**SQL Databases;**

CREATE DATABASE STUDENT;

SHOW DATABASES;

USE STUDENT;

SHOW TABLES;

CREATE TABLE STUDENT:

##RENAME DATABASE testDB TO tutorials\_DB;

SELECT DATABASE(); // to find the current working DB

**SQL TABLES;**

**CREATE TABLE table\_name(**

**column1 datatype,**

**column2 datatype,**

**column3 datatype,**

**.....**

**columnN datatype,**

**PRIMARY KEY( one or more columns )**

**);**

CREATE TABLE students (

id INTEGER PRIMARY KEY,

first\_name VARCHAR(30),

last\_name VARCHAR(30),

age INTEGER,

gender VARCHAR(30),

email VARCHAR(30),

address VARCHAR(30),

phone\_number VARCHAR(30)

);

**In this example, the students table has eight attributes:**

id - an integer column representing the unique identifier of each student (primary key).

first\_name - a VARCHAR(30) column representing the first name of the student.

last\_name - a VARCHAR(30) column representing the last name of the student.

age - an integer column representing the age of the student.

gender - a VARCHAR(30) column representing the gender of the student.

email - a VARCHAR(30) column representing the email address of the student.

address - a VARCHAR(30) column representing the address of the student.

phone\_number - a VARCHAR(30) column representing the phone number of the student.

INSERT INTO students (id, first\_name, last\_name, age, gender, email, address, phone\_number)

VALUES

(1, 'John', 'Doe', 20, 'Male', 'john.doe@example.com', '123 Main St', '123-456-7890'),

(2, 'Jane', 'Smith', 22, 'Female', 'jane.smith@example.com', '456 Elm St', '987-654-3210'),

(3, 'Michael', 'Johnson', 19, 'Male', 'michael.johnson@example.com', '789 Oak St', '555-123-4567'),

(4, 'Emily', 'Williams', 21, 'Female', 'emily.williams@example.com', '321 Maple Ave', '111-222-3333'),

(5, 'Daniel', 'Brown', 20, 'Male', 'daniel.brown@example.com', '444 Pine St', '444-555-6666'),

(6, 'Olivia', 'Jones', 23, 'Female', 'olivia.jones@example.com', '567 Cedar Ave', '777-888-9999'),

(7, 'William', 'Taylor', 20, 'Male', 'william.taylor@example.com', '890 Walnut St', '222-333-4444'),

(8, 'Sophia', 'Anderson', 21, 'Female', 'sophia.anderson@example.com', '123 Oakwood Ave', '888-999-0000'),

(9, 'Matthew', 'Thomas', 22, 'Male', 'matthew.thomas@example.com', '456 Maplewood Dr', '555-777-8888'),

(10, 'Ava', 'Jackson', 20, 'Female', 'ava.jackson@example.com', '789 Pinecrest Rd', '777-000-1111'),

(11, 'Jacob', 'White', 23, 'Male', 'jacob.white@example.com', '321 Elmwood Ln', '444-222-3333'),

(12, 'Mia', 'Harris', 21, 'Female', 'mia.harris@example.com', '654 Pinehill Ave', '666-777-8888'),

(13, 'Ethan', 'Martin', 20, 'Male', 'ethan.martin@example.com', '987 Cedarwood Rd', '111-222-3333'),

(14, 'Charlotte', 'Thompson', 22, 'Female', 'charlotte.thompson@example.com', '753 Oakhill Dr', '888-999-1111'),

(15, 'Alexander', 'Garcia', 19, 'Male', 'alexander.garcia@example.com', '369 Maple Ave', '333-444-5555'),

(16, 'Amelia', 'Davis', 21, 'Female', 'amelia.davis@example.com', '852 Willow St', '555-666-7777'),

(17, 'James', 'Rodriguez', 20, 'Male', 'james.rodriguez@example.com', '147 Elmwood Ave', '777-888-9999'),

(18, 'Lily', 'Lopez', 22, 'Female', 'lily.lopez@example.com', '963 Pine St', '222-333-4444'),

(19, 'Benjamin', 'Wilson', 23, 'Male', 'benjamin.wilson@example.com', '258 Oakwood Ln', '666-777-8888'),

(20, 'Grace', 'Lee', 20, 'Female', 'grace.lee@example.com', '753 Maple Ave', '111-222-3333'),

(21, 'Daniel', 'Hall', 22, 'Male', 'daniel.hall@example.com', '369 Cedarwood Dr', '444-555-6666'),

(22, 'Victoria', 'Young', 21, 'Female', 'victoria.young@example.com', '852 Elm St', '777-888-9999'),

(23, 'Henry', 'Gonzalez', 20, 'Male', 'henry.gonzalez@example.com', '456 Pinecrest Ave', '222-333-4444'),

(24, 'Sofia', 'Clark', 23, 'Female', 'sofia.clark@example.com', '963 Oakwood Rd', '555-666-7777'),

(25, 'Christopher', 'Lewis', 21, 'Male', 'christopher.lewis@example.com', '147 Maplewood Ave', '888-999-1111');

**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)

);

### **Verification**

Once your table is created, you can check if it has been created successfully or not. You can use SQL **DESC table\_name** command to list down the description of the table as follows:

DESC CUSTOMERS;

## SQL CREATE TABLE IF NOT EXISTS

Consider a situation where you will try to create a table which already exists, in such situation MySQL will throw the following error.

ERROR 1050 (42S01): Table 'CUSTOMERS' already exists

So, to avoid such error we can use SQL command **CREATE TABLE IF NOT EXISTS** to create a table.

### **Syntax**

Following is the basic syntax of a CREATE TABLE IF NOT EXISTS statement −

### **CREATE TABLE IF NOT EXISTS table\_name(**

### **column1 datatype,**

### **column2 datatype,**

### **column3 datatype,**

### **.....**

### **columnN datatype,**

### **PRIMARY KEY( one or more columns )**

### **);**

### **Example**

The following SQL command will create the **CUSTOMERS** table only when there is no table exists with the same name otherwise it will exit without any error.

CREATE TABLE IF NOT EXISTS CUSTOMERS(

Emp\_ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

Emp\_AGE INT NOT NULL,

ADDRESS CHAR (25),

Emp\_SALARY DECIMAL (18, 2),

PRIMARY KEY (ID)

);

## Creating a Table from an Existing Table

Instead of creating a new table every time, one can also copy an existing table and its contents including its structure, into a new table. This can be done using a combination of the CREATE TABLE statement and the SELECT statement. Since its structure is copied, the new table will have the same column definitions as the original table. Furthermore, the new table would be populated using the existing values from 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 CONDITION];**

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 −

**CREATE TABLE SALARY AS**

**SELECT ID, SALARY**

**FROM CUSTOMERS;**

CREATE TABLE Information AS

SELECT first\_name, last\_name, age

FROM students;

The **AS** keyword is used to specify the name of the new table (**Information**).

## The SQL RENAME TABLE Statement

RENAME TABLE CUSTOMERS to BUYERS;

## The SQL Delete Statement

The DELETE statement is used to delete rows from a table. If you want to remove a specific row from a table you should use WHERE condition.

**DELETE** **FROM** table\_name [**WHERE** condition];

But if you do not specify the WHERE condition it will remove all the rows from the table.

**DELETE** **FROM** table\_name;

There are some more terms similar to DELETE statement like as DROP statement and TRUNCATE statement, but they are not exactly same there are some differences between them.

Following SQL **TRUNCATE TABLE CUSTOMER** statement will remove all the records of the CUSTOMERS table −

* TRUNCATE TABLE CUSTOMERS;

### **Verification**

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

* SELECT \* FROM CUSTOMERS;

Following will be the output:

Empty set (0.00 sec)

TRUNCATE vs DELETE

Even though the TRUNCATE and DELETE commands work similar logically, there are some major differences that exist between them. They are detailed in the table below.

|  |  |
| --- | --- |
| **DELETE** | **TRUNCATE** |
| The DELETE command in SQL removes one or more rows from a table based on the conditions specified in a WHERE Clause. | SQL's TRUNCATE command is used to remove all of the rows from a table, regardless of whether or not any conditions are met. |
| It is a DML(Data Manipulation Language) command. | It is a DDL(Data Definition Language) command. |
| There is a need to make a manual COMMIT after making changes to the DELETE command, for the modifications to be committed. | When you use the TRUNCATE command, the modifications made to the table are committed automatically. |
| It deletes rows one at a time and applies same criteria to each deletion. | It removes all of the information in one go. |
| The WHERE clause serves as the condition in this case. | The WHERE Clause is not available. |
| All rows are locked after deletion. | TRUNCATE utilizes a table lock, which locks the pages so they cannot be deleted. |
| It makes a record of each and every transaction in the log file. | The only activity recorded is the deallocation of the pages on which the data is stored. |
| It consumes a greater amount of transaction space compared to TRUNCATE command. | It takes comparatively less amount of transaction space. |
| If there is an identity column, the table identity is not reset to the value it had when the table was created. | It returns the table identity to a value it was given as a seed. |
| It requires authorization to delete. | It requires table alter permission. |
| When it comes to large databases, it is much slower. | It is much faster. |

TRUNCATE vs DROP

Unlike TRUNCATE that resets the table structure, DROP command completely frees the tablespace from the memory. They are both Data Definition Language (DDL) operations as they interact with the definitions of database objects; which allows the database to automatically commit once these commands are executed with no chance to roll back.

However, there are still some differences exist between these two commands, which have been summarized in the following table −

|  |  |
| --- | --- |
| **DROP** | **TRUNCATE** |
| The DROP command in SQL removes an entire table from a database including its definition, indexes, constraints, data etc. | The TRUNCATE command is used to remove all of the rows from a table, regardless of whether or not any conditions are met and resets the table definition. |
| It is a DDL(Data Definition Language) command. | It is also a DDL(Data Definition Language) command. |
| The table space is completely freed from the memory. | The table still exists in the memory. |
| All the integrity constraints are removed. | The integrity constraints still exist in the table. |
| Requires ALTER and CONTROL permissions on the table schema and table respectively, to be able to perform this command. | Only requires the ALTER permissions to truncate the table. |
| DROP command is much slower than TRUNCATE but faster than DELETE. | TRUNCATE command is faster than both DROP and DELETE commands. |

# SQL - Clone Tables

There may be a situation when you need an exact copy of a table with the same columns, attributes, indexes, default values and so forth. Instead of spending time on creating the exact same version of an existing table, you can create a clone of the existing table.

SQL **Cloning Operation** allows to create the exact copy of an existing table along with its definition. There are three types of cloning possible using SQL in various RDBMS; they are listed below −

* Simple Cloning
* Shallow Cloning
* Deep Cloning

## Simple Cloning in MySQL

Simple cloning operation creates a new replica table from the existing table and copies all the records in newly created table.

To break this process down, a new table is created using the CREATE TABLE statement; and the data from the existing table, as a result of SELECT statement, is copied into the new table.

Here, clone table inherits only the basic column definitions like the NULL settings and default values from the original table. It does not inherit the indices and AUTO\_INCREMENT definitions.

### **Syntax**

Following is the basic syntax to perform simple cloning in MySQL−

CREATE TABLE new\_table SELECT \* FROM original\_table;

CREATE TABLE CUSTOMERS (

ID INT PRIMARY KEY,

NAME VARCHAR(20),

AGE INT,

ADDRESS VARCHAR(50),

SALARY DECIMAL(10, 2)

);

### 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, 'Pune', 4500.00);

### **Example**

Consider the following existing CUSTOMERS table which will be cloned in next new few steps.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 |

Now let's use the following SQL statement to create NEW\_CUSTOMERS table using the existing table CUSTOMERS.

CREATE TABLE NEW\_CUSTOMERS SELECT \* FROM CUSTOMERS;

### **Output**

The output is displayed as −

Query OK, 7 rows affected (0.06 sec)

Records: 7 Duplicates: 0 Warnings: 0

### **Verification**

To verify whether the table has been cloned successfully, we can use the following SELECT query −

SELECT \* FROM NEW\_CUSTOMERS;

If NEW\_CUSTOMERS table is created successfully, then it should get all the records which are available in CUSTOMERS table.

## Shallow Cloning in MySQL

Shallow cloning operation creates a new replica table from the existing table but does not copy any data records into newly created table, so only new but empty table is created.

Here, the clone table contains only the structure of the original table along with the column attributes including indices and AUTO\_INCREMENT definition.

### **Syntax**

Following is the basic syntax to perform shallow cloning in MySQL RDBMS –

**CREATE TABLE new\_table LIKE original\_table;**

### **Example**

Following is an example to create a shallow clone copy of the existing table CUSTOMERS.

CREATE TABLE SHALL\_CUSTOMERS LIKE CUSTOMERS;

### **Output**

The output is displayed as −

Query OK, 0 rows affected (0.06 sec)

### **Verification**

To verify whether the table has been cloned successfully, we can use the following DESC table\_name query −

DESC SHALL\_CUSTOMERS;

This will display the following information about the SHALL\_CUSTOMERS table which is just a replica of CUSTOMERS table −

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Field** | **Type** | **Null** | **Key** | **Default** | **Extra** |
| ID | int(11) | NO | PRI | NULL |  |
| NAME | varchar(20) | NO |  | NULL |  |
| AGE | int(11) | NO |  | NULL |  |
| ADDRESS | char(25) | YES |  | NULL |  |
| SALARY | decimal(18,2) | YES |  | NULL |  |

## Deep Cloning in MySQL

Deep cloning operation is a combination of simple cloning and shallow cloning. It not only copies the structure of the existing table but also its data into the newly created table. Hence, the new table will have all the contents from existing table and all the attributes including indices and the AUTO\_INCREMENT definitions.

Since it is a combination of shallow and simple cloning, this type of cloning will have two different queries to be executed: one with CREATE TABLE statement and one with INSERT INTO statement. The CREATE TABLE statement will create the new table by including all the attributes of existing table; and INSERT INTO statement will insert the data from existing table into new table.

### **Syntax**

Following is the basic syntax to perform deep cloning in MySQL RDBMS −

### CREATE TABLE new\_table LIKE original\_table;

CREATE TABLE DEEP\_CUSTOMERS LIKE CUSTOMERS;

### INSERT INTO new\_table SELECT \* FROM original\_table;

### **Example**

Following is an example to create a deep clone copy of the existing table CUSTOMERS. First step is to create a shallow clone of the existing table.

**CREATE TABLE DEEP\_CUSTOMERS LIKE CUSTOMERS;**

The output is displayed as −

Query OK, 0 rows affected (0.06 sec)

Now second step is to copy all the records from the CUSTOMERS table to DEEP\_CUSTOMERS.

