**MySQL**

**What is SQL?**

SQL is Structured Query Language, which is a computer language for storing, manipulating and retrieving data stored in a relational database.

### DDL - Data Definition Language

|  |  |
| --- | --- |
| **Sr.No.** | **Command & Description** |
| 1 | **CREATE**  Creates a new table, a view of a table, or other object in the database. |
| 2 | **ALTER**  Modifies an existing database object, such as a table. |
| 3 | **DROP**  Deletes an entire table, a view of a table or other objects in the database. |

### DML - Data Manipulation Language

|  |  |  |
| --- | --- | --- |
| **Sr.No.** |  | **Command & Description** |
| 1 |  | **SELECT**  Retrieves certain records from one or more tables. |
| 2 |  | **INSERT**  Creates a record. |
| 3 |  | **UPDATE**  Modifies records. |
| 4 |  | **DELETE**  Deletes records. |

### DCL - Data Control Language

|  |  |
| --- | --- |
| **Sr.No.** | **Command & Description** |
| 1 | **GRANT**  Gives a privilege to user. |
| 2 | **REVOKE**  Takes back privileges granted from user. |

### TCL - Transaction Control Language

# Commit, Rollback and Savepoint SQL command1.COMMIT command

To avoid that, we use the COMMIT command to mark the changes as permanent.

## 2.ROLLBACK command

This command restores the database to last commited state. It is also used with SAVEPOINT command to jump to a savepoint in an ongoing transaction.

3.SAVEPOINT command is used to temporarily save a transaction so that you can rollback to that point whenever required.

**What is RDBMS?**

RDBMS stands for **R**elational **D**atabase **M**anagement **S**ystem. RDBMS is the basis for SQL, and for all modern database systems like MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.

**What is a table?**

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

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

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

**What is a field?**

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

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

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

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

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

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

A record is a horizontal entity in a table.

**What is a column?**

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

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

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

| ADDRESS |

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

| Ahmedabad |

| Delhi |

| Kota |

| Mumbai |

| Bhopal |

| MP |

| Indore |

**What is a NULL value?**

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

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

**SQL Constraints**

Constraints are the rules enforced on data columns on a table.

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

* [NOT NULL Constraint](https://www.tutorialspoint.com/sql/sql-not-null.htm) − Ensures that a column cannot have a NULL value.
* [DEFAULT Constraint](https://www.tutorialspoint.com/sql/sql-default.htm) − Provides a default value for a column when none is specified.
* [UNIQUE Constraint](https://www.tutorialspoint.com/sql/sql-unique.htm) − Ensures that all the values in a column are different.
* [PRIMARY Key](https://www.tutorialspoint.com/sql/sql-primary-key.htm) − Uniquely identifies each row/record in a database table.
* [FOREIGN Key](https://www.tutorialspoint.com/sql/sql-foreign-key.htm) − Uniquely identifies a row/record in any another database table.
* [CHECK Constraint](https://www.tutorialspoint.com/sql/sql-check.htm) − The CHECK constraint ensures that all values in a column satisfy certain conditions.
* [INDEX](https://www.tutorialspoint.com/sql/sql-index.htm) − Used to create and retrieve data from the database very quickly.

**Database Normalization**

Database normalization is the process of efficiently organizing data in a database. There are two reasons of this normalization process −

* Eliminating redundant data, for example, storing the same data in more than one table.
* Ensuring data dependencies make sense.

[First Normal Form (1NF)](https://www.tutorialspoint.com/sql/first-normal-form.htm)

[Second Normal Form (2NF)](https://www.tutorialspoint.com/sql/second-normal-form.htm)

[Third Normal Form (3NF)](https://www.tutorialspoint.com/sql/third-normal-form.htm)

## Time for an Example

|  |  |  |
| --- | --- | --- |
| **roll\_no** | **name** | **subject** |
| 101 | Jayanth | OS, CN |
| 103 | Rakhi | Java |
| 102 | Madhura | C, C++ |

### How to solve this Problem?

It's very simple, because all we have to do is break the values into atomic values.

Here is our updated table and it now satisfies the First Normal Form.

|  |  |  |
| --- | --- | --- |
| **roll\_no** | **name** | **subject** |
| 101 | Jayanth | OS |
| 101 | Jayanth | CN |
| 103 | Rakhi | Java |
| 102 | Madhura | C |
| 102 | Madhura | C++ |

### DDL - Data Definition Language

# SQL: create command

**create** is a DDL SQL command used to create a table or a database in relational database management system.

## Creating a Database

CREATE DATABASE <DB\_NAME>;

### Example for creating Database

CREATE DATABASE Test;

## Creating a Table

Following is the syntax,

CREATE TABLE <TABLE\_NAME>

(

column\_name1 datatype1,

column\_name2 datatype2,

column\_name3 datatype3,

column\_name4 datatype4

);

### Example for creating Table

CREATE TABLE Student(

student\_id INT,

name VARCHAR(100),

age INT);

CREATE TABLE Test.Student(

student\_id INT,

name VARCHAR(100),

age INT);

|  |  |
| --- | --- |
| **Datatype** | **Use** |
| INT | used for columns which will store integer values. |
| FLOAT | used for columns which will store float values. |
| DOUBLE | used for columns which will store float values. |
| VARCHAR | used for columns which will be used to store any characters and integers, **basically a string**. |
| CHAR | used for columns which will store char values(single character). |
| DATE | used for columns which will store date values. |
| TEXT | used for columns which will store text which is generally long in length. For example, if you create a table for storing profile information of a social networking website, then for **about me** section you can have a column of type TEXT. |

# SQL: ALTER command

## ALTER Command: Add a new Column

ALTER TABLE table\_name ADD(

column\_name datatype);

ALTER TABLE student ADD(

address VARCHAR(200)

);

## ALTER Command: Add multiple new Columns

ALTER TABLE table\_name ADD(

column\_name1 datatype1,

column-name2 datatype2,

column-name3 datatype3);

Here is an Example for this,

ALTER TABLE student ADD(

father\_name VARCHAR(60),

mother\_name VARCHAR(60),

dob DATE);

# MySQL Drop a Column From Existing Table

## ****Syntax****

>> [ALTER](https://search.oracle.com/search/search?group=MySQL&q=ALTER) [TABLE](https://search.oracle.com/search/search?group=MySQL&q=TABLE) table\_name [DROP](https://search.oracle.com/search/search?group=MySQL&q=DROP) [COLUMN](https://search.oracle.com/search/search?group=MySQL&q=COLUMN) exisiting\_column\_name;

## ALTER Command: Add Column with default value

ALTER TABLE table\_name ADD(

column-name1 datatype1 DEFAULT some\_value

);

Here is an Example for this,

ALTER TABLE student ADD(

dob DATE DEFAULT '01-Jan-99'

);

## ALTER Command: Modify an existing Column

ALTER TABLE table\_name modify(

column\_name datatype

);

Here is an Example for this,

alter table mech modify column student\_name varchar(17);

Remember we added a new column address in the beginning? The above command will modify the address column of the **student** table, to now hold upto 300 characters.

## ALTER Command: Rename a Column

Using ALTER command you can rename an existing column. Following is the syntax,

ALTER TABLE table\_name RENAME column

old\_column\_name TO new\_column\_name;

Here is an example for this,

***alter table mech change old\_column\_name new\_column\_name int(50);***

The above command will rename address column to location.

# Truncate, Drop or Rename a Table

In this tutorial we will learn about the various DDL commands which are used to re-define the tables.

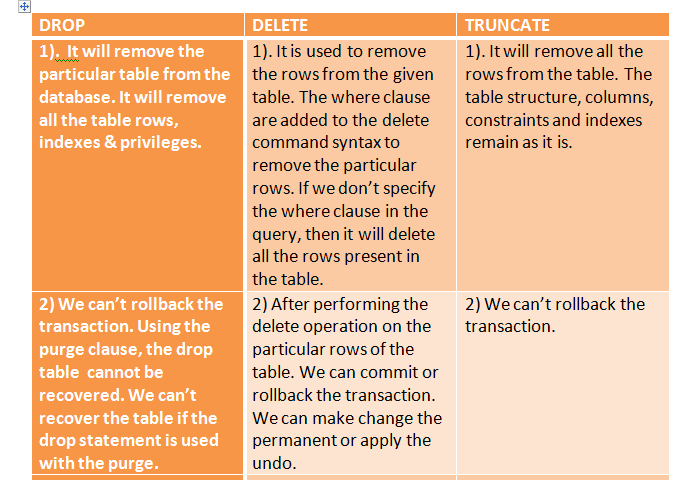
## TRUNCATE command

TRUNCATE command removes all the records from a table. But this command will not destroy the table's structure.

TRUNCATE TABLE table\_name

Here is an example explaining it,

TRUNCATE TABLE student;



The above query will delete all the records from the table **student**.

In DML commands, we will study about the DELETE command which is also more or less same as the TRUNCATE command. We will also learn about the difference between the two in that tutorial

## DROP command

DROP command completely removes a table from the database. This command will also destroy the table structure and the data stored in it. Following is its syntax,

DROP TABLE table\_name

Here is an example explaining it,

DROP TABLE student;

The above query will delete the **Student** table completely. It can also be used on Databases, to delete the complete database. For example, to drop a database,

DROP DATABASE Test;

The above query will drop the database with name **Test** from the system.

**How to Delete a row in MySQL**

To delete a row in MySQL, the DELETE FROM statement is used:

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

HERE

* DELETE FROM `table\_name` tells MySQL server to remove rows from the table ..
* [WHERE condition] is optional and is used to put a filter that restricts the number of rows affected by the MySQL DELETE row query.

