**SQL**

SQL is the standard language for dealing with Relational Databases.

SQL is used to insert, search, update, and delete database records.

The data type of a column defines what value the column can hold: integer, character, money, date and time, binary, and so on.

## MySQL Data Types (Version 8.0)

Each column in a database table is required to have a name and a data type.

An SQL developer must decide what type of data that will be stored inside each column when creating a table. The data type is a guideline for SQL to understand what type of data is expected inside of each column, and it also identifies how SQL will interact with the stored data.

In MySQL there are three main data types: string, numeric, and date and time.

### **String Data Types**

|  |  |
| --- | --- |
| **Data type** | **Description** |
| CHAR(size) | A FIXED length string (can contain letters, numbers, and special characters). The size parameter specifies the column length in characters - can be from 0 to 255. Default is 1 |
| VARCHAR(size) | A VARIABLE length string (can contain letters, numbers, and special characters). The size parameter specifies the maximum column length in characters - can be from 0 to 65535 |
| BINARY(size) | Equal to CHAR(), but stores binary byte strings. The size parameter specifies the column length in bytes. Default is 1 |
| VARBINARY(size) | Equal to VARCHAR(), but stores binary byte strings. The size parameter specifies the maximum column length in bytes. |
| TINYBLOB | For BLOBs (Binary Large OBjects). Max length: 255 bytes |
| TINYTEXT | Holds a string with a maximum length of 255 characters |
| TEXT(size) | Holds a string with a maximum length of 65,535 bytes |
| BLOB(size) | For BLOBs (Binary Large OBjects). Holds up to 65,535 bytes of data |
| MEDIUMTEXT | Holds a string with a maximum length of 16,777,215 characters |
| MEDIUMBLOB | For BLOBs (Binary Large OBjects). Holds up to 16,777,215 bytes of data |
| LONGTEXT | Holds a string with a maximum length of 4,294,967,295 characters |
| LONGBLOB | For BLOBs (Binary Large OBjects). Holds up to 4,294,967,295 bytes of data |
| ENUM(val1, val2, val3, ...) | A string object that can have only one value, chosen from a list of possible values. You can list up to 65535 values in an ENUM list. If a value is inserted that is not in the list, a blank value will be inserted. The values are sorted in the order you enter them |
| SET(val1, val2, val3, ...) | A string object that can have 0 or more values, chosen from a list of possible values. You can list up to 64 values in a SET list |

### **Numeric Data Types**

|  |  |
| --- | --- |
| **Data type** | **Description** |
| BIT(size) | A bit-value type. The number of bits per value is specified in size. The size parameter can hold a value from 1 to 64. The default value for size is 1. |
| TINYINT(size) | A very small integer. Signed range is from -128 to 127. Unsigned range is from 0 to 255. The size parameter specifies the maximum display width (which is 255) |
| BOOL | Zero is considered as false, nonzero values are considered as true. |
| BOOLEAN | Equal to BOOL |
| SMALLINT(size) | A small integer. Signed range is from -32768 to 32767. Unsigned range is from 0 to 65535. The size parameter specifies the maximum display width (which is 255) |
| MEDIUMINT(size) | A medium integer. Signed range is from -8388608 to 8388607. Unsigned range is from 0 to 16777215. The size parameter specifies the maximum display width (which is 255) |
| INT(size) | A medium integer. Signed range is from -2147483648 to 2147483647. Unsigned range is from 0 to 4294967295. The size parameter specifies the maximum display width (which is 255) |
| INTEGER(size) | Equal to INT(size) |
| BIGINT(size) | A large integer. Signed range is from -9223372036854775808 to 9223372036854775807. Unsigned range is from 0 to 18446744073709551615. The size parameter specifies the maximum display width (which is 255) |
| FLOAT(size, d) | A floating point number. The total number of digits is specified in size. The number of digits after the decimal point is specified in the d parameter. This syntax is deprecated in MySQL 8.0.17, and it will be removed in future MySQL versions |
| FLOAT(p) | A floating point number. MySQL uses the p value to determine whether to use FLOAT or DOUBLE for the resulting data type. If p is from 0 to 24, the data type becomes FLOAT(). If p is from 25 to 53, the data type becomes DOUBLE() |
| DOUBLE(size, d) | A normal-size floating point number. The total number of digits is specified in size. The number of digits after the decimal point is specified in the d parameter |
| DOUBLE PRECISION(size, d) |  |
| DECIMAL(size, d) | An exact fixed-point number. The total number of digits is specified in size. The number of digits after the decimal point is specified in the d parameter. The maximum number for size is 65. The maximum number for d is 30. The default value for size is 10. The default value for d is 0. |
| DEC(size, d) | Equal to DECIMAL(size,d) |

**Note:** All the numeric data types may have an extra option: UNSIGNED or ZEROFILL. If you add the UNSIGNED option, MySQL disallows negative values for the column. If you add the ZEROFILL option, MySQL automatically also adds the UNSIGNED attribute to the column.

### **Date and Time Data Types**

|  |  |  |  |
| --- | --- | --- | --- |
| **Data type** | | **Description** | |
| DATE | | A date. Format: YYYY-MM-DD. The supported range is from '1000-01-01' to '9999-12-31' | |
| DATETIME(fsp) | | A date and time combination. Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'. Adding DEFAULT and ON UPDATE in the column definition to get automatic initialization and updating to the current date and time | |
| TIMESTAMP(fsp) | | A timestamp. TIMESTAMP values are stored as the number of seconds since the Unix epoch ('1970-01-01 00:00:00' UTC). Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1970-01-01 00:00:01' UTC to '2038-01-09 03:14:07' UTC. Automatic initialization and updating to the current date and time can be specified using DEFAULT CURRENT\_TIMESTAMP and ON UPDATE CURRENT\_TIMESTAMP in the column definition | |
| TIME(fsp) | | A time. Format: hh:mm:ss. The supported range is from '-838:59:59' to '838:59:59' | |
| YEAR | | A year in four-digit format. Values allowed in four-digit format: 1901 to 2155, and 0000. MySQL 8.0 does not support year in two-digit format. | |
| SOME | TRUE if any of the subquery values meet the condition | |  |

MySQL Arithmetic Operators

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Add |  |
| - | Subtract |  |
| \* | Multiply |  |
| / | Divide |  |
| % | Modulo |  |

MySQL Bitwise Operators

|  |  |
| --- | --- |
| **Operator** | **Description** |
| & | Bitwise AND |
| | | Bitwise OR |
| ^ | Bitwise exclusive OR |

MySQL Comparison Operators

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Equal to | [Try it](https://www.w3schools.com/mysql/trymysql.asp?filename=trysql_op_equal_to) |
| > | Greater than | [Try it](https://www.w3schools.com/mysql/trymysql.asp?filename=trysql_op_greater_than) |
| < | Less than | [Try it](https://www.w3schools.com/mysql/trymysql.asp?filename=trysql_op_less_than) |
| >= | Greater than or equal to | [Try it](https://www.w3schools.com/mysql/trymysql.asp?filename=trysql_op_greater_than2) |
| <= | Less than or equal to | [Try it](https://www.w3schools.com/mysql/trymysql.asp?filename=trysql_op_less_than2) |
| <> | Not equal to |  |

MySQL Compound Operators

|  |  |
| --- | --- |
| **Operator** | **Description** |
| += | Add equals |
| -= | Subtract equals |
| \*= | Multiply equals |
| /= | Divide equals |
| %= | Modulo equals |
| &= | Bitwise AND equals |
| ^-= | Bitwise exclusive equals |
| |\*= | Bitwise OR equals |

MySQL Logical Operators

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| ALL | TRUE if all of the subquery values meet the condition |  |
| AND | TRUE if all the conditions separated by AND is TRUE |  |
| ANY | TRUE if any of the subquery values meet the condition |  |
| BETWEEN | TRUE if the operand is within the range of comparisons |  |
| EXISTS | TRUE if the subquery returns one or more records |  |
| IN | TRUE if the operand is equal to one of a list of expressions |  |
| LIKE | TRUE if the operand matches a pattern |  |
| NOT | Displays a record if the condition(s) is NOT TRUE |  |
| OR | TRUE if any of the conditions separated by OR is TRUE |  |

## The MySQL CREATE DATABASE Statement

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

### **Syntax**

CREATE DATABASE databasename;

## CREATE DATABASE Example

The following SQL statement creates a database called "testDB":

EG:

CREATE DATABASE testDB;

## The MySQL DROP DATABASE Statement

The DROP DATABASE statement is used to drop an existing SQL database.