INSERT INTO DEEP\_CUSTOMERS SELECT \* FROM CUSTOMERS;

### **Output**

The output is displayed as −

Query OK, 7 rows affected (0.01 sec)

Records: 7 Duplicates: 0 Warnings: 0

### **Verification**

To verify whether the table has been cloned successfully, we can use the following SELECT query −

SELECT \* FROM DEEP\_CUSTOMERS;

If DEEP\_CUSTOMERS table is cloned successfully, then it should get all the records which are available in CUSTOMERS.

# SQL - Temporary Tables

## What are Temporary Tables?

Temporary tables are pretty much what their name describes: they are the tables which are created in a database to store temporary data. We can perform SQL operations similar to the operations on permanent tables like CREATE, UPDATE, DELETE, INSERT, JOIN, etc. But these tables will be automatically deleted once the current client session is terminated. In addition to that, they can also be explicitly deleted if the users decide to drop them manually.

Various RDBMS, like MySQL, support temporary tables starting from version 3.23 onwards. If you are using an older version of MySQL than 3.23, you can't use temporary tables, but you can use **heap tables**.

As stated earlier, temporary tables will only last as long as the client session is alive. If you run the code in a PHP script, the temporary table will be destroyed automatically when the script finishes executing. If you are connected to the MySQL database server through a MySQL client program, then the temporary table will exist until you close the client connection or manually destroy the table.

## Creating Temporary Tables in MySQL

To create temporary tables in MySQL, we follow the same query as creating regular database tables. However, instead of using the CREATE TABLE statement, you use **CREATE TEMPORARY TABLE statement.**

### **Syntax**

Following is the syntax to create a temporary table –

CREATE TEMPORARY TABLE table\_name(

column1 datatype,

column2 datatype,

column3 datatype,

.....

columnN datatype,

PRIMARY KEY( one or more columns )

);

### **Example**

Following is the SQL Query to create a temporary table in MySQL database −

CREATE TEMPORARY 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)

);

Just like normal tables you can insert data into a temporary table using the INSERT statement. Following query inserts 3 records into the above created temporary table −

INSERT INTO CUSTOMERS VALUES

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

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

(3, 'kaushik', 23, 'Kota', 2000.00 );

The temporary table **CUSTOMERS** will be created and will have 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 |

When you issue a SHOW TABLES command, then your temporary table will not be displayed in the list of tables. To verify whether the temporary table is created you need to retrieve its data using the **SELECT** statement. Since all the temporary tables will be removed when the current session is closed, if you log out of the MySQL session and then issue a SELECT command, you will not find temporary table in the database.

## Dropping Temporary Tables in MySQL

Though all the temporary tables are deleted by MySQL when your database connection gets terminated, still, if you want to delete them manually, then you can do so by issuing a **DROP TEMPORARY TABLE** command.

### **Example**

Following is the basic syntax to delete a temporary table:

DROP TEMPORARY TABLE table\_name;

### **Example**

Following query drops the temporary table **CUSTOMERS** created in the previous example –

DROP TEMPORARY TABLE CUSTOMERS;

### **Verification**

Since we have removed the temporary table CUSTOMERS, if you try to retrieve the contents of it using the SELECT statement, it will generate an error saying the table does not exist.

SELECT \* FROM CUSTOMERS;

This will produce following result −

ERROR 1146: Table 'TUTORIALS.CUSTOMERS' doesn't exist

# SQL - ALTER TABLE

## SQL - ALTER TABLE Statement

The SQL **ALTER TABLE** command is a part of Data Definition Language (DDL) and modifies the structure of a table. The ALTER TABLE command can add or delete columns, create or destroy indexes, change the type of existing columns, or rename columns or the table itself.

The ALTER TABLE command can also change characteristics of a table such as the storage engine used for the table. We will make use of the following table in our examples.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 | Hyderabad | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |

### **Syntax**

Following is the basic syntax of an ALTER TABLE command −

ALTER TABLE table\_name [alter\_option ...];

Where, the **alter\_option** depends on the type of operation to be performed on a table. This article will discuss such important operations one by one.

## ALTER TABLE - ADD Column

If you need to add a new column to a table, you should use the **ADD COLUMN** option along with ALTER TABLE statement as shown below −

ALTER TABLE table\_name ADD column\_name datatype;

### **Example**

Following is the example to ADD a **New Column** to an existing table −

ALTER TABLE CUSTOMERS ADD Gender char(4);

### **Output**

Executing the query above will produce the following output −

(0 rows affected)

### **Verification**

To verify whether the CUSTOMERS table is altered by adding a new column SEX, use the SELECT statement to retrieve the records of the table −

SELECT \* FROM CUSTOMERS;

Now, the CUSTOMERS table will be displayed as follows −

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

## ALTER TABLE - DROP COLUMN

If you need to drop an existing column from a table, you should use the DROP COLUMN option along with ALTER TABLE statement as shown below.

ALTER TABLE table\_name DROP COLUMN column\_name;

### **Example**

Following is the example to DROP sex column from the existing table.

ALTER TABLE CUSTOMERS DROP COLUMN Gender;

### **Output**

Executing the query above will produce the following output −

(0 rows affected)

### **Verification**

To verify whether the CUSTOMERS table is altered by dropping an existing column SEX, use the SELECT statement to retrieve the records of the table −

SELECT \* FROM CUSTOMERS;

Now, the CUSTOMERS table is changed and following would be the output from the SELECT statement.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 | Hyderabad | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |

## ALTER TABLE - RENAME COLUMN

Following is the syntax to rename a column name of an existing table −

ALTER TABLE table\_name RENAME COLUMN old\_column\_name to new\_column\_name;

### **Example**

Following query renames NAME column in table CUSTOMERS −

ALTER TABLE CUSTOMERS RENAME COLUMN NAME to FULL\_NAME;

This will produce the following output −

Query OK, 0 rows affected (0.002 sec)

Records: 0 Duplicates: 0 Warnings: 0

## ALTER TABLE – Modify COLUMN

You can modify the properties of existing columns, such as changing the data type or size of a column. For example:

ALTER TABLE table\_name

MODIFY COLUMN column\_name new\_data\_type;

## Suppose you have a table named students with the following structure:

## +----+------------+------------+

## | id | first\_name | birthdate |

## +----+------------+------------+

## | 1 | John | 2000-01-15 |

## | 2 | Jane | 1999-05-23 |

## | 3 | Mark | 2001-03-10 |

## +----+------------+------------+

## Let's say you want to change the data type of the birthdate column from DATE to VARCHAR(10) to store dates as strings in the format 'YYYY-MM-DD':

ALTER TABLE buyers

MODIFY COLUMN Address VARCHAR(25);

## After running this SQL statement, the birthdate column's data type will be changed, and the table will look like this:

## +----+------------+------------+

## | id | first\_name | birthdate |

## +----+------------+------------+

## | 1 | John | 2000-01-15 |

## | 2 | Jane | 1999-05-23 |

## | 3 | Mark | 2001-03-10 |

## +----+------------+------------+

## In this example:

## We specify the table name as students.

## We use the MODIFY COLUMN clause to modify the properties of the birthdate column.

## We change the data type from DATE to VARCHAR(10) to store dates as strings in the 'YYYY-MM-DD' format.

## ALTER TABLE - ADD INDEX

## An index in MySQL is a database structure that improves the speed of data retrieval operations on a table by providing a fast and efficient way to look up rows based on the values in one or more columns. Think of an index like the index in a book—it allows you to quickly find information without having to read the entire book.

## Indexes are crucial for optimizing query performance, especially for tables with a large number of rows. They work by creating a data structure that stores a sorted list of values from the indexed column(s). When you query the table using the indexed column(s), MySQL can use the index to locate the desired rows much faster than scanning the entire table.

## Here's an example to illustrate the concept of indexes:

## Suppose you have a table called employees with the following structure:

## +----+-----------+-----------+--------+

## | id | first\_name| last\_name | salary |

## +----+-----------+-----------+--------+

## | 1 | John | Doe | 50000 |

## | 2 | Jane | Smith | 55000 |

## | 3 | Mark | Johnson | 48000 |

## | ...| ... | ... | ... |

## +----+-----------+-----------+--------+

## Now, let's say you frequently run queries to find employees by their last name. Without an index on the last\_name column, MySQL would have to scan the entire table row by row to find matching employees. This process can be slow and inefficient, especially for large datasets.

## To improve the performance of such queries, you can create an index on the last\_name column using the ALTER TABLE statement:

## ALTER TABLE customers

## ADD INDEX full\_name\_index (FULL\_NAME);

## In this SQL statement:

## ALTER TABLE employees:

## Specifies the table name (employees) that you want to modify.

## ADD INDEX last\_name\_index (last\_name): Adds an index called last\_name\_index to the last\_name column. This index will store a sorted list of values from the last\_name column, making it much faster to locate rows based on the last name.

## After creating this index, queries that search for employees by last name will be significantly faster because MySQL can use the index to quickly locate the relevant rows. For example:

## -- Query to find employees with the last name "Smith"

## SELECT \* FROM customers WHERE full\_name = 'Ramesh';

## SELECT \* FROM students WHERE last\_name = 'Williams';

## With the index in place, MySQL can use it to quickly identify the rows that match the condition, resulting in faster query execution.

## Indexes can be created on single columns or combinations of columns, and they are a critical part of database performance tuning. However, it's essential to balance the benefits of indexes with the trade-off of increased storage space and potential overhead in INSERT, UPDATE, and DELETE operations. You should create indexes strategically based on the types of queries you frequently run in your application.

## ALTER TABLE - DROP INDEX

You can drop an existing index from a table using the DROP INDEX statement along with the ALTER statement –

ALTER TABLE customers DROP INDEX full\_name\_index;

show index from customers;

### **Example**

Following query adds an index on the column NAME of CUSTOMERS table –

ALTER TABLE CUSTOMERS DROP INDEX name\_index;

ALTER TABLE STUDENTS DROP INDEX first\_name\_index;

### **Output**

The output will be displayed as −

Query OK, 0 rows affected (0.003 sec)

Records: 0 Duplicates: 0 Warnings: 0

## ALTER TABLE - ADD PRIMARY KEY

Following is the syntax to add a primary key in an existing table of a database –

ALTER TABLE table\_name ADD CONSTRAINT MyPrimaryKey PRIMARY KEY (column1, column2...);

### **Example**

Before we add a primary key to an existing table, first let's create a new table called EMPLOYEES as follows:

CREATE TABLE EMPLOYEES(

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR (25),

SALARY DECIMAL (18, 2)

);

Following query adds primary key constraint on the column ID of EMPLOYEES table –

ALTER TABLE EMPLOYEES ADD CONSTRAINT MyPrimaryKey PRIMARY KEY(ID);

This will produce the following output −

Query OK, 0 rows affected, 1 warning (0.003 sec)

Records: 0 Duplicates: 0 Warnings: 1

### **Verification**

To verify the above query if you describe the table using the DESC EMPLOYEES command −

DESC EMPLOYEES;

This will display the structure of the table created: column names, their respective data types, constraints (if any) etc.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Field** | **Type** | **Null** | **Key** | **Default** | **Extra** |
| ID | int(11) | NO | PRI | NULL |  |
| NAME | varchar(20) | NO |  | NULL |  |
| AGE | int(11) | NO |  | NULL |  |
| ADDRESS | char(25) | YES |  | NULL |  |
| SALARY | decimal(18,2) | YES |  | NULL |  |

## ALTER TABLE - DROP PRIMARY KEY

Following is the syntax to delete a primary key from an existing table of a database −

ALTER TABLE table\_name DROP PRIMARY KEY;

### **Example**

Following query deletes primary key constraint from the column ID of EMPLOYEES table −

ALTER TABLE EMPLOYEES DROP PRIMARY KEY;

This will produce the following output −

Query OK, 0 rows affected, 1 warning (0.003 sec)

Records: 0 Duplicates: 0 Warnings: 1

## ALTER TABLE - ADD CONSTRAINT

Following is the syntax to add a unique constraint to a column of an existing table −

ALTER TABLE table\_name ADD CONSTRAINT constraint\_name UNIQUE(column1, column2...);

### **Example**

Following query adds UNIQUE constraint to the table CUSTOMERS −

ALTER TABLE EMPLOYEES ADD CONSTRAINT CONST UNIQUE(NAME);

This will produce the following output −

Query OK, 0 rows affected (0.003 sec)

Records: 0 Duplicates: 0 Warnings: 0

## ALTER TABLE - DROP CONSTRAINT

Following is the syntax to drop a unique constraint from an existing table −

ALTER TABLE table\_name DROP CONSTRAINT constraint\_name;

### **Example**

Following query adds UNIQUE constraint to the table CUSTOMERS −

ALTER TABLE EMPLOYEES DROP CONSTRAINT CONST;

This will produce the following output −

Query OK, 0 rows affected (0.003 sec)

Records: 0 Duplicates: 0 Warnings: 0

In MySQL, you can use the **ALTER TABLE** statement with the **ADD CONSTRAINT** clause to add various types of constraints to an existing table. Constraints are rules that define the acceptable data values that can be stored in a table. They help ensure data integrity and consistency in the database. MySQL supports several types of constraints, including primary keys, foreign keys, unique constraints, and check constraints.

Here's a basic syntax for adding constraints to an existing table:

sqlCopy code

ALTER TABLE table\_name ADD CONSTRAINT constraint\_name constraint\_type (column\_name);

Let's break down the components of this syntax:

* **table\_name**: Replace this with the name of the table to which you want to add a constraint.
* **constraint\_name**: Specify a unique name for the constraint.
* **constraint\_type**: Choose the type of constraint you want to add (e.g., PRIMARY KEY, FOREIGN KEY, UNIQUE, CHECK).
* **column\_name**: Specify the column(s) to which the constraint applies.

Now, let's look at examples of different types of constraints you can add using the **ALTER TABLE** statement:

1. **PRIMARY KEY Constraint:**

A primary key constraint ensures that a column (or a combination of columns) contains unique values and cannot contain NULL values. It is often used to uniquely identify rows in a table.

sqlCopy code

ALTER TABLE employees ADD CONSTRAINT pk\_employee\_id PRIMARY KEY (employee\_id);

1. **FOREIGN KEY Constraint:**

A foreign key constraint establishes a relationship between two tables. It ensures that values in a column (usually in the child table) correspond to values in a primary key column (usually in the parent table).

sqlCopy code

ALTER TABLE orders ADD CONSTRAINT fk\_customer\_id FOREIGN KEY (customer\_id) REFERENCES customers (customer\_id);

1. **UNIQUE Constraint:**

A unique constraint ensures that values in a column (or combination of columns) are unique, but it allows NULL values.

sqlCopy code

ALTER TABLE products ADD CONSTRAINT uc\_product\_code UNIQUE (product\_code);

1. **CHECK Constraint:**

A check constraint enforces a condition on the values in a column. It ensures that only values that meet the specified condition are allowed.

sqlCopy code

ALTER TABLE employees ADD CONSTRAINT chk\_salary CHECK (salary >= 0);

These examples illustrate how to add various types of constraints to a table using the **ALTER TABLE** statement. Constraints play a vital role in maintaining data integrity and enforcing data rules in a database.

