# SQL

Presented By
Md. Jamal Uddin
Assistant Professor
Dept. of CSE, BSMRSTU

**What is SQL?**
SQL is Structured Query Language, which is a computer language for storing, manipulating and retrieving data stored in a relational database.
SQL is the standard language for Relational Database System. All the Relational Database Management Systems (RDMS) like MySQL, MS Access, Oracle, Sybase, Informix, Postgres and SQL Server use SQL as their standard database language.

Also, they are using different dialects, such as:
 MS SQL Server using T-SQL,
 Oracle using PL/SQL,
 MS Access version of SQL is called JET SQL (native format) etc.

**What Can SQL do?**

- SQL can execute queries against a database
- SQL can retrieve data from a database
- SQL can insert records in a database
- SQL can update records in a database
- SQL can delete records from a database
- SQL can create new databases
- SQL can create new tables in a database
- SQL can create stored procedures in a database
- SQL can create views in a database
- SQL can set permissions on tables, procedures, and views
-

**Using SQL in Your Web Site**

To build a web site that shows data from a database, you will need:
- An RDBMS database program (i.e. MS Access, SQL Server, MySQL)
- To use a server-side scripting language, like PHP or ASP
- To use SQL to get the data you want
- To use HTML / CSS

**RDBMS**

RDBMS stands for Relational Database Management System.

RDBMS is the basis for SQL, and for all modern database systems such as MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.

The data in RDBMS is stored in database objects called tables.

**What is a table?**

The data in an RDBMS is stored in database objects which are called as tables. This table is basically a collection of related data entries and it consists of numerous columns and rows.

Remember, a table is the most common and simplest form of data storage in a relational database. The following program is an example of a CUSTOMERS table:

```
+----+----------+-----+-----------+----------+
| 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 |
+----+----------+-----+-----------+----------+
```

**What is a field?**

Every table is broken up into smaller entities called fields. The fields in the CUSTOMERS table consist of ID, NAME, AGE, ADDRESS and SALARY.

A field is a column in a table that is designed to maintain specific information about every record in the table.

**What is a Record or a Row?**

A record is also called as a row of data is each individual entry that exists in a table. For example, there are 7 records in the above CUSTOMERS table. Following is a single row of data or record in the CUSTOMERS table:

```
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
+----+----------+-----+-----------+----------+
```

A record is a horizontal entity in a table.

**What is a column?**

A column is a vertical entity in a table that contains all information associated with a specific field in a table.

For example, a column in the CUSTOMERS table is ADDRESS, which represents location description and would be as shown below:

```
+-----------+
| ADDRESS   |
+-----------+
| Ahmedabad |
| Delhi     |
| Kota      |
| Mumbai    |
| Bhopal    |
| MP        |
| Indore    |
+----+------+
```

**What is a NULL value?**

A NULL value in a table is a value in a field that appears to be blank, which means a field with a NULL value is a field with no value.

It is very important to understand that a NULL value is different than a zero value or a field that contains spaces. A field with a NULL value is the one that has been left blank during a record creation.

**SQL Constraints**

Constraints are the rules enforced on data columns on 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 can either be column level or table level. Column level constraints are applied only to one column whereas, table level constraints are applied to the entire table.

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

- NOT NULL Constraint: Ensures that a column cannot have a NULL value.
- DEFAULT Constraint: Provides a default value for a column when none is specified.
- UNIQUE Constraint: Ensures that all the values in a column are different.
- PRIMARY Key: Uniquely identifies each row/record in a database table.

 FOREIGN Key: Uniquely identifies a row/record in any another database table.
 CHECK Constraint: The CHECK constraint ensures that all values in a column satisfy certain conditions.
 INDEX: Used to create and retrieve data from the database very quickly.

**Data Integrity**

The following categories of data integrity exist with each RDBMS:
 Entity Integrity: There are no duplicate rows in a table.
 Domain Integrity: Enforces valid entries for a given column by restricting the type, the format, or the range of values.
 Referential integrity: Rows cannot be deleted, which are used by other records.
 User-Defined Integrity: Enforces some specific business rules that do not fall into entity, domain or referential integrity.

**Database Normalization**

Database normalization is the process of efficiently organizing data in a database. There are two reasons of this normalization process:
 Eliminating redundant data. For example, storing the same data in more than one table.
 Ensuring data dependencies make sense.
Both these reasons are worthy goals as they reduce the amount of space a database consumes and ensures that data is logically stored. Normalization consists of a series of guidelines that help guide you in creating a good database structure. Normalization guidelines are divided into normal forms; think of a form as the format or the way a database structure is laid out. The aim of normal forms is to organize the database structure, so that it complies with the rules of first normal form, then second normal form and finally the third normal form.
It is your choice to take it further and go to the fourth normal form, fifth normal form and so on, but in general, the third normal form is more than enough.
 First Normal Form (1NF)
 Second Normal Form (2NF)
 Third Normal Form (3NF)

**Database – First Normal Form (1NF)**

The First normal form (1NF) sets basic rules for an organized database:

-  Define the data items required, because they become the columns in a table.
-  Place the related data items in a table.
-  Ensure that there are no repeating groups of data.
-  Ensure that there is a primary key.

**First Rule of 1NF**

You must define the data items. This means looking at the data to be stored, organizing the data into columns, defining what type of data each column contains and then finally putting the related columns into their own table.

For example, you put all the columns relating to locations of meetings in the Location table, those relating to members in the MemberDetails table and so on.

**Second Rule of 1NF**

The next step is ensuring that there are no repeating groups of data. Consider we have the following table:

```
CREATE TABLE CUSTOMERS(
      ID   INT              NOT NULL,
      NAME VARCHAR (20)     NOT NULL,
      AGE  INT              NOT NULL,
      ADDRESS  CHAR (25),
      ORDERS   VARCHAR(155)
);
```

So, if we populate this table for a single customer having multiple orders, then it would be something as shown below:

```
ID NAME AGE ADDRESS ORDERS
100 Sachin 36 Lower West Side Cannon XL-200
100 Sachin 36 Lower West Side Battery XL-200
100 Sachin 36 Lower West Side Tripod Large
```

But as per the 1NF, we need to ensure that there are no repeating groups of data. So, let us break the above table into two parts and then join them using a key as shown in the following program:

CUSTOMERS Table

```
CREATE TABLE CUSTOMERS(
    ID   INT              NOT NULL,
    NAME VARCHAR (20)     NOT NULL,
    AGE  INT              NOT NULL,
    ADDRESS  CHAR (25),
    PRIMARY KEY (ID)
);
```

This table would have the following record:

| ID | NAME | AGE | ADDRESS |
|---|---|---|---|
| 100 | Sachin | 36 | Lower West Side |

ORDERS Table

```
CREATE TABLE ORDERS(
    ID   INT              NOT NULL,
    CUSTOMER_ID INT       NOT NULL,
    ORDERS   VARCHAR(155),
    PRIMARY KEY (ID)
);
```

This table would have the following records:

| ID | CUSTOMER_ID | ORDERS |
|---|---|---|
| 10 | 100 | Cannon XL-200 |
| 11 | 100 | Battery XL-200 |
| 12 | 100 | Tripod Large |

**Third Rule of 1NF**
The final rule of the first normal form, create a primary key for each table which we have already created.