### **Syntax**

DROP DATABASE databasename;

## The MySQL CREATE TABLE Statement

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

### **Syntax**

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

The column parameters specify the names of the columns of the table.

The datatype parameter specifies the type of data the column can hold (e.g. varchar, integer, date, etc.).

**Tip:** For an overview of the available data types, go to our complete [Data Types Reference](https://www.w3schools.com/mysql/mysql_datatypes.asp).

## MySQL CREATE TABLE Example

The following example creates a table called "Persons" that contains five columns: PersonID, LastName, FirstName, Address, and City:

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

CREATE TABLE Persons (  
    PersonID int,  
    LastName varchar(255),  
    FirstName varchar(255),  
    Address varchar(255),  
    City varchar(255)  
);

The PersonID column is of type int and will hold an integer.

The LastName, FirstName, Address, and City columns are of type varchar and will hold characters, and the maximum length for these fields is 255 characters.

## The MySQL DROP TABLE Statement

The DROP TABLE statement is used to drop an existing table in a database.

### **Syntax**

DROP TABLE table\_name;

**Note:** Be careful before dropping a table. Deleting a table will result in loss of complete information stored in the table!

## MySQL DROP TABLE Example

The following SQL statement drops the existing table "Shippers":

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

DROP TABLE Shippers;

## MySQL TRUNCATE TABLE

The TRUNCATE TABLE statement is used to delete the data inside a table, but not the table itself.

### **Syntax**

TRUNCATE TABLE table\_name;

## MySQL ALTER TABLE Statement

The ALTER TABLE statement is used to add, delete, or modify columns in an existing table.

The ALTER TABLE statement is also used to add and drop various constraints on an existing table.

## ALTER TABLE - ADD Column

To add a column in a table, use the following syntax:

ALTER TABLE table\_name  
ADD column\_name datatype;

The following SQL adds an "Email" column to the "Customers" table:

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

ALTER TABLE Customers  
ADD Email varchar(255);

## ALTER TABLE - DROP COLUMN

To delete a column in a table, use the following syntax (notice that some database systems don't allow deleting a column):

ALTER TABLE table\_name  
DROP COLUMN column\_name;

The following SQL deletes the "Email" column from the "Customers" table:

### **Example**

ALTER TABLE Customers  
DROP COLUMN Email;

## ALTER TABLE - MODIFY COLUMN

To change the data type of a column in a table, use the following syntax:

ALTER TABLE table\_name  
MODIFY COLUMN column\_name datatype;

## MySQL ALTER TABLE Example

Look at the "Persons" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |

Now we want to add a column named "DateOfBirth" in the "Persons" table.

We use the following SQL statement:

### **Example**

ALTER TABLE Persons  
ADD DateOfBirth date;

Notice that the new column, "DateOfBirth", is of type date and is going to hold a date. The data type specifies what type of data the column can hold. For a complete reference of all the data types available in MySQL, go to our complete [Data Types reference](https://www.w3schools.com/mysql/mysql_datatypes.asp).

The "Persons" table will now look like this:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **LastName** | **FirstName** | **Address** | **City** | **DateOfBirth** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |  |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |  |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |  |

## Change Data Type Example

Now we want to change the data type of the column named "DateOfBirth" in the "Persons" table.

We use the following SQL statement:

### **Example**

ALTER TABLE Persons  
MODIFY COLUMN DateOfBirth year;

Notice that the "DateOfBirth" column is now of type year and is going to hold a year in a two- or four-digit format.

## DROP COLUMN Example

Next, we want to delete the column named "DateOfBirth" in the "Persons" table.

We use the following SQL statement:

### **Example**

ALTER TABLE Persons  
DROP COLUMN DateOfBirth;

The "Persons" table will now look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |

# **MySQL Constraints**

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

## 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*,  
    ....  
);

## MySQL 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/mysql/mysql_notnull.asp) - Ensures that a column cannot have a NULL value
* [UNIQUE](https://www.w3schools.com/mysql/mysql_unique.asp) - Ensures that all values in a column are different
* [PRIMARY KEY](https://www.w3schools.com/mysql/mysql_primarykey.asp) - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
* [FOREIGN KEY](https://www.w3schools.com/mysql/mysql_foreignkey.asp) - Prevents actions that would destroy links between tables
* [CHECK](https://www.w3schools.com/mysql/mysql_check.asp) - Ensures that the values in a column satisfies a specific condition
* [DEFAULT](https://www.w3schools.com/mysql/mysql_default.asp) - Sets a default value for a column if no value is specified
* [CREATE INDEX](https://www.w3schools.com/mysql/mysql_create_index.asp) - Used to create and retrieve data from the database very quickly

## MySQL 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.

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

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

## 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:

### **Example**

ALTER TABLE Persons  
MODIFY Age int NOT NULL;

MySQL 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.

UNIQUE Constraint on CREATE TABLE

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

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:

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

UNIQUE Constraint on ALTER TABLE

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

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:

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

DROP a UNIQUE Constraint

To drop a UNIQUE constraint, use the following SQL:

ALTER TABLE Persons  
DROP INDEX UC\_Person;

MySQL 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).

PRIMARY KEY on CREATE TABLE

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

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

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).

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:

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:

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

**Note:** If you use ALTER TABLE to add a primary key, the primary key column(s) must 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:

ALTER TABLE Persons  
DROP PRIMARY KEY;

## MySQL FOREIGN KEY Constraint

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

A FOREIGN KEY is a field (or collection of fields) in one table, that refers to the [PRIMARY KEY](https://www.w3schools.com/mysql/mysql_primarykey.asp) in another table.

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

Look at the following two tables:

### **Persons Table**

|  |  |  |  |
| --- | --- | --- | --- |
| **PersonID** | **LastName** | **FirstName** | **Age** |
| 1 | Hansen | Ola | 30 |
| 2 | Svendson | Tove | 23 |
| 3 | Pettersen | Kari | 20 |

### **Orders Table**

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

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 prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the parent table.

## FOREIGN KEY on CREATE TABLE

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

CREATE TABLE Orders (  
    OrderID int NOT NULL,  
    OrderNumber int NOT NULL,  
    PersonID int,  
    PRIMARY KEY (OrderID),  
    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:

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)  
);

## 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:

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:

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:

ALTER TABLE Orders  
DROP FOREIGN KEY FK\_PersonOrder;

MySQL CHECK Constraint

The CHECK constraint is used to limit the value range that can be placed in a column.

If you define a CHECK constraint on a column it will allow only certain values for this column.

If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.

CHECK on CREATE TABLE

The following SQL creates a CHECK constraint on the "Age" column when the "Persons" table is created. The CHECK constraint ensures that the age of a person must be 18, or older:

CREATE TABLE Persons (  
    ID int NOT NULL,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int,  
    CHECK (Age>=18)  
);

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

CREATE TABLE Persons (  
    ID int NOT NULL,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int,  
    City varchar(255),  
    CONSTRAINT CHK\_Person CHECK (Age>=18 AND City='Sandnes')  
);

CHECK on ALTER TABLE

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

ALTER TABLE Persons  
ADD CHECK (Age>=18);

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

ALTER TABLE Persons  
ADD CONSTRAINT CHK\_PersonAge CHECK (Age>=18 AND City='Sandnes');

DROP a CHECK Constraint

To drop a CHECK constraint, use the following SQL:

ALTER TABLE Persons  
DROP CHECK CHK\_PersonAge;

MySQL DEFAULT Constraint

The DEFAULT constraint is used to set a default value for a column.

The default value will be added to all new records, if no other value is specified.

