# What Is SQL?

SQL stands for *Structured Query Language* – a declarative language for manipulating and retrieving data stored in a relational database. It’s the most popular query language over databases.

Others:

* Comparison of different SQL implementations (MySQL, MSSQL, PostgreSQL, Oracle) and syntaxes: <http://troels.arvin.dk/db/rdbms/>

# Data Types

Each column in a database table is required to have a **name** and a **data type**. The data type is a guideline for SQL to understand what type of data is expected inside of each column, and it also identifies how SQL will interact with the stored data.

**Note**: Different database define different names for data types. And even if the name is the same, the size and other details may be different as well. Always check the documentation!

List of data types of MySQL, SQL Server and MS Access SQL:

<https://www.w3schools.com/sql/sql_datatypes.asp>

# Quoted Identifiers

**Question**

In MySQL, we can create queries with or without the backtick (`). Example:

SELECT \* FROM test;

SELECT \* FROM `test`;

Is there any technical difference between them?

**Answer:**

They are called **quoted identifiers** which tell the parser to handle the text between them as a literal string. They are useful for when you have a column/table that contains a **space** or **keyword**.

For example:

The following would not work:

CREATE TABLE my table (id INT);

But the following would:

CREATE TABLE `my table` (id INT);

Also:

The following would get an error, because COUNT is a reserved keyword:

SELECT count FROM some\_table

But the following works fine:

SELECT `count` FROM some\_table

**Double Quotes vs Backticks vs Square Brackets:**

The Standard SQL mechanism for quoting identifiers is enclosed in **double quotes**:

SELECT "select" FROM "from" WHERE "where" = "group by";

In MySQL, you can use **backticks**:

SELECT `select` FROM `from` WHERE `where` = `group by`;

In MS SQL Server, you can use **square brackets**:

SELECT [select] FROM [from] WHERE [where] = [group by];

# Permission

Normally, root should only be allowed to connect from 'localhost'. This ensures that someone cannot guess at the root password from the network. So disallow root login remotely.

# Database Statements

## CREATE DATABASE

The CREATE DATABASE statement is used to create a new SQL database.

**Syntax**

CREATE DATABASE Database-Name;

Notes:

* The database name must be **unique** within the RDBMS.
* Make sure you have the **admin privilege** before creating any database.

**Example**

If you want to create a new database named TestDB, use:

SQL> CREATE DATABASE TestDB;

Once a database is created, you can check it in the list of databases as follows:

SQL> SHOW DATABASES;

+---------------+

| Database      |

+---------------+

| Armood        |

| TutorialPoint |

| Orig          |

| TestDB        |

+---------------+

## DROP DATABASE

The DROP DATABASE statement is used to delete an existing database in SQL schema.

**Syntax**

DROP DATABASE Database-Name;

Note: Make sure you have the admin privilege before creating any database.

**Example**

If you want to delete an existing database named TestDB, use:

SQL> DROP DATABASE TestDB;

Once a database is dropped, you can check it in the list of databases as follows:

-- In MySQL:

SQL> SHOW DATABASES;

-- In SQL Server:

SQL> SELECT \* FROM sys.databases

+---------------+

| Database      |

+---------------+

| Armood        |

| TutorialPoint |

| Orig          |

+---------------+

## USE

When you have multiple databases in your SQL Schema, then before starting your operation, you would need to select a database where all the operations would be performed.

The USE statement is used to select any existing database in the SQL schema.

**Syntax**

USE Database-Name;

**Example**

You can check the available databases as shown below:

-- In MySQL:

SQL> SHOW DATABASES;

-- In SQL Server:

SQL> SELECT \* FROM sys.databases

+---------------+

| Database      |

+---------------+

| Armood        |

| TutorialPoint |

| Orig          |

| TestDB        |

+---------------+

Now, if you want to work with the TestDB database, use:

SQL> USE TestDB;

# Table Statements

## CREATE TABLE

Creating a basic table involves naming the table and defining its columns and each column's data type.

The CREATE TABLE statement is used to create a new table.

**Syntax**

CREATE TABLE Table-Name(

column1 datatype,

column2 datatype,

column3 datatype,

.....

columnN datatype,

PRIMARY KEY(one or more columns)

);

**Example**

The following example creates a Customers table with an ID as a primary key. The constraint NOT NULL show that these fields cannot be NULL:

SQL> CREATE TABLE Customers(

ID INT NOT NULL,

Name VARCHAR (20) NOT NULL,

Age INT NOT NULL,

Address CHAR (25),

Salary DECIMAL (18, 2),

PRIMARY KEY (ID)

);

You can verify if your table has been created successfully by looking at the message displayed by the SQL server. Or you can use the DESC or EXEC sp\_help command as follows:

-- In MySQL:

SQL> DESC Customers;

+---------+---------------+------+-----+---------+-------+

| Field | Type | Null | Key | Default | Extra |

+---------+---------------+------+-----+---------+-------+

| ID | int(11) | NO | PRI | | |

| Name | varchar(20) | NO | | | |

| Age | int(11) | NO | | | |

| Address | char(25) | YES | | NULL | |

| Salary | decimal(18,2) | YES | | NULL | |

+---------+---------------+------+-----+---------+-------+

-- In SQL Server:

SQL> EXEC sp\_help 'Customers'

+------------+---------+---------+--------+------+------+-------------+ ...

| Column\_name | Type | Computed | Length | Prec | Scale | Nullable | ...

+ ----------- + ------- + -------- + ------ + ---- + ----- + -------- + ...

| ID | int | no | 4 | 10 | 0 | no |

| Name | varchar | no | 20 | | | no |

| Age | int | no | 4 | 10 | 0 | no |

| Address | char | no | 25 | | | yes |

| Salary | decimal | no | 9 | 18 | 2 | yes |

+-------------+---------+----------+--------+------+-------+----------+ ...

## DROP TABLE

The DROP TABLE statement is used to remove a table definition and all of its data (values, indexes, triggers, constraints and permission specifications).

**Syntax**

DROP TABLE table-name;

**Example**

Let us first verify the Customers table and then we will delete it from the database as shown below:

SQL> DESC Customers;

+---------+---------------+------+-----+---------+-------+

| Field | Type | Null | Key | Default | Extra |

+---------+---------------+------+-----+---------+-------+

| ID | int(11) | NO | PRI | | |

| Name | varchar(20) | NO | | | |

| Age | int(11) | NO | | | |

| Address | char(25) | YES | | NULL | |

| Salary | decimal(18,2) | YES | | NULL | |

+---------+---------------+------+-----+---------+-------+

This means that the Customers table is available in the database.

Now drop it:

SQL> DROP TABLE Customers;

Now, if you would try the DESC command, then you will get the following error:

SQL> DESC Customers;

ERROR 1146 (42S02): Table 'TEST.Customers' doesn't exist

Here, TEST is the database name which we are using for our examples.

## TRUNCATE TABLE

The TRUNCATE TABLE command is used to **delete complete data from an existing table, but not the table itself**.

**Syntax**

TRUNCATE TABLE table-name;

**Example**

Consider a CUSTOMERS table having the following records:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | Kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

Let's truncate it:

SQL> TRUNCATE TABLE CUSTOMERS;

Now all records are deleted:

SQL> SELECT \* FROM CUSTOMERS;

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

## ALTER

The ALTER TABLE command is used to **add, delete or modify columns** in an existing table. You can also use it to **add and drop constraints** on an existing table.