**Database – Second Normal Form (2NF)**
The Second Normal Form states that it should meet all the rules for 1NF and there must be no partial dependences of any of the columns on the primary key:

Consider a customer-order relation and you want to store customer ID, customer name, order ID and order detail and the date of purchase:

```
CREATE TABLE CUSTOMERS(
      CUST_ID    INT             NOT NULL,
      CUST_NAME VARCHAR (20)      NOT NULL,
      ORDER_ID   INT             NOT NULL,
      ORDER_DETAIL VARCHAR (20)  NOT NULL,
      SALE_DATE  DATETIME,
      PRIMARY KEY (CUST_ID, ORDER_ID)
);
```

This table is in the first normal form; in that it obeys all the rules of the first normal form. In this table, the primary key consists of the CUST_ID and the ORDER_ID. Combined, they are unique assuming the same customer would hardly order the same thing.

However, the table is not in the second normal form because there are partial dependencies of primary keys and columns. CUST_NAME is dependent on CUST_ID and there's no real link between a customer's name and what he purchased. The order detail and purchase date are also dependent on the ORDER_ID, but they are not dependent on the CUST_ID, because there is no link between a CUST_ID and an ORDER_DETAIL or their SALE_DATE.

To make this table comply with the second normal form, you need to separate the columns into three tables.

First, create a table to store the customer details as shown in the code block below:

```
CREATE TABLE CUSTOMERS(
      CUST_ID    INT             NOT NULL,
      CUST_NAME VARCHAR (20)      NOT NULL,
      PRIMARY KEY (CUST_ID)
);
```

The next step is to create a table to store the details of each order:

```
CREATE TABLE ORDERS(
      ORDER_ID   INT             NOT NULL,
      ORDER_DETAIL VARCHAR (20)  NOT NULL,
    PRIMARY KEY (ORDER_ID)
);
```

Finally, create a third table storing just the CUST_ID and the ORDER_ID to keep a track of all the orders for a customer:

```
CREATE TABLE CUSTMERORDERS(
    CUST_ID    INT            NOT NULL,
    ORDER_ID   INT            NOT NULL,
    SALE_DATE  DATETIME,
    PRIMARY KEY (CUST_ID, ORDER_ID)
);
```

**Database – Third Normal Form (3NF)**

A table is in a third normal form when the following conditions are met:

-  It is in the second normal form.
-  All non-primary fields are dependent on the primary key.

The dependency of these non-primary fields is between the data. For example, in the following table – the street name, city and the state are unbreakably bound to their zip code.

```
CREATE TABLE CUSTOMERS(
    CUST_ID      INT            NOT NULL,
    CUST_NAME    VARCHAR (20)   NOT NULL,
    DOB          DATE,
    STREET       VARCHAR(200),
    CITY         VARCHAR(100),
    STATE        VARCHAR(100),
    ZIP          VARCHAR(12),
    EMAIL_ID     VARCHAR(256),
    PRIMARY KEY (CUST_ID)
);
```

The dependency between the zip code and the address is called as a transitive dependency. To comply with the third normal form, all you need to do is to move the Street, City and the State fields into their own table, which you can call as the Zip Code table.

```
CREATE TABLE ADDRESS(
    ZIP          VARCHAR(12),
    STREET      VARCHAR(200),
    CITY        VARCHAR(100),
    STATE       VARCHAR(100),
    PRIMARY KEY (ZIP)
);
```

The next step is to alter the CUSTOMERS table as shown below.

```
CREATE TABLE CUSTOMERS(
    CUST_ID     INT            NOT NULL,
    CUST_NAME   VARCHAR (20)   NOT NULL,
    DOB         DATE,
    ZIP         VARCHAR(12),
    EMAIL_ID    VARCHAR(256),
    PRIMARY KEY (CUST_ID)
);
```

The advantages of removing transitive dependencies are mainly two-fold. First, the amount of data duplication is reduced and therefore your database becomes smaller.

The second advantage is data integrity. When duplicated data changes, there is a big risk of updating only some of the data, especially if it is spread out in many different places in the database.

For example, if the address and the zip code data were stored in three or four different tables, then any changes in the zip codes would need to ripple out to every record in those three or four tables.

**What is MySQL?**

- MySQL is a database system used on the web
- MySQL is a database system that runs on a server
- MySQL is ideal for both small and large applications
- MySQL is very fast, reliable, and easy to use
- MySQL supports standard SQL
- MySQL compiles on a number of platforms
- MySQL is free to download and use

The **SQL CREATE DATABASE** statement is used to create a new SQL database.
Syntax The basic syntax of this CREATE DATABASE statement is as follows:

```
CREATE DATABASE DatabaseName;
```

Always the database name should be unique within the RDBMS.
Example If you want to create a new database <testDB>, then the CREATE DATABASE statement would be as shown below:

```
SQL> CREATE DATABASE testDB;
```

Make sure you have the admin privilege before creating any database. Once a database is created, you can check it in the list of databases as follows:

```
SQL> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| AMROOD             |
| TUTORIALSPOINT     |
| mysql              |
| orig               |
| test               |
| testDB             |
+--------------------+
7 rows in set (0.00 sec)
```

The **SQL DROP DATABASE** statement is used to drop an existing database in SQL schema.
Syntax The basic syntax of DROP DATABASE statement is as follows:

```
DROP DATABASE DatabaseName;
```

Always the database name should be unique within the RDBMS.
Example If you want to delete an existing database <testDB>, then the DROP DATABASE statement would be as shown below:

```
SQL> DROP DATABASE testDB;
```

NOTE: Be careful before using this operation because by deleting an existing database would result in loss of complete information stored in the database. Make sure you have the admin privilege before dropping any database. Once a database is dropped, you can check it in the list of the databases as shown below:

```
SQL> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| AMROOD             |
| TUTORIALSPOINT     |
| mysql              |
| orig               |
| test               |
+--------------------+
6 rows in set (0.00 sec)
```

**SQL ─ SELECT Database, USE Statement**

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

Syntax The basic syntax of the USE statement is as shown below:

```
USE DatabaseName;
```

Always the database name should be unique within the RDBMS.

Example You can check the available databases as shown below:

```
SQL> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| AMROOD             |

| TUTORIALSPOINT     |
| mysql              |
| orig               |
| test               |
+--------------------+
6 rows in set (0.00 sec)
```

Now, if you want to work with the AMROOD database, then you can execute the following SQL command and start working with the AMROOD database.

```
SQL> USE AMROOD;
```

## SQL ─ CREATE Table

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

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

Syntax The basic syntax of the CREATE TABLE statement is as follows:

```
CREATE TABLE table_name(
   column1 datatype,
   column2 datatype,
   column3 datatype,
   .....
   columnN datatype,
   PRIMARY KEY( one or more columns )
);
```

CREATE TABLE is the keyword telling the database system what you want to do. In this case, you want to create a new table. The unique name or identifier for the table follows the CREATE TABLE statement.