DEFAULT on CREATE TABLE

The following SQL sets a DEFAULT value for the "City" column when the "Persons" table is created:

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

The DEFAULT constraint can also be used to insert system values, by using functions like [CURRENT\_DATE()](https://www.w3schools.com/mysql/func_mysql_current_date.asp):

CREATE TABLE Orders (  
    ID int NOT NULL,  
    OrderNumber int NOT NULL,  
    OrderDate date DEFAULT CURRENT\_DATE()  
);

DEFAULT on ALTER TABLE

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

ALTER TABLE Persons  
ALTER City SET DEFAULT 'Sandnes';

DROP a DEFAULT Constraint

To drop a DEFAULT constraint, use the following SQL:

ALTER TABLE Persons  
ALTER City DROP DEFAULT;

1.SELECT \* FROM Customers;

 selects all the records in the "Customers" table:

Important SQL Commands

* SELECT - extracts data from a database
* UPDATE - updates data in a database
* DELETE - deletes data from a database
* INSERT INTO - inserts new data into a database
* CREATE DATABASE - creates a new database
* ALTER DATABASE - modifies a database
* CREATE TABLE - creates a new table
* ALTER TABLE - modifies a table
* DROP TABLE - deletes a table
* CREATE INDEX - creates an index (search key)
* DROP INDEX - deletes an index

## MySQL SELECT Statement

* The SELECT statement is used to select data from a database.
* The data returned is stored in a result table, called the result-set.
* SELECT column1, column2, ...  
  FROM table\_name;
* EG:SELECT CustomerName, City, Country FROM Customers;(WITH DISTINCT FIELD)
* SELECT \* FROM table\_name;
* EG: SELECT \* FROM Customers;(WITHOUT DISTINCT FIELD)
* SELECT COUNT(DISTINCT Country) FROM Customers;
* O/P:21
* SELECT DISTINCT Country FROM Customers;
* O/P:

|  |
| --- |
| **Country** |
| Germany |
| Mexico |
| UK |
| Sweden |
| France |
| Spain |
| Canada |
| Argentina |
| Switzerland |
| Brazil |
| Austria |
| Italy |
| Portugal |

## MySQL WHERE Clause

* The WHERE clause is used to filter records.
* It is used to extract only those records that fulfill a specified condition.
* SELECT column1, column2, ...  
  FROM table\_name  
  WHERE condition;
* The WHERE clause is not only used in SELECT statements, it is also used in UPDATE, DELETE

### **Example**

* SELECT \* FROM Customers  
  WHERE Country = 'Mexico';
* O/P:

|  |
| --- |
|  |
| **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 |
| 13 | Centro comercial Moctezuma | Francisco Chang | Sierras de Granada 9993 | México D.F. | 05022 | Mexico |
| 58 | Pericles Comidas clásicas | Guillermo Fernández | Calle Dr. Jorge Cash 321 | México D.F. |  |  |

SELECT \* FROM Customers  
WHERE CustomerID = 1;

* SELECT \* FROM Products

WHERE Price > 30;

* SELECT \* FROM Products

WHERE Price < 30;

EG:SELECT \* FROM Products

WHERE Price >= 30;

EG: SELECT \* FROM Products

WHERE Price <= 30;

EG: SELECT \* FROM Products

WHERE Price <> 18;

EG: SELECT \* FROM Products

WHERE Price BETWEEN 50 AND 60;

EG: SELECT \* FROM Customers

WHERE City LIKE 's%';

EG:SELECT \* FROM Customers

WHERE City IN ('Paris','London');

# **MySQL AND, OR and NOT Operators**

The WHERE clause can be combined with AND, OR, and NOT operators.

The AND and OR operators are used to filter records based on more than one condition:

* The AND operator displays a record if all the conditions separated by AND are TRUE.
* The OR operator displays a record if any of the conditions separated by OR is TRUE.

The NOT operator displays a record if the condition(s) is NOT TRUE.

### **AND Syntax**

SELECT column1, column2, ...  
FROM table\_name  
WHERE condition1 AND condition2 AND condition3 ...;

EG:

SELECT \* FROM Customers  
WHERE Country = 'Germany' AND City = 'Berlin';

O/P:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 1220 |  |

## OR Example

The following SQL statement selects all fields from "Customers" where city is "Berlin" OR "Stuttgart":

EG:

SELECT \* FROM Customers  
WHERE City = 'Berlin' OR City = 'Stuttgart';

O/P:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin |  |  |

## NOT Example

The following SQL statement selects all fields from "Customers" where country is NOT "Germany":

EG:

SELECT \* FROM Customers  
WHERE NOT Country = 'Germany';

O/P:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **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. | Lond |  |  |

## Combining AND, OR and NOT

You can also combine the AND, OR and NOT operators.

The following SQL statement selects all fields from "Customers" where country is "Germany" AND city must be "Berlin" OR "Stuttgart" (use parenthesis to form complex expressions):

EG:

SELECT \* FROM Customers  
WHERE Country = 'Germany' AND (City = 'Berlin' OR City = 'Stuttgart');

O/P:

The following SQL statement selects all fields from "Customers" where country is NOT "Germany" and NOT "USA":

SELECT \* FROM Customers  
WHERE NOT Country = 'Germany' AND NOT Country = 'USA';

## MySQL ORDER BY Keyword

The ORDER BY keyword is used to sort the result-set in ascending or descending order.

The ORDER BY keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.

### **ORDER BY Syntax**

## SELECT column1, column2, ... FROM table\_name ORDER BY column1, column2, ... ASC|DESC;

## ORDER BY Example

## selects all customers from the "Customers" table, sorted by the "Country" column:

## SELECT \* FROM Customers ORDER BY Country;

## ORDER BY DESC Example

## selects all customers from the "Customers" table, sorted DESCENDING by the "Country" column:

## EG:

## SELECT \* FROM Customers ORDER BY Country DESC;

## ORDER BY Several Columns Example

## selects all customers from the "Customers" table, sorted by the "Country" and the "CustomerName" column. This means that it orders by Country, but if some rows have the same Country, it orders them by CustomerName:

## SELECT \* FROM Customers ORDER BY Country, CustomerName;

## ORDER BY Several Columns

## selects all customers from the "Customers" table, sorted ascending by the "Country" and descending by the "CustomerName" column:

## EG: SELECT \* FROM Customers ORDER BY Country ASC, CustomerName DESC;

## MySQL INSERT INTO Statement

## The INSERT INTO statement is used to insert new records in a table.

### **INSERT INTO Syntax**

It is possible to write the INSERT INTO statement in two ways:

1. Specify both the column names and the values to be inserted:
2. INSERT INTO table\_name (column1, column2, column3, ...)  
   VALUES (value1, value2, value3, ...);
3. If you are adding values for all the columns of the table, you do not need to specify the column names in the SQL query. However, make sure the order of the values is in the same order as the columns in the table. Here, the INSERT INTO syntax would be as follows:
4. INSERT INTO *table\_name*  
   VALUES (*value1*,*value2*,*value3*, ...);

EG:

INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country)  
VALUES ('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006', 'Norway');

## Insert Data Only in Specified Columns

It is also possible to only insert data in specific columns.

The following SQL statement will insert a new record, but only insert data in the "CustomerName", "City", and "Country" columns (CustomerID will be updated automatically):

EG:

INSERT INTO Customers (CustomerName, City, Country)  
VALUES ('Cardinal', 'Stavanger', 'Norway');

# **MySQL NULL Values**

A field with a NULL value is a field with no value.

If a field in a table is optional, it is possible to insert a new record or update a record without adding a value to this field. Then, the field will be saved with a NULL value.

**Note:** A NULL value is different from a zero value or a field that contains spaces. A field with a NULL value is one that has been left blank during record creation!

It is not possible to test for NULL values with comparison operators, such as =, <, or <>.

We will have to use the IS NULL and IS NOT NULL operators instead.

### **IS NULL Syntax**

SELECT column\_namesFROM table\_name  
WHERE column\_name IS NULL;

### **IS NOT NULL Syntax**

SELECT column\_namesFROM table\_name  
WHERE column\_name IS NOT NULL;

EG:

SELECT CustomerName, ContactName, Address  
FROM Customers  
WHERE Address IS NULL;

O/P:

|  |  |  |
| --- | --- | --- |
| **CustomerName** | **ContactName** | **Address** |

## The MySQL UPDATE Statement

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

### **UPDATE Syntax**

UPDATE table\_name  
SET column1 = value1, column2 = value2, ...  
WHERE condition;

WHERE clause in the UPDATE statement. The WHERE clause specifies which record(s) that should be updated. If you omit the WHERE clause, all records in the table will be updated!