**Syntax**

-- To add column:

ALTER TABLE table-name ADD column-name data-type;

-- To drop column:

ALTER TABLE table-name DROP COLUMN column-name;

-- To change the data type:

ALTER TABLE table-name MODIFY COLUMN column-name data-type;

-- To add a NOT NULL constraint:

ALTER TABLE table-name MODIFY column-name data-type NOT NULL;

-- To add a UNIQUE constraint:

ALTER TABLE table-name

ADD CONSTRAINT constraint-name UNIQUE(column1, column2...);

-- To add a CHECK constraint:

ALTER TABLE table-name

ADD CONSTRAINT constraint-name CHECK (condition);

-- To add a PRIMARY KEY constraint:

ALTER TABLE table-name

ADD CONSTRAINT constraint-name PRIMARY KEY (column1, column2...);

-- To add a FOREIGN KEY constraint:

ALTER TABLE table1-name

ADD CONSTRAINT constraint-name

FOREIGN KEY (column-name) REFERENCES table2-name(column-name);

-- To drop PRIMARY KEY constraint:

-- In MySQL:

ALTER TABLE table-name

DROP PRIMARY KEY;

-- In SQL Server:

ALTER TABLE table-name

DROP CONSTRAINT primary-key-name;

-- To drop FOREIGN KEY constraint:

-- In MySQL:

ALTER TABLE table-name

DROP FOREIGN KEY;

-- In SQL Server:

ALTER TABLE table-name

DROP CONSTRAINT primary-key-name;

Many more...

**Example**

Consider the Customers table having the following records:

+----+----------+-----+-----------+----------+

| ID | Name | Age | Address | Salary |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | Kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

To add a new column to an existing table:

SQL> ALTER TABLE Customers ADD Gender char(1);

Output:

+----+---------+-----+-----------+----------+--------+

| ID | Name | Age | Address | Salary | Gender |

+----+---------+-----+-----------+----------+--------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 | NULL |

| 2 | Khilan | 25 | Delhi | 1500.00 | NULL |

| 3 | Kaushik | 23 | Kota | 2000.00 | NULL |

To drop the Gender column from the existing table:

SQL> ALTER TABLE Customers DROP Gender;

Output:

+----+----------+-----+-----------+----------+

| ID | Name | Age | Address | Salary |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | Kaushik | 23 | Kota | 2000.00 |

Examples to add / drop a PRIMARY KEY constraint, and add / drop a FOREIGN KEY constraint are at [this section](#_PRIMARY_KEY_and).

## SELECT

### Without Condition

The SELECT statement is used to fetch the data from a database table which returns this data in the form of a result table.

**Syntax**

SELECT column1, column2, ... columnN FROM table-name;

Tip: If you want to fetch all fields available in the table, just need to use a \* symbol instead of all column names:

SELECT \* FROM table-name;

**Example**

Consider the Customers table having the following records:

+----+----------+-----+-----------+----------+

| ID | Name | Age | Address | Salary |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | Kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

If you want to fetch the ID, Name and Salary fields in Customers table, use:

SQL> SELECT ID, Name, Salary FROM Customers;

Output:

+----+----------+----------+

| ID | Name | Salary |

+----+----------+----------+

| 1 | Ramesh | 2000.00 |

| 2 | Khilan | 1500.00 |

| 3 | Kaushik | 2000.00 |

| 4 | Chaitali | 6500.00 |

| 5 | Hardik | 8500.00 |

| 6 | Komal | 4500.00 |

| 7 | Muffy | 10000.00 |

+----+----------+----------+

If you want to fetch all the fields of the Customers table, use:

SQL> SELECT \* FROM Customers;

This would produce the result as shown below:

+----+----------+-----+-----------+----------+

| ID | Name | Age | Address | Salary |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | Kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

### With Condition

To specify a condition while fetching the data from a single table, you can use the SQL WHERE clause. It is very useful to filter the records and fetch only necessary ones.

**Syntax**

SELECT column1, column2, columnN

FROM table-name

WHERE [condition]

You can specify a condition using the comparison or logical operators like >, <, =, LIKE, NOT, etc. The following examples would make this concept clear.

**Example**

Consider the Customers table having the following records:

+----+----------+-----+-----------+----------+

| ID | Name | Age | Address | Salary |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | Kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

To fetch the ID, Name and Salary fields from the Customers table where the salary is greater than 2000, use:

SQL> SELECT ID, Name, Salary

FROM Customers

WHERE Salary > 2000;

Output:

+----+----------+----------+

| ID | Name | Salary |

+----+----------+----------+

| 4 | Chaitali | 6500.00 |

| 5 | Hardik | 8500.00 |

| 6 | Komal | 4500.00 |

| 7 | Muffy | 10000.00 |

+----+----------+----------+

### SELECT INTO

The SELECT INTO statement copies data from one table into a new table.

The new table will be created with the column names and types as defined in the old table. But you can create new column names using the AS clause.

**Syntax**

SELECT column1, column2, column3, ...

INTO new-table [IN external-db]

FROM old-table

WHERE condition;

**Examples**

The following statement creates a backup copy of Customers:

SQL> SELECT \* INTO CustomersBk

FROM Customers

The following statement uses the IN clause to copy the table into a new table in another database:

SQL> SELECT \* INTO CustomersBk IN 'Backup.mdb'

FROM Customers;

The following statement copies only CustomerName and ContactName columns into a new table:

SQL> SELECT CustomerName, ContactName INTO CustomersBk

FROM Customers;

The following statement copies only the German customers into a new table:

SQL> SELECT \* INTO CustomersGermany

FROM Customers

WHERE Country = 'Germany';

### SELECT DISTINCT

The SQL DISTINCT keyword is used in conjunction with the SELECT statement to eliminate all the duplicate records and fetching only unique records.

**Syntax**

SELECT DISTINCT column1, column2, ... columnN

FROM table-name

WHERE [condition]

**Example**

Consider the Customers table having the following records:

+----+----------+-----+-----------+----------+

| ID | Name | Age | Address | Salary |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | Kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

First, let us see how the following SELECT query returns the duplicate salary records.

SQL> SELECT Salary FROM Customers;

Output, where the salary (2000) is coming twice:

+----------+

| Salary |

+----------+

| 1500.00 |

| 2000.00 |

| 2000.00 |

| 4500.00 |

| 6500.00 |

| 8500.00 |

| 10000.00 |

+----------+

Now, let us use the DISTINCT with SELECT and see the result:

SQL> SELECT DISTINCT Salary FROM Customers;

Output where we do not have any duplicate entry:

+----------+

| Salary |

+----------+

| 1500.00 |

| 2000.00 |

| 4500.00 |

| 6500.00 |

| 8500.00 |

| 10000.00 |

+----------+

## INSERT INTO

The INSERT INTO statement is used to add new rows of data to a table in the database.

**Syntax**

INSERT INTO Table-Name (column1, column2, column3, ... columnN)

VALUES (value1, value2, value3, ... valueN);

Note: Make sure column names and values are in the same order as the columns in the table.

**Examples**

Following statements insert 2 new records to the Customers table:

SQL>

INSERT INTO Customers (ID, Name, Age, Address, Salary)

VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );

INSERT INTO Customers (ID, Name, Age, Address, Salary)

VALUES (2, 'Khilan', 25, 'Delhi', 1500.00 );

All the above statements would produce the following records in the Customers table as shown below:

+----+----------+-----+-----------+----------+

| ID | Name | Age | Address | Salary |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

+----+----------+-----+-----------+----------+

A neater way to insert multiple records to the Customers table is:

SQL>

INSERT INTO Customers (ID, Name, Age, Address, Salary)

VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 ),

(2, 'Khilan', 25, 'Delhi', 1500.00 ),

(3, 'Kaushik', 23, 'Kota', 2000.00 ),

(4, 'Chaitali', 25, 'Mumbai', 6500.00 ),

(5, 'Hardik', 27, 'Bhopal', 8500.00 ),

(6, 'Komal', 22, 'MP', 4500.00 ),

(7, 'Muffy', 24, 'Indore', 10000.00 );

Output:

+----+----------+-----+-----------+----------+

| ID | Name | Age | Address | Salary |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | Kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

### Populate One Table from Another Table

You can populate the data into a table over another table; provided the other table has a set of fields, which are required to populate the first table.

Here is the syntax:

INSERT INTO first-table-name [(column1, column2, ... columnN)]

SELECT column1, column2, ...columnN

FROM second-table-name

[WHERE condition];

### INSERT INTO SELECT

<https://www.w3schools.com/sql/sql_insert_into_select.asp>

## UPDATE

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

**Syntax**

UPDATE table-name

SET column1 = value1, column2 = value2...., columnN = valueN

WHERE [condition];

You can use the WHERE clause with the UPDATE query to update the selected rows, otherwise all the rows would be affected.