Then in brackets comes the list defining each column in the table and what sort of data type it is. The syntax becomes clearer with the following example.

A copy of an existing table can be created using a combination of the CREATE TABLE statement and the SELECT statement. You can check the complete details at Create Table Using another Table.

Example The following code block is an example, which creates a CUSTOMERS table with an ID as a primary key and NOT NULL are the constraints showing that these fields cannot be NULL while creating records in this table:

```
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)
);
```

Now, you have CUSTOMERS table available in your database which you can use to store the required information related to customers.

**SQL - Creating a Table from an Existing Table**

A copy of an existing table can be created using a combination of the CREATE TABLE statement and the SELECT statement. The new table has the same column definitions. All columns or specific columns can be selected. When you will create a new table using the existing table, the new table would be populated using the existing values in the old table.

Syntax The basic syntax for creating a table from another table is as follows:

```
CREATE TABLE NEW_TABLE_NAME AS
   SELECT [ column1, column2...columnN ]
   FROM EXISTING_TABLE_NAME
   [ WHERE ]
```

Here, column1, column2... are the fields of the existing table and the same would be used to create fields of the new table.

Example Following is an example which would create a table SALARY using the CUSTOMERS table and having the fields – customer ID and customer SALARY:

```
SQL> CREATE TABLE SALARY AS
   SELECT ID, SALARY
   FROM CUSTOMERS;
```

This would create a new table SALARY which will have the following records.

```
+----+----------+
| ID | SALARY   |
+----+----------+
|  1 |  2000.00 |
|  2 |  1500.00 |
|  3 |  2000.00 |
|  4 |  6500.00 |
|  5 |  8500.00 |
|  6 |  4500.00 |
|  7 | 10000.00 |
+----+----------+
```

**SQL ─ DROP or DELETE Table**

The SQL DROP TABLE statement is used to remove a table definition and all the data, indexes, triggers, constraints and permission specifications for that table. NOTE: You should be very careful while using this command because once a table is deleted then all the information available in that table will also be lost forever.

Syntax The basic syntax of this DROP TABLE statement is as follows:

```
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    |       |
+---------+---------------+------+-----+---------+-------+
5 rows in set (0.00 sec)
```

This means that the CUSTOMERS table is available in the database, so let us now drop it as shown below.

```
SQL> DROP TABLE CUSTOMERS;
Query OK, 0 rows affected (0.01 sec)
```

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.

**SQL ─ INSERT Query**

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

Syntax There are two basic syntaxes of the INSERT INTO statement which are shown below.

```
INSERT INTO TABLE_NAME (column1, column2, column3,...columnN)]
VALUES (value1, value2, value3,...valueN);
```

Here, column1, column2, column3,...columnN are the names of the columns in the table into which you want to insert the data.

You may not need to specify the column(s) name in the SQL query if you are adding values for all the columns of the table. But make sure the order of the values is in the same order as the columns in the table.

The SQL INSERT INTO syntax will be as follows:

```
INSERT INTO TABLE_NAME VALUES (value1,value2,value3,...valueN);
```

Example The following statements would create six records in the CUSTOMERS table.

```
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 );
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (3, 'kaushik', 23, 'Kota', 2000.00 );
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (4, 'Chaitali', 25, 'Mumbai', 6500.00 );
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (5, 'Hardik', 27, 'Bhopal', 8500.00 );
```

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (6, 'Komal', 22, 'MP', 4500.00 );
```

You can create a record in the CUSTOMERS table by using the second syntax as shown below.

```
INSERT INTO CUSTOMERS
VALUES (7, 'Muffy', 24, 'Indore', 10000.00 );
```

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 |
|  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 using another table**

You can populate the data into a table through the select statement 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];
```

## SQL ─ WHERE Clause

The SQL WHERE clause is used to specify a condition while fetching the data from a single table or by joining with multiple tables. If the given condition is satisfied, then only it returns a specific value from the table. You should use the WHERE clause to filter the records and fetching only the necessary records.

The WHERE clause is not only used in the SELECT statement, but it is also used in the UPDATE, DELETE statement, etc., which we would examine in the subsequent chapters.

Syntax The basic syntax of the SELECT statement with the WHERE clause is as shown below.

```
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 |
+----+----------+-----+-----------+----------+
```

The following code is an example which would fetch the ID, Name and Salary fields from the CUSTOMERS table, where the salary is greater than 2000:

```
SQL> SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE SALARY > 2000;
```

This would produce the following result:

```
+----+----------+----------+
| ID | NAME     | SALARY   |
+----+----------+----------+
|  4 | Chaitali |  6500.00 |
|  5 | Hardik   |  8500.00 |
|  6 | Komal    |  4500.00 |
|  7 | Muffy    | 10000.00 |
+----+----------+----------+
```

The following query is an example, which would fetch the ID, Name and Salary fields from the CUSTOMERS table for a customer with the name Hardik.
Here, it is important to note that all the strings should be given inside single quotes (''). Whereas, numeric values should be given without any quote as in the above example.

```
SQL> SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE NAME = 'Hardik';
```

This would produce the following result:

```
+----+----------+----------+
| ID | NAME     | SALARY   |
+----+----------+----------+
|  5 | Hardik   |  8500.00 |
+----+----------+----------+
```

## SQL ─ AND & OR Conjunctive Operators

The SQL AND & OR operators are used to combine multiple conditions to narrow data in an SQL statement. These two operators are called as the conjunctive operators.
These operators provide a means to make multiple comparisons with different operators in the same SQL statement.
The AND Operator The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause.

Syntax The basic syntax of the AND operator with a WHERE clause is as follows:

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition1] AND [condition2]...AND [conditionN];
```

You can combine N number of conditions using the AND operator. For an action to be taken by the SQL statement, whether it be a transaction or a query, all conditions separated by the AND must be TRUE.

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 |
+----+----------+-----+-----------+----------+
```

Following is an example, which would fetch the ID, Name and Salary fields from the CUSTOMERS table, where the salary is greater than 2000 and the age is less than 25 years.

```
SQL> SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE SALARY > 2000 AND age < 25;
```

This would produce the following result:

```
+----+-------+----------+
| ID | NAME  | SALARY   |
+----+-------+----------+
|  6 | Komal |  4500.00 |
|  7 | Muffy | 10000.00 |
+----+-------+----------+
```

The OR Operator The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.

Syntax The basic syntax of the OR operator with a WHERE clause is as follows:

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition1] OR [condition2]...OR [conditionN]
```

You can combine N number of conditions using the OR operator. For an action to be taken by the SQL statement, whether it be a transaction or query, the only any ONE of the conditions separated by the OR must be TRUE.

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 block has a query, which would 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.