## UPDATE Table

## The following SQL statement updates the first customer (CustomerID = 1) with a new contact person and a new city.

## EG:

## UPDATE Customers SET ContactName = 'Alfred Schmidt', City = 'Frankfurt' WHERE CustomerID = 1;

## UPDATE Multiple Records

It is the WHERE clause that determines how many records will be updated.

The following SQL statement will update the PostalCode to 00000 for all records where country is "Mexico":

### **Example**

## UPDATE Customers SET PostalCode = 00000 WHERE Country = 'Mexico';

## Be careful when updating records. If you omit the WHERE clause, ALL records will be updated!

### **Example**

## UPDATE Customers SET PostalCode = 00000;

## The MySQL DELETE Statement

## The DELETE statement is used to delete existing records in a table.

### **DELETE Syntax**

## DELETE FROM table\_name WHERE condition;

## the WHERE clause in the DELETE statement. The WHERE clause specifies which record(s) should be deleted. If you omit the WHERE clause, all records in the table will be deleted!

## DELETE FROM Customers WHERE CustomerName='Alfreds Futterkiste';

## deletes the customer "Alfreds Futterkiste" from the "Customers" table:

## Delete All Records

It is possible to delete all rows in a table without deleting the table. This means that the table structure, attributes, and indexes will be intact:

## DELETE FROM table\_name;

## deletes all rows in the "Customers" table, without deleting the table:

## EG:

## DELETE FROM Customers;

## The MySQL LIMIT Clause

The LIMIT clause is used to specify the number of records to return.

The LIMIT clause is useful on large tables with thousands of records. Returning a large number of records can impact performance.

### **LIMIT Syntax**

SELECT column\_name(s)  
FROM table\_nameWHERE condition  
LIMIT number;

## EG:

## SELECT \* FROM Customers LIMIT 3;

## ADD a WHERE CLAUSE

## SELECT \* FROM Customers WHERE Country='Germany' LIMIT 3;

## selects the first three records from the "Customers" table, where the country is "Germany":

## MySQL 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;

## EG: SQL statement finds the price of the cheapest product:

## SELECT MIN(Price) AS SmallestPrice FROM Products;

### **MAX() Syntax**

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

EG:

SELECT MAX(Price) AS LargestPrice  
FROM Products;

 statement finds the price of the most expensive product:

# **MySQL COUNT(), AVG() and SUM() Functions**

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

### **COUNT() Syntax**

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

EG:

SELECT COUNT(ProductID)  
FROM Products;

O/P:

23

### **AVG() Syntax**

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

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

EG:

SELECT AVG(Price)  
FROM Products;

O/P: NULL values are ignored.)

75.88899

### **SUM() Syntax**

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

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

EG: NULL values are ignored.

SELECT SUM(Quantity)  
FROM OrderDetails;

## The MySQL LIKE Operator

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

There are two wildcards often used in conjunction with the LIKE operator:

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

The percent sign and the underscore can also be used in combinations!

### **LIKE Syntax (ALSO** combine any number of conditions using AND or OR operators.)

SELECT column1, column2, ...  
FROM table\_name  
WHERE columnN LIKE pattern;

 examples showing different LIKE operators with '%' and '\_' wildcards:9

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

EG:

1.SELECT \* FROM Customers  
WHERE CustomerName LIKE 'a%';

selects all customers with a CustomerName starting with "a": 2.

2.SELECT \* FROM Customers  
WHERE CustomerName LIKE '%a';  selects all customers with a CustomerName ending with "a": 3.

3. SELECT \* FROM Customers  
WHERE CustomerName LIKE '%or%';

selects all customers with a CustomerName that have "or" in any position:

4. SELECT \* FROM Customers  
WHERE CustomerName LIKE '\_r%';

selects all customers with a CustomerName that have "r" in the second position:

5. SELECT \* FROM Customers  
WHERE CustomerName LIKE 'a\_\_%';

selects all customers with a CustomerName that starts with "a" and are at least 3 characters in length:

6. SELECT \* FROM Customers  
WHERE ContactName LIKE 'a%o';

selects all customers with a ContactName that starts with "a" and ends with "o":

7. SELECT \* FROM Customers  
WHERE CustomerName NOT LIKE 'a%';

 selects all customers with a CustomerName that does NOT start with "a":

## MySQL Wildcard Characters

A wildcard character is used to substitute one or more characters in a string.

Wildcard characters are used with the [LIKE](https://www.w3schools.com/mysql/mysql_like.asp) operator. The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

### **Wildcard Characters in MySQL**

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Description** | **Example** |
| % | Represents zero or more characters | bl% finds bl, black, blue, and blob |
| \_ | Represents a single character | h\_t finds hot, hat, and hi |

## Using the % Wildcard

1.SELECT \* FROM Customers  
WHERE City LIKE 'ber%';

selects all customers with a City starting with "ber":

2.SELECT \* FROM Customers  
WHERE City LIKE '%es%';

selects all customers with a City containing the pattern "es":

## Using the \_ Wildcard

1.SELECT \* FROM Customers  
WHERE City LIKE '\_ondon';

selects all customers with a City starting with any character, followed by "ondon":

2. SELECT \* FROM Customers  
WHERE City LIKE 'L\_n\_on';

selects all customers with a City starting with "L", followed by any character, followed by "n", followed by any character, followed by "on":

## The MySQL IN Operator

The IN operator allows you to specify multiple values in a WHERE clause.

The IN operator is a shorthand for multiple OR conditions.

### **IN Syntax**

SELECT column\_name(s)  
FROM table\_name  
WHERE column\_name IN (value1, value2, ...);

OR

SELECT column\_name(s)  
FROM table\_name  
WHERE column\_name IN (*SELECT* STATEMENT);

## IN Operator Examples

1.SELECT \* FROM Customers  
WHERE Country IN ('Germany', 'France', 'UK');

selects all customers that are located in "Germany", "France" or "UK":

2. SELECT \* FROM Customers  
WHERE Country NOT IN ('Germany', 'France', 'UK');

selects all customers that are NOT located in "Germany", "France" or "UK":

3. SELECT \* FROM Customers  
WHERE Country IN (SELECT Country FROM Suppliers);

selects all customers that are from the same countries as the suppliers:

# **MySQL BETWEEN Operator**

## The MySQL 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;

EG:

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

 selects all products with a price between 10 and 20:

## BETWEEN with IN Example

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

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

## BETWEEN Text Values Example

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

selects all products with a ProductName between "Carnarvon Tigers" and "Mozzarella di Giovanni":

2.SELECT \* FROM Products  
WHERE ProductName BETWEEN "Carnarvon Tigers" AND "Chef Anton's Cajun Seasoning"  
ORDER BY ProductName;1

selects all products with a ProductName between "Carnarvon Tigers" and "Chef Anton's Cajun Seasoning":

## NOT BETWEEN Text Values Example

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

selects all products with a ProductName not between "Carnarvon Tigers" and "Mozzarella di Giovanni":

## 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 '1996-07-01' AND '1996-07-31';

## MySQL Aliases

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;

## Alias for Columns Examples

1.SELECT CustomerID AS ID, CustomerName AS Customer  
FROM Customers;

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

2. SELECT CustomerName AS Customer, ContactName AS "Contact Person"  
FROM Customers;

statement creates two aliases, one for the CustomerName column and one for the ContactName column. **Note:** Single or double quotation marks are required if the alias name contains spaces:

3. SELECT CustomerName, CONCAT\_WS(', ', Address, PostalCode, City, Country) AS Address  
FROM Customers;

statement creates an alias named "Address" that combine four columns (Address, PostalCode, City and Country):

## Alias for Tables Example

The following SQL statement selects all the orders from the customer with CustomerID=4 (Around the Horn). We use the "Customers" and "Orders" tables, and give them the table aliases of "c" and "o" respectively (Here we use aliases to make the SQL shorter):