# SQL - Constraints

### **Table of Content**

* [SQL Constraints Introduction](https://www.tutorialspoint.com/sql/sql-constraints.htm#introduction)
* [SQL Create Constraints](https://www.tutorialspoint.com/sql/sql-constraints.htm#create_sql_constraints)
* [NOT NULL Constraint](https://www.tutorialspoint.com/sql/sql-constraints.htm#not_null_constraint)
* [UNIQUE Key Constraint](https://www.tutorialspoint.com/sql/sql-constraints.htm#unique_key_constraint)
* [DEFAULT Value Constraint](https://www.tutorialspoint.com/sql/sql-constraints.htm#default_value_constraint)
* [PRIMARY Key Constraint](https://www.tutorialspoint.com/sql/sql-constraints.htm#primary_key_constraint)
* [FOREIGN Key Constraint](https://www.tutorialspoint.com/sql/sql-constraints.htm#foreign_key_constraint)
* [CHECK Value Constraint](https://www.tutorialspoint.com/sql/sql-constraints.htm#check_value_constraint)
* [INDEX Constraint](https://www.tutorialspoint.com/sql/sql-constraints.htm#index_constraint)
* [Dropping SQL Constraints](https://www.tutorialspoint.com/sql/sql-constraints.htm#dropping_sql_constraints)
* [Data Integrity Constraints](https://www.tutorialspoint.com/sql/sql-constraints.htm#data_integrity_constraints)

## SQL Constraints

SQL Constraints are the rules applied to a data columns or the complete table to limit the type of data that can go into a table. When you try to perform any INSERT, UPDATE, or DELETE operation on the table, RDBMS will check whether that data violates any existing constraints and if there is any violation between the defined constraint and the data action, it aborts the operation and returns an error.

We can define a column level or a table level constraints. The column level constraints are applied only to one column, whereas the table level constraints are applied to the whole table.

## SQL Create Constraints

We can create constraints on a table at the time of a table creation using the CREATE TABLE statement, or after the table is created, we can use the ALTER TABLE statement to create or delete table constraints.

.

CREATE TABLE table\_name (

column1 datatype constraint,

column2 datatype constraint,

column3 datatype constraint,

....

);

Different RDBMS allows to define different constraints. This tutorial will discuss about 7 most important constraints available in MySQL.

## NOT NULL Constraint

When applied to a column, NOT NULL constraint ensure that a column cannot have a NULL value. Following is the example to create a NOT NULL constraint:

CREATE TABLE CUSTOMERS1 (

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR (25),

SALARY DECIMAL (18, 2)

);

Check further detail on [**NOT NULL Constraint**](https://www.tutorialspoint.com/sql/sql-not-null-constraint.htm)

## UNIQUE Key Constraint

When applied to a column, UNIQUE Key constraint ensure that a column accepts only UNIQUE values. Following is the example to create a UNIQUE Key constraint on column ID. Once this constraint is created, column ID can't be null and it will accept only UNIQUE values.

CREATE TABLE CUSTOMERS2 (

ID INT NOT NULL UNIQUE,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR (25),

SALARY DECIMAL (18, 2)

);

SELECT DISTINCT

TC.CONSTRAINT\_NAME,

KCU.COLUMN\_NAME,

TC.CONSTRAINT\_TYPE

FROM

INFORMATION\_SCHEMA.TABLE\_CONSTRAINTS TC

JOIN

INFORMATION\_SCHEMA.KEY\_COLUMN\_USAGE KCU

ON

TC.CONSTRAINT\_NAME = KCU.CONSTRAINT\_NAME

WHERE

KCU.TABLE\_NAME = 'customers2'

AND TC.CONSTRAINT\_SCHEMA = 'student';

* **TC**: Represents the **INFORMATION\_SCHEMA.TABLE\_CONSTRAINTS** table, which contains information about table constraints such as primary keys, unique constraints, etc.
* **KCU**: Represents the **INFORMATION\_SCHEMA.KEY\_COLUMN\_USAGE** table, which contains information about the columns that participate in table constraints.

## DEFAULT Value Constraint

When applied to a column, DEFAULT Value constraint provides a default value for a column when none is specified. Following is the example to create a DEFAULT constraint on column NAME. Once this constraint is created, column NAME will set to "Not Available" value if NAME is not set to a value.

CREATE TABLE CUSTOMERS1 (

ID INT NOT NULL UNIQUE,

NAME VARCHAR (20) DEFAULT 'Not Available',

AGE INT NOT NULL,

ADDRESS CHAR (25),

SALARY DECIMAL (18, 2)

);

Check further detail on [**Unique Key Constraint**](https://www.tutorialspoint.com/sql/sql-default-constraint.htm)

## PRIMARY Key Constraint

When applied to a column, PRIMARY Key constraint ensure that a column accepts only UNIQUE value and there can be a single PRIMARY Key on a table but multiple columns can constituet a PRIMARY Key. Following is the example to create a PRIMARY Key constraint on column ID. Once this constraint is created, column ID can't be null and it will accept only unique values.

CREATE TABLE CUSTOMERS3(

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR (25),

SALARY DECIMAL (18, 2),

PRIMARY KEY (ID)

);

Check further detail on [**PRIMARY Key Constraint**](https://www.tutorialspoint.com/sql/sql-primary-key.htm)

## FOREIGN Key Constraint

FOREIGN Key constraint maps with a column in another table and uniquely identifies a row/record in that table. Following is an example to create a foreign key constraint on column ID available in CUSTOMERS table as shown in the statement below −

CREATE TABLE Authors (

AuthorID INT PRIMARY KEY,

Name VARCHAR(100) NOT NULL

);

CREATE TABLE Books (

BookID INT PRIMARY KEY,

Title VARCHAR(100) NOT NULL,

AuthorID INT,

FOREIGN KEY (AuthorID) REFERENCES Authors(AuthorID)

);

INSERT INTO Authors (AuthorID, Name) VALUES (1, 'J.K. Rowling');

INSERT INTO Authors (AuthorID, Name) VALUES (2, 'George R.R. Martin');

INSERT INTO Books (BookID, Title, AuthorID) VALUES (1, 'Harry Potter', 1);

INSERT INTO Books (BookID, Title, AuthorID) VALUES (2, 'Game of Thrones', 2);

SELECT Books.Title, Authors.Name

FROM Books

JOIN Authors ON Books.AuthorID = Authors.AuthorID;

SELECT Title, (SELECT Name FROM Authors WHERE Authors.AuthorID = Books.AuthorID) AS AuthorName

FROM Books;

This query will return a list of books along with the names of their authors. The JOIN clause is used to combine rows from two or more tables, based on a related column between them, which in this case is AuthorID

Check further detail on [**FOREIGN Key Constraint**](https://www.tutorialspoint.com/sql/sql-foreign-key.htm)

## CHECK Value Constraint

When applied to a column, CHECK Value constraint works like a validation and it is used to check the validity of the data entered into the particular column of the table. table and uniquely identifies a row/record in that table. Following is an example to create a CHECK validation on AGE column which will not accept if its value is below to 18.

CREATE TABLE CUSTOMERS11(

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)

);

Check further detail on [**CHECK Value Constraint**](https://www.tutorialspoint.com/sql/sql-check-constraint.htm)

## INDEX Constraint

The INDEX constraints are created to speed up the data retrieval from the database. An Index can be created by using a single or group of columns in a table. A table can have a single PRIMARY Key but can have multiple INDEXES. An Index can be Unique or Non Unique based on requirements. Following is an example to create an Index on Age Column of the CUSTOMERS table.

CREATE INDEX idx\_age ON CUSTOMERS ( AGE );

Check further detail on [**INDEX Constraint**](https://www.tutorialspoint.com/sql/sql-index-constraint.htm)

## Dropping SQL Constraints

Any constraint that you have defined can be dropped using the **ALTER TABLE** command with the DROP CONSTRAINT option. For example, to drop the primary key constraint from the CUSTOMERS table, you can use the following command.

ALTER TABLE CUSTOMERS DROP CONSTRAINT PRIMARY KEY;

Some RDBMS allow you to disable constraints instead of permanently dropping them from the database, which you may want to temporarily disable the constraints and then enable them later.

## Data Integrity Constraints

Data integrity constraints are used to ensure the overall accuracy, completeness, and consistency of data. Now a days data integrity also refers to the data safety in regard to regulatory compliance — such as GDPR compliance etc.

Data integrity is handled in a relational database through the concept of referential integrity. There are many types of integrity constraints that play a role in **Referential Integrity (RI)**. These constraints include Primary Key, Foreign Key, Unique Constraints and other constraints which are mentioned above.

HOW TO CHECK WHICH CONSTRAINT IS APPLIED ON ANY PARTICULAR TABLE IN THE DATABASE:

SELECT \* FROM information\_schema.table\_constraints WHERE table\_schema = 'student' AND table\_name = 'customers';

# SQL - SELECT Query

## The SQL SELECT Statement

The SQL **SELECT** Statement is used to fetch the data from a database table which returns this data in the form of a table. These tables are called result-sets.

*CLAUSES and OPERATORS available in SQL can be used with the SELECT statement in order to retrieve the filtered records of a database table.*

### **Syntax**

The basic syntax of the SELECT Query is as follows −

SELECT column1, column2, columnN FROM table\_name;

Here, column1, column2... are the fields of a table whose values you want to fetch. If you want to fetch all the columns available in a table, then you can use the following syntax −

SELECT \* FROM table\_name;

### **Example**

Assume we have created a table named **CUSTOMERS** using the CREATE TABLE statement as shown 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)

);

Now, insert values into this table using the INSERT statement as follows −

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, 'Hyderabad', 4500.00 ),

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

The table will be created as −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 | Hyderabad | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |

### **Retrieving Selected Fields/Columns**

The following statement fetches the ID, Name and Salary fields of the records available in CUSTOMERS table.

SELECT ID, NAME, SALARY FROM CUSTOMERS;

### **Output**

The above query would produce the following table −

|  |  |  |
| --- | --- | --- |
| **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 |

### **Retrieving All Fields/Columns**

If you want to fetch all the fields of the CUSTOMERS table, then you should use the query of SELECT statement with an **Asterisk (\*)** instead of the column names, as shown below −

SELECT \* FROM CUSTOMERS;

### **Output**

The resultant table will be &minus

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 | Hyderabad | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |

## Computing using SELECT

The SQL SELECT statement can also be used to retrieve the results of various mathematical computations in the form of a table. In such cases, you do not need to specify any database table in the statement.

Following is the syntax to do so −

SELECT mathematical\_expression;

### **Example**

Following is an example which multiply two given numbers using SQL statement.

SELECT 56\*65;

The query above produces the following output −

|  |
| --- |
| **56\*65** |
| 3640 |

## Aliasing a Column in SELECT Statement

Whenever a column name in a table is too difficult to read and understand, SQL provides a method to alias this column name into another understandable and relative name. This is done using the **AS keyword**. You can use the AS keyword in a SELECT statement to display the column names of a table as an alias name.

Following is the syntax to do so −

SELECT column\_name AS alias\_name FROM table\_name;

You can also use an alias to display SELECT expressions with the same syntax; you should use a mathematical statement instead of *column\_name*.

### **Example**

In the example below, we are trying to retrieve customer details NAME and AGE in a single column of the resultant table using the **concat()** expression and aliasing the column as DETAILS along with the customer addresses from the CUSTOMERS table. This will be done using SELECT statement in the following query −

SELECT CONCAT(NAME,' ',AGE) AS DETAILS, ADDRESS FROM CUSTOMERS ORDER BY NAME;

The query above produces the following output −

|  |  |
| --- | --- |
| **DETAILS** | **ADDRESS** |
|  |  |
| Chaitali 25 | Mumbai |
| Hardik 27 | Bhopal |
| kaushik 23 | Kota |
| Khilan 25 | Delhi |
| Komal 22 | Hyderabad |
| Muffy 24 | Indore |
| Ramesh 32 | Ahmedabad |

# SQL - Select Into Statement

## The SQL Select Into Statement

The **SQL SELECT INTO** Statement creates a new table and inserts data from an existing table into the newly created table. The new table is automatically created based on the structure of the columns in the SELECT statement and can be created in the same database or in a different database.

However, it's important to note that the SELECT INTO statement does not preserve any indexes, constraints, or other properties of the original table, and the new table will not have any primary keys or foreign keys defined by default. Therefore, you may need to add these properties to the new table manually if necessary.

*MySQL doesn't support the SELECT ... INTO TABLE Sybase SQL extension i.e. in MySQL you cannot use the****SELECT ... INTO****statement to insert data from one table to another. Instead of this, we can use****INSERT INTO ... SELECT****statement or,****CREATE TABLE ... SELECT****.*

### **Syntax**

Following is the basic syntax of the SQL SELECT INTO statement in SQL Server −

SELECT \* INTO new\_table\_name FROM existing\_table\_name

### **Example**

Let us create the CUSTOMERS table which contains the personal details of customers including their name, age, address and salary etc. as shown below −

CREATE TABLE CUSTOMERS\_BACKUP (

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR (25),

SALARY DECIMAL (18, 2),

PRIMARY KEY (ID)

);

Now, insert values into this table using the INSERT statement as follows −

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, 'Hyderabad', 4500.00 ),

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

The CUSTOMERS table will be creates 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 | Hyderabad | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |

The following SELECT INTO statement creates a new table called **CUSTOMER\_BACKUP** and copies the data from the CUSTOMERS table into it −

SELECT \* INTO CUSTOMERS\_BACKUP FROM CUSTOMERS;

CREATE TABLE CUSTOMERS\_BACKUP AS

SELECT \* FROM CUSTOMERS;

### **Output**

We get the following result. We can observe that 7 rows have been modified.

(7 rows affected)

### **Verification**

We can verify whether the changes are reflected in a table by retrieving its contents using the SELECT statement. Following is the query to display the records in the CUSTOMER\_BACKUP table −

SELECT \* from CUSTOMER\_BACKUP;

The table displayed 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 | Hyderabad | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |

## Copying Data From Specific Columns

We can also copy data from specific columns from an existing table into the new table using the SQL SELECT INTO statement. To do so, we just need to include the required column names after the select keyword.