If the WHERE clause is not used in the MySQL DELETE query, then all the rows in a given table will be deleted.

**Example of MySQL Delete Query**

Before we go into more details discussion the DELETE command, let's insert some sample data into the movies table to work with.

INSERT INTO `movies` (`title`, `director`, `year\_released`, `category\_id`) VALUES ('The Great Dictator', 'Chalie Chaplie', 1920, 7);

INSERT INTO `movies` (`title`, `director`, `category\_id`) VALUES ('sample movie', 'Anonymous', 8);

INSERT INTO movies (`title`, `director`, `year\_released`, `category\_id`) VALUES ('movie 3', 'John Brown', 1920, 8);

Executing the above script adds three (3) movies into the movies table. Before we go any further into our lesson, let's get all the movies in our table. The script shown below does that.

SELECT \* FROM `movies`;

Executing the above script gives us the following results.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **movie\_id** | **itle** | **director** | **year\_released** | **category\_id** |
| 1 | Pirates of the Caribean 4 | Rob Marshall | 2011 | 1 |
| 2 | Forgetting Sarah Marshal | Nicholas Stoller | 2008 | 2 |
| 3 | X-Men | NULL | 2008 | NULL |
| 4 | Code Name Black | Edgar Jimz | 2010 | NULL |
| 5 | Daddy's Little Girls | NULL | 2007 | 8 |
| 6 | Angels and Demons | NULL | 2007 | 6 |
| 7 | Davinci Code | NULL | 2007 | 6 |
| 9 | Honey mooners | John Schultz | 2005 | 8 |
| 16 | 67% Guilty | NULL | 2012 | NULL |
| 18 | The Great Dictator | Chalie Chaplie | 1920 | 7 |
| 19 | sample movie | Anonymous | NULL | 8 |
| 20 | movie 3 | John Brown | 1920 | 8 |

 Let's suppose that the Myflix video library no longer wishes to be renting out "The Great Dictator" to its members and they want it removed from the database. Its movie id is 18, we can use the script shown below to delete its row from the movies table.

DELETE FROM `movies` WHERE `movie\_id` = 18;

## RENAME query

RENAME command is used to set a new name for any existing table. Following is the syntax,

RENAME TABLE old\_table\_name to new\_table\_name

Here is an example explaining it.

RENAME TABLE student to students\_info;

## Introduction to the MySQL TRUNCATE TABLE statement

The MySQL TRUNCATE TABLE statement allows you to delete all data in a table.

Logically, the TRUNCATE TABLE statement is like a [DELETE](https://www.mysqltutorial.org/mysql-delete-statement.aspx) statement without a [WHERE](https://www.mysqltutorial.org/mysql-where/) clause that deletes all rows from a table, or a sequence of [DROP TABLE](https://www.mysqltutorial.org/mysql-drop-table) and [CREATE TABLE](https://www.mysqltutorial.org/mysql-create-table/) statements.

However, the TRUNCATE TABLE statement is more efficient than the DELETE statement because it drops and recreates the table instead of deleting rows one by one.

Here is the basic syntax of theTRUNCATE TABLE statement:

**TRUNCATE [TABLE] table\_name;**

## INSERT command

Insert command is used to insert data into a table. Following is its general syntax,

INSERT INTO table\_name VALUES(data1, data2, ...)

Lets see an example,

Consider a table **student** with the following fields.

|  |  |  |
| --- | --- | --- |
| **s\_id** | **name** | **age** |

INSERT INTO student VALUES(101, 'Adam', 15);

The above command will insert a new record into **student** table.

|  |  |  |
| --- | --- | --- |
| **s\_id** | **name** | **age** |
| 101 | Adam | 15 |

INSERT INTO student(id, name) values(102, 'Alex');

### Insert NULL value to a column

Both the statements below will insert NULL value into **age** column of the **student** table.

INSERT INTO student(id, name) values(102, 'Alex');

Or,

INSERT INTO Student VALUES(102,'Alex', null);

The above command will insert only two column values and the other column is set to null.

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex |  |

### Insert Default value to a column

INSERT INTO Student VALUES(103,'Chris', default)

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex |  |
| 103 | chris | 14 |

Suppose the column age in our tabel has a default value of 14.

Also, if you run the below query, it will insert default value into the age column, whatever the default value may be.

INSERT INTO Student VALUES(103,'Chris')

# Using UPDATE SQL command

## UPDATE command

UPDATE table\_name SET column\_name = new\_value WHERE some\_condition;

Lets take a sample table **student**,

|  |  |  |
| --- | --- | --- |
| **student\_id** | **name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex |  |
| 103 | chris | 14 |

UPDATE student SET age=18 WHERE student\_id=102;

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |
| 103 | chris | 14 |

In the above statement, if we do not use the WHERE clause, then our update query will update age for all the columns of the table to **18**.

### Updating Multiple Columns

We can also update values of multiple columns using a single UPDATE statement.

UPDATE student SET name='Abhi', age=17 where s\_id=103;

The above command will update two columns of the record which has s\_id 103.

|  |  |  |
| --- | --- | --- |
| **s\_id** | **name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |
| 103 | Abhi | 17 |

### UPDATE Command: Incrementing Integer Value

When we have to update any integer value in a table, then we can fetch and update the value in the table in a single statement.

For example, if we have to update the age column of **student** table every year for every student, then we can simply run the following UPDATE statement to perform the following operation:

UPDATE student SET age = age+1;

As you can see, we have used age = age + 1 to increment the value of age by 1.

**NOTE:** This style only works for integer values.

# Using DELETE SQL command

When you ask any question in [Studytonight's Forum](https://www.studytonight.com/studyroom/) it gets saved into a table. And using the **Delete** option, you can even delete a question asked by you. How do you think that works? Yes, using the Delete DML command.

Let's study about the syntax and the usage of the Delete command.

## DELETE command

DELETE command is used to delete data from a table.

Following is its general syntax,

DELETE FROM table\_name;

Let's take a sample table **student**:

|  |  |  |
| --- | --- | --- |
| **s\_id** | **name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |
| 103 | Abhi | 17 |

### Delete all Records from a Table

DELETE FROM student;

The above command will delete all the records from the table **student**.

### Delete a particular Record from a Table

In our **student** table if we want to delete a single record, we can use the WHERE clause to provide a condition in our DELETE statement.

DELETE FROM student WHERE s\_id=103;

The above command will delete the record where s\_id is 103 from the table **student**.

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |

## TRUNCATE TABLE

The TRUNCATE TABLE command deletes the data inside a table, but not the table itself.

The following SQL truncates the table "Categories":

### Example

TRUNCATE TABLE Categories;

# TCL

# Commit, Rollback and Savepoint SQL commands

Transaction Control Language(TCL) commands are used to manage transactions in the database. These are used to manage the changes made to the data in a table by DML statements. It also allows statements to be grouped together into logical transactions.

## COMMIT command

COMMIT command is used to permanently save any transaction into the database.

When we use any DML command like INSERT, UPDATE or DELETE, the changes made by these commands are not permanent, until the current session is closed, the changes made by these commands can be rolled back.

To avoid that, we use the COMMIT command to mark the changes as permanent.

Following is commit command's syntax,

COMMIT;

## ROLLBACK command

This command restores the database to last commited state. It is also used with SAVEPOINT command to jump to a savepoint in an ongoing transaction.

If we have used the UPDATE command to make some changes into the database, and realise that those changes were not required, then we can use the ROLLBACK command to rollback those changes, if they were not commited using the COMMIT command.

Following is rollback command's syntax,

ROLLBACK TO savepoint\_name;

## SAVEPOINT command

SAVEPOINT command is used to temporarily save a transaction so that you can rollback to that point whenever required.

Following is savepoint command's syntax,

SAVEPOINT savepoint\_name;

In short, using this command we can **name** the different states of our data in any table and then rollback to that state using the ROLLBACK command whenever required.

### Using Savepoint and Rollback

Following is the table **class**,

|  |  |
| --- | --- |
| **id** | **name** |
| 1 | Abhi |
| 2 | Adam |
| 4 | Alex |

Lets use some SQL queries on the above table and see the results.

INSERT INTO class VALUES(5, 'Rahul');

START TRANSACTION;

COMMIT;

UPDATE class SET name = 'Abhijit' WHERE id = '5';

SAVEPOINT A;

INSERT INTO class VALUES(6, 'Chris');

SAVEPOINT B;

INSERT INTO class VALUES(7, 'Bravo');

SAVEPOINT C;

SELECT \* FROM class;

**NOTE:** SELECT statement is used to show the data stored in the table.

The resultant table will look like,

|  |  |
| --- | --- |
| **id** | **name** |
| 1 | Abhi |
| 2 | Adam |
| 4 | Alex |
| 5 | Abhijit |
| 6 | Chris |
| 7 | Bravo |

Now let's use the ROLLBACK command to roll back the state of data to the **savepoint B**.

ROLLBACK TO B;

SELECT \* FROM class;

Now our **class** table will look like,

|  |  |
| --- | --- |
| **id** | **name** |
| 1 | Abhi |
| 2 | Adam |
| 4 | Alex |
| 5 | Abhijit |
| 6 | Chris |

Now let's again use the ROLLBACK command to roll back the state of data to the **savepoint A**

ROLLBACK TO A;

SELECT \* FROM class;

Now the table will look like,

|  |  |
| --- | --- |
| **id** | **name** |
| 1 | Abhi |
| 2 | Adam |
| 4 | Alex |
| 5 | Abhijit |

So now you know how the commands COMMIT, ROLLBACK and SAVEPOINT works.