### **Example**

SELECT o.OrderID, o.OrderDate, c.CustomerName  
FROM Customers AS c, Orders AS o  
WHERE c.CustomerName='Around the Horn' AND c.CustomerID=o.CustomerID;

BELOW WITHOUT ALIASES

EG: SELECT Orders.OrderID, Orders.OrderDate, Customers.CustomerName  
FROM Customers, Orders  
WHERE Customers.CustomerName='Around the Horn' AND Customers.CustomerID=Orders.CustomerID;

Aliases can be useful when:

* There are more than one table involved in a query
* Functions are used in the query
* Column names are big or not very readable
* Two or more columns are combined together

## MySQL Joining Tables

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

look at a selection from the "Orders" table:

|  |  |  |
| --- | --- | --- |
| **OrderID** | **CustomerID** | **OrderDate** |
| 10308 | 2 | 1996-09-18 |
| 10309 | 37 | 1996-09-19 |
| 10310 | 77 | 1996-09-20 |

Then, look at a selection from the "Customers" table:

|  |  |  |  |
| --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Country** |
| 1 | Alfreds Futterkiste | Maria Anders | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mexico |

Notice that the "CustomerID" column in the "Orders" table refers to the "CustomerID" in the "Customers" table. The relationship between the two tables above is the "CustomerID" column.

Then, we can create the following SQL statement (that contains an INNER JOIN), that selects records that have matching values in both tables:

EG:

SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate  
FROM Orders  
INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID;

O/P:

OrderIDCustomerNameOrderDate

10308Ana Trujillo Emparedados y helados9/18/1996

10365Antonio Moreno Taquería11/27/1996

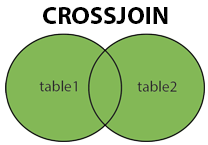
10383Around the Horn12/16/1996

10355Around the Horn11/15/1996

10278Berglunds snabbköp8/12/1996

Supported Types of Joins in MySQL

* INNER JOIN: Returns records that have matching values in both tables
* LEFT JOIN: Returns all records from the left table, and the matched records from the right table
* RIGHT JOIN: Returns all records from the right table, and the matched records from the left table
* CROSS JOIN: Returns all records from both tables

## MySQL INNER JOIN Keyword

The INNER JOIN keyword selects records that have matching values in both tables.



### **INNER JOIN Syntax**

SELECT column\_name(s)  
FROM table1  
INNER JOIN table2ON table1.column\_name = table2.column\_name;

Below is a selection from the "Orders" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OrderID** | **CustomerID** | **EmployeeID** | **OrderDate** | **ShipperID** |
| 10308 | 2 | 7 | 1996-09-18 | 3 |
| 10309 | 37 | 3 | 1996-09-19 | 1 |
| 10310 | 77 | 8 | 1996-09-20 | 2 |

And 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 |

## MySQL INNER JOIN Example

The following SQL statement selects all orders with customer information:

EG:

SELECT Orders.OrderID, Customers.CustomerName  
FROM Orders  
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID;

The INNER JOIN keyword selects all rows from both tables as long as there is a match between the columns. If there are records in the "Orders" table that do not have matches in "Customers", these orders will not be shown!

O/P:

|  |  |
| --- | --- |
| **OrderID** | **CustomerName** |
| 10248 | Wilman Kala |
| 10249 | Tradição Hipermercados |
| 10250 | Hanari Carnes |
| 10251 | Victuailles en stock |
| 10252 | Suprêmes délices |
| 10253 | Hanari Carnes |
| 10254 | Chop-suey Chinese |
| 10255 | Richter Supermarkt |
| 10256 | Wellington Importadora |
| 10257 | HILARIÓN-Abastos |
| 10258 | Ernst Handel |
| 10259 | Centro comercial Moctezuma |
| 10260 | Old World Delicatessen |
| 10261 | Que Delícia |
| 10262 | Rattlesnake Canyon Grocery |
| 10263 | Ernst Handel |
| 10264 | Folk och fä HB |
| 10265 | Blondel père et fils |
| 10266 | Wartian Herkku |
| 10267 | Frankenversand |
| 10268 | GROSELLA-Restaurante |
| 10269 | White Clover Markets |
| 10270 | Wartian Herkku |
| 10271 | Split Rail Beer & Ale |
| 10272 | Rattlesnake Canyon Grocery |

## JOIN Three Tables

The following SQL statement selects all orders with customer and shipper information:

### **Ex:**

SELECT Orders.OrderID, Customers.CustomerName, Shippers.ShipperName  
FROM ((Orders  
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID)  
INNER JOIN Shippers ON Orders.ShipperID = Shippers.ShipperID);

O/P:

|  |  |  |
| --- | --- | --- |
| **OrderID** | **CustomerName** | **ShipperName** |
| 10248 | Wilman Kala | Federal Shipping |
| 10249 | Tradição Hipermercados | Speedy Express |
| 10250 | Hanari Carnes | United Package |
| 10251 | Victuailles en stock | Speedy Express |
| 10252 | Suprêmes délices | United Package |
| 10253 | Hanari Carnes | United Package |
| 10254 | Chop-suey Chinese | United Package |
| 10255 | Richter Supermarkt | Federal Shipping |
| 10256 | Wellington Importadora | United Package |
| 10257 | HILARIÓN-Abastos | Federal Shipping |
| 10258 | Ernst Handel | Speedy Express |
| 10259 | Centro comercial Moctezuma | Federal Shipping |
| 10260 | Old World Delicatessen | Speedy Express |
| 10261 | Que Delícia | United Package |
| 10262 | Rattlesnake Canyon Grocery | Federal Shipping |
| 10263 | Ernst Handel | Federal Shipping |
| 10264 | Folk och fä HB | Federal Shipping |
| 10265 | Blondel père et fils | Speedy Express |
| 10266 | Wartian Herkku | Federal Shipping |
| 10267 | Frankenversand | Speedy Express |
| 10268 | GROSELLA-Restaurante | Federal Shipping |

## MySQL LEFT JOIN Keyword

The LEFT JOIN keyword returns all records from the left table (table1), and the matching records (if any) from the right table (table2).



### **LEFT JOIN Syntax**

SELECT column\_name(s)  
FROM table1  
LEFT JOIN table2ON table1.column\_name = table2.column\_name;

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 "Orders" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OrderID** | **CustomerID** | **EmployeeID** | **OrderDate** | **ShipperID** |
| 10308 | 2 | 7 | 1996-09-18 | 3 |
| 10309 | 37 | 3 | 1996-09-19 | 1 |
| 10310 | 77 | 8 | 1996-09-20 | 2 |

## MySQL LEFT JOIN Example

SQL statement will select all customers, and any orders they might have:

EG:

SELECT Customers.CustomerName, Orders.OrderID  
FROM Customers  
LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID  
ORDER BY Customers.CustomerName;

O/P:

|  |  |
| --- | --- |
| **CustomerName** | **OrderID** |
| Alfreds Futterkiste | 10643 |
| Alfreds Futterkiste | 10835 |
| Alfreds Futterkiste | 11011 |
| Alfreds Futterkiste | 10952 |
| Alfreds Futterkiste | 10692 |
| Alfreds Futterkiste | 10702 |
| Ana Trujillo Emparedados y helados | 10926 |
|  |  |

The LEFT JOIN keyword returns all records from the left table (Customers), even if there are no matches in the right table (Orders).

## MySQL RIGHT JOIN Keyword

The RIGHT JOIN keyword returns all records from the right table (table2), and the matching records (if any) from the left table (table1).