### **Syntax**

Following is the syntax −

SELECT column1, column2, ..., columnN

INTO new\_table\_name

FROM existing\_table\_name;

### **Example**

In the following query, we are creating a new table called **CUSTOMER\_DETAILS** with only the NAME, AGE, and ADDRESS columns from the CUSTOMERS table, and populate it with the corresponding data.

SELECT name, age, address INTO CUSTOMER\_DETAILS FROM CUSTOMERS;

CREATE TABLE CUSTOMER\_DETAILS

SELECT NAME, AGE, ADDRESS FROM CUSTOMERS;

### **Output**

We get the following result. We can observe that 7 rows have been modified.

(7 rows affected)

### **Verification**

We can verify whether the changes are reflected in a table by retrieving its contents using the SELECT statement. Following is the query to display the records in the CUSTOMER\_DETAILS table −

SELECT \* from CUSTOMER\_DETAILS;

The table displayed is as follows −

|  |  |  |
| --- | --- | --- |
| **NAME** | **AGE** | **ADDRESS** |
| Ramesh | 32 | Ahmedabad |
| Khilan | 25 | Delhi |
| Kaushik | 23 | Kota |
| Chaitali | 25 | Mumbai |
| Hardik | 27 | Bhopal |
| Komal | 22 | Hyderabad |
| Muffy | 24 | Indore |

**Note:** The new table will not include any other columns from the original table. Also the original table remains unchanged.

## Fetching Data From Multiple Tables

Using the SQL SELECT INTO statement we can also copy data from multiple tables to a new table. This is accomplished using the JOIN clause which combines the data from multiple tables (based on a common column).

### **Syntax**

Following is the syntax to copy data from multiple tables using the SELECT INTO statement −

SELECT column1, column2, ..., columnN

INTO new\_table\_name

FROM table1

JOIN table2 ON table1.column = table2.column

### **Example**

First of all, let us create another table named **ORDERS** −

CREATE TABLE ORDERS (

OID INT NOT NULL,

DATE VARCHAR (20) NOT NULL,

CUSTOMER\_ID INT NOT NULL,

AMOUNT DECIMAL (18, 2));

Using the INSERT statement, insert values into this table as follows −

INSERT INTO ORDERS (OID, DATE, CUSTOMER\_ID, AMOUNT) VALUES

(102, '2009-10-08 00:00:00', 3, 3000.00),

(100, '2009-10-08 00:00:00', 3, 1500.00),

(101, '2009-11-20 00:00:00', 2, 1560.00),

(103, '2008-05-20 00:00:00', 4, 2060.00);

The table is created as −

|  |  |  |  |
| --- | --- | --- | --- |
| **OID** | **DATE** | **CUSTOMER\_ID** | **AMOUNT** |
| 102 | 2009-10-08 00:00:00 | 3 | 3000.00 |
| 100 | 2009-10-08 00:00:00 | 3 | 1500.00 |
| 101 | 2009-11-20 00:00:00 | 2 | 1560.00 |
| 103 | 2008-05-20 00:00:00 | 4 | 2060.00 |

Now, we are creating a new table called CUSTOMER\_ORDERS that includes the customer name from the CUSTOMERS table and the customer id from the ORDERS table, where the id of customers from the CUSTOMERS table matches with the id of customers from the ORDERS table −

SELECT CUSTOMERS.Name, ORDERS.customer\_id

INTO CUSTOMER\_ORDERS

FROM CUSTOMERS

LEFT JOIN ORDERS ON CUSTOMERS.ID = ORDERS.customer\_id;

### **Output**

We get the following result. We can observe that 8 rows have been modified.

(8 rows affected)

### **Verification**

We can verify whether the changes are reflected in a table by retrieving its contents using the SELECT statement. Following is the query to display the records in the CUSTOMER\_ORDERS table −

SELECT \* FROM CUSTOMER\_ORDERS;

The table displayed is as follows −

|  |  |
| --- | --- |
| **NAME** | **customer\_id** |
| Ramesh | NULL |
| Khilan | 2 |
| Kaushik | 3 |
| Kaushik | 3 |
| Chailtali | 4 |
| Hardik | NULL |
| Komal | NULL |
| Muffy | NULL |

## Copying Specific Records

We can also use the SQL SELECT INTO statement with a WHERE clause to create a new table and copy specific rows from an existing table into it.

### **Syntax**

Following is the syntax for using SELECT INTO statement with a WHERE clause −

SELECT \*

INTO new\_table\_name

FROM existing\_table\_name

WHERE condition;

### **Example**

Using the following query we are creating a new table called **NameStartsWith\_K** that includes all columns from the CUSTOMERS table, but it only stores the records of the customers whose name starts with **‘k’**.

SELECT \*

INTO NameStartsWith\_K

FROM CUSTOMERS

WHERE NAME LIKE 'k%';

### **Output**

We get the following result. We can observe that 3 rows have been modified.

(3 rows affected)

### **Verification**

We can verify whether the changes are reflected in a table by retrieving its contents using the SELECT statement.

SELECT \* from NameStartsWith\_K;

The table displayed is as follows −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | Kaushik | 23 | Kota | 2000.00 |
| 6 | Komal | 22 | Hyderabad | 4500.00 |

# SQL - Insert Into... Select Statement

## The Insert Into... Select Statement

The SQL **INSERT INTO... SELECT** statement is used to add/insert one or more new rows from an existing table to another table. This statement is a combination of two different statements: INSERT INTO and SELECT.

* The INSERT INTO statement is one of the most fundamental and frequently used statements in database management and requires only the name of the table and the values to be inserted. However, it is important to ensure that the data being inserted satisfies the constraints if the columns of a table (if any) and its type matches the data types of the table columns.
* The SELECT statement is used to retrieve data from an existing database table.

When these statements are used together, the SELECT statement first retrieves the data from an existing table and the INSERT INTO statement inserts the retrieved data into another table (if they have same table structures).

### **Syntax**

Following is the syntax of the SQL INSERT INTO... SELECT statement −

INSERT INTO table\_new SELECT (column1, column2, ...columnN) FROM table\_old;

Before using this query, we have to make sure that −

* In the database where we are going to insert data, source and target tables already exist.
* The structure of the source and target tables are same.

### **Example**

Assume we have created a table named **CUSTOMERS** which contains the personal details of customers including their name, age, address and salary etc.., as shown 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)

);

Now, insert values into this table using the INSERT statement as follows −

INSERT INTO CUSTOMERS 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, 'Hyderabad', 4500.00),

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

The table will be created as −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 | Hyderabad | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |

Create another table named **BUYERS** with same structure as the CUSTOMERS table.

CREATE TABLE BUYERS (

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR (25),

SALARY DECIMAL (18, 2),

PRIMARY KEY (ID)

);

Following query copies all the records from the **CUSTOMERS** table to **BUYERS** −

INSERT INTO BUYERS SELECT \* FROM CUSTOMERS;

### **Verification**

If you verify the contents of the BUYERS table using the SELECT statement as −

SELECT \* FROM BUYERS;

The table will be created as −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 | Hyderabad | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |

## SQL - Inserting Specific Records

Sometimes we only need to add a small number of records to another table. This can be accomplished by using a WHERE clause along with the SQL INSERT INTO... SELECT statement.

## Example

Let us create a table named **NAMESTARTSWITH\_K** with the same structure as the CUSTOMER table using the CREATE statement as −

CREATE TABLE NAMESTARTSWITH\_K (

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR (25),

SALARY DECIMAL (18, 2),

PRIMARY KEY (ID)

);

Following query, inserts the records of the customers whose name starts with the letter **k** from the CUSTOMERS table to the BUYERS table −

INSERT INTO NAMESTARTSWITH\_K

SELECT \* FROM CUSTOMERS

WHERE NAME LIKE 'k%';

### **Verification**

Following is the SELECT statement to verify the contents of the above created table −

SELECT \* FROM NAMESTARTSWITH\_K;

The table will be created as −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | Kaushik | 23 | Kota | 2000.00 |
| 6 | Komal | 22 | Hyderabad | 4500.00 |

## SQL - Inserting Top N Rows

The LIMIT clause filters the number of rows from the query. You can use this to filter the top N records that should be added to the target table.

### **Example**

But, before proceeding further, let us truncate all rows in the BUYERS table using the following statement −

TRUNCATE TABLE BUYERS;

Following query inserts the top 3 records from the CUSTOMERS table to the BUYERS table −

INSERT INTO BUYERS SELECT \* FROM CUSTOMERS ORDER BY ID ASC LIMIT 3;

### **Verification**

Let us verify the contents of the BUYERS table −

SELECT \* FROM BUYERS;

The resultant table will be 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 |

# SQL - UPDATE Query

## The SQL UPDATE Statement

The SQL **UPDATE** Statement is used to modify the existing records in a table. This statement is a part of Data Manipulation Language (DML), as it only modifies the data present in a table without affecting the table's structure.

To filter records that needs to be modified, you can use a WHERE clause with UPDATE statement. Using a WHERE clause, you can either update a single row or multiple rows.

*Since it only interacts with the data of a table, the SQL UPDATE statement needs to used cautiously. If the rows to be modified aren't selected properly, all the rows in the table will be affected and the correct table data is either lost or needs to be reinserted.*

The SQL UPDATE statement makes use of locks on each row while modifying them in a table, and once the row is modified, the lock is released. Therefore, it can either make changes to a single row or multiple rows with a single query.

### **Syntax**

The basic syntax of the SQL UPDATE statement 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**

Assume we have created a table named **CUSTOMERS** using the CREATE TABLE statement as shown 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)

);

Now, insert values into this table using the INSERT statement as follows −

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, 'Hyderabad', 4500.00 ),

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

The table will be created as −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 | Hyderabad | 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.

UPDATE CUSTOMERS SET ADDRESS = 'Pune' WHERE ID = 6;

### **Output**

The query produces the following output −

Query OK, 1 row affected (0.13 sec)

Rows matched: 1 Changed: 1 Warnings: 0

### **Verification**

To verify whether the records of the table are modified or not, use the following SELECT query below −

SELECT \* FROM CUSTOMERS WHERE ID=6;

Now, the CUSTOMERS table would have the following records −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
|  |  |  |  |  |
| 6 | Komal | 22 | Pune | 4500.00 |

## Update Multiple ROWS and COLUMNS

Using SQL UPDATE statement, multiple rows and columns in a table can also be updated. To update multiple rows, specify the condition in a WHERE clause such that only the required rows would satisfy it.

However, to update multiple columns, set the new values to all the columns that need to be updated. In this case, using the WHERE clause would narrow down the records of the table and not using the clause would change all the values in these columns.

### **Syntax**

Following is the syntax to update multiple rows and columns −

UPDATE table\_name

SET column\_name1 = new\_value, column\_name2 = new\_value...

WHERE condition(s)

### **Example**

If you want to modify all the AGE 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. Following query increases the age of all the customers by 5 years and adds 3000 to all the salary values −

UPDATE CUSTOMERS SET AGE = AGE+5, SALARY = SALARY+3000;

### **Output**

The query produces the following output −

Query OK, 7 rows affected (0.12 sec)

Rows matched: 7 Changed: 7 Warnings: 0

### **Verification**

To verify whether the records of the table are modified or not, use the following SELECT query below −

SELECT \* FROM CUSTOMERS;

Now, CUSTOMERS table would have the following records −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | Ramesh | 37 | Ahmedabad | 5000.00 |
| 2 | Khilan | 30 | Delhi | 4500.00 |
| 3 | kaushik | 28 | Kota | 5000.00 |
| 4 | Chaitali | 30 | Mumbai | 9500.00 |
| 5 | Hardik | 32 | Bhopal | 11500.00 |
| 6 | Komal | 27 | Pune | 7500.00 |
| 7 | Muffy | 29 | Indore | 13000.00 |

### **Example**

But, if you want to modify the ADDRESS and the SALARY columns of selected records in the CUSTOMERS table, you need to specify a condition to filter the records to be modified, using the WHERE clause, as shown in the following query −

UPDATE CUSTOMERS SET ADDRESS = 'Goa', SALARY = 10000.00 WHERE NAME = 'Ramesh';

### **Output**

This query produces the following output −

Query OK, 1 row affected (0.04 sec)

Rows matched: 1 Changed: 1 Warnings: 0

### **Verification**

To verify whether the records of the table are modified or not, use the following SELECT query below −

SELECT \* FROM CUSTOMERS WHERE NAME = 'Ramesh';

Now, CUSTOMERS table would have the following records −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | Ramesh | 37 | Pune | 1000.00 |

# SQL - SORTING Results

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

In addition to that, ORDER BY clause can also sort the data in a database table in a preferred order. This case may not sort the records of a table in any standard order (like alphabetical or lexicographical), but, they could be sorted based on any external condition. For instance, in an ORDERS table containing the list of orders made by various customers of an organization, the details of orders placed can be sorted based on the dates on which those orders are made. This need not be alphabetically sorted, instead, it is based on "first come first serve".

### **Syntax**

The basic syntax of the ORDER BY clause which would be used to sort the result in an ascending or descending order 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 that whatever column you are using to sort, that column should be in the column-list.

## Sorting Results in Ascending Order

Using Order By Clause in SQL, the records in a database table can be sorted in ascending order, either by default or by specifying the "ASC" keyword in the clause condition. Let us see an example to understand this.

### **Example**

Assume we have created a table named **CUSTOMERS** using the CREATE TABLE statement as shown 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)

);

Now, insert values into this table using the INSERT statement as follows −

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, 'Hyderabad', 4500.00 ),

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

The table will be created as −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 | Hyderabad | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |

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

SELECT \* FROM CUSTOMERS ORDER BY NAME;

### **Output**

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 | Hyderabad | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

## Sorting Results in Descending Order

But, to sort the records in a database table in descending order, we need to specify the "DESC" keyword in the clause condition. Let us see an example to understand this.

### **Example**

The following query sorts the records of the CUSTOMERS tables in descending order based on the column **NAME**.

SELECT \* FROM CUSTOMERS ORDER BY NAME DESC;

### **Output**

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 | Hyderabad | 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 |

## Sorting Results in a Preferred Order

One can also sort the records of a table in their own preferred order using the CASE statement within the ORDER BY clause. All the values are specified in the clause along with the position they are supposed to be sorted in; if the values are not given any number, they are automatically sorted in ascending order.

### **Example**

To fetch the rows with their own preferred order, the SELECT query used would be as follows −

SELECT \* FROM CUSTOMERS

ORDER BY (CASE ADDRESS

WHEN 'DELHI' THEN 1

WHEN 'BHOPAL' THEN 2

WHEN 'KOTA' THEN 3

WHEN 'AHMEDABAD' THEN 4

WHEN 'Hyderabad' THEN 5

ELSE 100 END) ASC, ADDRESS DESC;

### **Output**

In this SQL query, the CASE statement is being used to order the results based on the ADDRESS column. Each city (Delhi, Bhopal, Kota, Ahmedabad, Hyderabad) is assigned a specific order number (1 through 5).

