER BY clause orders the data in ascending order.

# Joins: SQL

Joins in SQL are used to combine records from one or more tables in a database. A JOIN is a means for combining fields from multiple tables by using values common to each.

The common columns are typically the [**primary key**](https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-primary-key/) columns of the first table and [**foreign key**](https://www.postgresqltutorial.com/postgresql-foreign-key/) columns of the second table.

Let’s assume we have two tables as –

1. *employee (emp\_id, fname, lname, salary, dept\_id)* - *emp\_id* is primary key, *dept\_id* is foreign key

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Emp\_id | Fname | Lname | Salary | Dept\_id |
| E001 | Tom | Hardy | 50000 | D001 |
| E002 | Sheldon | Cooper | 45000 | D002 |
| E003 | John | Snow | 60000 | D001 |
| E004 | Ramesh | K | 80000 | D004 |
| E005 | Rocky | D | 90000 | D002 |
| E006 | Micheal | Nanty | 12000 | D001 |
| E007 | Sunny | Koop | 70000 | D008 |

2. *department (dept\_id, name, supervisor)* – *dept\_id* is the primary key.

|  |  |  |
| --- | --- | --- |
| Dept\_id | Name | Supervisor |
| D001 | Meterology | Ramesh |
| D002 | Physics | Koonk |
| D003 | Chemistry | Tony |
| D004 | Meta | Quont |

There are different types of Joins available –

## **INNER** Join

The SQL **INNER JOIN** joins two tables based on a common column, and selects records that have matching values in these columns.

*#SELECT employee.emp\_id, department.name, department.supervisor*

*#FROM employee*

*#INNER JOIN department*

*#ON employee.dept\_id = department.dept\_id;*

The inner join returns a result set that contains row in the left table that matches the row in the right table.

The output should be –

|  |  |  |
| --- | --- | --- |
| Emp\_id | Name | supervisor |
| E001 | Meterology | Ramesh |
| E002 | Physics | Koonk |
| E003 | Meterology | Ramesh |
| E004 | Meta | Quont |
| E005 | Physics | Koonk |
| E006 | Meterology | Ramesh |

## **LEFT** Join

The SQL **LEFT JOIN** joins two tables based on a common column, and selects records that have matching values in these columns and remaining rows from the left table.

*#SELECT employee.emp\_id, department.name, department.supervisor*

*#FROM employee*

*#LEFT JOIN department*

*#ON employee.dept\_id = department.dept\_id;*

The left join returns a complete set of rows from the left table with the matching rows if available from the right table. If there is no match, the right side will have null values.

The output should be –

|  |  |  |
| --- | --- | --- |
| Emp\_id | Name | supervisor |
| E001 | Meterology | Ramesh |
| E002 | Physics | Koonk |
| E003 | Meterology | Ramesh |
| E004 | Meta | Quont |
| E005 | Physics | Koonk |
| E006 | Meterology | Ramesh |
| E007 |  |  |

## **RIGHT** Join

The SQL **RIGHT JOIN** joins two tables based on a common column, and selects records that have matching values in these columns and remaining rows from the right table.

*#SELECT employee.emp\_id, department.name, department.supervisor*

*#FROM employee*

*#RIGHT JOIN department*

*#ON employee.dept\_id = department.dept\_id;*

The right join returns a complete set of rows from the right table with the matching rows if available from the left table. If there is no match, the left side will have null values.

The output should be –

|  |  |  |
| --- | --- | --- |
| Emp\_id | Name | supervisor |
| E001 | Meterology | Ramesh |
| E002 | Physics | Koonk |
| E003 | Meterology | Ramesh |
| E004 | Meta | Quont |
| E005 | Physics | Koonk |
| E006 | Meterology | Ramesh |
|  | Chemistry | Tony |

## **FULL** Join

The SQL **FULL OUTER** JOIN joins two tables based on a common column, and selects records that have matching values in these columns and remaining rows from both of the tables.

*#SELECT employee.emp\_id, department.name, department.supervisor*

*#FROM employee*

*#FULL OUTER JOIN department*

*#ON employee.dept\_id = department.dept\_id;*

The full outer join or full join returns a result set that contains all rows from both the left and right tables, with the matching rows from both sides where available. If there is no match, the missing side contains null values.

The output should be –

|  |  |  |
| --- | --- | --- |
| Emp\_id | Name | supervisor |
| E001 | Meterology | Ramesh |
| E002 | Physics | Koonk |
| E003 | Meterology | Ramesh |
| E004 | Meta | Quont |
| E005 | Physics | Koonk |
| E006 | Meterology | Ramesh |
| E007 |  |  |
|  | Chemistry | Tony |

## **CROSS** JOIN

A **CROSS JOIN** allows you to produce a Cartesian Product of rows in two or more tables.

If table 1 has n rows, and table 2 has m rows, then after cross join the combined table will have ‘n\*m’ rows.

There are two tables (T1, and T2) –

|  |
| --- |
| score |
| 1 |
| 2 |

|  |
| --- |
| player |
| A |
| B |

Then cross join, can be calculated as -

*#SELECT \**

*#FROM T1*

*#CROSS JOIN T2;*

The output should have 2\*2 = 4 rows –

|  |  |
| --- | --- |
| Score | Player |
| 1 | A |
| 2 | A |
| 1 | B |
| 2 | B |

## **SELF** JOIN

SQL has a special type of join called the SELF JOIN which is used to join a table with itself. It comes in handy when comparing the column of rows within the same table.

To form a self-join, you specify the same table twice with [different table aliases](https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-alias/) and provide the join predicate after the ON keyword.

It is also important to note that there is**no such keyword as SELF JOIN**, but it is achieved with the help of LEFT JOIN, RIGHT JOIN or INNER JOIN using aliases.

Let’s take an example. We will take the same employee table, and add one more field to it as ‘supervisor\_id’.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Emp\_id | Fname | Lname | Salary | supervisor\_id |
|  |  |  |  |  |
| E001 | Tom | Hardy | 50000 | E001 |
| E002 | Sheldon | Cooper | 45000 | E001 |
| E003 | John | Snow | 60000 | E002 |
| E004 | Ramesh | K | 80000 | E003 |
| E005 | Rocky | D | 90000 | E001 |

Now, if we are interested to get the employee and its supervisor, we can do self join –

*#SELECT e.fname employee, s.fname supervisor*

*#FROM employee as e*

*#INNER JOIN employee as s*

*#ON e.emp\_id = s.emp\_id;*

The answer should be -

|  |  |
| --- | --- |
| *employee* | *supervisor* |
| Tom | Tom |
| Sheldon | Tom |
| John | Sheldon |
| Ramesh | John |
| Rocky | Tom |

# Grouping and Subqueries