# Using GRANT and REVOKE

Data Control Language(DCL) is used to control privileges in Database. To perform any operation in the database, such as for creating tables, sequences or views, a user needs privileges. Privileges are of two types,

* **System:** This includes permissions for creating session, table, etc and all types of other system privileges.
* **Object:** This includes permissions for any command or query to perform any operation on the database tables.

In DCL we have two commands,

* GRANT: Used to provide any user access privileges or other priviliges for the database.
* REVOKE: Used to take back permissions from any user.

# SELECT SQL Query

SELECT query is used to retrieve data from a table. It is the most used SQL query.

### Syntax of SELECT query

SELECT query is used to retieve records from a table. We can specify the names of the columns which we want in the resultset.

SELECT

column\_name1,

column\_name2,

column\_name3,

...

column\_nameN

FROM table\_name;

### Time for an Example

Consider the following **student** table,

|  |  |  |  |
| --- | --- | --- | --- |
| **s\_id** | **name** | **age** | **address** |
| 101 | Adam | 15 | Chennai |
| 102 | Alex | 18 | Delhi |
| 103 | Abhi | 17 | Banglore |
| 104 | Ankit | 22 | Mumbai |

SELECT s\_id, name, age FROM student;

The above query will fetch information of s\_id, name and age columns of the **student** table and display them,

|  |  |  |
| --- | --- | --- |
| **s\_id** | **name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |
| 103 | Abhi | 17 |
| 104 | Ankit | 22 |

As you can see the data from address column is absent, because we did not specif it in our SELECT query.

SELECT \* FROM student;

The above query will show all the records of **student** table, that means it will show complete dataset of the table.

|  |  |  |  |
| --- | --- | --- | --- |
| **s\_id** | **name** | **age** | **address** |
| 101 | Adam | 15 | Chennai |
| 102 | Alex | 18 | Delhi |
| 103 | Abhi | 17 | Banglore |
| 104 | Ankit | 22 | Mumbai |

### Select a particular record based on a condition

We can use the [WHERE clause](https://www.studytonight.com/dbms/where-clause.php" \t "_blank) to set a condition,

SELECT \* FROM student WHERE name = 'Abhi';

The above query will return the following result,

|  |  |  |  |
| --- | --- | --- | --- |
| 103 | Abhi | 17 | Rohtak |

### Performing Simple Calculations using SELECT Query

Consider the following **employee** table.

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 101 | Adam | 26 | 5000 |
| 102 | Ricky | 42 | 8000 |
| 103 | Abhi | 25 | 10000 |
| 104 | Rohan | 22 | 5000 |

Here is our SELECT query,

SELECT eid, name, salary+3000 FROM employee;

The above command will display a new column in the result, with **3000** added into existing salaries of the employees.

|  |  |  |
| --- | --- | --- |
| **eid** | **name** | **salary+3000** |
| 101 | Adam | 8000 |
| 102 | Ricky | 11000 |
| 103 | Abhi | 13000 |
| 104 | Rohan | 8000 |

So you can also perform simple mathematical operations on the data too using the SELECT query to fetch data from table.

# Using the WHERE SQL clause

### Syntax for WHERE clause

Here is how you can use the WHERE clause with a DELETE statement, or any other statement,

DELETE FROM table\_name WHERE [condition];

The WHERE clause is used at the end of any SQL query, to specify a condition for execution.

### Time for an Example

Consider a table **student**,

|  |  |  |  |
| --- | --- | --- | --- |
| **s\_id** | **name** | **age** | **address** |
| 101 | Adam | 15 | Chennai |
| 102 | Alex | 18 | Delhi |
| 103 | Abhi | 17 | Banglore |
| 104 | Ankit | 22 | Mumbai |

Now we will use the SELECT statement to display data of the table, based on a condition, which we will add to our SELECT query using WHERE clause.

Let's write a simple SQL query to display the record for student with s\_id as 101.

SELECT s\_id,

name,

age,

address

FROM student WHERE s\_id = 101;

Following will be the result of the above query.

|  |  |  |  |
| --- | --- | --- | --- |
| **s\_id** | **name** | **age** | **address** |
| 101 | Adam | 15 | Noida |

#### Applying condition on Text Fields

In the above example we have applied a condition to an integer value field, but what if we want to apply the condition on name field. In that case we must enclose the value in single quote ' '. Some databases even accept double quotes, but single quotes is accepted by all.

SELECT s\_id,

name,

age,

address

FROM student WHERE name = 'Adam';

Following will be the result of the above query.

|  |  |  |  |
| --- | --- | --- | --- |
| **s\_id** | **name** | **age** | **address** |
| 101 | Adam | 15 | Noida |

## Operators for WHERE clause condition

Following is a list of operators that can be used while specifying the WHERE clause condition.

#### WHERE clause Syntax

The basic syntax for the WHERE clause when used in a SELECT statement is as follows.

SELECT \* FROM tableName WHERE condition;

**HERE**

* **"SELECT \* FROM tableName"** is the standard SELECT statement
* **"WHERE"** is the keyword that restricts our select query result set and **"condition"** is the filter to be applied on the results. The filter could be a range, single value or sub query.

Let's now look at a **practical example**.

Suppose we want to get a member's personal details from members table given the membership number 1, we would use the following script to achieve that.

SELECT \* FROM `members` WHERE `membership\_number` = 1;

Executing the above script in MySQL workbench on the "myflixdb" would produce the following results.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **membership\_number** | **full\_names** | **gender** | **date\_of\_birth** | **physical\_address** | **postal\_address** | **contct\_number** | **email** |
| 1 | Janet Jones | Female | 21-07-1980 | First Street Plot No 4 | Private Bag | 0759 253 542 | [janetjones@yagoo.cm](mailto:janetjones@yagoo.cm) |

### WHERE clause combined with - *****AND*****LOGICAL Operator

The WHERE clause when used together with the AND logical operator, is only executed if ALL filter criteria specified are met.

Let's now look at a practical example - Suppose we want to get a list of all the movies in category 2 that were released in 2008, we would use the script shown below is achieve that.

SELECT \* FROM `movies` WHERE `category\_id` = 2 AND `year\_released` = 2008;

Executing the above script in MySQL workbench against the "myflixdb" produces the following results.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **movie\_id** | **title** | **director** | **year\_released** | **category\_id** |
| 2 | Forgetting Sarah Marshal | Nicholas Stoller | 2008 | 2 |

### WHERE clause combined with - *****OR*****LOGICAL Operator

The WHERE clause when used together with the OR operator, is only executed if any or the entire specified filter criteria is met.

The following script gets all the movies in either category 1 or category 2

SELECT \* FROM `movies` WHERE `category\_id` = 1 OR `category\_id` = 2;

Executing the above script in MySQL workbench against the "myflixdb" produces the following results.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **movie\_id** | **title** | **director** | **year\_released** | **category\_id** |
| 1 | Pirates of the Caribean 4 | Rob Marshall | 2011 | 1 |
| 2 | Forgetting Sarah Marshal | Nicholas Stoller | 2008 | 2 |

### WHERE clause combined with - *****IN*****Keyword

The WHERE clause when used together with the IN keyword only affects the rows whose values matches the list of values provided in the IN keyword. IN helps reduces number of OR clauses you may have to use

The following query gives rows where membership\_number is either 1 , 2 or 3

SELECT \* FROM `members` WHERE `membership\_number` IN (1,2,3);

Executing the above script in MySQL workbench against the "myflixdb" produces the following results.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **membership\_number** | **full\_names** | **gender** | **date\_of\_birth** | **physical\_address** | **postal\_address** | **contct\_number** | **email** |
| 1 | Janet Jones | Female | 21-07-1980 | First Street Plot No 4 | Private Bag | 0759 253 542 | [janetjones@yagoo.cm](mailto:janetjones@yagoo.cm) |
| 2 | Janet Smith Jones | Female | 23-06-1980 | Melrose 123 | NULL | NULL | [jj@fstreet.com](mailto:jj@fstreet.com) |
| 3 | Robert Phil | Male | 12-07-1989 | 3rd Street 34 | NULL | 12345 | [rm@tstreet.com](mailto:rm@tstreet.com) |

### WHERE clause combined with - NOT IN Keyword

The  WHERE clause when used together with the NOT IN keyword  DOES NOT affects the rows whose values matches the list of values provided in the NOT IN keyword.

The following query gives rows where membership\_number is NOT  1 , 2 or 3

SELECT \* FROM `members` WHERE `membership\_number` NOT IN (1,2,3);

Executing the above script in MySQL workbench against the "myflixdb" produces the following results.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **membership\_number** | **full\_names** | **gender** | **date\_of\_birth** | **physical\_address** | **postal\_address** | **contct\_number** | **email** |
| 4 | Gloria Williams | Female | 14-02-1984 | 2nd Street 23 | NULL | NULL | NULL |

### WHERE clause combined with - ****COMPARISON OPERATORS****

The less than (), equal to (=), not equal to () comparison operators can be  used with the Where clause

### *****=*****Equal To

The following script gets all the female members from the members table using the equal to comparison operator.