### **RIGHT JOIN Syntax**

SELECT column\_name(s)  
FROM table1  
RIGHT JOIN table2ON table1.column\_name = table2.column\_name;

a selection from the "Orders" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OrderID** | **CustomerID** | **EmployeeID** | **OrderDate** | **ShipperID** |
| 10308 | 2 | 7 | 1996-09-18 | 3 |
| 10309 | 37 | 3 | 1996-09-19 | 1 |
| 10310 | 77 | 8 | 1996-09-20 | 2 |

And a selection from the "Employees" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EmployeeID** | **LastName** | **FirstName** | **BirthDate** | **Photo** |
| 1 | Davolio | Nancy | 12/8/1968 | EmpID1.pic |
| 2 | Fuller | Andrew | 2/19/1952 | EmpID2.pic |
| 3 | Leverling | Janet | 8/30/1963 | EmpID3.pic |

## MySQL RIGHT JOIN Example

The following SQL statement will return all employees, and any orders they might have placed:

EG:

SELECT Orders.OrderID, Employees.LastName, Employees.FirstName  
FROM Orders  
RIGHT JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID  
ORDER BY Orders.OrderID;

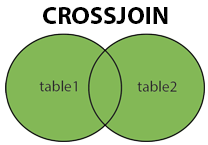
O/P:

|  |  |  |
| --- | --- | --- |
| **OrderID** | **LastName** | **FirstName** |
| 10248 | Buchanan | Steven |
| 10249 | Suyama | Michael |
| 10250 | Peacock | Margaret |
| 10251 | Leverling | Janet |
| 10252 | Peacock | Margaret |
| 10253 | Leverling | Janet |
| 10254 | Buchanan | Steven |
| 10255 | Dodsworth | Anne |
| 10256 | Leverling | Janet |
| 10257 | Peacock | Margaret |
| 10258 | Davolio | Nancy |
| 10259 | Peacock | Margaret |
| 10260 | Peacock | Margaret |
| 10261 | Peacock | Margaret |
| 10262 | Callahan | Laura |
| 10263 | Dodsworth | Anne |

The RIGHT JOIN keyword returns all records from the right table (Employees), even if there are no matches in the left table (Orders).

## SQL CROSS JOIN Keyword

The CROSS JOIN keyword returns all records from both tables (table1 and table2).



### **CROSS JOIN Syntax**

SELECT column\_name(s)  
FROM table1  
CROSS JOIN table2;

**Note:** CROSS JOIN can potentially return very large result-sets!

 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 "Orders" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OrderID** | **CustomerID** | **EmployeeID** | **OrderDate** | **ShipperID** |
| 10308 | 2 | 7 | 1996-09-18 | 3 |
| 10309 | 37 | 3 | 1996-09-19 | 1 |
| 10310 | 77 | 8 | 1996-09-20 | 2 |

MySQL CROSS JOIN Example

The following SQL statement selects all customers, and all orders:

EG:

SELECT Customers.CustomerName, Orders.OrderID  
FROM Customers  
CROSS JOIN Orders;

O/P:

|  |  |
| --- | --- |
| **CustomerName** | **OrderID** |
| Alfreds Futterkiste | 10248 |
| Ana Trujillo Emparedados y helados | 10248 |
| Antonio Moreno Taquería | 10248 |
| Around the Horn | 10248 |

The CROSS JOIN keyword returns all matching records from both tables whether the other table matches or not. So, if there are rows in "Customers" that do not have matches in "Orders", or if there are rows in "Orders" that do not have matches in "Customers", those rows will be listed as well.

If you add a WHERE clause (if table1 and table2 has a relationship), the CROSS JOIN will produce the same result as the INNER JOIN clause:

EG:

SELECT Customers.CustomerName, Orders.OrderID  
FROM Customers  
CROSS JOIN Orders  
WHERE Customers.CustomerID=Orders.CustomerID;

O/P:

|  |  |
| --- | --- |
| **CustomerName** | **OrderID** |
| Wilman Kala | 10248 |
| Tradição Hipermercados | 10249 |
| Hanari Carnes | 10250 |
| Victuailles en stock | 10251 |

What is subquery in SQL?

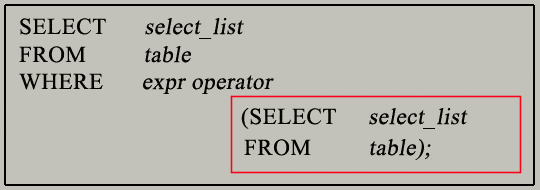
A subquery is a SQL query nested inside a larger query.

* A subquery may occur in :
  + - A SELECT clause
  + - A FROM clause
  + - A WHERE clause
* The subquery can be nested inside a SELECT, INSERT, UPDATE, or DELETE statement or inside another subquery.
* A subquery is usually added within the WHERE Clause of another SQL SELECT statement.
* You can use the comparison operators, such as >, <, or =. The comparison operator can also be a multiple-row operator, such as IN, ANY, or ALL.
* A subquery is also called an inner query or inner select, while the statement containing a subquery is also called an outer query or outer select.
* The inner query executes first before its parent query so that the results of an inner query can be passed to the outer query.

You can use a subquery in a SELECT, INSERT, DELETE, or UPDATE statement to perform the following tasks:

* Compare an expression to the result of the query.
* Determine if an expression is included in the results of the query.
* Check whether the query selects any rows.

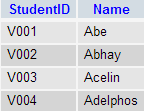
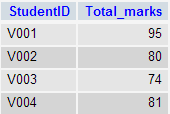
**Syntax :**



* The subquery (inner query) executes once before the main query (outer query) executes.
* The main query (outer query) use the subquery result.

SQL Subqueries Example :

In this section, you will learn the requirements of using subqueries. We have the following two tables 'student' and 'marks' with common field 'StudentID'.

  
           
            student                                        marks

Now we want to write a query to identify all students who get better marks than that of the student who's StudentID is 'V002', but we do not know the marks of 'V002'.  
- To solve the problem, we require two queries. One query returns the marks (stored in Total\_marks field) of 'V002' and a second query identifies the students who get better marks than the result of the first query.

**First query:**

SELECT \*

FROM `marks`

WHERE studentid = 'V002';

Copy

**Query result:**

student query

The result of the query is 80.  
- Using the result of this query, here we have written another query to identify the students who get better marks than 80. Here is the query :

**Second query:**

SELECT a.studentid, a.name, b.total\_marks

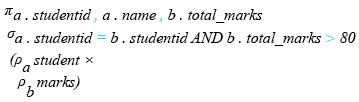
FROM student a, marks b

WHERE a.studentid = b.studentid

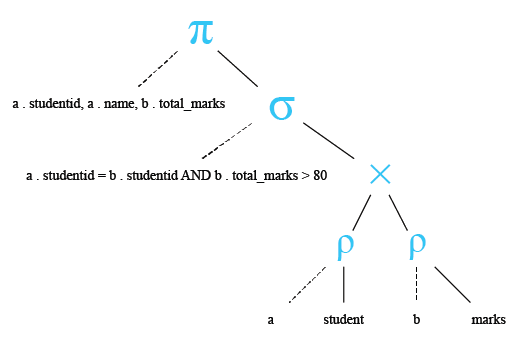
AND b.total\_marks >80;

Copy

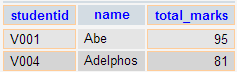
**Relational Algebra Expression:**



**Relational Algebra Tree:**



**Query result:**



Above two queries identified students who get the better number than the student who's StudentID is 'V002' (Abhay).