**Example**

Consider the Customers table having the following records:

+----+----------+-----+-----------+----------+

| ID | Name | Age | Address | Salary |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | Kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

The following query will update the Address for a customer whose ID number is 6 in the table.

SQL> UPDATE Customers

SET Address = 'Pune'

WHERE ID = 6;

Now, the Customers table would have the following records:

+----+----------+-----+-----------+----------+

| ID | Name | Age | Address | Salary |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | Kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | Pune | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

If you want to modify all the Address and the Salary column values in the Customers table, you do not need to use the WHERE clause:

SQL> UPDATE Customers

SET Address = 'Pune', Salary = 1000.00;

Now, Customers table would have the following records:

+----+----------+-----+---------+---------+

| ID | Name | Age | Address | Salary |

+----+----------+-----+---------+---------+

| 1 | Ramesh | 32 | Pune | 1000.00 |

| 2 | Khilan | 25 | Pune | 1000.00 |

| 3 | Kaushik | 23 | Pune | 1000.00 |

| 4 | Chaitali | 25 | Pune | 1000.00 |

| 5 | Hardik | 27 | Pune | 1000.00 |

| 6 | Komal | 22 | Pune | 1000.00 |

| 7 | Muffy | 24 | Pune | 1000.00 |

+----+----------+-----+---------+---------+

## DELETE

The DELETE statement is used to delete the existing records from a table.

**Syntax**

DELETE FROM table-name

WHERE [condition];

You can use the WHERE clause with a DELETE query to delete the selected rows, otherwise all the records would be deleted.

**Example**

Consider the Customers table having the following records:

+----+----------+-----+-----------+----------+

| ID | Name | Age | Address | Salary |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | Kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

The following code has a query, which will DELETE a customer, whose ID is 6.

SQL> DELETE FROM Customers

WHERE ID = 6;

Now, the Customers table would have the following records:

+----+----------+-----+-----------+----------+

| ID | Name | Age | Address | Salary |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | Kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

If you want to DELETE all the records from the Customers table, you do not need to use the WHERE clause:

SQL> DELETE FROM Customers;

Now, the Customers table would not have any record.

## JOIN

The JOIN clause is used to combine records from two or more tables in a database by using values common to each.

**Different Types of JOIN:**

* INNER JOIN − returns rows when there is a match in both tables.
* LEFT JOIN − returns all rows from the left table, even if there are no matches in the right table.
* RIGHT JOIN − returns all rows from the right table, even if there are no matches in the left table.
* FULL JOIN − returns rows when there is a match in one of the tables.
* SELF JOIN − join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.
* CARTESIAN JOIN − returns the Cartesian product of the sets of records from the two or more joined tables.

### INNER JOIN



**Syntax**

SELECT column1.column1, column2.column2, ...

FROM table1

INNER JOIN table2

ON table1.common\_column = table2.common\_column;

**Example**

Consider the Customers table having the following records:

+----+----------+-----+-----------+----------+

| ID | Name | Age | Address | Salary |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | Kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

And the Orders table having the following records:

+-----+---------------------+-------------+--------+-----------+

| OID | Date | CustomerID | Amount | ShipperID |

+-----+---------------------+-------------+--------+-----------+

| 102 | 2009-10-08 00:00:00 | 3 | 3000 | S1 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 | S2 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 | S3 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 | S6 |

+-----+---------------------+-------------+--------+-----------+

And the Shippers table having the following records:

+-----+-------------------+------------------+

| ID | ShipperName | Phone |

+-----+-------------------+------------------+

| S1 | Speedy Express | (503) 555-9831 |

| S2 | United Package | (503) 555-3199 |

| S3 | Federal Shipping | (503) 555-9931 |

| S4 | Plane Service | (503) 555-2060 |

+-----+-------------------+------------------+

The following statement selects all common orders (with customer ID and name, order amount and date) from 2 tables – Customers and Orders:

SQL> SELECT Customers.ID, Customers.Name, Orders.Amount, Orders.Date

FROM Customers

INNER JOIN Orders ON Customers.ID = Orders.CustomerID;

Output:

+----+----------+--------+---------------------+

| ID | Name | Amount | Date |

+----+----------+--------+---------------------+

| 3 | Kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | Kaushik | 1500 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

+----+----------+--------+---------------------+

The following SQL statement selects all orders (with customer ID and name, shipper name) from 3 tables – Customers, Orders, and Shippers:

SQL> SELECT Customers.ID, Customers.Name, Shippers.Name

FROM ((Customers

INNER JOIN Orders ON Customers.ID = Orders.CustomerID)

INNER JOIN Shippers ON Orders.ShipperID = Shippers.ID);

Output:

+----+----------+--------------------+

| ID | Name | ShipperName |

+----+----------+--------------------+

| 3 | Kaushik | Speedy Express |

| 3 | Kaushik | United Package |

| 2 | Khilan | Federal Shipping |

+----+----------+--------------------+

### LEFT JOIN



**Syntax**

SELECT column1.column1, column2.column2, ...

FROM table1

LEFT JOIN table2

ON table1.common\_column = table2.common\_column;

**Example**

Consider the Customers table having the following records:

+----+----------+-----+-----------+----------+

| ID | Name | Age | Address | Salary |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | Kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

And the Orders table having the following records:

+-----+---------------------+-------------+--------+-----------+

| OID | Date | CustomerID | Amount | ShipperID |

+-----+---------------------+-------------+--------+-----------+

| 102 | 2009-10-08 00:00:00 | 3 | 3000 | S1 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 | S2 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 | S3 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 | S6 |

+-----+---------------------+-------------+--------+-----------+

The following statement selects all orders of the Customers table (with customer ID and name, order amount and date) from 2 tables – Customers and Orders:

SQL> SELECT Customers.ID, Customers.Name, Orders.Amount, Orders.Date

FROM Customers

LEFT JOIN Orders ON Customers.ID = Orders.CustomerID;

Output:

+----+----------+--------+---------------------+

| ID | Name | Amount | Date |

+----+----------+--------+---------------------+

| 1 | Ramesh | NULL | NULL |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 3 | Kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | Kaushik | 1500 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

| 5 | Hardik | NULL | NULL |

| 6 | Komal | NULL | NULL |

| 7 | Muffy | NULL | NULL |

+----+----------+--------+---------------------+

### RIGHT JOIN



**Syntax**

SELECT column1.column1, column2.column2, ...

FROM table1

RIGHT JOIN table2

ON table1.common\_column = table2.common\_column;

**Example**

Consider the Customers table having the following records:

+----+----------+-----+-----------+----------+

| ID | Name | Age | Address | Salary |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | Kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

And the Orders table having the following records:

+-----+---------------------+-------------+--------+-----------+

| OID | Date | CustomerID | Amount | ShipperID |

+-----+---------------------+-------------+--------+-----------+

| 102 | 2009-10-08 00:00:00 | 3 | 3000 | S1 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 | S2 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 | S3 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 | S6 |

+-----+---------------------+-------------+--------+-----------+

The following statement selects all orders of the Orders table (with customer ID and name, order amount and date) from 2 tables – Customers and Orders:

SQL> SELECT Customers.ID, Customers.Name, Orders.Amount, Orders.Date

FROM Customers

RIGHT JOIN Orders ON Customers.ID = Orders.CustomerID;

Output:

+------+----------+--------+---------------------+

| ID | Name | Amount | Date |

+------+----------+--------+---------------------+

| 3 | Kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | Kaushik | 1500 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

+------+----------+--------+---------------------+

### FULL JOIN (OUTER JOIN)



**Syntax**

SELECT column1.column1, column2.column2, ...