The ELSE 100 part is a catch-all for any addresses that are not one of the five specified cities. If the address is not ‘Delhi’, ‘Bhopal’, ‘Kota’, ‘Ahmedabad’, or ‘Hyderabad’, it is assigned an order number of 100.

This means that in the ordered list of results, addresses from Delhi will appear first, followed by addresses from Bhopal, Kota, Ahmedabad, and Hyderabad. Addresses from any other city will appear last because they have been assigned the highest order number (100). The ASC keyword means that results are ordered in ascending order.

Finally, ADDRESS DESC means that within each group of cities, the addresses are sorted in descending order. So for example, if there are multiple customers from Delhi, their addresses will be listed in reverse alphabetical order.

This would produce the following result −

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

This will sort the customers by ADDRESS in your **own order** of preference first, and in a natural order for the remaining addresses. Also, the remaining Addresses will be sorted in the reverse alphabetical order.

**SQL Operators and Clauses**

# SQL - WHERE Clause

## The SQL Where Clause

The SQL **WHERE** clause is used to filter the results obtained by the DML statements such as SELECT, UPDATE and DELETE etc. We can retrieve the data from a single table or multiple tables(after join operation) using the WHERE clause.

For instance, you can use the WHERE clause to retrieve details of employees of a department in an organization, or employees earning salary above/below certain amount, or details of students eligible for scholarships etc. This clause basically provides the specification of which records to be retrieved and which are to be to be neglected.

### **Syntax**

The basic syntax of the SQL WHERE clause is as shown below −

DML\_Statement column1, column2,... columnN

FROM table\_name

WHERE [condition];

Here, the DML\_Statement can be any statement, such as SELECT, UPDATE, DELETE etc.

You can specify a condition using the [comparison or logical operators](https://www.tutorialspoint.com/sql/sql-operators.htm) such as, >, <, =, **LIKE, NOT**, etc.

## WHERE Clause with SELECT Statement

Typically, the SELECT statement is used to retrieve data from a table. If we use the WHERE clause with the SELECT statement, we can filter the rows to be retrieved based on a specific condition (or expression). Following is the syntax for it −

SELECT column1, column2, ...

FROM table\_name

WHERE condition;

### **Example**

Assume we have created a table named CUSTOMERS in MySQL database using CREATE TABLE statement as shown 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)

);

Following INSERT query inserts 7 records into this table −

INSERT INTO CUSTOMERS 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, 'Hyderabad', 4500.00),

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

The table created is 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 | Hyderabad | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |

In the following query, we are fetching the ID, NAME and SALARY fields from the CUSTOMERS table for the records where the SALARY is greater than 2000 −

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

### **Output**

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 |

## WHERE Clause with UPDATE Statement

The UPDATE statement is used to modify the existing records in a table. Using the SQL WHERE clause with the UPDATE statement, we can update particular records. If the WHERE clause is not used, the UPDATE statement would affect all the records of a table. Following is the syntax −

UPDATE table\_name

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

WHERE condition;

### **Example**

In the following query, we are incrementing the salary of the customer named Ramesh by 10000 by using the WHERE clause along with the UPDATE statement −

UPDATE CUSTOMERS set SALARY = SALARY+10000 where NAME = 'Ramesh';

### **Output**

We get the following result. We can observe that the age of 2 customers have been modified −

Query OK, 2 rows affected (0.02 sec)

Rows matched: 2 Changed: 2 Warnings: 0

### **Verification**

To verify if the changes are reflected in the table, we can use SELECT statement as shown in the following query −

SELECT \* FROM CUSTOMERS WHERE NAME = 'Ramesh';

The table is displayed as follows −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | Ramesh | 32 | Ahmedabad | 12000.00 |

## WHERE Clause with IN Operator

Using the IN operator you can specify the list of values or sub query in the where clause. If you use WHERE and IN with the SELECT statement, it allows us to retrieve the rows in a table that match any of the values in the specified list. Following is the syntax for it −

WHERE column\_name IN (value1, value2, ...);

Where, the **column\_name** is the column of a table and **value1, value2,** etc. are the list of values that we want to compare with the column\_name.

### **Example**

Suppose you want to display records with NAME values ‘Khilan’, ‘Hardik’ and ‘Muffy’ from the CUSTOMERS table, you can use the following query −

SELECT \* from CUSTOMERS WHERE NAME IN ('Khilan', 'Hardik', 'Muffy');

### **Output**

The result obtained is as follows −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |

## WHERE Clause with NOT IN Operator

The WHERE clause with NOT IN operator is the negation of WHERE clause with the IN operator.

* If you use WHERE with the IN operator, the DML statement will act on the the list of values (of a column) specified
* Whereas, if you use WHERE with the NOT IN operator, the DML operation is performed on the values (of a column) that are not there in the specified list.

Hence, if you use WHERE Clause with NOT IN Operator along with the SELECT statement, the rows that do not match the list of values are retrieved. Following is the syntax −

WHERE column\_name NOT IN (value1, value2, ...);

### **Example**

In this example, we are displaying the records from CUSTOMERS table, where AGE is NOT equal to ‘25’, ‘23’ and ‘22’.

SELECT \* from CUSTOMERS WHERE AGE NOT IN (25, 23, 22);

### **Output**

We obtain the result as given below −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | Ramesh | 32 | Ahmedabad | 12000.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |

## WHERE Clause with LIKE Operator

The WHERE clause with LIKE operator allows us to filter rows that matches a specific pattern. This specific pattern is represented by wildcards (such as %, \_, [] etc). Following is the syntax −

WHERE column\_name LIKE pattern;

Where, **column\_name** is the column that we want to compare the pattern against and **pattern** is a string that can contain wildcards (such as %, \_, [] etc).

### **Example**

Following is the query which would display all the records where the name starts with K and is at least 4 characters in length −

SELECT \* FROM CUSTOMERS WHERE NAME LIKE 'K\_\_\_%';

### **Output**

The result obtained is given below −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | Kaushik | 23 | Kota | 2000.00 |
| 6 | Komal | 22 | Hyderabad | 4500.00 |

## WHERE Clause with AND, OR Operators

We can use AND and OR operators together in SQL to combine multiple conditions in a WHERE clause to filter rows that meets the specified criteria. The AND operator will make sure only those rows are filtered that satisfy all the conditions and the OR operator will filter records that satisfy any one of the specified conditions. However, this is only used when specifying one condition is not enough to filter all the required rows.

Following is the syntax for using the AND and OR operators in a WHERE clause −

WHERE (condition1 OR condition2) AND condition3;

### **Example**

In the following query, we are retrieving all rows from the CUSTOMERS table based on some conditions. The parentheses control the order of evaluation so that the OR operator is applied first, followed by the AND operator −

SELECT \* FROM CUSTOMERS

WHERE (AGE = 25 OR salary < 4500) AND (name = 'Komal' OR name = 'Kaushik');

### **Output**

This would produce the following result −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 3 | Kaushik | 23 | Kota | 2000.00 |

# SQL - TOP Clause

## The SQL TOP Clause

While we are retrieving data from an SQL table, the SQL **TOP clause** is used to restrict the number of rows returned by a **SELECT query in SQL server**. In addition, we can also use it with UPDATE and DELETE statements to limit (restrict) the resultant records.

For instance, if you have a large number of data stored in a database table, and you only want to perform operations on first N rows, you can use the TOP clause in your SQL server query.

*MySQL database does not support TOP clause instead of this, we can use the*[*LIMIT*](https://www.tutorialspoint.com/sql/sql-limit-clause.htm)*clause to select a limited number of records from a MySQL table. Similarly, Oracle supports the ROWNUM clause to restrict the records of a table. The TOP clause is similar to the LIMIT clause.*

### **Syntax**

The basic syntax of the SQL TOP clause is as follows −

SELECT TOP value column\_name(s)

FROM table\_name

WHERE [condition]

Where, **value** is the number/ percentage of number of rows to return from the top.

### **Example**

To understand it better let us consider the CUSTOMERS table which contains the personal details of customers including their name, age, address and salary etc. as shown 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)

);

Now, insert values into this table using the INSERT statement as follows −

INSERT INTO CUSTOMERS 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, 'Hyderabad', 4500.00 ),

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

The table will be created 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 | Hyderabad | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |

Now, we are using the TOP clause to fetch the top 4 records from the CUSTOMERS table without specifying any conditional clauses such as WHERE, ORDER BY, etc. −

SELECT TOP 4 \* FROM CUSTOMERS;

### **SELECT \* FROM CUSTOMERS**

### **LIMIT 4;**

### **Output**

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 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |

## TOP with ORDER BY Clause

The ORDER BY clause in SQL is used to sort the result-set of a query in ascending or descending order. We can use it along with the TOP clause to retrieve (or, update or, delete etc.) first **N** records in sorted order.

### **Example**

Using the following query we are retrieving the top **4** records of the **CUSTOMERS** table in a sorted order. Here, we are sorting the table in descending order based on the **SALARY** column −

SELECT TOP 4 \* FROM CUSTOMERS ORDER BY SALARY DESC;

**SELECT \* FROM CUSTOMERS**

**ORDER BY SALARY DESC**

**LIMIT 4;**

### **Output**

We obtain the result as follows −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 7 | Muffy | 24 | Indore | 10000.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 6 | Komal | 22 | Hyderabad | 4500.00 |

**Note −** By default, the ORDER BY clause sorts the data in ascending order. So, if we need to sort the data in descending order, we must use the DESC keyword.

## TOP Clause with PERCENT

We can also restrict the records by specifying percentage value instead of number, using the PERCENT clause along with the TOP clause.

### **Example**

The following query selects the first 40% of the records from the CUSTOMERS table sorted in the ascending order by their SALARY −

SELECT TOP 40 PERCENT \* FROM CUSTOMERS ORDER BY SALARY

### **SELECT \***

### **FROM Customers**

### **WHERE SALARY > (**

### **SELECT 0.4 \* MAX(SALARY)**

### **FROM Customers**

### **)**

### **ORDER BY SALARY DESC;**

### **Output**

We have the total of 7 records in our table. So 40% of 7 is 2.8. Therefore, SQL server rounds the result to three rows (the next whole number) as shown in the output below −

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

## TOP with WHERE Clause

We can use the TOP clause with the WHERE clause to limit the given number of rows and filter them based on a specified condition.

### **Example**

Following is the query to show the details of the first two customers whose name starts with ‘K’ from the CUSTOMERS table −

SELECT TOP 2 \* FROM CUSTOMERS WHERE NAME LIKE 'k%'

### **SELECT \***

### **FROM CUSTOMERS**

### **WHERE NAME LIKE 'K%'**

### **LIMIT 2;**

### **Output**

Following result is produced −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | Kaushik | 23 | Kota | 2000.00 |

## TOP Clause With DELETE Statement

The TOP clause can be used with the DELETE statement to delete a specific number of rows that meet the given criteria.

### **Example**

In the following query, we are using DELETE statement with TOP clause. Here, we are deleting the top 2 customers whose NAME starts with ‘K’ −

DELETE TOP(2) FROM CUSTOMERS WHERE NAME LIKE 'K%';

### **DELETE FROM CUSTOMERS**

### **WHERE NAME LIKE 'K%'**

### **LIMIT 2;**

### **Output**

We get the output as shown below −

(2 rows affected)

### **Verification**

We can verify whether the changes are reflected in a table by retrieving its contents using the SELECT statement as shown below −

SELECT \* FROM CUSTOMERS;

The table is displayed as follows −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 6 | Komal | 22 | Hyderabad | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |

## TOP and WITH TIES Clause

While sorting the data in a table using the ORDER BY clause based on a column, some times multiple rows may contain same values in the column(s) specified in the ORDER BY clause.

If you try to restrict the number of records using the TOP clause, all the eligible columns may not be filtered.

The **WITH TIES** clause is used to ensure that the records having the same values (records with **"tied"** values) are included in the query results.

### **Example**

Consider the above created table **CUSTOMERS**. If we need to retrieve the top 2 customers sorted by the ascending order of their SALARY values, the query would be −

SELECT TOP 2 \* FROM CUSTOMERS ORDER BY SALARY;

The resultant table would be −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | Kaushik | 23 | Kota | 2000.00 |

**SELECT \***

**FROM CUSTOMERS**

**ORDER BY SALARY**

**LIMIT 2;**

But, the first two salary values (in ascending order) in the table are 1500 and 2000 and there is another column in the CUSTOMERS table with salary value 2000 which is not included in the result.

If you want to retrieve all the columns with first two salary values (when arranged in the ascending order). We need to use the **WITH TIES** clause as shown below −

SELECT TOP 2 WITH TIES \* FROM CUSTOMERS ORDER BY SALARY;

In SQL, LIMIT and OFFSET are used to set the number of records to return and where to start returning records from a result set, respectively.

In your query, LIMIT 1 means that only one record should be returned from the subquery.

The OFFSET 1 clause means to skip the first record in the result set. So in this case, the subquery is returning the second lowest salary from the CUSTOMERS table.

The outer query then returns all records from the CUSTOMERS table where the salary is less than or equal to this second lowest salary, ordered by salary. So essentially, this query is finding all customers whose salaries are less than or equal to the second lowest salary in the table.

SELECT \*

FROM CUSTOMERS

WHERE SALARY <= (

SELECT DISTINCT SALARY

FROM CUSTOMERS

ORDER BY SALARY

LIMIT 1 OFFSET 1

)

ORDER BY SALARY;

### **Output**

The resultant table would be −

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

**For finding top 3 salaries:**

SELECT DISTINCT SALARY

FROM CUSTOMERS

ORDER BY SALARY DESC

LIMIT 3;

# SQL - DISTINCT Keyword

## The SQL DISTINCT Keyword

The SQL **DISTINCT** keyword is used in conjunction with the SELECT statement to fetch unique records from a table.

We use DISTINCT keyword with the SELECT statement when there is a need to avoid duplicate values present in any specific columns/tables. When we use DISTINCT keyword, SELECT statement returns only the unique records available in the table.

*The SQL DISTINCT Keyword can be associated with SELECT statement to fetch unique records from single or multiple columns/tables.*

### **Syntax**

The basic syntax of SQL DISTINCT keyword is as follows −

SELECT DISTINCT column1, column2,.....columnN FROM table\_name;

Where, **column1, column2,** etc. are the columns we want to retrieve the unique or distinct values from; and **table\_name** represents the name of the table containing the data.