SELECT \* FROM `members` WHERE `gender` = 'Female';

Executing the above script in MySQL workbench against the "myflixdb" produces the following results.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **membership\_number** | **full\_names** | **gender** | **date\_of\_birth** | **physical\_address** | **postal\_address** | **contct\_number** | **email** |
| 1 | Janet Jones | Female | 21-07-1980 | First Street Plot No 4 | Private Bag | 0759 253 542 | [janetjones@yagoo.cm](mailto:janetjones@yagoo.cm) |
| 2 | Janet Smith Jones | Female | 23-06-1980 | Melrose 123 | NULL | NULL | [jj@fstreet.com](mailto:jj@fstreet.com) |
| 4 | Gloria Williams | Female | 14-02-1984 | 2nd Street 23 | NULL | NULL | NULL |

### ****>****Greater than

The following script gets all the payments that are greater than 2,000 from the payments table.

SELECT \* FROM `payments` WHERE `amount\_paid` >2000;

Executing the above script in MySQL workbench against the "myflixdb" produces the following results.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **payment\_id** | **membership\_number** | **payment\_date** | **description** | **amount\_paid** | **external\_reference\_number** |
| 1 | 1 | 23-07-2012 | Movie rental payment | 2500 | 11 |
| 3 | 3 | 30-07-2012 | Movie rental payment | 6000 | NULL |

### < > Not Equal To

The following script gets all the movies whose category id is not 1.

SELECT \* FROM `movies` WHERE `category\_id`<> 1;

Executing the above script in MySQL workbench against the "myflixdb" produces the following results.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **movie\_id** | **title** | **director** | **year\_released** | **category\_id** |
| 2 | Forgetting Sarah Marshal | Nicholas Stoller | 2008 | 2 |
| 5 | Daddy's Little Girls | NULL | 2007 | 8 |
| 6 | Angels and Demons | NULL | 2007 | 6 |
| 7 | Davinci Code | NULL | 2007 | 6 |
| 9 | Honey mooners | John Schultz | 2005 | 8 |

|  |  |
| --- | --- |
| **Operator** | **Description** |
| = | Equal to |
| != | Not Equal to |
| < | Less than |
| > | Greater than |
| <= | Less than or Equal to |
| >= | Greate than or Equal to |
| BETWEEN | Between a specified range of values |
| LIKE | This is used to search for a pattern in value. |
| IN | In a given set of values |

# SQL BETWEEN Operator

## The SQL BETWEEN Operator

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

The BETWEEN operator is inclusive: begin and end values are included.

### BETWEEN Syntax

SELECT column\_name(s)  
FROM table\_name  
WHERE column\_name BETWEEN value1 AND value2;

## Demo Database

Below is a selection from the "Products" table in the Northwind sample database:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ProductID** | **ProductName** | **SupplierID** | **CategoryID** | **Unit** | **Price** |
| 1 | Chais | 1 | 1 | 10 boxes x 20 bags | 18 |
| 2 | Chang | 1 | 1 | 24 - 12 oz bottles | 19 |
| 3 | Aniseed Syrup | 1 | 2 | 12 - 550 ml bottles | 10 |
| 4 | Chef Anton's Cajun Seasoning | 1 | 2 | 48 - 6 oz jars | 22 |
| 5 | Chef Anton's Gumbo Mix | 1 | 2 | 36 boxes | 21.35 |

## BETWEEN Example

The following SQL statement selects all products with a price BETWEEN 10 and 20:

### Example

SELECT \* FROM Products  
WHERE Price BETWEEN 10 AND 20;

## NOT BETWEEN Example

To display the products outside the range of the previous example, use NOT BETWEEN:

### Example

SELECT \* FROM Products  
WHERE Price NOT BETWEEN 10 AND 20;

## BETWEEN with IN Example

The following SQL statement selects all products with a price BETWEEN 10 and 20. In addition; do not show products with a CategoryID of 1,2, or 3:

### Example

SELECT \* FROM Products  
WHERE Price BETWEEN 10 AND 20  
AND CategoryID NOT IN (1,2,3);

## BETWEEN Text Values Example

The following SQL statement selects all products with a ProductName BETWEEN Carnarvon Tigers and Mozzarella di Giovanni:

### Example

SELECT \* FROM Products  
WHERE ProductName BETWEEN 'Carnarvon Tigers' AND 'Mozzarella di Giovanni'  
ORDER BY ProductName;

The following SQL statement selects all products with a ProductName BETWEEN Carnarvon Tigers and Chef Anton's Cajun Seasoning:

### Example

SELECT \* FROM Products  
WHERE ProductName BETWEEN "Carnarvon Tigers" AND "Chef Anton's Cajun Seasoning"  
ORDER BY ProductName;

## NOT BETWEEN Text Values Example

The following SQL statement selects all products with a ProductName NOT BETWEEN Carnarvon Tigers and Mozzarella di Giovanni:

### Example

SELECT \* FROM Products  
WHERE ProductName NOT BETWEEN 'Carnarvon Tigers' AND 'Mozzarella di Giovanni'  
ORDER BY ProductName;

## Sample Table

Below is a selection from the "Orders" table in the Northwind sample database:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OrderID** | **CustomerID** | **EmployeeID** | **OrderDate** | **ShipperID** |
| 10248 | 90 | 5 | 7/4/1996 | 3 |
| 10249 | 81 | 6 | 7/5/1996 | 1 |
| 10250 | 34 | 4 | 7/8/1996 | 2 |
| 10251 | 84 | 3 | 7/9/1996 | 1 |
| 10252 | 76 | 4 | 7/10/1996 | 2 |

## BETWEEN Dates Example

The following SQL statement selects all orders with an OrderDate BETWEEN '01-July-1996' and '31-July-1996':

### Example

SELECT \* FROM Orders  
WHERE OrderDate BETWEEN #01/07/1996# AND #31/07/1996#;

OR:

### Example

SELECT \* FROM Orders  
WHERE OrderDate BETWEEN '1996-07-01' AND '1996-07-31';

# SQL MIN() and MAX() Functions

## The SQL MIN() and MAX() Functions

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

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

### MIN() Syntax

SELECT MIN(column\_name)  
FROM table\_name  
WHERE condition;

### MAX() Syntax

SELECT MAX(column\_name)  
FROM table\_name  
WHERE condition;

## Demo Database

Below is a selection from the "Products" table in the Northwind sample database:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ProductID** | **ProductName** | **SupplierID** | **CategoryID** | **Unit** | **Price** |
| 1 | Chais | 1 | 1 | 10 boxes x 20 bags | 18 |
| 2 | Chang | 1 | 1 | 24 - 12 oz bottles | 19 |
| 3 | Aniseed Syrup | 1 | 2 | 12 - 550 ml bottles | 10 |
| 4 | Chef Anton's Cajun Seasoning | 2 | 2 | 48 - 6 oz jars | 22 |
| 5 | Chef Anton's Gumbo Mix | 2 | 2 | 36 boxes | 21.35 |

## MIN() Example

The following SQL statement finds the price of the cheapest product:

### Example

SELECT MIN(Price) AS SmallestPrice  
FROM Products;

## MAX() Example

The following SQL statement finds the price of the most expensive product:

### Example

SELECT MAX(Price) AS LargestPrice  
FROM Products;

Top of Form

# Test SQL COUNT(), AVG() and SUM() Functions

## The SQL COUNT(), AVG() and SUM() Functions

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

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

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

### COUNT() Syntax

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

### AVG() Syntax

SELECT AVG(column\_name)  
FROM table\_name  
WHERE condition;

### SUM() Syntax

SELECT SUM(column\_name)  
FROM table\_name  
WHERE condition;

## Demo Database

Below is a selection from the "Products" table in the Northwind sample database:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ProductID** | **ProductName** | **SupplierID** | **CategoryID** | **Unit** | **Price** |
| 1 | Chais | 1 | 1 | 10 boxes x 20 bags | 18 |
| 2 | Chang | 1 | 1 | 24 - 12 oz bottles | 19 |
| 3 | Aniseed Syrup | 1 | 2 | 12 - 550 ml bottles | 10 |
| 4 | Chef Anton's Cajun Seasoning | 2 | 2 | 48 - 6 oz jars | 22 |
| 5 | Chef Anton's Gumbo Mix | 2 | 2 | 36 boxes | 21.35 |

## COUNT() Example

The following SQL statement finds the number of products:

### Example

SELECT COUNT(ProductID)  
FROM Products;

**Note:** NULL values are not counted.

## AVG() Example

The following SQL statement finds the average price of all products:

### Example

SELECT AVG(Price)  
FROM Products;

**Note:** NULL values are ignored.

## Demo Database

Below is a selection from the "OrderDetails" table in the Northwind sample database:

|  |  |  |  |
| --- | --- | --- | --- |
| **OrderDetailID** | **OrderID** | **ProductID** | **Quantity** |
| 1 | 10248 | 11 | 12 |
| 2 | 10248 | 42 | 10 |
| 3 | 10248 | 72 | 5 |
| 4 | 10249 | 14 | 9 |
| 5 | 10249 | 51 | 40 |

## SUM() Example

The following SQL statement finds the sum of the "Quantity" fields in the "OrderDetails" table:

### Example

SELECT SUM(Quantity)  
FROM OrderDetails;

**Note:** NULL values are ignored.

## Yours

# SQL UNION Operator

## The SQL UNION Operator

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

* Each SELECT statement within UNION must have the same number of columns
* The columns must also have similar data types
* The columns in each SELECT statement must also be in the same order

### UNION Syntax

SELECT column\_name(s) FROM table1  
UNION  
SELECT column\_name(s) FROM table2;

### UNION ALL Syntax

The UNION operator selects only distinct (unique) values by default. To allow duplicate values, use UNION ALL:

SELECT column\_name(s) FROM table1  
UNION ALL  
SELECT column\_name(s) FROM table2;

**Note:** The column names in the result-set are usually equal to the column names in the first SELECT statement in the UNION.

## Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Customers" table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |

And a selection from the "Suppliers" table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **SupplierID** | **SupplierName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 1 | Exotic Liquid | Charlotte Cooper | 49 Gilbert St. | London | EC1 4SD | UK |
| 2 | New Orleans Cajun Delights | Shelley Burke | P.O. Box 78934 | New Orleans | 70117 | USA |
| 3 | Grandma Kelly's Homestead | Regina Murphy | 707 Oxford Rd. | Ann Arbor | 48104 | USA |

## SQL UNION Example

The following SQL statement returns the cities (only distinct values) from both the "Customers" and the "Suppliers" table:

### Example

SELECT City FROM Customers  
UNION  
SELECT City FROM Suppliers  
ORDER BY City;

**Note:** If some customers or suppliers have the same city, each city will only be listed once, because UNION selects only **distinct values**. Use UNION ALL to also select duplicate values!

## SQL UNION ALL Example

The following SQL statement returns the cities (duplicate values also) from both the "Customers" and the "Suppliers" table:

### Example

SELECT City FROM Customers  
UNION ALL  
SELECT City FROM Suppliers  
ORDER BY City;

## SQL UNION With WHERE

The following SQL statement returns the German cities (only distinct values) from both the "Customers" and the "Suppliers" table:

### Example

SELECT City, Country FROM Customers  
WHERE Country='Germany'  
UNION  
SELECT City, Country FROM Suppliers  
WHERE Country='Germany'  
ORDER BY City;

## SQL UNION ALL With WHERE

The following SQL statement returns the German cities (duplicate values also) from both the "Customers" and the "Suppliers" table:

### Example

SELECT City, Country FROM Customers  
WHERE Country='Germany'  
UNION ALL  
SELECT City, Country FROM Suppliers  
WHERE Country='Germany'  
ORDER BY City;

## Another UNION Example

The following SQL statement lists all customers and suppliers:

### Example

SELECT 'Customer' AS Type, ContactName, City, Country  
FROM Customers  
UNION  
SELECT 'Supplier', ContactName, City, Country  
FROM Suppliers;

# SQL CASE Statement

## The SQL CASE Statement

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

If there is no ELSE part and no conditions are true, it returns NULL.

## CASE Syntax

CASE  
    WHEN condition1 THEN result1  
    WHEN condition2 THEN result2  
    WHEN conditionN THEN resultN  
    ELSE result  
END;

## Demo Database

Below is a selection from the "OrderDetails" table in the Northwind sample database:

|  |  |  |  |
| --- | --- | --- | --- |
| **OrderDetailID** | **OrderID** | **ProductID** | **Quantity** |
| 1 | 10248 | 11 | 12 |
| 2 | 10248 | 42 | 10 |
| 3 | 10248 | 72 | 5 |
| 4 | 10249 | 14 | 9 |
| 5 | 10249 | 51 | 40 |

## SQL CASE Examples

The following SQL goes through conditions and returns a value when the first condition is met:

### Example

SELECT OrderID, Quantity,  
CASE  
    WHEN Quantity > 30 THEN 'The quantity is greater than 30'  
    WHEN Quantity = 30 THEN 'The quantity is 30'  
    ELSE 'The quantity is under 30'  
END

AS QuantityResult  
FROM OrderDetails;

The following SQL will order the customers by City. However, if City is NULL, then order by Country:

### Example

SELECT Customer Name, City, Country  
FROM Customers  
ORDER BY  
(CASE  
    WHEN City IS NULL THEN Country  
    ELSE City  
END);

# MySQL ORDER BY

**Summary**: in this tutorial, you will learn how to sort a result set using the MySQL ORDER BY clause.

## Introduction to MySQL ORDER BY clause

When you use the [SELECT](https://www.mysqltutorial.org/mysql-select-statement-query-data.aspx) statement to query data from a table, the result set is not sorted. It means that the rows in the result set can be in any order.

To sort the result set, you add the ORDER BY clause to the SELECT statement. The following illustrates the syntax of the ORDER BY  clause:

**Syntax**

SELECT

select\_list

FROM

table\_name

ORDER BY

column1 [ASC|DESC],

column2 [ASC|DESC],

...;

**Real Ex**:

SELECT

contactLastname,

contactFirstname

FROM

customers

ORDER BY

contactLastname DESC;

### B) Using MySQL ORDER BY clause to sort values in multiple columns example

SELECT

contactLastname,

contactFirstname

FROM

customers

ORDER BY

contactLastname DESC,

contactFirstname ASC;

# SQL Constraints

SQL constraints are used to specify rules for data in a table.

## SQL Create Constraints

Constraints can be specified when the table is created with the CREATE TABLE statement, or after the table is created with the ALTER TABLE statement.

### Syntax

CREATE TABLE table\_name (  
    column1 datatype *constraint*,  
    column2 datatype *constraint*,  
    column3 datatype *constraint*,  
    ....  
);

## SQL Constraints

SQL constraints are used to specify rules for the data in a table.

Constraints are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the table. If there is any violation between the constraint and the data action, the action is aborted.

Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to the whole table.

The following constraints are commonly used in SQL:

* [**NOT NULL**](https://www.w3schools.com/sql/sql_notnull.asp) - Ensures that a column cannot have a NULL value
* [**UNIQUE**](https://www.w3schools.com/sql/sql_unique.asp) - Ensures that all values in a column are different
* [**PRIMARY KEY**](https://www.w3schools.com/sql/sql_primarykey.asp) - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
* [**FOREIGN KEY**](https://www.w3schools.com/sql/sql_foreignkey.asp) - Uniquely identifies a row/record in another table
* [**CHECK**](https://www.w3schools.com/sql/sql_check.asp) - Ensures that all values in a column satisfies a specific condition
* [**DEFAULT**](https://www.w3schools.com/sql/sql_default.asp) - Sets a default value for a column when no value is specified
* [**INDEX**](https://www.w3schools.com/sql/sql_create_index.asp) - Used to create and retrieve data from the database very quickly

# SQL NOT NULL Constraint

## SQL NOT NULL Constraint

By default, a column can hold NULL values.

The NOT NULL constraint enforces a column to NOT accept NULL values.

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

## SQL NOT NULL on CREATE TABLE

The following SQL ensures that the "ID", "LastName", and "FirstName" columns will NOT accept NULL values when the "Persons" table is created:

### Example

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

## SQL NOT NULL on ALTER TABLE

To create a NOT NULL constraint on the "Age" column when the "Persons" table is already created, use the following SQL:

ALTER TABLE Persons  
MODIFY Age int NOT NULL;

# SQL UNIQUE Constraint

## SQL UNIQUE Constraint

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

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

A PRIMARY KEY constraint automatically has a UNIQUE constraint.

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

## SQL UNIQUE Constraint on CREATE TABLE

The following SQL creates a UNIQUE constraint on the "ID" column when the "Persons" table is created:

**SQL Server / Oracle / MS Access:**

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

**MySQL:**

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

To name a UNIQUE constraint, and to define a UNIQUE constraint on multiple columns, use the following SQL syntax:

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

CREATE TABLE Persons (  
    ID int NOT NULL,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int,  
    CONSTRAINT UC\_Person UNIQUE (ID,LastName)  
);

## SQL UNIQUE Constraint on ALTER TABLE

To create a UNIQUE constraint on the "ID" column when the table is already created, use the following SQL:

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

ALTER TABLE Persons  
ADD UNIQUE (ID);

To name a UNIQUE constraint, and to define a UNIQUE constraint on multiple columns, use the following SQL syntax:

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

ALTER TABLE Persons  
ADD CONSTRAINT UC\_Person UNIQUE (ID,LastName);

## DROP a UNIQUE Constraint

To drop a UNIQUE constraint, use the following SQL:

**MySQL:**

ALTER TABLE Persons  
DROP INDEX UC\_Person;

**SQL Server / Oracle / MS Access:**

ALTER TABLE Persons  
DROP CONSTRAINT UC\_Person;

# SQL PRIMARY KEY Constraint

## SQL PRIMARY KEY Constraint

The PRIMARY KEY constraint uniquely identifies each record in a table.

Primary keys must contain UNIQUE values, and cannot contain NULL values.

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

## SQL PRIMARY KEY on CREATE TABLE

The following SQL creates a PRIMARY KEY on the "ID" column when the "Persons" table is created:

**MySQL:**

CREATE TABLE Persons (  
    ID int NOT NULL,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int,  
    PRIMARY KEY (ID)  
);

**SQL Server / Oracle / MS Access:**

CREATE TABLE Persons (  
    ID int NOT NULL PRIMARY KEY,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int  
);

To allow naming of a PRIMARY KEY constraint, and for defining a PRIMARY KEY constraint on multiple columns, use the following SQL syntax:

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

CREATE TABLE Persons (  
    ID int NOT NULL,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int,  
    CONSTRAINT PK\_Person PRIMARY KEY (ID,LastName)  
);

**Note:** In the example above there is only ONE PRIMARY KEY (PK\_Person). However, the VALUE of the primary key is made up of TWO COLUMNS (ID + LastName).

## SQL PRIMARY KEY on ALTER TABLE

To create a PRIMARY KEY constraint on the "ID" column when the table is already created, use the following SQL:

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

ALTER TABLE Persons  
ADD PRIMARY KEY (ID);

To allow naming of a PRIMARY KEY constraint, and for defining a PRIMARY KEY constraint on multiple columns, use the following SQL syntax:

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

ALTER TABLE Persons  
ADD CONSTRAINT PK\_Person PRIMARY KEY (ID,LastName);

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

## DROP a PRIMARY KEY Constraint

To drop a PRIMARY KEY constraint, use the following SQL:

**MySQL:**

ALTER TABLE Persons  
DROP PRIMARY KEY;

# SQL FOREIGN KEY Constraint

## SQL FOREIGN KEY Constraint

A FOREIGN KEY is a key used to link two tables together.