FROM table1

FULL JOIN table2

ON table1.common\_column = table2.common\_column;

**Example**

Consider the Customers table having the following records:

+----+----------+-----+-----------+----------+

| ID | Name | Age | Address | Salary |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | Kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

And the Orders table having the following records:

+-----+---------------------+-------------+--------+-----------+

| OID | Date | CustomerID | Amount | ShipperID |

+-----+---------------------+-------------+--------+-----------+

| 102 | 2009-10-08 00:00:00 | 3 | 3000 | S1 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 | S2 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 | S3 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 | S6 |

+-----+---------------------+-------------+--------+-----------+

The following statement selects all orders of both tables (with customer ID and name, order amount and date) from 2 tables – Customers and Orders:

SQL> SELECT Customers.ID, Customers.Name, Orders.Amount, Orders.Date

FROM Customers

FULL JOIN Orders ON Customers.ID = Orders.CustomerID;

Output:

+------+----------+--------+---------------------+

| ID | Name | Amount | Date |

+------+----------+--------+---------------------+

| 1 | Ramesh | NULL | NULL |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 3 | Kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | Kaushik | 1500 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

| 5 | Hardik | NULL | NULL |

| 6 | Komal | NULL | NULL |

| 7 | Muffy | NULL | NULL |

| 3 | Kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | Kaushik | 1500 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

+------+----------+--------+---------------------+

**You might not know**:

If your Database does not support FULL JOIN (For example, MySQL), then you can use [UNION ALL](#_UNION) clause to combine LEFT JOIN and RIGHT JOIN to create FULL JOIN.

### SELF JOIN

<https://www.tutorialspoint.com/sql/sql-self-joins.htm>

### CARTESIAN JOIN

<https://www.tutorialspoint.com/sql/sql-cartesian-joins.htm>

## UNION

<https://www.w3schools.com/sql/sql_union.asp>

The UNION clause/operator is used to combine the results of two or more SELECT statements without **returning any duplicate rows**.

To use this UNION clause, each SELECT statement must have:

* The same number of columns selected
* The same number of column expressions
* The same data type and
* Have them in the same order. But they need not have to be in the same length.

**Syntax:**

SELECT column1 [, column2 ]

FROM table1 [, table2 ]

[WHERE condition]

UNION [ALL]

SELECT column1 [, column2 ]

FROM table1 [, table2 ]

[WHERE condition]

**Example:**

Consider the Customers table having the following records:

+----+----------+-----+-----------+----------+

| ID | Name | Age | Address | Salary |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | Kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

And the Orders table having the following records:

+-----+---------------------+-------------+--------+

|OID | Date | CustomerID | Amount |

+-----+---------------------+-------------+--------+

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

+-----+---------------------+-------------+--------+

Now join these two tables (excluding duplicate rows) in our SELECT statement as follows:

SQL> SELECT Customers.ID, Customers.Name, Orders.Amount, Orders.Date

FROM Customers

LEFT JOIN Orders ON Customers.ID = Orders.CustomerID;

UNION

SELECT Customers.ID, Customers.Name, Orders.Amount, Orders.Date

FROM Customers

RIGHT JOIN Orders ON Customers.ID = Orders.CustomerID;

Result:

+------+----------+--------+---------------------+

| ID | Name | Amount | Date |

+------+----------+--------+---------------------+

| 1 | Ramesh | NULL | NULL |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 3 | Kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | Kaushik | 1500 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

| 5 | Hardik | NULL | NULL |

| 6 | Komal | NULL | NULL |

| 7 | Muffy | NULL | NULL |

+------+----------+--------+---------------------+

Now join these two tables (including duplicate rows) in our SELECT statement as follows:

SQL> SELECT Customers.ID, Customers.Name, Orders.Amount, Orders.Date

FROM Customers

LEFT JOIN Orders ON Customers.ID = Orders.CustomerID;

UNION ALL

SELECT Customers.ID, Customers.Name, Orders.Amount, Orders.Date

FROM Customers

RIGHT JOIN Orders ON Customers.ID = Orders.CustomerID;

Result:

+------+----------+--------+---------------------+

| ID | Name | Amount | Date |

+------+----------+--------+---------------------+

| 1 | Ramesh | NULL | NULL |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 3 | Kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | Kaushik | 1500 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

| 5 | Hardik | NULL | NULL |

| 6 | Komal | NULL | NULL |

| 7 | Muffy | NULL | NULL |

| 3 | Kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | Kaushik | 1500 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

+------+----------+--------+---------------------+

# Operators

## Arithmetic Operators

Same as C: +, -, \*, / and %

## Comparison Operators

Same as C: =, !=, >, >=, <, <=

Others:

* <>: Same as !=
* !<: Not less than (equal or greater)
* !>: Not greater than (equal or less)

## Logical Operators

### AND

The AND operator is used to combine multiple conditions to narrow data in an SQL statement.

**Example**:

Consider the Customers table having the following records:

+----+----------+-----+-----------+----------+

| ID | Name | Age | Address | Salary |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | Kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

To fetch the ID, Name and Salary fields from the Customers table for a customer with name Hardik and salary greater than 2000, use:

SQL> SELECT ID, Name, Salary

FROM Customers

WHERE Name = 'Hardik' AND Salary > 2000;

Output:

+----+----------+----------+

| ID | Name | Salary |

+----+----------+----------+

| 5 | Hardik | 8500.00 |

+----+----------+----------+

### OR

The OR operator is used to combine multiple conditions to expand data in an SQL statement.

**Example**:

Consider the Customers table having the following records:

+----+----------+-----+-----------+----------+

| ID | Name | Age | Address | Salary |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | Kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

To fetch the ID, Name and Salary fields from the Customers table where the salary is greater than 2000 or the age is less than 25 years, use:

SQL> SELECT ID, Name, Salary

FROM Customers

WHERE Salary > 2000 OR age < 25;

Output:

+----+----------+----------+

| ID | Name | Salary |

+----+----------+----------+

| 3 | Kaushik | 2000.00 |

| 4 | Chaitali | 6500.00 |

| 5 | Hardik | 8500.00 |

| 6 | Komal | 4500.00 |

| 7 | Muffy | 10000.00 |

+----+----------+----------+

### NOT

The NOT operator **reverses the meaning of the logical operator** with which it is used.

E.g.: NOT EXISTS, NOT BETWEEN, NOT IN, etc.

### IN

The IN operator is a **shorthand for multiple OR conditions**.

**Example**:

Consider the Customers table having the following records:

+----+----------+-----+-----------+----------+

| ID | Name | Age | Address | Salary |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | Kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

To fetch the all fields which age is 32, 25 or 22 from all columns, use:

SQL> SELECT \*

FROM Customers

WHERE Age IN (32, 25, 22); -- Same as: WHERE Age = 32 OR Age = 25 OR Age = 22

Output:

+----+----------+-----+-----------+----------+

| ID | Name | Age | Address | Salary |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

### BETWEEN

The BETWEEN operator **selects values within a given range**. The values can be numbers, text, or dates.

**Example**:

Consider the Customers table having the following records:

+----+----------+-----+-----------+----------+

| ID | Name | Age | Address | Salary |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | Kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

To fetch the all fields which age is from 22 to 25 from all columns, use:

SQL> SELECT \*

FROM Customers

WHERE Age BETWEEN 22 AND 25; -- Same as: WHERE Age >= 22 AND Age <= 25

Output:

+----+----------+-----+-----------+----------+

| ID | Name | Age | Address | Salary |

+----+----------+-----+-----------+----------+

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | Kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

### LIKE

The LIKE operator is used to search for a specified pattern in a column.