```
SQL> SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE SALARY > 2000 OR age < 25;
```

This would produce the following result:

```
+----+----------+----------+
| ID | NAME     | SALARY   |
+----+----------+----------+
|  3 | kaushik  |  2000.00 |
|  4 | Chaitali |  6500.00 |
|  5 | Hardik   |  8500.00 |
|  6 | Komal    |  4500.00 |
|  7 | Muffy    | 10000.00 |
+----+----------+----------+
```

**SQL ─ UPDATE Query**

The SQL UPDATE Query is used to modify the existing records in a table. You can use the WHERE clause with the UPDATE query to update the selected rows, otherwise all the rows would be affected.

Syntax The basic syntax of the UPDATE query with a WHERE clause is as follows:

```
UPDATE table_name
```

```
SET column1 = value1, column2 = value2...., columnN = valueN
WHERE [condition];
```

You can combine N number of conditions using the AND or the OR operators.
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 as the UPDATE query would be enough as shown in the following code block.

```
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 |
+----+----------+-----+---------+---------+
```

**SQL ─ DELETE Query**

The SQL DELETE Query is used to delete the existing records from a table.
You can use the WHERE clause with a DELETE query to delete the selected rows, otherwise all the records would be deleted.
Syntax The basic syntax of the DELETE query with the WHERE clause is as follows:

```
DELETE FROM table_name
WHERE [condition];
```

You can combine N number of conditions using AND or OR operators.
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 and **the** DELETE query would be as follows:

```
SQL> DELETE FROM CUSTOMERS;
```
Now, the CUSTOMERS table would not have any record.

**SQL ─ LIKE Clause**

The SQL LIKE clause is used to compare a value to similar values using wildcard operators. There are two wildcards used in conjunction with the LIKE operator.

-  The percent sign (%)
-  The underscore (_)

The percent sign represents zero, one or multiple characters. The underscore represents a single number or character. These symbols can be used in combinations.

Syntax The basic syntax of % and _ is as follows:

```
SELECT FROM table_name
WHERE column LIKE 'XXXX%'
or
SELECT FROM table_name
WHERE column LIKE '%XXXX%'
```

or

```
SELECT FROM table_name
WHERE column LIKE 'XXXX_'
```

or

```
SELECT FROM table_name
WHERE column LIKE '_XXXX'
```

or

```
SELECT FROM table_name
WHERE column LIKE '_XXXX_'
```

You can combine N number of conditions using AND or OR operators. Here, XXXX could be any numeric or string value.

Example The following table has a few examples showing the WHERE part having different LIKE clause with '%' and '_' operators:

| Statement | Description |
|---|---|
| WHERE SALARY LIKE '200%' | Finds any values that start with 200. |
| WHERE SALARY LIKE '%200%' | Finds any values that have 200 in any position. |
| WHERE SALARY LIKE '_00%' | Finds any values that have 00 in the second and third positions. |
| WHERE SALARY LIKE '2_%_%' | Finds any values that start with 2 and are at least 3 characters in length. |
| WHERE SALARY LIKE '%2' | Finds any values that end with 2. |
| WHERE SALARY LIKE '_2%3' | Finds any values that have a 2 in the second position and end with a 3. |
| WHERE SALARY LIKE '2___3' | Finds any values in a five-digit number that start with 2 and end with 3. |

Let us take a real example, consider the CUSTOMERS table having the records 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 |
+----+----------+-----+-----------+----------+
```

Following is an example, which would display all the records from the CUSTOMERS table, where the SALARY starts with 200.

```
SQL> SELECT * FROM CUSTOMERS
WHERE SALARY LIKE '200%';
```

This would produce the following result:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
+----+----------+-----+-----------+----------+
```

**SQL ─ TOP, LIMIT or ROWNUM Clause**

The SQL TOP clause is used to fetch a TOP N number or X percent records from a table.

Note: All the databases do not support the TOP clause. For example, MySQL supports the LIMIT clause to fetch a limited number of records, while Oracle uses the ROWNUM command to fetch a limited number of records.

Syntax The basic syntax of the TOP clause with a SELECT statement would be as follows.

```
SELECT TOP number|percent column_name(s)
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 |
+----+----------+-----+-----------+----------+
```

The following query is an example on the SQL server, which would fetch the top 3 records from the CUSTOMERS table.

```
SQL> SELECT TOP 3 * FROM CUSTOMERS;
```

This would produce the following result:

```
+----+---------+-----+-----------+---------+
| ID | NAME    | AGE | ADDRESS   | SALARY  |
+----+---------+-----+-----------+---------+
|  1 | Ramesh  |  32 | Ahmedabad | 2000.00 |
|  2 | Khilan  |  25 | Delhi     | 1500.00 |
|  3 | kaushik |  23 | Kota      | 2000.00 |
+----+---------+-----+-----------+---------+
```

If you are using MySQL server, then here is an equivalent example:

```
SQL> SELECT * FROM CUSTOMERS
LIMIT 3;
```

This would produce the following result:

```
+----+---------+-----+-----------+---------+
| ID | NAME    | AGE | ADDRESS   | SALARY  |
+----+---------+-----+-----------+---------+
|  1 | Ramesh  |  32 | Ahmedabad | 2000.00 |

|  2 | Khilan  |  25 | Delhi     | 1500.00 |

|  3 | kaushik |  23 | Kota      | 2000.00 |

+----+---------+-----+-----------+---------+
```

If you are using an Oracle server, then the following code block has an equivalent example.

```
SQL> SELECT * FROM CUSTOMERS
WHERE ROWNUM <= 3;
```

This would produce the following result:

```
+----+---------+-----+-----------+---------+
| ID | NAME    | AGE | ADDRESS   | SALARY  |
+----+---------+-----+-----------+---------+

|  1 | Ramesh  |  32 | Ahmedabad | 2000.00 |
|  2 | Khilan  |  25 | Delhi     | 1500.00 |
|  3 | kaushik |  23 | Kota      | 2000.00 |
+----+---------+-----+-----------+---------+
```

**SQL ─ ORDER BY Clause**

The SQL ORDER BY clause is used to sort the data in ascending or descending order, based on one or more columns. Some databases sort the query results in an ascending order by default.

Syntax The basic syntax of the ORDER BY clause is as follows:

```
SELECT column-list
FROM table_name
[WHERE condition]
[ORDER BY column1, column2, .. columnN] [ASC | DESC];
```

You can use more than one column in the ORDER BY clause. Make sure whatever column you are using to sort that column should be in the column-list.

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 block has an example, which would sort the result in an ascending order by the NAME and the SALARY.

```
SQL> SELECT * FROM CUSTOMERS
    ORDER BY NAME, SALARY;
```

This would produce the following result:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
+----+----------+-----+-----------+----------+
```

The following code block has an example, which would sort the result in the descending order by NAME.

```
SQL> SELECT * FROM CUSTOMERS
   ORDER BY NAME DESC;
```

This would produce the following result:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
+----+----------+-----+-----------+----------+
```

## SQL ─ Group By

The SQL GROUP BY clause is used in collaboration with the SELECT statement to arrange identical data into groups. This GROUP BY clause follows the WHERE clause in a SELECT statement and precedes the ORDER BY clause.
Syntax The basic syntax of a GROUP BY clause is shown in the following code block. The GROUP BY clause must follow the conditions in the WHERE clause and must precede the ORDER BY clause if one is used.