## DISTINCT Keyword on Single Columns

We can use the DISTINCT keyword on a single column to retrieve all unique values in that column, i.e. with duplicates removed. This is often used to get a summary of the distinct values in a particular column or to eliminate redundant data.

### **Example**

Assume we have created a table with name CUSTOMERS in MySQL database using CREATE TABLE statement as shown 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)

);

Following query inserts values into this table using the INSERT statement −

INSERT INTO CUSTOMERS 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, 'Hyderabad', 4500.00),

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

The table obtained is 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 | Hyderabad | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |

First, let us retrieve the SALARY values from the CUSTOMERS table using the SELECT query −

SELECT SALARY FROM CUSTOMERS ORDER BY SALARY;

This would produce the following result. Here, you can observe that the salary value **2000** is appearing twice −

|  |
| --- |
| **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 −

SELECT DISTINCT SALARY FROM CUSTOMERS ORDER BY SALARY;

### **Output**

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 |

## DISTINCT Keyword on Multiple Columns

We can also use the DISTINCT keyword on multiple columns to retrieve all unique combinations of values across those columns. This is often used to get a summary of distinct values in multiple columns, or to eliminate redundant data.

### **Example**

In the following query, we are retrieving a list of all unique combinations of customer's age and salary using the DISTINCT keyword −

SELECT DISTINCT AGE, SALARY FROM CUSTOMERS ORDER BY AGE;

### **Output**

Though the AGE column have the value "25" in two records, each combination of "25" with it's specific 'salary' is unique, so both rows are included in the result set −

|  |  |
| --- | --- |
| **AGE** | **SALARY** |
| 22 | 4500.00 |
| 23 | 2000.00 |
| 24 | 10000.00 |
| 25 | 1500.00 |
| 25 | 6500.00 |
| 27 | 8500.00 |
| 32 | 2000.00 |

## DISTINCT Keyword with COUNT() Function

The **COUNT()** function is used to get the number of records retuned by the SELECT query. We need to pass an expression to this function so that the SELECT query returns the number of records that satisfy the specified expression.

If we pass the DISTINCT keyword to the COUNT() function as an expression, it returns the number of unique values in a column of a table.

### **Syntax**

Following is the syntax for using the DISTINCT keyword with COUNT() function −

SELECT COUNT(DISTINCT column\_name) FROM table\_name WHERE condition;

Where, **column\_name** is the name of the column for which we want to count the unique values; and **table\_name** is the name of the table that contains the data.

### **Example**

In the following query, we are retrieving the count of distinct age of the customers −

SELECT COUNT(DISTINCT AGE) as UniqueAge FROM CUSTOMERS;

### **Output**

Following is the result produced −

|  |
| --- |
| **UniqueAge** |
| 6 |

## DISTINCT Keyword with NULL Values

In SQL, when there are NULL values in the column, DISTINCT treats them as unique values and includes them in the result set.

### **Example**

First of all let us update two records of the CUSTOMERS table and modify their salary values to **NULL**

UPDATE CUSTOMERS SET SALARY = NULL WHERE ID IN(6,4);

The resultant CUSTOMERS table would be −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 | NULL |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 6 | Komal | 22 | Hyderabad | NULL |
| 7 | Muffy | 24 | Indore | 10000.00 |

Now, we are retrieving the distinct salary of the customers using the following query −

SELECT DISTINCT SALARY FROM CUSTOMERS ORDER BY SALARY;

### **Output**

Following is the output of the above query −

|  |
| --- |
| **SALARY** |
| NULL |
| 1500.00 |
| 2000.00 |
| 8500.00 |
| 10000.00 |

# SQL - ORDER BY Clause

## The SQL ORDER BY Clause

The SQL **ORDER BY** clause is used to sort the data in either ascending or descending order, based on one or more columns. This clause can sort data by a single column or by multiple columns. Sorting by multiple columns can be helpful when you need to sort data hierarchically, such as sorting by state, city, and then by the person's name.

ORDER BY is used with the SQL SELECT statement and is usually specified after the WHERE, HAVING, and GROUP BY clauses.

Following are the important points about ORDER BY Clause −

* Some databases sort the query results in an ascending order by default.
* To sort the data in ascending order, we use the keyword **ASC**.
* To sort the data in descending order, we use the keyword **DESC**.

In addition to sorting records in ascending order or descending order, the ORDER BY clause can also sort the data in a database table in a preferred order.

This preferred order may not sort the records of a table in any standard order (like alphabetical or lexicographical), but they could be sorted based on external condition(s).

For instance, in the CUSTOMERS table containing the details of the customers of an organization, the records can be sorted based on the population of the cities they are from. This need not be alphabetically sorted, instead, we need to define the order manually using the [**CASE**](https://www.tutorialspoint.com/sql/sql-case.htm) statement.

### **Syntax**

The basic syntax of the ORDER BY clause is as follows −

SELECT column-list

FROM table\_name

[ORDER BY column1, column2, .. columnN] [ASC | DESC];

Where, **column-list** is list of the columns we want to retrieve; and ASC or DESC specifies the sort order.

**Note:** We can use more than one column in the ORDER BY clause, but we need to make sure that the column we are using to sort is specified in the column-list.

## ORDER BY Clause with ASC

We can sort the result-set of a query in ascending order (based on one or more columns) using the **SQL ORDER BY** clause by specifying **ASC** as the sort order. ASC is the default sort order for this clause, i.e. while using the ORDER BY clause if you do not explicitly specify the sort order, the data will be sorted in ascending order.

### **Example**

Assume we have created a table with name CUSTOMERS in the MySQL database using CREATE TABLE statement as shown 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)

);

Following query inserts values into this table using the INSERT statement −

insert INTO CUSTOMERS 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, 'Hyderabad', 4500.00),

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

The table obtained is 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 | Hyderabad | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |

In the following query, we are sorting the records of the CUSTOMERS table in ascending order based on the column NAME −

SELECT \* FROM CUSTOMERS ORDER BY NAME ASC;

### **Output**

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 | Hyderabad | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

## ORDER BY Clause with DESC

To sort the result-set of a query in descending order (based on one or more columns), we need to use the ORDER BY clause by specifying DESC as the sort order.

### **Example**

The following query sorts the records of the CUSTOMER table based on the descending order of the name of the customers −

SELECT \* FROM CUSTOMERS ORDER BY NAME DESC;

### **Output**

This would produce the result as follows −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |
| 6 | Komal | 22 | Hyderabad | 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 |

## ORDER BY Clause on Multiple Columns

We can use the ORDER BY clause to sort the result-set of a query by multiple (more than one) columns. When sorting by multiple columns, the sorting is done in the order that is specified in the ORDER BY clause. In other words, the table will be sorted based on the first column (specified in the query), then the second column, and so on.

### **Example**

In the following query, we are retrieving all records from the CUSTOMERS table and sorting them first by their address in ascending order, and then by their salary in descending order −

SELECT \* FROM CUSTOMERS ORDER BY AGE ASC, SALARY DESC;

### **Output**

Following is the result produced −

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

## ORDER BY with WHERE Clause

We can also use the WHERE clause with the ORDER BY clause to sort the rows that meet certain conditions. This can be useful when we want to sort a subset of the data in a table based on the specific criteria.

### **Example**

Now, we are retrieving all records from the CUSTOMERS table where the age of the customer is 25, and sorting them as per the descending order of their names −

SELECT \* FROM CUSTOMERS WHERE AGE = 25 ORDER BY NAME DESC;

### **Output**

Following is the output of the above query −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |

## ORDER BY with LIMIT Clause

We can use the LIMIT clause with ORDER BY clause to limit the specified number of rows by sorting them either in ascending or in descending order.

### **Syntax**

Following is the syntax of using the LIMIT clause with the ORDER BY clause in MySQL database −

SELECT column1, column2, ...

FROM table\_name

ORDER BY column\_name1 [ASC | DESC], column\_name2 [ASC | DESC], ...

LIMIT N;

### **Example**

In here, we are retrieving the top 4 records from the CUSTOMERS table based on their salary, and sorting them in ascending order based on their name −

SELECT SALARY FROM CUSTOMERS ORDER BY NAME LIMIT 4;

### **Output**

Following is the output of the above query −

|  |
| --- |
| **SALARY** |
| 6500.00 |
| 8500.00 |
| 2000.00 |
| 1500.00 |

## Sorting Results in a Preferred Order

One can also sort the records of a table in their own preferred order using the **CASE** statement within the ORDER BY clause. All the values are specified in the clause along with the position they are supposed to be sorted in; if the values are not given any number, they are automatically sorted in ascending order.

### **Example**

To fetch the rows with their own preferred order, the SELECT query used would be as follows −

SELECT \* FROM CUSTOMERS

ORDER BY (CASE ADDRESS

WHEN 'MUMBAI' THEN 1

WHEN 'DELHI' THEN 2

WHEN 'HYDERABAD' THEN 3

WHEN 'AHMEDABAD' THEN 4

WHEN 'INDORE' THEN 5

WHEN 'BHOPAL' THEN 6

WHEN 'KOTA' THEN 7

ELSE 100 END);

### **Output**

The above query sorts the CUSTOMERS table based on the custom order defined using the CASE statement. Here, we are sorting the records based on the population of the cities specified in the ADDRESS column.

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

# SQL - Group By Clause

## The SQL GROUP BY Clause

The SQL **GROUP BY** clause is used in conjunction with the SELECT statement to arrange identical data into groups. This clause follows the WHERE clause in a SELECT statement and precedes the ORDER BY and HAVING clauses (if they exist).

The main purpose of grouping the records of a table based on particular columns is to perform calculations on these groups. Therefore, The GROUP BY clause is typically used with aggregate functions such as SUM(), COUNT(), AVG(), MAX(), or MIN() etc.

For example, if you have a table named SALES\_DATA containing the sales data with the columns YEAR, PRODUCT, and SALES. To calculate the total sales in an year, the GROUP BY clause can be used to group the records in this table based on the year and calculate the sum of sales in each group using the SUM() function.

### **Syntax**

Following is the basic syntax of the SQL GROUP BY clause −

SELECT column\_name(s)

FROM table\_name

GROUP BY column\_name(s);

Where, **column\_name(s)** refers to the name of one or more columns in the table that we want to group the data by and the **table\_name** refers to the name of the table that we want to retrieve data from.

## GROUP BY Clause with Aggregate Functions

Typically, we group the record of a table to perform calculations on them. Therefore, the SQL GROUP BY clause is often used with the aggregate functions such as SUM(), AVG(), MIN(), MAX(), COUNT(), etc.

### **Example**

Assume we have created a table named CUSTOMERS, which contains the personal details of customers including their name, age, address and salary, using the following query −

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 insert values into this table using the INSERT statement as follows −

INSERT INTO CUSTOMERS VALUES

(1, 'Ramesh', 32, 'Hyderabad', 2000.00),

(2, 'Khilan', 32, 'Hyderabad', 1500.00),

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

(4, 'Chaitali', 32, 'Delhi', 6500.00),

(5, 'Hardik', 23, 'Bhopal', 8500.00),

(6, 'Komal', 25, 'Indore', 4500.00),

(7, 'Muffy', 25, 'Indore', 10000.00);

The table created is as shown below −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | Ramesh | 32 | Hyderabad | 2000.00 |
| 2 | Khilan | 32 | Hyderabad | 1500.00 |
| 3 | Kaushik | 23 | Delhi | 2000.00 |
| 4 | Chaitali | 32 | Delhi | 6500.00 |
| 5 | Hardik | 23 | Bhopal | 8500.00 |
| 6 | Komal | 25 | Indore | 4500.00 |
| 7 | Muffy | 25 | Indore | 10000.00 |

The following SQL query groups the CUSTOMERS table based on AGE and counts the number of records in each group −

SELECT AGE, COUNT(Name) FROM CUSTOMERS GROUP BY AGE;

### **Output**

Following is the result produced −

|  |  |
| --- | --- |
| **AGE** | **COUNT(Name)** |
| 32 | 3 |
| 23 | 2 |
| 25 | 2 |

### **Example**

In the following query, we are finding the highest salary for each age −

SELECT AGE, MAX(salary) AS MAX\_SALARY FROM CUSTOMERS GROUP BY AGE;

### **Output**

Following is the output of the above query −

|  |  |
| --- | --- |
| **AGE** | **MAX\_SALARY** |
| 32 | 6500.00 |
| 23 | 8500.00 |
| 25 | 10000.00 |

Similarly we can group the records of the CUSTOMERS table based on the AGE column and calculate the maximum salary, average and sum of the SALARY values in each group using the MIN(), AVG() and SUM() functions respectively.

## GROUP BY Clause on Single Columns

When we use the GROUP BY clause with a single column, all the rows in the table that have the same value in that particular column will be merged into a single record.

### **Example**

In the following example we are grouping the above created CUSTOMERS table by the ADDRESS column and calculating the average salary of the customer from each city −

SELECT ADDRESS, AVG(SALARY) as AVG\_SALARY FROM CUSTOMERS GROUP BY ADDRESS;

### **Output**

This would produce the following result −

|  |  |
| --- | --- |
| **ADDRESS** | **AVG\_SALARY** |
| Hyderabad | 1750.000000 |
| Delhi | 4250.000000 |
| Bhopal | 8500.000000 |
| Indore | 7250.000000 |

## GROUP BY Clause with Multiple Columns

When we use the GROUP BY clause with multiple columns, all the rows in the table that have the same values in all of the specified columns will be merged into a single group.

### **Example**

In the following query we are grouping the records of the CUSTOMERS table based on the columns ADDRESS and AGE and −

SELECT ADDRESS, AGE, SUM(SALARY) AS TOTAL\_SALARY FROM CUSTOMERS GROUP BY ADDRESS, AGE;

### **Output**

This would produce the following result −

|  |  |  |
| --- | --- | --- |
| **ADDRESS** | **AGE** | **TOTAL\_SALARY** |
| Hyderabad | 32 | 3500.00 |
| Delhi | 23 | 2000.00 |
| Delhi | 32 | 6500.00 |
| Bhopal | 23 | 8500.00 |
| Indore | 25 | 14500.00 |

## GROUP BY with ORDER BY Clause

We can use the ORDER BY clause with GROUP BY in SQL to sort the grouped data by one or more columns.

### **Syntax**

Following is the syntax for using ORDER BY clause with GROUP BY clause in SQL −

SELECT column1, column2, ..., aggregate\_function(columnX) AS alias

FROM table

GROUP BY column1, column2, ...