There are some wildcards (used to substitute one or more characters in a string) often used in conjunction with the LIKE operator:

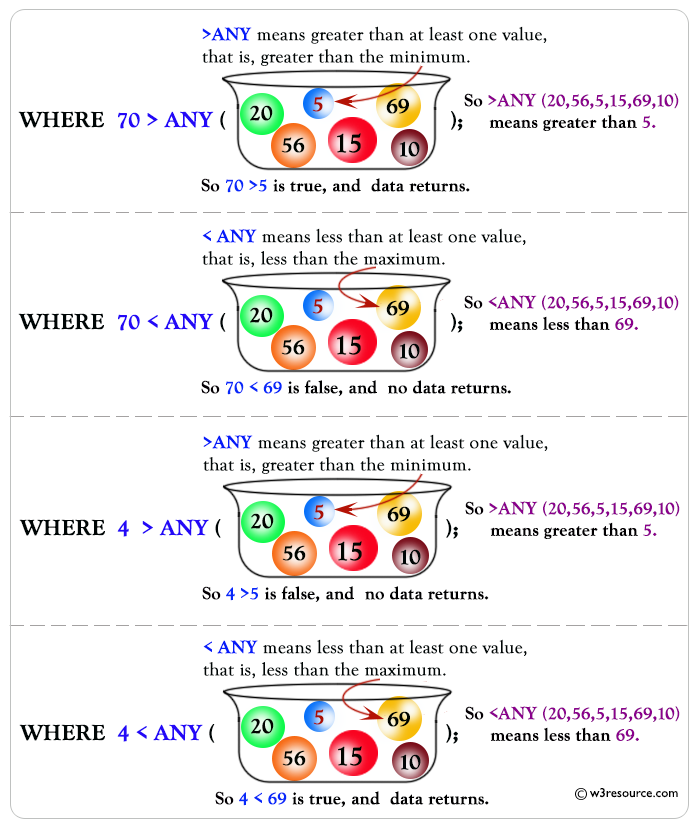
|  |  |  |
| --- | --- | --- |
| **Symbol** | **Description** | **Example** |
| % | Represents zero or more characters | bl% finds bl, black, blue, and blob |
| \_ | Represents a single character | h\_t finds hot, hat, and hit |
| [] | Represents any single character within the brackets | h[oa]t finds hot and hat, but not hit |
| ^ | Represents any character not in the brackets | h[^oa]t finds hit, but not hot and hat |
| - | Represents a range of characters | c[a-b]t finds cat and cbt |

Here are some examples showing different LIKE operators with '%' and '\_' wildcards:

|  |  |
| --- | --- |
| **LIKE Operator** | **Description** |
| WHERE Name LIKE 'a%' | Finds any values that start with "a" |
| WHERE Name LIKE '%a' | Finds any values that end with "a" |
| WHERE Name LIKE '%or%' | Finds any values that have "or" in any position |
| WHERE Name LIKE '\_r%' | Finds any values that have "r" in the second position |
| WHERE Name LIKE 'a\_%' | Finds any values that start with "a" and are at least 2 characters in length |
| WHERE Name LIKE 'a\_\_%' | Finds any values that start with "a" and are at least 3 characters in length |
| WHERE Name LIKE 'a%o' | Finds any values that start with "a" and ends with "o" |

### ANY

The ANY operator returns TRUE if **ANY of the subquery values meet the condition**. It's used with a WHERE or HAVING clause.



**Example:**

Consider the Customers table having the following records:

+----+----------+-----+-----------+----------+

| ID | Name | Age | Address | Salary |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | Kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

And the Orders table having the following records:

+-----+---------------------+-------------+--------+-----------+

| OID | Date | CustomerID | Amount | ShipperID |

+-----+---------------------+-------------+--------+-----------+

| 100 | 2009-10-08 00:00:00 | 2 | 1500 | S2 |

| 101 | 2009-11-20 00:00:00 | 7 | 1560 | S3 |

| 102 | 2009-10-08 00:00:00 | 3 | 3000 | S1 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 | S6 |

+-----+---------------------+-------------+--------+-----------+

Because the subquery returns CustomerID 3 (the smallest) and 4, the main query will return all ID and Name of customers having ID greater than 3, which is 4, 5, 6 and 7.

SQL> SELECT ID, Name

FROM Customers

WHERE ID > ANY (SELECT CustomerID FROM Orders WHERE Amount > 2000);

Output:

+----+----------+

| ID | Name |

+----+----------+

| 4 | Chantal |

| 5 | Hardik |

| 6 | Komal |

| 7 | Muffy |

Because the subquery returns CustomerID 3 (also the smallest), the main query will return all ID and Name of customers having ID equal to 3.

SQL> SELECT ID, Name

FROM Customers

WHERE ID = ANY (SELECT CustomerID FROM Orders WHERE Amount > 2500);

Output:

+----+----------+

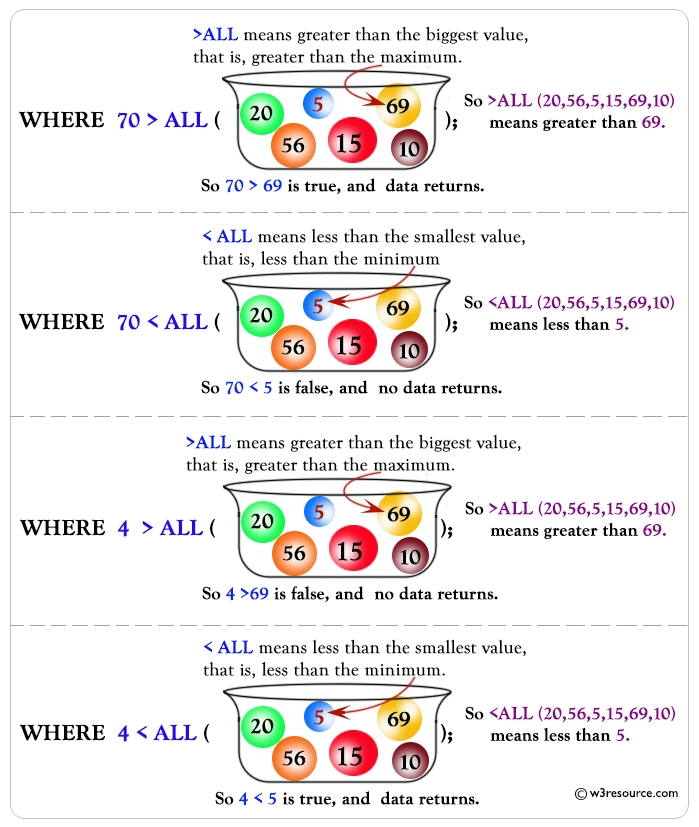
| ID | Name |

+----+----------+

| 3 | Kaushik |

### ALL

The ALL operator returns TRUE if **ALL of the subquery values meet the condition**. It's used with a WHERE or HAVING clause.



**Example:**

Consider the Customers table having the following records:

+----+----------+-----+-----------+----------+

| ID | Name | Age | Address | Salary |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | Kaushik | 23 | Kota | 2000.00 |

| 4 | Chantal | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

And the Orders table having the following records:

+-----+---------------------+-------------+--------+-----------+

| OID | Date | CustomerID | Amount | ShipperID |

+-----+---------------------+-------------+--------+-----------+

| 100 | 2009-10-08 00:00:00 | 2 | 1500 | S2 |

| 101 | 2009-11-20 00:00:00 | 7 | 1560 | S3 |

| 102 | 2009-10-08 00:00:00 | 3 | 3000 | S1 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 | S6 |

+-----+---------------------+-------------+--------+-----------+

Because the subquery returns CustomerID 3 (also the largest), the main query will return all ID and Name of customers having ID greater than 3, which is 4, 5, 6 and 7.

SQL> SELECT ID, Name

FROM Customers

WHERE ID > ALL (SELECT CustomerID FROM Orders WHERE Amount > 2500);

Output:

+----+----------+

| ID | Name |

+----+----------+

| 4 | Chantal |

| 5 | Hardik |

| 6 | Komal |

| 7 | Muffy |

Because the subquery returns CustomerID 2, 3, 4, 7 (the largest), the main query will return nothing because it finds no customers having ID greater than 7.

SQL> SELECT ID, Name

FROM Customers

WHERE ID > ALL (SELECT CustomerID FROM Orders WHERE Amount > 1500);

Output (no return record):

+----+----------+

| ID | Name |

+----+----------+

Because the subquery returns CustomerID 3 and 4, the main query will return nothing because it finds no customers having ID equal to both 3 and 4.