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

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

Look at the following two tables:

"Persons" table:

|  |  |  |  |
| --- | --- | --- | --- |
| **PersonID** | **FirstName** | **LastName** | **Age** |
| 101 | Jayanth | V | 30 |
| 102 | Ragini | P | 23 |
| 103 | Pooja | B | 20 |

"Orders" table:

|  |  |  |
| --- | --- | --- |
| **OrderID** | **OrderNumber** | **PersonID** |
| 1 | 77895 | 103 |
| 2 | 44678 | 103 |
| 3 | 22456 | 102 |
| 4 | 24562 | 101 |

Notice that the "PersonID" column in the "Orders" table points to the "PersonID" column in the "Persons" table.

The "PersonID" column in the "Persons" table is the PRIMARY KEY in the "Persons" table.

The "PersonID" column in the "Orders" table is a FOREIGN KEY in the "Orders" table.

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

The FOREIGN KEY constraint also prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the table it points to.

## SQL FOREIGN KEY on CREATE TABLE

The following SQL creates a FOREIGN KEY on the "PersonID" column when the "Orders" table is created:

**MySQL:**

CREATE TABLE Orders (  
    OrderID int NOT NULL,  
    OrderNumber int NOT NULL,  
    PersonID int,  
    PRIMARY KEY (OrderID),  
    FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)  
);

**SQL Server / Oracle / MS Access:**

CREATE TABLE Orders (  
    OrderID int NOT NULL PRIMARY KEY,  
    OrderNumber int NOT NULL,  
    PersonID int FOREIGN KEY REFERENCES Persons(PersonID)  
);

To allow naming of a FOREIGN KEY constraint, and for defining a FOREIGN KEY constraint on multiple columns, use the following SQL syntax:

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

CREATE TABLE Orders (  
    OrderID int NOT NULL,  
    OrderNumber int NOT NULL,  
    PersonID int,  
    PRIMARY KEY (OrderID),  
    CONSTRAINT FK\_PersonOrder FOREIGN KEY (PersonID)  
    REFERENCES Persons(PersonID)  
);

## SQL FOREIGN KEY on ALTER TABLE

To create a FOREIGN KEY constraint on the "PersonID" column when the "Orders" table is already created, use the following SQL:

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

ALTER TABLE Orders  
ADD FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);

To allow naming of a FOREIGN KEY constraint, and for defining a FOREIGN KEY constraint on multiple columns, use the following SQL syntax:

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

ALTER TABLE Orders  
ADD CONSTRAINT FK\_PersonOrder  
FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);

## DROP a FOREIGN KEY Constraint

To drop a FOREIGN KEY constraint, use the following SQL:

**MySQL:**

ALTER TABLE Orders  
DROP FOREIGN KEY FK\_PersonOrder;

# MySQL – ON DELETE CASCADE Constraint

**ON DELETE CASCADE** constraint is used in MySQL to delete the rows from the child table automatically, when the rows from the parent table are deleted. For example when a student registers in an online learning platform, then all the details of the student are recorded with their unique number/id. All the courses in these online learning platforms had their own code, title, and name. Students can enroll in any course according to their wishes.

There is no rule that all students must enroll in all courses, or they have to join the course on the same date. A student can enroll in one or more courses. Suppose you delete a row from the “Student” table, now you will also want to delete all rows in the “Enroll” table that references the row in the “Student” table. For that, we need ON DELETE CASCADE.  Below are the steps that explain how ON DELETE CASCADE referential action works.

# MySQL – ON update CASCADE Constraint

**ON update CASCADE** constraint is used in MySQL to update the rows from the child table automatically, when the rows from the parent table are updated. For example when a student registers in an online learning platform, then all the details of the student are recorded with their unique number/id. All the courses in these online learning platforms had their own code, title, and name. Students can enroll in any course according to their wishes.

**Step 1:**Create the**Student**table

CREATE TABLE Student (

sno INT PRIMARY KEY,

sname VARCHAR(20),

age INT

);

**Step 2:** Insert rows intothe **Student**table

INSERT INTO Student(sno, sname,age)

VALUES(1,'Ankit',17),

(2,'Ramya',18),

(3,'Ram',16);

**Step 3:** Executethe **SELECT**query to check the data in the **STUDENT** table.

SELECT \*

FROM Student;

**Output:**

| sno | sname | age |
| --- | --- | --- |
| 1 | Ankit | 17 |
| 2 | Ramya | 18 |
| 3 | Ram | 16 |

**Step 4:**Create the**Course**table

CREATE TABLE Course (

cno INT PRIMARY KEY,

cname VARCHAR(20)

);

**Step 5:** Insert rows intothe **Course**table

INSERT INTO Course(cno, cname)

VALUES(101,'c'),

(102,'c++'),

(103,'DBMS');

**Step 6:** Executethe **SELECT**query to check the data in the **Course** table.

SELECT \*

FROM Course;

**Output:**

| cno | cname |
| --- | --- |
| 101 | c |
| 102 | c++ |
| 103 | DBMS |

**Step 7:**Create the **Enroll** table

CREATE TABLE Enroll (

sno INT,

cno INT,

jdate date,

PRIMARY KEY(sno,cno),

FOREIGN KEY(sno)

REFERENCES Student(sno)

ON DELETE CASCADE

FOREIGN KEY(cno)

REFERENCES Course(cno)

ON DELETE CASCADE

);

**Step 8:** Insert rows intothe **Enroll**table

INSERT INTO Enroll(sno,cno,jdate)

VALUES(1, 101, '5-jun-2021'),

(1, 102, '5-jun-2021'),

(2, 103, '6-jun-2021');

**Step 9:** Executethe **SELECT**query to check the data in the **Enroll** table.

SELECT \*

FROM Enroll;

**Output:**

| sno | cno | jdate |
| --- | --- | --- |
| 1 | 101 | 5-jun-2021 |
| 1 | 102 | 5-jun-2021 |
| 2 | 103 | 6-jun-2021 |

**Step 10**: Here the parent tables are **Student** and **Course** whereas the child table is **Enroll**. If a student drops from the course or a course is removed from the offering list it must affect the child table also.

DELETE FROM Student

WHERE sname="Ramya";

**Step 11:** Executethe **SELECT**query to check the data.

Select \* from Student;

**Output:**

| sno | sname | age |
| --- | --- | --- |
| 1 | Ankit | 17 |
| 3 | Ram | 16 |

Select \* from Enroll;

**Output:**

| sno | cno | jdate |
| --- | --- | --- |
| 1 | 101 | 5-jun-2021 |
| 1 | 102 | 5-jun-2021 |

As you delete the contents of sno=2 in the parent table it automatically deletes the details of sno=2 from the child table also. In the same way, if you remove a course from the Course table it automatically deletes the rows of that course in the child table Enroll. This works out because the foreign key constraint ON DELETE CASCADE is specified.

**CHECK - Ensures that the values in a column satisfies a specific condition**

Example: CREATE TABLE Persons (

ID int NOT NULL,

LastName varchar(255) NOT NULL,

FirstName varchar(255),

Age int,

CHECK (Age>=18)

);

ALTER TABLE Persons

ADD CHECK (Age>=18);

**DEFAULT - Sets a default value for a column if no value is specified**

Example: CREATE TABLE Persons (

ID int NOT NULL,

LastName varchar(255) NOT NULL,

FirstName varchar(255),

Age int,

City varchar(255) DEFAULT 'Sandnes'

);

## SQL Aliases

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

Aliases are often used to make column names more readable.

An alias only exists for the duration of that query.

An alias is created with the AS keyword.

### Alias Column Syntax

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

### Alias Table Syntax

SELECT column\_name(s)  
FROM table\_name AS alias\_name;

## Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Customers" table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |

And a selection from the "Orders" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OrderID** | **CustomerID** | **EmployeeID** | **OrderDate** | **ShipperID** |
| 10354 | 58 | 8 | 1996-11-14 | 3 |
| 10355 | 4 | 6 | 1996-11-15 | 1 |
| 10356 | 86 | 6 | 1996-11-18 | 2 |

## Alias for Columns Examples

The following SQL statement creates two aliases, one for the CustomerID column and one for the CustomerName column:

### Example

SELECT CustomerID AS ID, CustomerName AS Customer  
FROM Customers;

);Bottom of Form

# SQL LIKE clause

LIKE clause is used in the condition in SQL query with the WHERE clause. LIKE clause compares data with an expression using wildcard operators to match pattern given in the condition.

### Wildcard operators

There are two wildcard operators that are used in LIKE clause.

* **Percent sign %**: represents zero, one or more than one character.
* **Underscore sign \_**: represents only a single character.

### Example of LIKE clause

Consider the following **Student** table.

|  |  |  |
| --- | --- | --- |
| **s\_id** | **s\_Name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |
| 103 | Abhi | 17 |

SELECT \* FROM Student WHERE s\_name LIKE 'A%';

The above query will return all records where **s\_name** starts with character 'A'.

|  |  |  |
| --- | --- | --- |
| **s\_id** | **s\_Name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |
| 103 | Abhi | 17 |

### Using \_ and %

SELECT \* FROM Student WHERE s\_name LIKE '\_d%';

The above query will return all records from **Student** table where **s\_name** contain 'd' as second character.

|  |  |  |
| --- | --- | --- |
| **s\_id** | **s\_Name** | **age** |
| 101 | Adam | 15 |

### Using % only

SELECT \* FROM Student WHERE s\_name LIKE '%x';

The above query will return all records from **Student** table where **s\_name** contain 'x' as last character.

|  |  |  |
| --- | --- | --- |
| **s\_id** | **s\_Name** | **age** |
| 102 | Alex | 18 |

# SQL Views

## SQL CREATE VIEW Statement

In SQL, a view is a virtual table based on the result-set of an SQL statement.