ORDER BY column1 [ASC | DESC], column2 [ASC | DESC], ...;

### **Example**

In here, we are finding the highest salary for each age, sorted by high to low −

SELECT AGE, MIN(SALARY) AS MIN\_SALARY FROM CUSTOMERS GROUP BY AGE ORDER BY MIN\_SALARY DESC;

### **Output**

Following is the result produced −

|  |  |
| --- | --- |
| **AGE** | **MIN\_SALARY** |
| 25 | 4500.00 |
| 23 | 2000.00 |
| 32 | 1500.00 |

## GROUP BY with HAVING Clause

We can also use the GROUP BY clause with the HAVING clause filter the grouped data in a table based on specific criteria.

### **Syntax**

Following is the syntax for using ORDER BY clause with HAVING clause in SQL −

SELECT column1, column2, aggregate\_function(column)

FROM table\_name

GROUP BY column1, column2

HAVING condition;

### **Example**

In the following query, we are grouping the customers by their age and calculating the minimum salary for each group. Using the HAVING clause we are filtering the groups where the age is greater than 24 −

SELECT ADDRESS, AGE, MIN(SALARY) AS MIN\_SAL FROM CUSTOMERS GROUP BY ADDRESS, AGE HAVING AGE>24;

### **Output**

The result produced is as follows −

|  |  |  |
| --- | --- | --- |
| **ADDRESS** | **AGE** | **MIN\_SUM** |
| Hyderabad | 32 | 1500.00 |
| Delhi | 32 | 6500.00 |
| Indore | 25 | 4500.00 |

# SQL - Having Clause

## The SQL HAVING Clause

The SQL **HAVING clause** is similar to the WHERE clause; both are used to filter rows in a table based on specified criteria. However, the HAVING clause is used to filter grouped rows instead of single rows. These rows are grouped together by the GROUP BY clause, so, the HAVING clause must always be followed by the GROUP BY clause.

Moreover, the HAVING clause can be used with aggregate functions such as COUNT(), SUM(), AVG(), etc., whereas the WHERE clause cannot be used with them.

### **Syntax**

Following is the basic syntax of the SQL HAVING clause −

SELECT column1, column2, aggregate\_function(column)

FROM table\_name

GROUP BY column1, column2

HAVING condition;

The following code block shows the position of the HAVING Clause in a query −

SELECT

FROM

WHERE

GROUP BY

HAVING

ORDER BY

## HAVING with GROUP BY Clause

We can use the HAVING clause with the GROUP BY clause to filter groups of rows that meet certain conditions. It is used to apply a filter to the result set after the aggregation has been performed.

### **Example**

Assume we have created a table named CUSTOMERS, which contains the personal details of customers including their name, age, address and salary, using the following query −

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 insert values into this table using the INSERT statement as follows −

INSERT INTO CUSTOMERS VALUES

(1, 'Ramesh', 32, 'Hyderabad', 2000.00),

(2, 'Khilan', 32, 'Hyderabad', 1500.00),

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

(4, 'Chaitali', 32, 'Delhi', 6500.00),

(5, 'Hardik', 23, 'Bhopal', 8500.00),

(6, 'Komal', 25, 'Indore', 4500.00),

(7, 'Muffy', 25, 'Indore', 10000.00);

The table created is as shown below −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | Ramesh | 32 | Hyderabad | 2000.00 |
| 2 | Khilan | 32 | Hyderabad | 1500.00 |
| 3 | Kaushik | 23 | Delhi | 2000.00 |
| 4 | Chaitali | 32 | Delhi | 6500.00 |
| 5 | Hardik | 23 | Bhopal | 8500.00 |
| 6 | Komal | 25 | Indore | 4500.00 |
| 7 | Muffy | 25 | Indore | 10000.00 |

Now, we are grouping the records of the CUSTOMERS table based on the columns ADDRESS and AGE and filtering the groups where the AGE value is less than 25.

SELECT ADDRESS, AGE, MIN(SALARY) AS MIN\_SAL FROM CUSTOMERS GROUP BY ADDRESS, AGE HAVING AGE > 25;

### **Output**

The result produced is as follows −

|  |  |  |
| --- | --- | --- |
| **ADDRESS** | **AGE** | **MIN\_SUM** |
| Hyderabad | 32 | 1500.00 |
| Delhi | 32 | 6500.00 |

## HAVING with ORDER BY Clause

The ORDER BY clause is used to arrange/sort the records of the result of a SELECT query based on a specific column (either in ascending order or in descending order). If we use the ORDER BY clause with the HAVING clause we can sort the filtered groups in the desired order.

### **Example**

Following query groups the records of the CUSTOMERS table based on the columns AGE and ADDRESS, filters the groups where the SALARY value is less than 5000 and, arranges the remaining groups in descending order based the total salaries of each group.

SELECT ADDRESS, AGE, SUM(SALARY) AS TOTAL\_SALARY FROM CUSTOMERS GROUP BY ADDRESS, AGE HAVING TOTAL\_SALARY >=5000 ORDER BY TOTAL\_SALARY DESC;

### **Output**

The result produced is as follows −

|  |  |  |
| --- | --- | --- |
| **ADDRESS** | **AGE** | **TOTAL\_SALARY** |
| INDORE | 25 | 14500.00 |
| BHOPAL | 23 | 8500.00 |
| DELHI | 32 | 6500.00 |

## HAVING Clause with COUNT() Function

The HAVING clause can be used with the COUNT() function to filter groups based on the number of rows they contain.

### **Example**

Following query groups the records of the CUSTOMERS table based on the AGE column and, retrieves the details of the group that has more than two entities −

SELECT AGE, COUNT(AGE) FROM CUSTOMERS GROUP BY AGE HAVING COUNT(age) > 2;

### **Output**

This would produce the following result −

|  |  |
| --- | --- |
| **AGE** | **COUNT(AGE)** |
| 32 | 3 |

## HAVING Clause with AVG() Function

The HAVING clause can also be used with the AVG() function to filter groups based on the average value of a specified column.

### **Example**

Now, we are retrieving the city of the customers whose average salary is greater than 5240 −

SELECT ADDRESS, AVG(SALARY) as AVG\_SALARY FROM CUSTOMERS GROUP BY ADDRESS HAVING AVG(SALARY) > 5240;

### **Output**

Following is the output of the above query −

|  |  |
| --- | --- |
| **ADDRESS** | **AVG\_SALARY** |
| Bhopal | 8500.000000 |
| Indore | 7250.000000 |

## HAVING Clause with MAX() Function

We can also use the HAVING clause with MAX() function to filter groups based on the maximum value of a specified column.

### **Example**

Now, we are retrieving the city of the customers whose maximum salary is greater than 5240 −

SELECT ADDRESS, MAX(SALARY) as MAX\_SALARY FROM CUSTOMERS GROUP BY ADDRESS HAVING MAX(SALARY) > 7000;

### **Output**

The result obtained is as follows −

|  |  |
| --- | --- |
| **ADDRESS** | **MAX\_SALARY** |
| Bhopal | 8500.00 |
| Indore | 10000.00 |

# SQL - AND and OR Conjunctive Operators

An operator is a reserved word or a character used primarily in SQL to manipulate and retrieve data from a database object. They are used with SQL queries in a WHERE clause to perform binary or unary operation(s) based on the operator used. In a unary operation, the operation is performed with a single operator and a single operand; whereas in a binary operation, the operation is performed with a single operator and two operands.

These operators are used to specify conditions in an SQL statement with the purpose of filtering data or to serve as conjunctions for multiple conditions in a statement.

The SQL **AND** & **OR** are logical operators, that serve as conjunctive operators, used to combine multiple conditions in an SQL statement with the purpose of filtering data in a database table.

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. All of these conditions must be true for the overall condition to be true.

### **Syntax**

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

WHERE [condition1] AND [condition2]...AND [conditionN];

where, **condition1, condition2,** ... are the conditions we want to apply to the query. Each condition is separated by the AND operator.

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**

Assume we have created a table with name CUSTOMERS in SQL database using CREATE TABLE statement as shown 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)

);

Following query inserts values into this table using the INSERT statement −

insert INTO CUSTOMERS VALUES(1, 'Ramesh', 32, 'Ahmedabad', 2000.00);

insert INTO CUSTOMERS VALUES(2, 'Khilan', 25, 'Delhi', 1500.00);

insert INTO CUSTOMERS VALUES(3, 'kaushik', 23, 'Kota', 2000.00);

insert INTO CUSTOMERS VALUES(4, 'Chaitali', 25, 'Mumbai', 6500.00);

insert INTO CUSTOMERS VALUES(5, 'Hardik', 27, 'Bhopal', 8500.00);

insert INTO CUSTOMERS VALUES(6, 'Komal', 22, 'MP', 4500.00);

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

Using the following SELECT query, you can verify if the records are properly inserted into the table −

SELECT \* from CUSTOMERS;

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

| 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 −

SELECT ID, NAME, SALARY

FROM CUSTOMERS

WHERE SALARY > 2000 AND age < 25;

### **Output**

This would produce the following result −

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

| ID | NAME | SALARY |

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

| 6 | Komal | 4500.00 |

| 7 | Muffy | 10000.00 |

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

## Multiple AND Operators

You can also use multiple 'AND' operators in a query to combine multiple conditions or expressions together. Conditions combined with these multiple 'AND' operators are evaluated from left to right. If any of the conditions evaluate to false, the entire compound condition will be false and the record will not be included in the result set.

### **Example**

In the following query, we are selecting all records from the CUSTOMERS table where the name of the customer starts with 'K', the age of the customer is greater than or equal to 22, and their salary is less than 3742 −

SELECT \* FROM CUSTOMERS

WHERE NAME LIKE 'k%' AND AGE >= 22 AND SALARY < 3742;

### **Output**

Following is the result produced −

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

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

## AND with other Logical Operators

The "AND" operator can be used in combination with other logical operators to filter records from a database table more strictly.

It's important to note that when using multiple logical operators in SQL, the order of operations is important. Parentheses can be used to control the order of operations and ensure that the conditions are evaluated in the correct order. Additionally, using too many logical operators or complex expressions can negatively impact query performance, so it's important to carefully consider the design of the WHERE clause when working with large datasets.

### **Example**

In here, we are combining the **AND** operator with the **NOT** operator to create a **NAND** operation. The 'NAND' operation returns true if at least one of the input conditions is false, and false if both input conditions are true.

In the following query we are selecting all records from the CUSTOMERS table where the condition (salary is greater than 4500 and the age is less than 26) is false. The "NOT" operator negates the entire condition, and the "AND" operator combines two conditions −

SELECT \* FROM CUSTOMERS

WHERE NOT (SALARY > 4500 AND AGE < 26);

### **Output**

Following is the output of the above query −

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

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

## AND with UPDATE statement

The "AND" operator can be used with the "UPDATE" statement to modify rows in a table that meet certain specified criteria.

### **Syntax**

Following is the syntax of using the AND operator with the UPDATE statement −

UPDATE table\_name

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

WHERE condition1 AND condition2 AND ...;

Where, **table\_name** is the name of the table we want to update, **column1, column2,** etc. are the columns we want to modify, and **value1, value2,** etc. are the new values we want to set for those columns.

### **Example**

In the following query we are trying to update the salary of all the customers whose age is greater than 27 and updating it to ‘55000’ using UPDATE statement −

UPDATE CUSTOMERS

SET salary = 55000

WHERE AGE > 27;

### **Output**

We get the following result. We can observe that the salary of 1 customer has been modified −

(1 row affected)

### **Verification**

To verify if the changes are reflected in the tables, we can use SELECT statement to print the tables.

To display the Customers table, use the following query −

select \* from customers;

The table is displayed as follows −

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 55000.00 |

| 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 |

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

As we can see in the above table, the salary of ‘Ramesh’ has been updated to ‘55000’ because his age is 32 i.e. greater than 27.

## The OR Operator

The **OR** operator is used to combine multiple conditions in an SQL statement's WHERE clause. It returns true if at least one of the conditions is true, and false if all conditions are false.

### **Syntax**

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

WHERE [condition1] OR [condition2]...OR [conditionN]

where, **condition1, condition2,** ... are the conditions we want to apply to the query. Each condition is separated by the OR operator.

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**

The following query fetches 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.

SELECT ID, NAME, SALARY

FROM CUSTOMERS

WHERE SALARY > 2000 OR age < 25;

### **Output**

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 |

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

## Multiple OR operators

In SQL, it is common to use multiple 'OR' operators to combine multiple conditions or expressions together. When using multiple "OR" operators, any rows that meet at least one of the conditions will be returned.

### **Example**

In the following query, we are selecting all records from the CUSTOMERS table where either the name of the customer ends with 'l', or the salary of the customer is greater than 10560, or their age is less than 25 −

SELECT \* FROM CUSTOMERS

WHERE NAME LIKE '%l' OR SALARY > 10560 OR AGE < 25

### **Output**

Following is the result obtained −

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 3 | kaushik | 23 | Kota | 2000.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore |10000.00 |

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

## OR with AND operator

We can use AND and OR operators together in SQL to combine multiple conditions in a WHERE clause to filter rows that meets the specified criteria.

### **Syntax**

Following is the syntax for using the AND and OR operators together −

WHERE (condition1 OR condition2) AND condition3;

Where, **condition1, condition2, and condition3** represent the conditions that we want to combine with the AND and OR operators. The parentheses group the first two conditions and combine them with the OR operator. The result of that operation is combined with the third condition using the AND operator.

### **Example**

In the following query we are trying to retrieve all rows from the "CUSTOMERS" table where the age is equal to 25 or the salary is less than 4500 and the name is either Komal or Kaushik. The parentheses control the order of evaluation so that the OR operator is applied first, followed by the AND operator −

SELECT \* FROM CUSTOMERS

WHERE (AGE = 25 OR salary < 4500) AND (name = 'Komal' OR name = 'Kaushik');

### **Output**

This would produce the following result −

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 3 | kaushik | 23 | Kota | 2000.00 |

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

## OR with DELETE statement

We can also use the OR operator with the DELETE statement to delete rows that meet any one of multiple conditions.

### **Syntax**

Following is the syntax of using OR operator with DELETE statement −

DELETE FROM table\_name

WHERE column1 = 'value1' OR column2 = 'value2';

### **Example**

In the following query we are trying to delete the records from the customers table where either the age of the customer equals 25 or their salary is less than 2000 −

DELETE FROM CUSTOMERS

WHERE AGE = 25 OR SALARY < 2000

### **Output**

We get the following result −

(2 rows affected)

### **Verification**

To verify if the changes are reflected in the tables, we can use SELECT statement to print the tables.

To display the Customers table, use the following query −

select \* from customers;

The table is displayed as follows −

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 55000.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

# SQL - LIKE Operator

The **SQL LIKE** is a logical operator that is used to retrieve the data in a column of a table, based on a specified pattern.