SQL> SELECT ID, Name

FROM Customers

WHERE ID = ALL (SELECT CustomerID FROM Orders WHERE Amount > 2000);

Output (no return record):

+----+----------+

| ID | Name |

+----+----------+

### EXISTS

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

**Example**:

Consider the Customers table having the following records:

+----+----------+-----+-----------+----------+

| ID | Name | Age | Address | Salary |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | Kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

And the Orders table having the following records:

+-----+---------------------+-------------+--------+-----------+

| OID | Date | CustomerID | Amount | ShipperID |

+-----+---------------------+-------------+--------+-----------+

| 102 | 2009-10-08 00:00:00 | 3 | 3000 | S1 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 | S2 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 | S3 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 | S6 |

+-----+---------------------+-------------+--------+-----------+

Because the subquery returns at least one record (so, it means TRUE), the main query will list all ID and Name of customers:

SQL> SELECT ID, Name

FROM Customers

WHERE EXISTS (SELECT OID FROM Orders WHERE ShipperID = 'S1');

Output:

+----+----------+

| ID | Name |

+----+----------+

| 1 | Ramesh |

| 2 | Khilan |

| 3 | Kaushik |

| 4 | Chaitali |

| 5 | Hardik |

| 6 | Komal |

| 7 | Muffy |

# Constraints

Constraints are the rules enforced on the data columns of a table. These are used to **limit the type of data that can go into a table**. This ensures the accuracy and reliability of the data in the database.

Constraints could be either on a column level or a table level. The column level constraints are applied only to one column, whereas the table level constraints are applied to the whole table.

Following are some of the most commonly used constraints available in SQL:

* NOT NULL − Ensures that a column cannot have NULL value.
* DEFAULT − Provides a default value for a column when none is specified.
* UNIQUE − Ensures that all values in a column are different.
* PRIMARY KEY − Uniquely identifies each record in a table.
* FOREIGN KEY − Uniquely identifies a record in any of the given table.
* CHECK − Ensures that all the values in a column satisfies certain conditions.
* INDEX − Creates and retrieves data from the database very quickly.

## NOT NULL

By default, a column can hold NULL values. If you do not want a column to have a NULL value, you need to define a NOT NULL constraint on that column.

**Note**: NULL is not the same as no data, rather, it represents unknown data.

**Example**

The following SQL query creates a table called Customers with 5 columns. Three of which, ID, Name and Age, do NOT accept NULL:

SQL> CREATE TABLE Customers(

ID INT NOT NULL,

Name VARCHAR (20) NOT NULL,

Age INT NOT NULL,

Address CHAR (25),

Salary DECIMAL (18, 2),

PRIMARY KEY (ID)

);

## DEFAULT

The DEFAULT constraint **provides a default value** to a column when the INSERT INTO statement does not provide a specific value.

**Example**

The following SQL creates a table called Customers with 5 columns. Here, the Salary column is set to 5000.00 by default, so in case the INSERT INTO statement does not provide a value for this column, then by default this column would be set to 5000.00.

SQL> CREATE TABLE Customers(

ID INT NOT NULL,

Name VARCHAR (20) NOT NULL,

Age INT NOT NULL,

Address CHAR (25),

Salary DECIMAL (18, 2) DEFAULT 5000.00,

PRIMARY KEY (ID)

);

## UNIQUE

The UNIQUE constraint **prevents two records from having identical values in a column**.

**Example**

The following SQL query creates a table called Customers with 5 columns. Here, the Age column is set to UNIQUE, so that it cannot have two records with the same value.

SQL> CREATE TABLE Customers(

ID INT NOT NULL,

Name VARCHAR (20) NOT NULL,

Age INT NOT NULL UNIQUE,

Address CHAR (25),

Salary DECIMAL (18, 2) ,

PRIMARY KEY (ID)

);

## CHECK

The CHECK constraint enables a condition to **check the value being entered into a record**. If the condition evaluates to false, the record violates the constraint and isn't entered the table.

**Example**

The following program creates a table called Customers with 5 columns. Here, we add a CHECK with Age column, so that you cannot have any customer who is below 18 years.

SQL> CREATE TABLE Customers(

ID INT NOT NULL,

Name VARCHAR (20) NOT NULL,

Age INT NOT NULL CHECK (Age >= 18),

Address CHAR (25),

Salary DECIMAL (18, 2) ,

PRIMARY KEY (ID)

);

## INDEX

The INDEX constraint is used to **create and retrieve data from the database very quickly**. An index can be created by using a single or group of columns in a table. When it is created, it is assigned a ROWID for each row before it sorts out the data.

Proper indexes are good for performance in large databases, but you need to be careful while creating them. A selection of fields depends on what you are using in your SQL queries.

Syntax

CREATE INDEX index-name

ON table-name ( column1, column2, columnN ...);

**Example**

The following SQL syntax creates a table called Customers with 5 columns.

SQL> CREATE TABLE Customers(

ID INT NOT NULL,

Name VARCHAR (20) NOT NULL,

Age INT NOT NULL,

Address CHAR (25),

Salary DECIMAL (18, 2) ,

PRIMARY KEY (ID)

);

To create an INDEX on the Age column, to optimize the search on customers for a specific age, run:

SQL> CREATE INDEX idx-age

ON Customers (Age);

## PRIMARY KEY and FOREIGN KEY

### PRIMARY KEY

Primary keys are used to **uniquely identify each record in a table**. That means it must contain UNIQUE values, and cannot contain NULL value. Each table is allowed to have only one PK.

**Example 1 – PRIMARY KEY on CREATE TABLE:**

The following SQL creates a Customers table with a PRIMARY KEY on the ID column:

-- MySQL / SQL Server:

SQL> CREATE TABLE Customers (

ID int NOT NULL,

Name varchar(255) NOT NULL,

Age int,

PRIMARY KEY (ID)

);

-- MySQL / SQL Server / Oracle / MS Access:

SQL> CREATE TABLE Customers (

ID int NOT NULL PRIMARY KEY,

Name varchar(255) NOT NULL,

Age int

);

To give PRIMARY KEY constraint a **name**:

-- MySQL / SQL Server / Oracle / MS Access:

SQL> CREATE TABLE Customers (

ID int NOT NULL,

Name varchar(255) NOT NULL,

Age int,

CONSTRAINT PK\_Customer PRIMARY KEY (ID)

);

To define PRIMARY KEY constraint on **multiple columns**:

-- MySQL / SQL Server / Oracle / MS Access:

SQL> CREATE TABLE Customers (

ID int NOT NULL,

Name varchar(255) NOT NULL,

Age int,

CONSTRAINT PK\_Customer PRIMARY KEY (ID, Name)

);

-- There is only 1 constrain (PK\_Customer). However, its value is made up of 2 columns (ID + Name).

**Example 2 – PRIMARY KEY on ALTER TABLE:**

To create a PRIMARY KEY constraint on the ID column when the table is already created, run:

-- MySQL / SQL Server / Oracle / MS Access:

SQL> ALTER TABLE Customers

ADD PRIMARY KEY (ID);

To give PRIMARY KEY constraint a **name**:

-- MySQL / SQL Server / Oracle / MS Access:

SQL> ALTER TABLE Customer

ADD CONSTRAINT PK\_Customer PRIMARY KEY (ID);

To define PRIMARY KEY constraint on **multiple columns**:

-- MySQL / SQL Server / Oracle / MS Access:

SQL> ALTER TABLE Customer

ADD CONSTRAINT PK\_Customer PRIMARY KEY (ID, Name);

**Example 3 – DROP PRIMARY KEY Constraint:**

To drop a PRIMARY KEY constraint, run:

-- MySQL:

ALTER TABLE Customers

DROP PRIMARY KEY;

-- SQL Server / Oracle / MS Access:

ALTER TABLE Customers

DROP CONSTRAINT PK\_Customer;

**TIPS**:

* Instead of adding value for primary key manually, we can make it automatically with [auto-increment](#_Auto_Increment) feature.

### FOREIGN KEY

Foreign keys are used to **link two tables together**. It is a field (or collection of fields) in one table that refers to the primary key in another table.

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

In the below example 1, the CustomerID column in the Orders table points to the ID column (also primary key) in the Customers table. So it is called a foreign key in the Orders table.