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

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

### CREATE VIEW Syntax

CREATE VIEW view\_name AS  
SELECT column1, column2, ...  
FROM table\_name  
WHERE condition;

**Note:** A view always shows up-to-date data! The database engine recreates the data, using the view's SQL statement, every time a user queries a view.

## SQL CREATE VIEW Examples

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

### Example

CREATE VIEW [Brazil Customers] AS  
SELECT CustomerName, ContactName  
FROM Customers  
WHERE Country = 'Brazil';

We can query the view above as follows:

### Example

SELECT \* FROM [Brazil Customers];

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

### Example

CREATE VIEW [Products Above Average Price] AS  
SELECT ProductName, Price  
FROM Products  
WHERE Price > (SELECT AVG(Price) FROM Products);

We can query the view above as follows:

### Example

SELECT \* FROM [Products Above Average Price];

## SQL Updating a View

A view can be updated with the CREATE OR REPLACE VIEW command.

### SQL CREATE OR REPLACE VIEW Syntax

CREATE OR REPLACE VIEW view\_name AS  
SELECT column1, column2, ...  
FROM table\_name  
WHERE condition;

The following SQL adds the "City" column to the "Brazil Customers" view:

### Example

CREATE OR REPLACE VIEW [Brazil Customers] AS  
SELECT CustomerName, ContactName, City  
FROM Customers  
WHERE Country = 'Brazil';

## SQL Dropping a View

A view is deleted with the DROP VIEW command.

### SQL DROP VIEW Syntax

DROP VIEW view\_name;

The following SQL drops the "Brazil Customers" view:

### Example

DROP VIEW [Brazil Customers];

# ORDER BY Clause

## Syntax of Order By

SELECT column-list|\* FROM table-name ORDER BY ASC | DESC;

### Using default Order by

Consider the following **Emp** table,

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SELECT \* FROM Emp ORDER BY salary;

The above query will return the resultant data in ascending order of the **salary**.

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 403 | Rohan | 34 | 6000 |
| 402 | Shane | 29 | 8000 |
| 405 | Tiger | 35 | 8000 |
| 401 | Anu | 22 | 9000 |
| 404 | Scott | 44 | 10000 |

### Using Order by DESC

Consider the **Emp** table described above,

SELECT \* FROM Emp ORDER BY salary DESC;

The above query will return the resultant data in descending order of the **salary**.

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 404 | Scott | 44 | 10000 |
| 401 | Anu | 22 | 9000 |
| 405 | Tiger | 35 | 8000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |

# DISTINCT keyword

The distinct keyword is used with SELECT statement to retrieve unique values from the table. Distinct removes all the duplicate records while retrieving records from any table in the database.

### Syntax for DISTINCT Keyword

SELECT DISTINCT column-name FROM table-name;

### Example using DISTINCT Keyword

Consider the following **Emp** table. As you can see in the table below, there is employee **name**, along with employee **salary** and **age**.

In the table below, multiple employees have the same salary, so we will be using DISTINCT keyword to list down distinct salary amount, that is currently being paid to the employees.

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 5000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 10000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SELECT DISTINCT salary FROM Emp;

The above query will return only the unique salary from **Emp** table.

|  |
| --- |
| **salary** |
| 5000 |
| 8000 |
| 10000 |

# AND & OR operator

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 5000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 12000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 9000 |

SELECT \* FROM Emp WHERE salary < 10000 **AND** age > 25

The above query will return records where **salary** is less than **10000** and **age** greater than **25**. Hope you get the concept here. We have used the AND operator to specify two conditions with WHERE clause.

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 402 | Shane | 29 | 8000 |
| 405 | Tiger | 35 | 9000 |

## OR operator

OR operator is also used to combine multiple conditions with WHERE clause. The only difference between AND and OR is their behaviour.

When we use AND to combine two or more than two conditions, records satisfying all the specified conditions will be there in the result.

But in case of OR operator, atleast one condition from the conditions specified must be satisfied by any record to be in the resultset.

### Example of OR operator

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 5000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 12000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 9000 |

SELECT \* FROM Emp WHERE salary > 10000 OR age > 25

The above query will return records where **either** salary is greater than 10000 **or** age is greater than 25.

|  |  |  |  |
| --- | --- | --- | --- |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 12000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 9000 |

# SQL GROUP BY Statement

## The SQL GROUP BY Statement

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

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

### GROUP BY Syntax

SELECT column\_name(s)  
FROM table\_name  
WHERE condition  
GROUP BY column\_name(s)ORDER BY column\_name(s);

## Demo Database

Below is a selection from the "Customers" table in the Northwind sample database:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

## SQL GROUP BY Examples

The following SQL statement lists the number of customers in each country:

### Example

SELECT COUNT(CustomerID), Country  
FROM Customers  
GROUP BY Country;

The following SQL statement lists the number of customers in each country, sorted high to low:

### Example

SELECT COUNT(CustomerID), Country  
FROM Customers  
GROUP BY Country  
ORDER BY COUNT(CustomerID) DESC;

## Demo Database

Below is a selection from the "Orders" table in the Northwind sample database:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OrderID** | **CustomerID** | **EmployeeID** | **OrderDate** | **ShipperID** |
| 10248 | 90 | 5 | 1996-07-04 | 3 |
| 10249 | 81 | 6 | 1996-07-05 | 1 |
| 10250 | 34 | 4 | 1996-07-08 | 2 |

And a selection from the "Shippers" table:

|  |  |
| --- | --- |
| **ShipperID** | **ShipperName** |
| 1 | Speedy Express |
| 2 | United Package |
| 3 | Federal Shipping |

## GROUP BY With JOIN Example

The following SQL statement lists the number of orders sent by each shipper:

### Example

SELECT Shippers.ShipperName, COUNT(Orders.OrderID) AS NumberOfOrders FROM Orders  
LEFT JOIN Shippers ON Orders.ShipperID = Shippers.ShipperID  
GROUP BY ShipperName;

# SQL HAVING Clause

## The SQL HAVING Clause

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

### HAVING Syntax

SELECT column\_name(s)  
FROM table\_name  
WHERE condition  
GROUP BY column\_name(s)HAVING conditionORDER BY column\_name(s);

## Demo Database

Below is a selection from the "Customers" table in the Northwind sample database:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

## SQL HAVING Examples

The following SQL statement lists the number of customers in each country. Only include countries with more than 5 customers:

### Example

SELECT COUNT(CustomerID), Country  
FROM Customers  
GROUP BY Country  
HAVING COUNT(CustomerID) > 5;

The following SQL statement lists the number of customers in each country, sorted high to low (Only include countries with more than 5 customers):

### Example

SELECT COUNT(CustomerID), Country  
FROM Customers  
GROUP BY Country  
HAVING COUNT(CustomerID) > 5  
ORDER BY COUNT(CustomerID) DESC;

## Demo Database

Below is a selection from the "Orders" table in the Northwind sample database:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OrderID** | **CustomerID** | **EmployeeID** | **OrderDate** | **ShipperID** |
| 10248 | 90 | 5 | 1996-07-04 | 3 |
| 10249 | 81 | 6 | 1996-07-05 | 1 |
| 10250 | 34 | 4 | 1996-07-08 | 2 |

And a selection from the "Employees" table:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **EmployeeID** | **LastName** | **FirstName** | **BirthDate** | **Photo** | **Notes** |
| 1 | Davolio | Nancy | 1968-12-08 | EmpID1.pic | Education includes a BA.... |
| 2 | Fuller | Andrew | 1952-02-19 | EmpID2.pic | Andrew received his BTS.... |
| 3 | Leverling | Janet | 1963-08-30 | EmpID3.pic | Janet has a BS degree.... |

## More HAVING Examples

The following SQL statement lists the employees that have registered more than 10 orders:

### Example

SELECT Employees.LastName, COUNT(Orders.OrderID) AS NumberOfOrders  
FROM (Orders  
INNER JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID)  
GROUP BY LastName  
HAVING COUNT(Orders.OrderID) > 10;

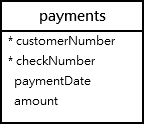
The following SQL statement lists if the employees "Davolio" or "Fuller" have registered more than 25 orders:

### Example[Get your own SQL Server](https://www.w3schools.com/spaces/)

SELECT Employees.LastName, COUNT(Orders.OrderID) AS NumberOfOrders  
FROM Orders  
INNER JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID  
WHERE LastName = 'Davolio' OR LastName = 'Fuller'  
GROUP BY LastName  
HAVING COUNT(Orders.OrderID) > 25;

## Using a MySQL subquery in the WHERE clause

We will use the table payments in the [sample database](https://www.mysqltutorial.org/mysql-sample-database.aspx) for the demonstration.



### MySQL subquery with comparison operators

You can use comparison operators e.g., =, >, < to compare a single value returned by the subquery with the expression in the [WHERE](https://www.mysqltutorial.org/mysql-where/) clause.

For example, the following query returns the customer who has the highest payment.