You can combine the above two queries by placing one query inside the other. The subquery (also called the 'inner query') is the query inside the parentheses. See the following code and query result :

**SQL Code:**

SELECT a.studentid, a.name, b.total\_marks

FROM student a, marks b

WHERE a.studentid = b.studentid AND b.total\_marks >

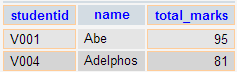
(SELECT total\_marks

FROM marks

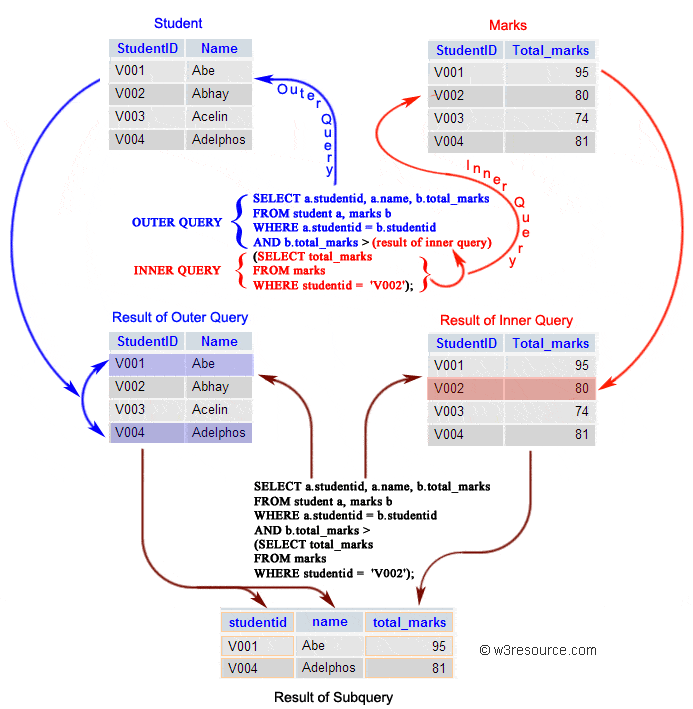
WHERE studentid = 'V002');

Copy

**Query result:**



**Pictorial Presentation of SQL Subquery:**

[](https://www.w3resource.com/sql/subqueries/sql-subqueries.gif)

Subqueries: General Rules

A subquery SELECT statement is almost similar to the SELECT statement and it is used to begin a regular or outer query. Here is the syntax of a subquery:

**Syntax:**

(SELECT [DISTINCT] subquery\_select\_argument

FROM {table\_name | view\_name}

{table\_name | view\_name} ...

[WHERE search\_conditions]

[GROUP BY aggregate\_expression [, aggregate\_expression] ...]

[HAVING search\_conditions])

Subqueries: Guidelines

There are some guidelines to consider when using subqueries :

* A subquery must be enclosed in parentheses.
* A subquery must be placed on the right side of the comparison operator.
* Subqueries cannot manipulate their results internally, therefore ORDER BY clause cannot be added into a subquery. You can use an ORDER BY clause in the main SELECT statement (outer query) which will be the last clause.
* Use single-row operators with single-row subqueries.
* If a subquery (inner query) returns a null value to the outer query, the outer query will not return any rows when using certain comparison operators in a WHERE clause.

Type of Subqueries

* Single row subquery : Returns zero or one row.
* Multiple row subquery : Returns one or more rows.
* Multiple column subqueries : Returns one or more columns.
* Correlated subqueries : Reference one or more columns in the outer SQL statement. The subquery is known as a correlated subquery because the subquery is related to the outer SQL statement.
* Nested subqueries : Subqueries are placed within another subquery.

In the next session, we have thoroughly discussed the above topics. Apart from the above type of subqueries, you can use a subquery inside INSERT, UPDATE and DELETE statement. Here is a brief discussion :

Subqueries with INSERT statement

INSERT statement can be used with subqueries. Here are the syntax and an example of subqueries using INSERT statement.

**Syntax:**

INSERT INTO table\_name [ (column1 [, column2 ]) ]

SELECT [ \*|column1 [, column2 ]

FROM table1 [, table2 ]

[ WHERE VALUE OPERATOR ];

If we want to insert those orders from 'orders' table which have the advance\_amount 2000 or 5000 into 'neworder' table the following SQL can be used:

Sample table: orders

**SQL Code:**

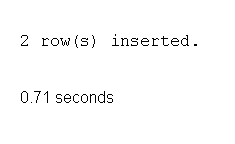
INSERT INTO neworder

SELECT \* FROM orders

WHERE advance\_amount in(2000,5000);

Copy

Output:



To see more details of subqueries using INSERT statement [click here](https://www.w3resource.com/sql/insert-statement/insert-using-subqueries.php).

Subqueries with UPDATE statement

In a UPDATE statement, you can set new column value equal to the result returned by a single row subquery. Here are the syntax and an example of subqueries using UPDATE statement.

**Syntax:**

UPDATE table SET column\_name = new\_value

[ WHERE OPERATOR [ VALUE ]

(SELECT COLUMN\_NAME

FROM TABLE\_NAME)

[ WHERE) ]

If we want to update that ord\_date in 'neworder' table with '15-JAN-10' which have the difference of ord\_amount and advance\_amount is less than the minimum ord\_amount of 'orders' table the following SQL can be used:

Sample table: neworder

**SQL Code:**

UPDATE neworder

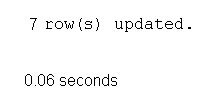
SET ord\_date='15-JAN-10'

WHERE ord\_amount-advance\_amount<

(SELECT MIN(ord\_amount) FROM orders);

Copy

Output:



To see more details of subqueries using UPDATE statement [click here](https://www.w3resource.com/sql/update-statement/update-using-subqueries.php).

Subqueries with DELETE statement

DELETE statement can be used with subqueries. Here are the syntax and an example of subqueries using DELETE statement.

**Syntax:**

DELETE FROM TABLE\_NAME

[ WHERE OPERATOR [ VALUE ]

(SELECT COLUMN\_NAME

FROM TABLE\_NAME)

[ WHERE) ]

If we want to delete those orders from 'neworder' table which advance\_amount are less than the maximum advance\_amount of 'orders' table, the following SQL can be used:

Sample table: neworder

**SQL Code:**

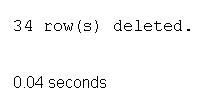
DELETE FROM neworder

WHERE advance\_amount<

(SELECT MAX(advance\_amount) FROM orders);

Copy

Output:



# SQL Correlated Subqueries

Last update on August 19 2022 21:50:45 (UTC/GMT +8 hours)

## Correlated Subqueries

SQL Correlated Subqueries are used to select data from a table referenced in the outer query. The subquery is known as a correlated because the subquery is related to the outer query. In this type of queries, a table alias (also called a correlation name) must be used to specify which table reference is to be used.

Uncorrelated subquery executes the subquery first and provides the value to the outer query, whereas correlated subquery references a column in the outer query and executes the subquery once for each row in the outer query..

The alias is the pet name of a table which is brought about by putting directly after the table name in the FROM clause. This is suitable when anybody wants to obtain information from two separate tables.

**Example: SQL Correlated Subqueries**

The following correlated subqueries retrive ord\_num, ord\_amount, cust\_code and agent\_code from the table orders ( **'a'** and **'b'** are the aliases of orders and agents table) with following conditions -

the agent\_code of orders table must be the same agent\_code of agents table and agent\_name of agents table must be Alex**,**

the following SQL statement can be used:

Sample table: orders

Sample table: agents

**SQL Code:**

SELECT a.ord\_num,a.ord\_amount,a.cust\_code,a.agent\_code

FROM orders a

WHERE a.agent\_code=(

SELECT b.agent\_code

FROM agents b WHERE b.agent\_name='Alex');

Copy

Output:

ORD\_NUM ORD\_AMOUNT CUST\_CODE AGENT\_CODE

---------- ---------- ---------- ----------

200127 2500 C00015 A003

200100 1000 C00015 A003

The inner of the above query returns the 'agent\_code' A003.

The simplified form of above code is:

**SQL Code:**

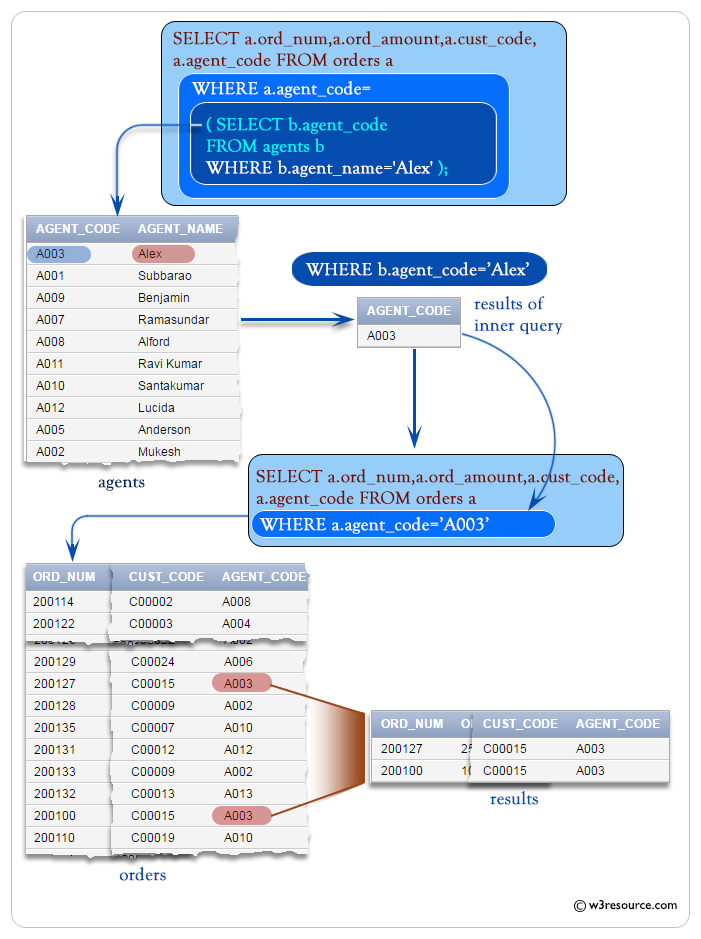
SELECT a.ord\_num,a.ord\_amount,a.cust\_code,a.agent\_code