The foreign key constraint is used to prevent actions that would destroy links between tables. It also prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the table it points to. Examples are shown in [this section](#_How_Primary_Key).

**Example 1 – FOREIGN KEY on CREATE TABLE:**

The following SQL creates an Orders table with a FOREIGN KEY on the CustomerID column:

-- MySQL / SQL Server:

SQL> CREATE TABLE Orders (

OrderID int NOT NULL,

ProductName varchar(255) NOT NULL,

CustomerID int,

PRIMARY KEY (OrderID),

FOREIGN KEY (CustomerID) REFERENCES Customers(ID)

);

-- MySQL / SQL Server / Oracle / MS Access:

SQL> CREATE TABLE Orders (

OrderID int NOT NULL PRIMARY KEY,

ProductName varchar(255) NOT NULL,

CustomerID int FOREIGN KEY REFERENCES Customers(ID)

);

To give FOREIGN KEY constraint a **name**:

-- MySQL / SQL Server / Oracle / MS Access:

SQL> CREATE TABLE Orders (

OrderID int NOT NULL,

ProductName varchar(255) NOT NULL,

CustomerID int,

PRIMARY KEY (OrderID),

CONSTRAINT FK\_CustomerOrder FOREIGN KEY (CustomerID)

REFERENCES Customers(ID)

);

**Example 2 – FOREIGN KEY on ALTER TABLE:**

To create a FOREIGN KEY constraint on the CustomerID column when the table is already created, run:

-- MySQL / SQL Server / Oracle / MS Access:

ALTER TABLE Orders

ADD FOREIGN KEY (CustomerID) REFERENCES Customers(ID);

To give FOREIGN KEY constraint a **name**:

-- MySQL / SQL Server / Oracle / MS Access:

ALTER TABLE Orders

ADD CONSTRAINT FK\_CustomerOrder

FOREIGN KEY (CustomerID) REFERENCES Customers(ID);

**Example 3 – DROP FOREIGN KEY Constraint:**

To drop a FOREIGN KEY constraint, run:

-- MySQL:

ALTER TABLE Orders

DROP FOREIGN KEY PK\_CustomerOrder;

-- SQL Server / Oracle / MS Access:

ALTER TABLE Orders

DROP CONSTRAINT PK\_CustomerOrder;

### How Primary Key and Foreign Key Affect Each Other?

**When inserting new records:**

Suppose the Customers table has following records:

+----+----------+-----+

| ID | Name | Age |

+----+----------+-----+

| 1 | Ramesh | 23 |

| 2 | Khilan | 24 |

+----+----------+-----+

If you try to insert a record to the Orders table with CustomerID not available in the ID column of the Customers table, you'll get error "*The INSERT statement conflicted with the FOREIGN KEY constraint "FK\_CustomerOrder". The conflict occurred in database "master", table "dbo.Customers", column 'ID'.*"

INSERT INTO Orders VALUES (1, 'iPhone', 1)

INSERT INTO Orders VALUES (2, 'Samsung', 2)

INSERT INTO Orders VALUES (3, 'iPhone', 3) -- ERROR

**When deleting records:**

You can delete records in the Orders table.

But if you try to delete records (only whose ID available in the Orders table) in the Customers table, you'll get error "*The DELETE statement conflicted with the REFERENCE constraint "FK\_CustomerOrder". The conflict occurred in database "master", table "dbo.Orders", column 'CustomerID'.*".

DELETE FROM Customers WHERE ID = 2 -- ERROR

DELETE FROM Orders WHERE CustomerID = 2 -- OK

To make such a deleting valid, you have to delete records in the Orders table first, then the Customers table:

DELETE FROM Orders WHERE CustomerID = 2 -- OK

DELETE FROM Customers WHERE ID = 2 -- OK

**When dropping tables:**

The below SQL statements run fine:

DROP TABLE Orders -- OK

DROP TABLE Customers -- OK

But if you reverse the table order, you will get error "*Could not drop object 'Customers' because it is referenced by a FOREIGN KEY constraint.*":

DROP TABLE Customers -- ERROR

DROP TABLE Orders -- OK

**When updating tables:**

If you update the Age column of the Customers table, everything runs fine.

But if you try to update the ID column of the Customers table, you will get error "*The UPDATE statement conflicted with the REFERENCE constraint "FK\_CustomerOrder". The conflict occurred in database "master", table "dbo.Orders", column 'CustomerID'.*"

UPDATE Customers SET Age = 25 WHERE ID = 1 -- OK

update Customers SET ID = 3 WHERE Age = 24 -- ERROR

# Auto Increment

## MySQL

The MySQL uses AUTO INCREMENT field to automatically generate a unique number every time a new record is inserted. For example:

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 increases by 1 for each new record.

To let the sequence starts with another value, use:

ALTER TABLE Persons AUTO\_INCREMENT=100;

## MS SQL Server

The MS SQL Server uses the IDENTITY keyword to perform an auto-increment feature. For example:

CREATE TABLE Persons (

PersonId INT IDENTITY(1,1) PRIMARY KEY,

LastName VARCHAR(255) NOT NULL,

FirstName VARCHAR(255),

Age INT

);

By default, the starting value for IDENTITY is 1 and increases by 1 for each new record.

To let the sequence starts with another value, for example start at value 10 and increment by 5, change IDENTITY(1, 1) to IDENTITY(10, 5).

## Notes

* We can only increase number (int, float, etc.), but **not string** (char, varchar, etc.)
* When inserting a new record, we will **NOT have to specify value for the column having auto increment**. For example:

-- Table Persons created above

INSERT INTO Persons (FirstName, LastName)

VALUES ('Lars', 'Monsen');

But this one will cause "unmatched column count" error:

-- Table Persons created above

INSERT INTO Persons ~~(FirstName, LastName)~~ -- ERROR

VALUES ('Lars', 'Monsen');

* Auto incrementing is **NOT be affected when a record is removed**. For example, the table 'A' originally has 3 record with id as 1, 2 and 3 respectively. If we remove the last record (id of 3) then insert a new record to 'A', this new record's id will be assigned to 4, not 3.

# Sorting and Grouping

## ORDER BY

<https://www.tutorialspoint.com/sql/sql-order-by.htm>

<https://www.tutorialspoint.com/sql/sql-sorting-results.htm>

## GROUP BY

<https://www.tutorialspoint.com/sql/sql-group-by.htm>

# Aliases

<https://www.w3schools.com/sql/sql_alias.asp>

# Wildcards

<https://www.w3schools.com/sql/sql_wildcards.asp>

# Functions

## MIN() and MAX()

<https://www.w3schools.com/sql/sql_min_max.asp>

## COUNT(), AVG() and SUM()

<https://www.w3schools.com/sql/sql_count_avg_sum.asp>

# Backup and Restore

## MS SQL Server

### Backup

**Syntax**:

-- Data:

BACKUP DATABASE Database-Name

TO DISK DB-File-Path

[WITH option …]

-- Transaction log:

BACKUP LOG Database-Name

TO DISK Log-File-Path

[WITH option …]

**Common options**:

* DIFFERENTIAL
* FORMAT
* MEDIANAME
* NAME

**Example**:

Assume you want to backup your database named TestDB to a new file named TestDB.bak, run:

BACKUP DATABASE TestDB

TO DISK = 'D:\SQL\TestDB.bak'

### Restore

**Syntax**:

-- Data:

RESTORE DATABASE Database-Name

FROM DISK DB-File-Path

[WITH option …]

-- Transaction log:

RESTORE LOG Database-Name

FROM DISK DB-File-Path

[WITH option …]

**Common options**:

* WITH RECOVERY
* WITH NORECOVERY

**Example**:

Assume you want to restore your database named TestDB from the backup file named TestDB.bak, run:

RESTORE DATABASE TestDB

FROM DISK = 'D:\SQL\TestDB.bak'

More details:

<https://sqlbackupandftp.com/blog/how-to-automate-sql-server-database-backups>

<https://docs.microsoft.com/en-us/sql/t-sql/statements/backup-transact-sql?view=sql-server-ver15>

<https://sqlbackupandftp.com/blog/restore-database-backup>

## MySQL

<https://www.hivelocity.net/kb/create-mysql-backup-from-command-line-mysql-import-sql-file/>

# SQL Client Applications and Extensions

Quick comparison: **HeidiSQL** vs **phpMyAdmin** vs **TablePlus**

<https://tableplus.com/blog/2018/10/heidisql-vs-phpmyadmin-vs-tableplus.html>

## HeidiSQL

### MS SQL Server

How to connect:

<https://medium.com/@bchewy/connecting-to-an-mssql-server-through-heidisql-e46e57ef3f74>

<https://stackoverflow.com/a/33748584>

Attach existing database (.mdf and .ldf files) to HeidiSQL: <https://serverfault.com/a/579845>

## VS Code Extension

Check Personal\Tutorials\Visual Studio\VS Code Tutorial.docx

# Anti-Hacking

## SQL Injection

<https://www.tutorialspoint.com/mysql/mysql-sql-injection.htm>

# Procedural SQL

Procedural languages are designed to extend SQL's abilities while being able to integrate with SQL seamlessly. Many features (such as stored procedures, variables, conditions, loops, etc. ) are added, making the language Turing-complete.