```
SELECT column1, column2
FROM table_name
WHERE [ conditions ]
GROUP BY column1, column2
ORDER BY column1, column2
```

Example Consider the CUSTOMERS table is 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 know the total amount of the salary on each customer, then the GROUP BY query would be as follows.

```
SQL> SELECT NAME, SUM(SALARY) FROM CUSTOMERS
    GROUP BY NAME;
```

This would produce the following result:

```
+----------+-------------+
| NAME     | SUM(SALARY) |
+----------+-------------+
| Chaitali |     6500.00 |
| Hardik   |     8500.00 |
| kaushik  |     2000.00 |
| Khilan   |     1500.00 |
| Komal    |     4500.00 |
| Muffy    |    10000.00 |
| Ramesh   |     2000.00 |
+----------+-------------+
```

Now, let us look at a table where the CUSTOMERS table has the following records with duplicate names:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Ramesh   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | kaushik  |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```
Now again, if you want to know the total amount of salary on each customer, then the GROUP BY query would be as follows:

```
SQL> SELECT NAME, SUM(SALARY) FROM CUSTOMERS
    GROUP BY NAME;
```

This would produce the following result:

```
+---------+-------------+
| NAME    | SUM(SALARY) |
+---------+-------------+
| Hardik  |     8500.00 |
| kaushik |     8500.00 |
| Komal   |     4500.00 |
| Muffy   |    10000.00 |
| Ramesh  |     3500.00 |
+---------+-------------+
```

**SQL ─ Distinct Keyword**

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

There may be a situation when you have multiple duplicate records in a table. While fetching such records, it makes more sense to fetch only those unique records instead of fetching duplicate records.

Syntax The basic syntax of DISTINCT keyword to eliminate the duplicate records is as follows:

```
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
     ORDER BY SALARY;
```

This would produce the following result, where the salary (2000) is coming twice which is a duplicate record from the original table.

```
+----------+
| SALARY   |
+----------+
|  1500.00 |
|  2000.00 |
|  2000.00 |
|  4500.00 |
|  6500.00 |
|  8500.00 |
| 10000.00 |
+----------+
```

Now, let us use the DISTINCT keyword with the above SELECT query and then see the result.

```
SQL> SELECT DISTINCT SALARY FROM CUSTOMERS
     ORDER BY SALARY;
```

This would produce the following result where we do not have any duplicate entry.

```
+----------+
| SALARY   |
+----------+
|  1500.00 |
|  2000.00 |
|  4500.00 |
|  6500.00 |
|  8500.00 |
| 10000.00 |
+----------+
```

**SQL ─ 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. These constraints have already been discussed in SQL - RDBMS Concepts chapter, but it’s worth to revise them at this point.

 NOT NULL Constraint: Ensures that a column cannot have a NULL value.

 DEFAULT Constraint: Provides a default value for a column when none is specified.

 UNIQUE Constraint: Ensures that all values in a column are different.

 PRIMARY Key: Uniquely identifies each row/record in a database table.

 FOREIGN Key: Uniquely identifies row/record in any of the given database tables.

 CHECK Constraint: The CHECK constraint ensures that all the values in a column satisfies certain conditions.

 INDEX: Used to create and retrieve data from the database very quickly.

Constraints can be specified when a table is created with the CREATE TABLE statement or you can use the ALTER TABLE statement to create constraints even after the table is created.

**SQL - NOT NULL Constraint**

By default, a column can hold NULL values. If you do not want a column to have a NULL value, then you need to define such a constraint on this column specifying that NULL is now not allowed for that column.

A NULL is not the same as no data, rather, it represents unknown data.

Example For example, the following SQL query creates a new table called CUSTOMERS and adds five columns, three of which are – ID, NAME and AGE. In this we specify not to accept NULLs:

```
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)
);
```

If CUSTOMERS table has already been created, then to add a NOT NULL constraint to the SALARY column in Oracle and MySQL, you would write a query like the one that is shown in the following code block.

```
ALTER TABLE CUSTOMERS
   MODIFY SALARY  DECIMAL (18, 2) NOT NULL;
```

**SQL - DEFAULT Constraint**

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

Example For example, the following SQL creates a new table called CUSTOMERS and adds five 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.

```
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)
);
```

If the CUSTOMERS table has already been created, then to add a DEFAULT constraint to the SALARY column, you would write a query like the one which is shown in the code block below.

```
ALTER TABLE CUSTOMERS
   MODIFY SALARY  DECIMAL (18, 2) DEFAULT 5000.00;
```

Drop Default Constraint To drop a DEFAULT constraint, use the following SQL query.

```
ALTER TABLE CUSTOMERS
   ALTER COLUMN SALARY DROP DEFAULT;
```

**SQL - UNIQUE Constraint**

The UNIQUE Constraint prevents two records from having identical values in a column. In the CUSTOMERS table, for example, you might want to prevent two or more people from having an identical age.

Example For example, the following SQL query creates a new table called CUSTOMERS and adds five columns. Here, the AGE column is set to UNIQUE, so that you cannot have two records with the same age.

```
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)
);
```

If the CUSTOMERS table has already been created, then to add a UNIQUE constraint to the AGE column. You would write a statement like the query that is given in the code block below.

```
ALTER TABLE CUSTOMERS
   MODIFY AGE INT NOT NULL UNIQUE;
```

DROP a UNIQUE Constraint

To drop a UNIQUE constraint, use the following SQL query.

```
ALTER TABLE CUSTOMERS
   DROP CONSTRAINT myUniqueConstraint;
```

If you are using MySQL, then you can use the following syntax:

```
ALTER TABLE CUSTOMERS
   DROP INDEX myUniqueConstraint;
```

**SQL ─ Primary Key**

A primary key is a field in a table which uniquely identifies each row/record in a database table. Primary keys must contain unique values. A primary key column cannot have NULL values.

A table can have only one primary key, which may consist of single or multiple fields. When multiple fields are used as a primary key, they are called a composite key.

If a table has a primary key defined on any field(s), then you cannot have two records having the same value of that field(s).

Note: You would use these concepts while creating database tables.

Create Primary Key Here is the syntax to define the ID attribute as a primary key in a CUSTOMERS table.

```
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 a PRIMARY KEY constraint on the "ID" column when the CUSTOMERS table already exists, use the following SQL syntax:

```
ALTER TABLE CUSTOMER ADD PRIMARY KEY (ID);
```

NOTE: If you use the ALTER TABLE statement to add a primary key, the primary key column(s) should have already been declared to not contain NULL values (when the table was first created).

For defining a PRIMARY KEY constraint on multiple columns, use the SQL syntax given below.

```
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, NAME)
);
```

To create a PRIMARY KEY constraint on the "ID" and "NAMES" columns when CUSTOMERS table already exists, use the following SQL syntax.

```
ALTER TABLE CUSTOMERS
   ADD CONSTRAINT PK_CUSTID PRIMARY KEY (ID, NAME);
```

Delete Primary Key You can clear the primary key constraints from the table with the syntax given below.

```
ALTER TABLE CUSTOMERS DROP PRIMARY KEY ;
```

**SQL ─ Foreign Key**
A foreign key is a key used to link two tables together. This is sometimes also called as a referencing key.
A Foreign Key is a column or a combination of columns whose values match a Primary Key in a different table.
The relationship between 2 tables matches the Primary Key in one of the tables with a Foreign Key in the second table.
If a table has a primary key defined on any field(s), then you cannot have two records having the same value of that field(s).