SELECT

customerNumber,

checkNumber,

amount

FROM

payments

WHERE

amount = (SELECT MAX(amount) FROM payments);

Code language: SQL (Structured Query Language) (sql)

[**Try It Out**](https://www.mysqltutorial.org/tryit/query/mysql-subquery/#1)

mysql subquery with equal operator

Besides the = operator, you can use other comparison operators such as greater than (>), greater than or equal to (>=) less than(<), and less than or equal to (<=).

For example, you can find customers whose payments are greater than the average payment using a subquery:

SELECT

customerNumber,

checkNumber,

amount

FROM

payments

WHERE

amount > (SELECT

AVG(amount)

FROM

payments);

Code language: SQL (Structured Query Language) (sql)

### mysql subquery with greater than operator

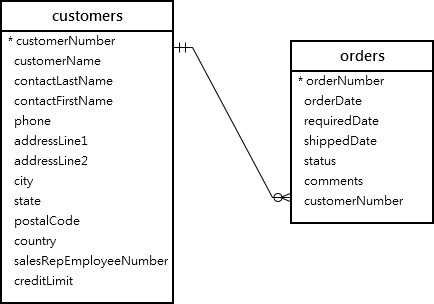
In this example:

* First, get the average payment by using a subquery.
* Then, select the payments that are greater than the average payment returned by the subquery in the outer query.

### MySQL subquery with IN and NOT IN operators

If a subquery returns more than one value, you can use other operators such as [IN](https://www.mysqltutorial.org/mysql-basics/mysql-in/) or [NOT IN](https://www.mysqltutorial.org/mysql-basics/mysql-in/) operator in the WHERE clause.

See the following customers and orders tables:



For example, you can use a subquery with NOT IN operator to find the customers who have not placed any orders as follows:

SELECT

customerName

FROM

customers

WHERE

customerNumber NOT IN (SELECT DISTINCT

customerNumber

FROM

orders);

Code language: SQL (Structured Query Language) (sql)

### mysql subquery not in

## MySQL subquery in the FROM clause

When you use a subquery in the FROM clause, the result set returned from a subquery is used as a [temporary table.](https://www.mysqltutorial.org/mysql-temporary-table/) This table is referred to as a [derived table](https://www.mysqltutorial.org/mysql-derived-table/) or materialized subquery.

The following subquery finds the [maximum](https://www.mysqltutorial.org/mysql-max-function/), [minimum,](https://www.mysqltutorial.org/mysql-min/)and [average](https://www.mysqltutorial.org/mysql-avg/)number of items in sale orders:

SELECT

MAX(items),

MIN(items),

FLOOR(AVG(items))

FROM

(SELECT

orderNumber, COUNT(orderNumber) AS items

FROM

orderdetails

GROUP BY orderNumber) AS lineitems;

Code language: SQL (Structured Query Language) (sql)

mysql subquery from clause example

# SQL JOIN

SQL Join is used to fetch data from two or more tables, which is joined to appear as single set of data. It is used for combining column from two or more tables by using values common to both tables.

## Types of JOIN

Following are the types of JOIN that we can use in SQL:

* Inner - Equi
* Outer

1. Left Outer Join
2. Right Outer Join
3. Full Outer Join

* Cross - cartesian
* Natural -

**Cross JOIN or Cartesian Product**

This type of JOIN returns the cartesian product of rows from the tables in Join. It will return a table which consists of records which combines each row from the first table with each row of the second table.

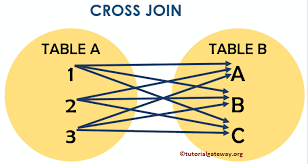
Cross JOIN Syntax is,

SELECT column-name-list

FROM

table-name1 CROSS JOIN table-name2;

#### Example of Cross JOIN

Following is the **class** table, 

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 4 | alex |

and the **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |

Cross JOIN query will be,

SELECT \* FROM

class CROSS JOIN class\_info;

The resultset table will look like,

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **ID** | **Address** |
| 1 | abhi | 1 | DELHI |
| 2 | adam | 1 | DELHI |
| 4 | alex | 1 | DELHI |
| 1 | abhi | 2 | MUMBAI |
| 2 | adam | 2 | MUMBAI |
| 4 | alex | 2 | MUMBAI |
| 1 | abhi | 3 | CHENNAI |
| 2 | adam | 3 | CHENNAI |
| 4 | alex | 3 | CHENNAI |

As you can see, this join returns the cross product of all the records present in both the tables.

## INNER Join or EQUI Join

## SQL INNER JOIN Keyword

This is a simple JOIN in which the result is based on matched data as per the equality condition specified in the SQL query.

Inner Join Syntax is,

SELECT column-name-list FROM

table-name1 INNER JOIN table-name2

WHERE table-name1.column-name = table-name2.column-name;

#### Example of INNER JOIN

Consider a **class** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 3 | alex |
| 4 | anu |

and the **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |

**Inner** JOIN query will be,

SELECT \* from class INNER JOIN class\_info where class.id = class\_info.id;

The resultset table will look like,

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **ID** | **Address** |
| 1 | abhi | 1 | DELHI |
| 2 | adam | 2 | MUMBAI |
| 3 | alex | 3 | CHENNAI |

### Natural JOIN

Natural Join is a type of Inner join which is based on column having **same name and same datatype** present in both the tables to be joined.

The syntax for Natural Join is,

SELECT \* FROM

table-name1 NATURAL JOIN table-name2;

#### Example of Natural JOIN

Here is the **class** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 3 | alex |
| 4 | anu |

and the **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |

**Natural join query will be,**

SELECT \* from class NATURAL JOIN class\_info;

The resultset table will look like,

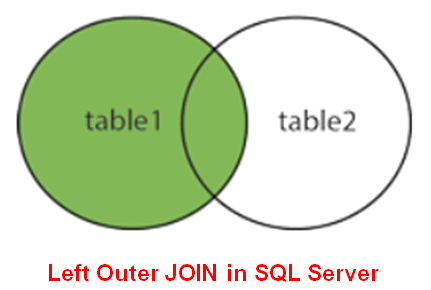
|  |  |  |
| --- | --- | --- |
| **ID** | **NAME** | **Address** |
| 1 | abhi | DELHI |
| 2 | adam | MUMBAI |
| 3 | alex | CHENNAI |

## OUTER JOIN

Outer Join is based on both matched and unmatched data. Outer Joins subdivide further into,

1. Left Outer Join
2. Right Outer Join
3. Full Outer Join

### LEFT Outer Join



The left outer join returns a resultset table with the **matched data** from the two tables and then the remaining rows of the **left** table and null from the **right** table's columns.

Syntax for Left Outer Join is,

SELECT column-name-list FROM

table-name1 LEFT OUTER JOIN table-name2

ON table-name1.column-name = table-name2.column-name;

To specify a condition, we use the ON keyword with Outer Join.

Left outer Join Syntax for **Oracle** is,

SELECT column-name-list FROM

table-name1, table-name2 on table-name1.column-name = table-name2.column-name(+);

#### Example of Left Outer Join

Here is the **class** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 3 | alex |
| 4 | anu |
| 5 | ashish |

and the **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **State** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |
| 7 | NOIDA |
| 8 | PANIPAT |

**Left Outer Join** query will be,

SELECT \* FROM class LEFT OUTER JOIN class\_info ON (class.id = class\_info.id);

The resultset table will look like,

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **ID** | **Address** |
| 1 | abhi | 1 | DELHI |
| 2 | adam | 2 | MUMBAI |
| 3 | alex | 3 | CHENNAI |
| 4 | anu | null | null |
| 5 | ashish | null | null |

### RIGHT Outer Join

The right outer join returns a resultset table with the **matched data** from the two tables being joined, then the remaining rows of the **right** table and null for the remaining **left** table's columns.

Syntax for Right Outer Join is,



SELECT column-name-list FROM

table-name1 RIGHT OUTER JOIN table-name2

ON table-name1.column-name = table-name2.column-name;

Right outer Join Syntax for **Oracle** is,

SELECT column-name-list FROM

table-name1, table-name2

ON table-name1.column-name(+) = table-name2.column-name;

#### Example of Right Outer Join

Once again the **class** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 3 | alex |
| 4 | anu |
| 5 | ashish |

and the **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |
| 7 | NOIDA |
| 8 | PANIPAT |

**Right Outer Join** query will be,

SELECT \* FROM class RIGHT OUTER JOIN class\_info ON (class.id = class\_info.id);

The resultant table will look like,

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **ID** | **Address** |
| 1 | abhi | 1 | DELHI |
| 2 | adam | 2 | MUMBAI |
| 3 | alex | 3 | CHENNAI |
| null | null | 7 | NOIDA |
| null | null | 8 | PANIPAT |

### Full Outer Join



The full outer join returns a resultset table with the **matched data** of two table then remaining rows of both **left** table and then the **right** table.

Syntax of Full Outer Join is,

SELECT column-name-list FROM

table-name1 FULL OUTER JOIN table-name2

ON table-name1.column-name = table-name2.column-name;

#### Example of Full outer join is,

The **class** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 3 | alex |
| 4 | anu |
| 5 | ashish |

and the **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |
| 7 | NOIDA |
| 8 | PANIPAT |

**Full Outer Join** query will be like,

SELECT \* FROM class FULL OUTER JOIN class\_info ON (class.id = class\_info.id);

The resultset table will look like,

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **ID** | **Address** |
| 1 | abhi | 1 | DELHI |
| 2 | adam | 2 | MUMBAI |
| 3 | alex | 3 | CHENNAI |
| 4 | anu | null | null |
| 5 | ashish | null | null |
| null | null | 7 | NOIDA |
| null | null | 8 | PANIPAT |