It is used along with the WHERE clause of the UPDATE, DELETE and SELECT statements, to filter the rows based on the given pattern. These patterns are specified using **‘Wildcards’**.

Suppose we need to submit the list of all the students whose name starts with ‘K’. We can obtain this with the help of the LIKE operator as follows −

WHERE student\_name LIKE 'K%';

Here, the **“%”** is a wild card which represents zero, one or multiple characters. And the expression **“K%”** specifies that it will display the list of all the students whose name **starts with** ‘k’.

*The LIKE operator can be used with strings, numbers, or date values. However, using the string values is recommended.*

### **Syntax**

The basic syntax of the SQL LIKE operator is as follows −

SELECT column1, column2, ...

FROM table\_name

WHERE columnn LIKE specified\_pattern;

## What are wild cards?

SQL wildcards are special characters used in SQL queries to match patterns in the data. Following are the four wildcards used in conjunction with the LIKE operator −

|  |  |
| --- | --- |
| **S.No** | **WildCard & Definition** |
| 1 | **%**  The percent sign represents zero, one or multiple characters. |
| 2 | **\_**  The underscore represents a single number or character. |
| 3 | **[]**  This matches any single character within the given range in the []. |
| 4 | **[^]**  This matches any single character excluding the given range in the [^]. |

**Note** − In the LIKE operator, the above wildcard characters can be used individually as well as in combinations with each other. The two mainly used wildcard characters are **‘%’** and **‘\_’**.

The table given below has a few examples showing the WHERE clause having different LIKE operators with '%' , '\_' , [] and [^] pattern −

|  |  |
| --- | --- |
| **S.No** | **Statement & Description** |
| 1 | **WHERE SALARY LIKE '200%'**  Finds any values that start with 200. |
| 2 | **WHERE SALARY LIKE '%200%'**  Finds any values that have 200 in any position. |
| 3 | **WHERE SALARY LIKE '\_00%'**  Finds any values that have 00 in the second and third positions. |
| 4 | **WHERE SALARY LIKE '2\_%\_%'**  Finds any values that start with 2 and are at least 3 characters in length. |
| 5 | **WHERE SALARY LIKE '%2'**  Finds any values that end with 2. |
| 6 | **WHERE SALARY LIKE '\_2%3'**  Finds any values that have a 2 in the second position and end with a 3. |
| 7 | **WHERE SALARY LIKE '2\_\_\_3'**  Finds any values in a five-digit number that start with 2 and end with 3. |
| 8 | **WHERE NAME LIKE 'A[a, l, j, y, a, k]'**  Finds any name that starts with A and has the specified alphabets. For e.g. Ajay |
| 9 | **WHERE NAME LIKE 'A[a, l, j, y, a, k]'**  Finds any name that starts with A and does not consists of the specified alphabets. For e.g. Abdu. |

## Using the “%” Wildcard character

The % sign represents zero or multiple characters. The ‘%’ wildcard matches any length of a string which even includes the zero length.

### **Example**

To understand it better let us consider the CUSTOMERS table which contains the personal details of customers including their name, age, address and salary etc. as shown 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)

);

Now, insert values into this table using the INSERT statement as follows −

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 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

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

The table will be created 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 try to display all the records from the CUSTOMERS table, where the SALARY starts with 200.

SELECT \* FROM CUSTOMERS

WHERE SALARY LIKE '200%';

### **Output**

This would produce the following result −

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

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

### **Example**

Below is the query, that displays all the records from the CUSTOMERS table previously created, with the NAME that has ‘al’ in any position. Here we are using multiple ‘%’ wildcards in the LIKE condition −

SELECT \* FROM CUSTOMERS

WHERE NAME LIKE '%al%';

### **Output**

The following result is produced −

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

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

## Using the “\_” wildcard character

The **underscore** wild card represents a single number or character. A single ‘\_’ looks for exactly one character similar to the ‘%’ wildcard.

### **Example**

Following is the query, which would display all the records from the CUSTOMERS table previously created, where the Name starts with **K** and is at least 4 characters in length −

SELECT \* FROM CUSTOMERS

WHERE NAME LIKE 'K\_\_\_%';

### **Output**

The result obtained is given below −

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 6 | Komal | 22 | MP | 4500.00 |

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

### **Example**

Following is the query to display all the records from the CUSTOMERS table, where the Name has ‘m’ in the third position −

SELECT \* from customers WHERE NAME LIKE '\_\_m%';

### **Output**

We get the following result on executing the above query −

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 6 | Komal | 22 | MP | 4500.00 |

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

## Using the “[ ]” wildcard character

The square bracket with a list of characters matches any single character within the given range [b-k] or set [xyz].

### **Example**

In the query given below we are trying to display all the records from the CUSTOMERS table, where the NAME starts with **K** and has the specified characters set **[h,i,o,m,l,a,n]** −

select \* from customers

where NAME LIKE 'k[h,i,o,m,l,a,n]%';

### **SELECT \***

### **FROM customers**

### **WHERE NAME LIKE 'k%'**

### **AND (NAME LIKE 'kh%'**

### **OR NAME LIKE 'ki%'**

### **OR NAME LIKE 'ko%'**

### **OR NAME LIKE 'km%'**

### **OR NAME LIKE 'kl%'**

### **OR NAME LIKE 'ka%'**

### **OR NAME LIKE 'kn%');**

### **Output**

We get the following result −

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 6 | Komal | 22 | MP | 4500.00 |

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

### **Example**

### In MySQL, you can use the **REGEXP** operator with a regular expression to achieve the desired pattern matching where the **NAME** column starts with any character from 'b' to 'i'. Here's how you can write the query:

### **SELECT \***

### **FROM customers**

### **WHERE NAME REGEXP '^[b-i]';**

### **Output**

The result given below is displayed −

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

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

## Using the “[^]” wildcard character

The [^ character list or range] matches any single character that is not present in the given range or character list.

### **Example**

### In MySQL, you can use the **REGEXP** operator with a regular expression to achieve the desired pattern matching where the **NAME** column does not start with any character from 'b' to 'k'. Here's how you can write the query:

### **SELECT \***

### **FROM customers**

### **WHERE NAME NOT REGEXP '^[b-k]';**

### **Output**

The following result is obtained −

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 7 | Muffy | 24 | Indore | 10000.00|

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

## Using LIKE operator with OR operator

We can also use the LIKE operator with multiple string patterns for selecting rows by using the **AND** or **OR** operators.

### **Syntax**

Following is the basic syntax of using LIKE operator with OR operator −

SELECT column1, column2, ...

FROM table\_name

WHERE column1 LIKE pattern1 OR column2 LIKE pattern2 OR ...;

### **Example**

Here, the SQL command select the customers whose NAME starts with **C** and ends with **i**, or customers whose NAME ends with **k** −

SELECT \* FROM CUSTOMERS

WHERE NAME LIKE 'C%i' OR NAME LIKE '%k';

### **Output**

This will produce the following result −

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

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

## Using NOT operator with the LIKE condition

We use the NOT LIKE operator to extract the rows which does not contain a particular string provided in the search pattern.

### **Syntax**

Following is the basic syntax of NOT LIKE operator in SQL −

SELECT column1, column2, ...

FROM table\_name

WHERE column1 NOT LIKE pattern;

### **Example**

In the query given below we are trying to fetch all the customers whose NAME does not start with **K** −

select \* from customers

WHERE NAME NOT LIKE 'K%';

### **Output**

This will produce the following result −

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

## Using Escape characters with LIKE operator

The escape character in SQL is used to exclude certain wildcard characters from the expression of the **LIKE** operator. By doing so, we can use these characters in their general sense.

Using escape, we can also avoid using the characters that are reserved in SQL syntax to denote specific commands, such as the single quote **“ ' ”**, **“%”** and **“\_”**.

For example, if you need to search for **“%”** as a literal in the LIKE condition, then it is done using Escape character.

**Note** − An escape character is only defined as a single character. It is suggested to choose the character which is not present in our data.

### **Syntax**

The syntax for using the LIKE operator with escape characters is as follows −

SELECT column1, column2, ...

FROM table\_name

WHERE column1 LIKE 'pattern ESCAPE escape\_character';

Where,

* **pattern** is the pattern you want to match.
* **ESCAPE** is the keyword that indicates the escape character
* **escape\_character** is the character that you want to use as the escape character.

### **Example**

Let us create a new table EMPLOYEE using the query below −

CREATE TABLE EMPLOYEE (

SALARY DECIMAL (18,2) NOT NULL,

BONUS\_PERCENT VARCHAR (20)

);

Now, we can insert values into this empty tables using the INSERT statement as follows −

INSERT INTO EMPLOYEE VALUES (67000.00, '45.00');

INSERT INTO EMPLOYEE VALUES (54000.00, '20.34%');

INSERT INTO EMPLOYEE VALUES (75000.00, '51.00');

INSERT INTO EMPLOYEE VALUES (84000.00, '56.82%');

The **Employee** table consists of the salary of employees in an organization and the bonus percentage in their salary as shown below −

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

| SALARY | BONUS\_PERCENT |

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

| 67000.00 | 45.00 |

| 54000.00 | 20.34% |

| 75000.00 | 51.00 |

| 84000.00 | 56.82% |

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

Now, let us try to display all the records from the EMPLOYEE table, where the BONUS\_PERCENT contains the **%** literal.

select \* from employee WHERE BONUS\_PERCENT LIKE '%!%' escape '!';

### **Output**

This will produce the following result −

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

| SALARY | BONUS\_PERCENT |

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

| 54000.00 | 20.34% |

| 84000.00 | 56.82% |

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

### **Example**

In here, we are trying to return the BONUS\_PERCENT that starts with **‘2’** and contains the **‘%’** literal.

select \* from employee

WHERE BONUS\_PERCENT LIKE'2%!%%' escape '!';

### **Output**

Following result is obtained −

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

| SALARY | BONUS\_PERCENT |

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

| 54000.00 | 20.34% |

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

# SQL - IN Operator

The **SQL IN** is a logical operator that allows us to specify multiple values or sub query in the **WHERE** clause.

It returns all rows in which the specified column matches one of the values in the list. The list of values or sub query must be specified in the parenthesis e.g. IN **(select query)** or IN **(Value1, Value2, Value3, …)**.

The IN operator can be used with any data type in SQL. It is used to filter data from a database table based on specified values.

In some scenarios we may use multiple OR statements to include multiple conditions in SELECT, DELETE, UPDATE, or INSERT statements. Instead of multiples OR statements we can use the IN operator.

*The IN operator is useful when you want to select all rows that match one of a specific set of values. While the OR operator is useful when you want to select all rows that match any one of multiple conditions.*

### **Syntax**

The basic syntax of the SQL IN operator to specify multiple values is as follows −

WHERE column\_name IN (value1, value2, value3, ...);

Where,

* **value1, value2, value3, …** are the values in the list to be tested against the expression. The IN operator returns TRUE if any of these values is found in the list, and FALSE if it is not.

## Using IN operator in SELECT Statement

We can use the **SQL IN** operator to specify multiple values in a WHERE clause, and we can also use it in a SELECT statement to retrieve data that matches any of the specified values.

Here, we are using the IN operator to specify multiple values in SELECT statement.

### **Example**

To understand it better let us consider the CUSTOMERS table which contains the personal details of customers including their name, age, address and salary etc. as shown 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)

);

Now, insert values into this table using the INSERT statement as follows −

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 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

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

The table will be created 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 |

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

Suppose based on the above table we want to display records with NAME equal to ‘Khilan’, ‘Hardik’ and ‘Muffy’ (string values). This can be achieved using **IN** operator as follows −

select \* from CUSTOMERS WHERE NAME IN ('Khilan', 'Hardik', 'Muffy');

### **Output**

The result obtained is as follows −

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

**Note** − We cannot use the wildcard characters '%', '\_', etc. with the string values. The above query can also be done using OR operator as follows −

select \* from CUSTOMERS WHERE NAME = 'Khilan' OR NAME = 'Hardik' OR NAME = 'Muffy';

### **Output**

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

## Using IN operator in UPDATE statement

We can also use the **SQL IN** operator in an UPDATE statement to update rows that match any of the specified values in a WHERE clause. The UPDATE statement is used to modify existing data in a database table.

### **Example**

Here, we are using the IN operator to specify multiple values in the UPDATE statement and updating the CUSTOMERS table previously created. Here, are changing the records of the customers with age ‘25’ or ‘27’ and updating the age value to ‘30’ −

update customers set AGE = 30 where AGE IN (25, 27);

### **Output**

We get the following result. We can observe that the age of 3 customers has been modified −

(3 rows affected)

### **Verification**

We can verify whether the changes are reflected in a table by retrieving its contents using the SELECT statement. Following is the query to display the records in the Customers table −

select \* from customers;

The table is displayed as follows −

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 30 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 30 | Mumbai | 6500.00 |

| 5 | Hardik | 30 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

As we can see in the above table, the AGE of ‘Khilan’, ‘Chaitali’ and ‘Hardik’ has been updated to ‘30’.

## Using IN operator with NOT operator

To negate a condition, we use the NOT operator. The SQL IN operator can be used in combination with the NOT operator to exclude specific values in a WHERE clause. In other words, the absence of a list from an expression will be checked.

### **Syntax**

Following is the basic syntax of NOT IN operator −

WHERE column\_name NOT IN (value1, value2, …);

### **Example**

Now, we are trying to display all the records from the CUSTOMERS table, where the AGE is NOT equal to ‘25’, ‘23’ and ‘22’ −

select \* from CUSTOMERS

WHERE AGE NOT IN (25, 23, 22);

### **Output**

We obtain the result as given below −

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

## Using IN operator with column

We can also use the SQL IN operator with a column name to compare the values of one column to another. It is used to select the rows in which a specific value exists for the given column.

### **Example**

In the below query, we are trying to select the rows with the value of thSALARY column −

select \* from CUSTOMERS

WHERE 2000 IN (SALARY);

### **Output**

This would produce the following result −

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

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

## Using subquery with IN operator

We can use the subquery with the IN operator that is used to return records from the single column. This means that more than one column in the SELECT column list cannot be included in the subquery.

### **Syntax**

The basic syntax of the IN operator to specify a query is as follows −

WHERE column\_name IN (subquery);

Where,

* **Subquery** − This is the SELECT statement that has a result set to be tested against the expression. The IN condition evaluates to true if any of these values match the expression.

### **Example**

In the query given below we are displaying all the records from the CUSTOMERS table where the NAME of the customer is obtained with SALARY greater than 2000 −

select \* from CUSTOMERS

WHERE NAME IN (SELECT NAME from CUSTOMERS WHERE SALARY > 2000);

### **Output**

This will produce the following result −

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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