Example Consider the structure of the following two tables.
CUSTOMERS Table:

```
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)
);
```

ORDERS Table

```
CREATE TABLE ORDERS (
      ID         INT       NOT NULL,
      DATE       DATETIME,
      CUSTOMER_ID INT references CUSTOMERS(ID),
      AMOUNT    double,
      PRIMARY KEY (ID)
);
```

If the ORDERS table has already been created and the foreign key has not yet been set, the use the syntax for specifying a foreign key by altering a table.

```
ALTER TABLE ORDERS
  ADD FOREIGN KEY (Customer_ID) REFERENCES CUSTOMERS (ID);
```

DROP a FOREIGN KEY Constraint

To drop a FOREIGN KEY constraint, use the following SQL syntax.

```
ALTER TABLE ORDERS
  DROP FOREIGN KEY;
```

**SQL ─ CHECK Constraint**

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 For example, the following program creates a new table called CUSTOMERS and adds five columns. Here, we add a CHECK with AGE column, so that you cannot have any CUSTOMER who is below 18 years.

```
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)
);
```

If the CUSTOMERS table has already been created, then to add a CHECK constraint to AGE column, you would write a statement like the one given below.

```
ALTER TABLE CUSTOMERS
  MODIFY AGE INT NOT NULL CHECK (AGE >= 18 );
```

You can also use the following syntax, which supports naming the constraint in multiple columns as well:

```
ALTER TABLE CUSTOMERS
  ADD CONSTRAINT myCheckConstraint CHECK(AGE >= 18);
```

DROP a CHECK Constraint To drop a CHECK constraint, use the following SQL syntax. This syntax does not work with MySQL.

```
ALTER TABLE CUSTOMERS
  DROP CONSTRAINT myCheckConstraint;
```

**SQL ─ INDEX Constraint**

The INDEX is used to create and retrieve data from the database very quickly. An Index can be created by using a single or a group of columns in a table. When the index 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 an index. A selection of fields depends on what you are using in your SQL queries.

Example For example, the following SQL syntax creates a new table called CUSTOMERS and adds five columns:

```
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)
);
```

Now, you can create an index on a single or multiple columns using the syntax given below.

```
CREATE INDEX index_name
   ON table_name ( column1, column2.....);
```

To create an INDEX on the AGE column, to optimize the search on customers for a specific age, follow the SQL syntax which is given below.

```
CREATE INDEX idx_age
   ON CUSTOMERS ( AGE );
```

DROP an INDEX Constraint To drop an INDEX constraint, use the following SQL syntax.

```
ALTER TABLE CUSTOMERS
  DROP INDEX idx_age;
```

**SQL ─ Using Joins**

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

Consider the following two tables:

Table 1: CUSTOMERS Table

```
+----+----------+-----+-----------+----------+
| 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 |
+----+----------+-----+-----------+----------+
```

Table 2: ORDERS Table

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | 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, let us join these two tables in our SELECT statement as shown below.

```
SQL> SELECT ID, NAME, AGE, AMOUNT
        FROM CUSTOMERS, ORDERS
        WHERE  CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result.

```
+----+----------+-----+--------+
| ID | NAME     | AGE | AMOUNT |
+----+----------+-----+--------+
|  3 | kaushik  |  23 |   3000 |
|  3 | kaushik  |  23 |   1500 |
|  2 | Khilan   |  25 |   1560 |
|  4 | Chaitali |  25 |   2060 |
+----+----------+-----+--------+
```

Here, it is noticeable that the join is performed in the WHERE clause. Several operators can be used to join tables, such as =, <, >, <>, <=, >=, !=, BETWEEN, LIKE, and NOT; they can all be used to join tables. However, the most common operator is the equal to symbol.

There are different types of joins available in SQL:

 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: is used to 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.

Let us now discuss each of these joins in detail.

**SQL - INNER JOIN**

The most important and frequently used of the joins is the INNER JOIN. They are also referred to as an EQUIJOIN.

The INNER JOIN creates a new result table by combining column values of two tables (table1 and table2) based upon the join-predicate. The query compares each row of table1 with each row of table2 to find all pairs of rows which satisfy the join-predicate. When the join-predicate is satisfied, column values for each matched pair of rows of A and B are combined into a result row.

Syntax

The basic syntax of the INNER JOIN is as follows.

```
SELECT table1.column1, table2.column2...
FROM table1
INNER JOIN table2
ON table1.common_field = table2.common_field;
```

Example Consider the following two tables.

Table 1: CUSTOMERS Table is as follows.

```
+----+----------+-----+-----------+----------+
| 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 |
+----+----------+-----+-----------+----------+
```

Table 2: ORDERS Table is as follows.

```
+-----+---------------------+-------------+--------+
| OID | DATE                | CUSTOMER_ID | 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, let us join these two tables using the INNER JOIN as follows:

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
     FROM CUSTOMERS
     INNER JOIN ORDERS
     ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result.

```
+----+----------+--------+---------------------+
| 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 |
+----+----------+--------+---------------------+
```

SQL ─ LEFT JOIN

The SQL LEFT JOIN returns all rows from the left table, even if there are no matches in the right table. This means that if the ON clause matches 0 (zero) records in the right table; the join will still return a row in the result, but with NULL in each column from the right table.

This means that a left join returns all the values from the left table, plus matched values from the right table or NULL in case of no matching join predicate.

Syntax The basic syntax of a LEFT JOIN is as follows.

```
SELECT table1.column1, table2.column2...
FROM table1
LEFT JOIN table2
ON table1.common_field = table2.common_field;
```

Here, the given condition could be any given expression based on your requirement.

SQL

86

Example Consider the following two tables,

Table 1: CUSTOMERS Table is as follows.

```
+----+----------+-----+-----------+----------+
| 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 |
+----+----------+-----+-----------+----------+
```

Table 2: Orders Table is as follows.

```
+-----+---------------------+-------------+--------+
| OID | DATE                | CUSTOMER_ID | 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, let us join these two tables using the LEFT JOIN as follows.
```
SQL> SELECT  ID, NAME, AMOUNT, DATE
     FROM CUSTOMERS
     LEFT JOIN ORDERS
     ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```
This would produce the following result:
```
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
```
SQL
   87
```
|  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                |
+----+----------+--------+---------------------+
```
SQL - RIGHT JOIN
The SQL RIGHT JOIN returns all rows from the right table, even if there are no matches in the left table. This means that if the ON clause matches 0 (zero) records in the left table; the join will still return a row in the result, but with NULL in each column from the left table.
This means that a right join returns all the values from the right table, plus matched values from the left table or NULL in case of no matching join predicate.
Syntax The basic syntax of a RIGHT JOIN is as follow.
```
SELECT table1.column1, table2.column2...
FROM table1
RIGHT JOIN table2
ON table1.common_field = table2.common_field;
```
Example Consider the following two tables,
Table 1: CUSTOMERS Table is as follows.
```
+----+----------+-----+-----------+----------+
| 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 |
SQL
   88
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Table 2: ORDERS Table is as follows.

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | 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, let us join these two tables using the RIGHT JOIN as follows.

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
     FROM CUSTOMERS
     RIGHT JOIN ORDERS
     ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result:

```
+------+----------+--------+---------------------+
| 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 |
+------+----------+--------+---------------------+
```

SQL ─ FULL JOIN The SQL FULL JOIN combines the results of both left and right outer joins.

The joined table will contain all records from both the tables and fill in NULLs for missing matches on either side.

SQL

89

Syntax The basic syntax of a FULL JOIN is as follows:

```
SELECT table1.column1, table2.column2...
FROM table1
FULL JOIN table2
ON table1.common_field = table2.common_field;
```

Here, the given condition could be any given expression based on your requirement.

Example Consider the following two tables.

Table 1: CUSTOMERS Table is as follows.

```
+----+----------+-----+-----------+----------+
| 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 |
+----+----------+-----+-----------+----------+
```

Table 2: ORDERS Table is as follows.

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | 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 |
+-----+---------------------+-------------+--------+
```

us join these two tables using FULL JOIN as follows.

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
     FROM CUSTOMERS
     FULL JOIN ORDERS
```

```
    ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```
This would produce the following 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 |
+------+----------+--------+---------------------+
```
If your Database does not support FULL JOIN (MySQL does not support FULL JOIN), then you can use UNION ALL clause to combine these two JOINS as shown below.
```
SQL> SELECT  ID, NAME, AMOUNT, DATE
    FROM CUSTOMERS
    LEFT JOIN ORDERS
    ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
UNION ALL
    SELECT  ID, NAME, AMOUNT, DATE
    FROM CUSTOMERS
    RIGHT JOIN ORDERS
    ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
```

SQL ─ SELF JOIN The SQL SELF JOIN is used to join a table to itself as if the table were two tables; temporarily renaming at least one table in the SQL statement.

Syntax The basic syntax of SELF JOIN is as follows:
```
SELECT a.column_name, b.column_name...
FROM table1 a, table1 b
```

```
WHERE a.common_field = b.common_field;
```

Here, the WHERE clause could be any given expression based on your requirement.

Example Consider the following table.

CUSTOMERS Table is as follows.

```
+----+----------+-----+-----------+----------+
| 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 |
+----+----------+-----+-----------+----------+
```

Now, let us join this table using SELF JOIN as follows:

```
SQL> SELECT  a.ID, b.NAME, a.SALARY
     FROM CUSTOMERS a, CUSTOMERS b
     WHERE a.SALARY < b.SALARY;
```

This would produce the following result:

```
+----+----------+---------+
| ID | NAME     | SALARY  |
+----+----------+---------+
|  2 | Ramesh   | 1500.00 |
|  2 | kaushik  | 1500.00 |
|  1 | Chaitali | 2000.00 |
|  2 | Chaitali | 1500.00 |
|  3 | Chaitali | 2000.00 |
|  6 | Chaitali | 4500.00 |
|  1 | Hardik   | 2000.00 |
|  2 | Hardik   | 1500.00 |
|  3 | Hardik   | 2000.00 |
|  4 | Hardik   | 6500.00 |
|  6 | Hardik   | 4500.00 |
|  1 | Komal    | 2000.00 |
|  2 | Komal    | 1500.00 |
|  3 | Komal    | 2000.00 |
|  1 | Muffy    | 2000.00 |
|  2 | Muffy    | 1500.00 |
|  3 | Muffy    | 2000.00 |
|  4 | Muffy    | 6500.00 |
|  5 | Muffy    | 8500.00 |
|  6 | Muffy    | 4500.00 |
+----+----------+---------+
```

SQL ─ CARTESIAN or CROSS JOIN

The CARTESIAN JOIN or CROSS JOIN returns the Cartesian product of the sets of records from two or more joined tables. Thus, it equates to an inner join where the join-condition always evaluates to either True or where the join-condition is absent from the statement.

Syntax The basic syntax of the CARTESIAN JOIN or the CROSS JOIN is as follows:

```
SELECT table1.column1, table2.column2...
FROM  table1, table2 [, table3 ]
```

Example Consider the following two tables.

Table 1: CUSTOMERS table is as follows.

```
+----+----------+-----+-----------+----------+
| 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 |
+----+----------+-----+-----------+----------+
```

Table 2: ORDERS Table is as follows:

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | 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, let us join these two tables using INNER JOIN as follows:

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
     FROM CUSTOMERS, ORDERS;
```

This would produce the following result:

```
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
|  1 | Ramesh   |   3000 | 2009-10-08 00:00:00 |
|  1 | Ramesh   |   1500 | 2009-10-08 00:00:00 |
|  1 | Ramesh   |   1560 | 2009-11-20 00:00:00 |
|  1 | Ramesh   |   2060 | 2008-05-20 00:00:00 |
|  2 | Khilan   |   3000 | 2009-10-08 00:00:00 |
|  2 | Khilan   |   1500 | 2009-10-08 00:00:00 |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  2 | Khilan   |   2060 | 2008-05-20 00:00:00 |
|  3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1560 | 2009-11-20 00:00:00 |
|  3 | kaushik  |   2060 | 2008-05-20 00:00:00 |
|  4 | Chaitali |   3000 | 2009-10-08 00:00:00 |
|  4 | Chaitali |   1500 | 2009-10-08 00:00:00 |
|  4 | Chaitali |   1560 | 2009-11-20 00:00:00 |
|  4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
|  5 | Hardik   |   3000 | 2009-10-08 00:00:00 |
|  5 | Hardik   |   1500 | 2009-10-08 00:00:00 |
|  5 | Hardik   |   1560 | 2009-11-20 00:00:00 |
|  5 | Hardik   |   2060 | 2008-05-20 00:00:00 |
|  6 | Komal    |   3000 | 2009-10-08 00:00:00 |
|  6 | Komal    |   1500 | 2009-10-08 00:00:00 |
|  6 | Komal    |   1560 | 2009-11-20 00:00:00 |
|  6 | Komal    |   2060 | 2008-05-20 00:00:00 |
|  7 | Muffy    |   3000 | 2009-10-08 00:00:00 |
|  7 | Muffy    |   1500 | 2009-10-08 00:00:00 |
|  7 | Muffy    |   1560 | 2009-11-20 00:00:00 |
|  7 | Muffy    |   2060 | 2008-05-20 00:00:00 |
+----+----------+--------+---------------------+
```

## SQL ─ UNIONS CLAUSE

The SQL 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 The basic syntax of a UNION clause is as follows:

```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
UNION
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
```

Here, the given condition could be any given expression based on your requirement.

Example Consider the following two tables.
Table 1: CUSTOMERS Table is as follows.

```
+----+----------+-----+-----------+----------+
| 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 |
+----+----------+-----+-----------+----------+
```

Table 2: ORDERS Table is as follows.