## Common Extensions

Both PL/SQL and T-SQL are popular procedural extensions of SQL.

**Differences between SQL and PL/SQL and T-SQL:**

|  |  |
| --- | --- |
| **SQL** | **PL/SQL and T-SQL** |
| Cannot contain procedural code in it. | Be a non-standard **extension** of SQL. Containing almost all of the features of standard SQL |
| Allows **single query** that is used to perform DML and DDL operations. | Allows **blocks of codes** that used to write the entire program blocks / procedure / function, etc. |
| Be declarative (define **what** need to be done, rather than how things need to be done) | Be procedural (defines **how** the things needs to be done) |
| Execute as a single statement. | Execute as a whole block. |
| Mainly used to manipulate data. | Mainly used to create an application. |

**Difference between T-SQL and PL-SQL:**

|  |  |
| --- | --- |
| **T-SQL** | **PL-SQL** |
| Stands for "Transact-SQL" | Stands for "Procedural Language extensions to SQL" |
| Developed by **Sybase** and **Microsoft**. | Developed by **Oracle**. |
| Used in **Sybase server** and **MS** **SQL server**. | Used in Oracle Server. |
| Focuses on a high degree of control to programmers. | A natural **programming language** that blends easily with the SQL |
| Main features: variables, functions, conditional statements, loops, error and exception handling, string operation, mathematical operations, date and time processing, etc. | Main features: variables, functions, conditional statements, loops, error and exception handling, OOP concepts (data encapsulation, function overloading, and information hiding). |
|  |  |
| All commands are submitted to the server in a single go | Executes the block of code at a time. |

**Notes**:

* Different databases use different languages for stored procedures. So, you CANNOT use T-SQL for Oracle server nor PL-SQL for MS SQL server.

### T-SQL

<https://www.tutorialspoint.com/t_sql/index.htm>

### PL/SQL

<https://www.tutorialspoint.com/plsql/index.htm>

### MySQL Extension

## Decision Making

### MySQL

#### IF EXISTS and IF NOT EXISTS

Add these clauses after CREATE or DROPT statements to check the existence of a database/table before create/delete it. This is also a simple way to prevent "already exists" or "not exists" SQL error while manipulating DB.

For examples:

CREATE DATABASE IF NOT EXISTS;

DROPT DATABASE IF EXISTS;

CREATE TABLE IF NOT EXISTS (…);

DROPT TABLE IF EXIST;

...

## Stored Procedures

**SP vs Functions:**

|  |  |
| --- | --- |
| **Stored Procedure (SP)** | **Function (UDF - User Defined)** |
| Can return zero, single or multiple values. | Must return a single value (which may be a scalar or a table). |
| Can use transaction in SP. | Can't use transaction in UDF. |
| Allows input/output parameter. | Allows only input parameter. |
| Can call function from SP. | Can't call SP from function. |
| Can't be used in SELECT/WHERE/HAVING statement. | Can be used in SELECT/WHERE/HAVING statement. |
| Allow exception handling using Try-Catch block. | Don't allow exception handling. |

### MySQL

<https://www.mssqltips.com/sqlservertip/1495/getting-started-with-stored-procedures-in-sql-server/>

<https://www.w3resource.com/mysql/mysql-procedure.php#:~:text=A%20procedure%20(often%20called%20a,MySQL%205%20introduce%20stored%20procedure>.

<https://www.w3resource.com/mysql/mysql-procedure.php#:~:text=A%20procedure%20(often%20called%20a,MySQL%205%20introduce%20stored%20procedure>.

### MS SQL Server

## Trigger

## Built-In Functions

### TOP

It indicates how many rows should be returned by a query in a result set. It's put after SELECT along with the number of rows to return.

For example, the following query return only the first 12 results:

SELECT TOP 12 Id, Name, Description

FROM Products ORDER BY Name;

You don't find this clause in standard SQL.

# MySQL

## Installation

**On Linux**: [How To Install MySQL on Ubuntu 18.04 | DigitalOcean](https://www.digitalocean.com/community/tutorials/how-to-install-mysql-on-ubuntu-18-04)

**On Windows**:

* XAMPP

## Usage

### Start / Stop

**On Linux**:

* Start: sudo systemctl start mysql
* Stop: sudo systemctl stop mysql
* Check status: systemctl status mysql.service

**On Windows**:

### Import CSV file into MySQL

<https://phoenixnap.com/kb/import-csv-file-into-mysql>

# SQL Embedded Database

Embedded database systems are database management systems (DBMSs) which are tightly **integrated with an application** (rather than shared among multiple applications) and completely controlled by the application. They **DON'T need a server**.

Structurally, embedded databases may be relational, or non-relational/NoSQL. But in this document, we only talk about relational SQL databases.

For example, SQLite is a SQL embedded database because it's **server-less** (it's self-contained in local machine). On the other hand, other popular DBMS (such as MySQL, MS SQL Server, etc.) requires a client and server architecture to interact over a network.

**Note:**

There are 2 definitions of embedded databases:

1. The above one

2. Database systems designed for the "embedded" space (mobile devices, etc.). They perform reasonably in tight environments (memory/CPU wise)

So, don't be confused!

## SQLite

### SQLite vs MySQL

|  |  |  |
| --- | --- | --- |
|  | **SQLite** | **MySQL** |
| **Architecture** | Runs on **local** machine  + Directly stores data in a single file  + No configurations are required  + Easy to set up | Runs on **server** |
| **Data type support** | **Limited** support  (Blob, Integer, Null, Text, Real) | **Wide-range** support  (Tinyint, Smallint, Mediumint, Int, Bigint, Double, Float, Real, Decimal, Double precision, Numeric, Timestamp, Date, Datetime, Char, Varchar, Year, Tinytext, Tinyblob, Blob, Text, MediumBlob, MediumText, Enum, Set, Longblob, Longtext) |
| **Storage** | SQLite library is about 250 KB in size | MySQL server is about 600 MB |
| **Multiple access** | Does **not have any specific user management** functionality  + Not suitable for multiple user access | Has a **well-constructed user management** system  + Can handle multiple users and grant various levels of permission |
| **Scalability** | Suitable for **smaller databases**  + As the database grows, the memory requirement gets larger.  + Performance optimization is harder. | Easily scalable and can handle a **bigger database** with less effort. |
| **Security** | Does **not have an inbuilt authentication mechanism**  + Database files can be accessed by anyone | **Has many inbuilt security features**. This includes authentication with a username, password, and SSH. |
| **Syntax** | Uses **standard SQL** syntax with minor alterations | Slightly different syntax as compared to conventional SQL |
| **When to Use** | - Developing small standalone apps  - Read and write directly from the disk  - Basic development and testing | - Web-based applications  - Large database and more scalability  - Multiple user access  - Require strong security and authentication features |

### Migration from MySQL to SQLite

<https://stackoverflow.com/a/9933603>

## SQL Server Compact