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | 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, let us join these two tables in our SELECT statement as follows:

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
     FROM CUSTOMERS
     LEFT JOIN ORDERS
     ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
UNION
     SELECT  ID, NAME, AMOUNT, DATE
     FROM CUSTOMERS
     RIGHT JOIN ORDERS
     ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following 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                |
+------+----------+--------+---------------------+
```

The UNION ALL Clause

The UNION ALL operator is used to combine the results of two SELECT statements including duplicate rows.
The same rules that apply to the UNION clause will apply to the UNION ALL operator.
Syntax The basic syntax of the UNION ALL is as follows.

```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
UNION ALL
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
```

Here, the given condition could be any given expression based on your requirement.
Example
SQL
98
Consider the following two tables,
Table 1: CUSTOMERS Table is as follows.

```
+----+----------+-----+-----------+----------+
| 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 |
+----+----------+-----+-----------+----------+
```

Table 2: ORDERS table is as follows.

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | 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, let us join these two tables in our SELECT statement as follows:
```
SQL> SELECT  ID, NAME, AMOUNT, DATE
     FROM CUSTOMERS
     LEFT JOIN ORDERS
     ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
UNION ALL
     SELECT  ID, NAME, AMOUNT, DATE
     FROM CUSTOMERS
     RIGHT JOIN ORDERS
     ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```
This would produce the following 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 |
+------+----------+--------+---------------------+
```
There are two other clauses (i.e., operators), which are like the UNION clause.

- SQL INTERSECT Clause: This is used to combine two SELECT statements, but returns rows only from the first SELECT statement that are identical to a row in the second SELECT statement.
- SQL EXCEPT Clause: This combines two SELECT statements and returns rows from the first SELECT statement that are not returned by the second SELECT statement.

**SQL ─ INTERSECT Clause**

The SQL INTERSECT clause/operator is used to combine two SELECT statements, but returns rows only from the first SELECT statement that are identical to a row in the second SELECT statement. This means INTERSECT returns only common rows returned by the two SELECT statements.

Just as with the UNION operator, the same rules apply when using the INTERSECT operator. MySQL does not support the INTERSECT operator.

Syntax The basic syntax of INTERSECT is as follows.

```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
INTERSECT
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
```

Here, the given condition could be any given expression based on your requirement.

Example Consider the following two tables.

Table 1: CUSTOMERS Table is as follows.

```
+----+----------+-----+-----------+----------+
| 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 |
+----+----------+-----+-----------+----------+
```

Table 2: ORDERS Table is as follows.

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | 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, let us join these two tables in our SELECT statement as follows.

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
     FROM CUSTOMERS
     LEFT JOIN ORDERS
     ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
INTERSECT
     SELECT  ID, NAME, AMOUNT, DATE
     FROM CUSTOMERS
     RIGHT JOIN ORDERS
     ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```
This would produce the following result.
```
+------+---------+--------+---------------------+
| ID   | NAME    | AMOUNT | DATE                |
+------+---------+--------+---------------------+
|    3 | kaushik |   3000 | 2009-10-08 00:00:00 |
|    3 | kaushik |   1500 | 2009-10-08 00:00:00 |
|    2 | Ramesh  |   1560 | 2009-11-20 00:00:00 |
|    4 | kaushik |   2060 | 2008-05-20 00:00:00 |
+------+---------+--------+---------------------+
```
SQL ─ EXCEPT Clause The SQL EXCEPT clause/operator is used to combine two SELECT statements and returns rows from the first SELECT statement that are not returned by the second SELECT statement. This means EXCEPT returns only rows, which are not available in the second SELECT statement.
Just as with the UNION operator, the same rules apply when using the EXCEPT operator. MySQL does not support the EXCEPT operator.
Syntax The basic syntax of EXCEPT is as follows.
```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
EXCEPT
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
```
SQL
  102

[WHERE condition]
Here, the given condition could be any given expression based on your requirement.
Example Consider the following two tables.
Table 1: CUSTOMERS Table is as follows.

```
+----+----------+-----+-----------+----------+
| 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 |
+----+----------+-----+-----------+----------+
```

Table 2: ORDERS table is as follows.

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | 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, let us join these two tables in our SELECT statement as shown below.

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS
   LEFT JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
EXCEPT
SQL
```

## SQL - TRUNCATE TABLE Command

The SQL TRUNCATE TABLE command is used to delete complete data from an existing table.

You can also use DROP TABLE command to delete complete table but it would remove complete table structure form the database and you would need to re-create this table once again if you wish you store some data.
Syntax The basic syntax of a TRUNCATE TABLE command is as follows.
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 |
+----+----------+-----+-----------+----------+
```

Following is the example of a Truncate command.

```
SQL > TRUNCATE TABLE CUSTOMERS;
```

Now, the CUSTOMERS table is truncated and the output from SELECT statement will be as shown in the code block below:

```
SQL> SELECT * FROM CUSTOMERS;
Empty set (0.00 sec)
```

## SQL Functions

SQL has many built-in functions for performing calculations on data.

## SQL Aggregate Functions

SQL aggregate functions return a single value, calculated from values in a column.

Useful aggregate functions:

- AVG() - Returns the average value
- COUNT() - Returns the number of rows

- FIRST() - Returns the first value
- LAST() - Returns the last value
- MAX() - Returns the largest value
- MIN() - Returns the smallest value
- SUM() - Returns the sum

## SQL Scalar functions

SQL scalar functions return a single value, based on the input value.

Useful scalar functions:

- UCASE() - Converts a field to upper case
- LCASE() - Converts a field to lower case
- MID() - Extract characters from a text field
- LEN() - Returns the length of a text field
- ROUND() - Rounds a numeric field to the number of decimals specified
- NOW() - Returns the current system date and time
- FORMAT() - Formats how a field is to be displayed

## The AVG() Function

The AVG() function returns the average value of a numeric column.
SQL AVG() Syntax
SELECT AVG(column_name) FROM table_name

**The COUNT() function** returns the number of rows that matches a specified criteria.
SQL COUNT(column_name) Syntax
The COUNT(column_name) function returns the number of values (NULL values will not be counted) of the specified column:
SELECT COUNT(column_name) FROM table_name;

## The MAX() Function

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

### SQL MAX() Syntax

SELECT MAX(column_name) FROM table_name;

## The MIN() Function

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

### SQL MIN() Syntax

```
SELECT MIN(column_name) FROM table_name;
```

## The SUM() Function

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

### SQL SUM() Syntax

```
SELECT SUM(column_name) FROM table_name;
```

## The UCASE() Function

The UCASE() function converts the value of a field to uppercase.

### SQL UCASE() Syntax

```
SELECT UCASE(column_name) FROM table_name;
```

## The LCASE() Function

The LCASE() function converts the value of a field to lowercase.

### SQL LCASE() Syntax

```
SELECT LCASE(column_name) FROM table_name;
```

## The NOW() Function

The NOW() function returns the current system date and time.

### SQL NOW() Syntax

```
SELECT NOW() FROM table_name;
```