FROM orders a

WHERE a.agent\_code='A003';

Copy

**Pictorical Presentation:**



## MySQL CREATE INDEX Statement

The CREATE INDEX statement is used to create indexes in tables.

Indexes are used to retrieve data from the database more quickly than otherwise. The users cannot see the indexes, they are just used to speed up searches/queries.

### **CREATE INDEX Syntax**

Creates an index on a table. Duplicate values are allowed:

CREATE INDEX index\_name  
ON table\_name (column1, column2, ...);

### **CREATE UNIQUE INDEX Syntax**

Creates a unique index on a table. Duplicate values are not allowed:

CREATE UNIQUE INDEX index\_name  
ON table\_name (column1, column2, ...);

## MySQL CREATE INDEX Example

The SQL statement below creates an index named "idx\_lastname" on the "LastName" column in the "Persons" table:

CREATE INDEX idx\_lastname  
ON Persons (LastName);

If you want to create an index on a combination of columns, you can list the column names within the parentheses, separated by commas:

CREATE INDEX idx\_pname  
ON Persons (LastName, FirstName);

## DROP INDEX Statement

The DROP INDEX statement is used to delete an index in a table.

ALTER TABLE table\_nameDROP INDEX index\_name;

## MySQL 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 statements and functions to a view and present the data as if the data were coming from one single table.

A view is created with the CREATE VIEW statement.

### **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 view, every time a user queries it.

## MySQL 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];

## MySQL Updating a View

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

### **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';

## MySQL Dropping a View

A view is deleted with the DROP VIEW statement.

### **DROP VIEW Syntax**

DROP VIEW view\_name;

The following SQL drops the "Brazil Customers" view:

### **Example**

DROP VIEW [Brazil Customers];

# **MySQL AUTO INCREMENT Field**

Auto-increment allows a unique number to be generated automatically when a new record is inserted into a table.

Often this is the primary key field that we would like to be created automatically every time a new record is inserted.

MySQL AUTO\_INCREMENT Keyword

MySQL uses the AUTO\_INCREMENT keyword to perform an auto-increment feature.

By default, the starting value for AUTO\_INCREMENT is 1, and it will increment by 1 for each new record.

The following SQL statement defines the "Personid" column to be an auto-increment primary key field in the "Persons" table:

CREATE TABLE Persons (  
    Personid int NOT NULL AUTO\_INCREMENT,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int,  
    PRIMARY KEY (Personid)  
);

To let the AUTO\_INCREMENT sequence start with another value, use the following SQL statement:

ALTER TABLE Persons AUTO\_INCREMENT=100;

When we insert a new record into the "Persons" table, we do NOT have to specify a value for the "Personid" column (a unique value will be added automatically):

INSERT INTO Persons (FirstName,LastName)  
VALUES ('Lars','Monsen');

The SQL statement above would insert a new record into the "Persons" table. The "Personid" column would be assigned a unique value automatically. The "FirstName" column would be set to "Lars" and the "LastName" column would be set to "Monsen".

## MySQL Dates

As long as your data contains only the date portion, your queries will work as expected. However, if a time portion is involved, it gets more complicated.

MySQL Date Data Types

MySQL comes with the following data types for storing a date or a date/time value in the database:

* DATE - format YYYY-MM-DD
* DATETIME - format: YYYY-MM-DD HH:MI:SS
* TIMESTAMP - format: YYYY-MM-DD HH:MI:SS
* YEAR - format YYYY or YY

**Note:** The date data type are set for a column when you create a new table in your database!

### **Orders Table**

|  |  |  |
| --- | --- | --- |
| **OrderId** | **ProductName** | **OrderDate** |
| 1 | Geitost | 2008-11-11 |
| 2 | Camembert Pierrot | 2008-11-09 |
| 3 | Mozzarella di Giovanni | 2008-11-11 |
| 4 | Mascarpone Fabioli | 2008-10-29 |

Now we want to select the records with an OrderDate of "2008-11-11" from the table above.

We use the following SELECT statement:

SELECT \* FROM Orders WHERE OrderDate='2008-11-11'

The result-set will look like this:

|  |  |  |
| --- | --- | --- |
| **OrderId** | **ProductName** | **OrderDate** |
| 1 | Geitost | 2008-11-11 |
| 3 | Mozzarella di Giovanni | 2008-11-11 |

# **MySQL Functions**

MySQL has many built-in functions.

This reference contains string, numeric, date, and some advanced functions in MySQL.

## The MySQL 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 |

## MySQL GROUP BY Examples

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

EG:

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

## The MySQL 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 |

## MySQL HAVING Examples

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

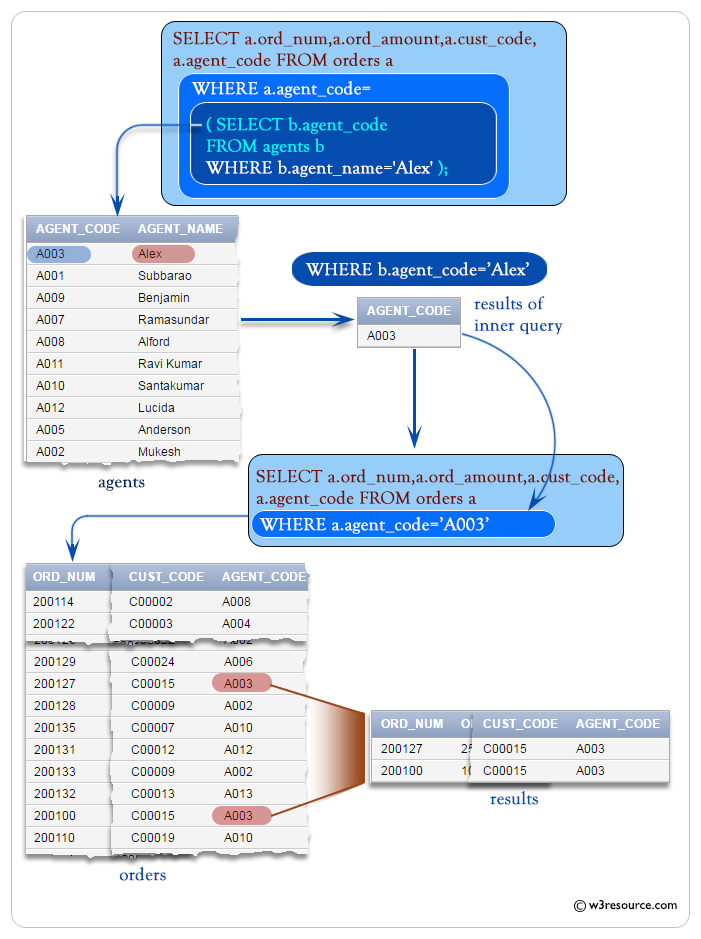
EG:

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):

EG:

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



## The MySQL 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;

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 |

## MySQL CASE Examples

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

EG:

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 QuantityText  
FROM OrderDetails;

O/P:

|  |  |  |
| --- | --- | --- |
| **OrderID** | **Quantity** | **QuantityText** |
| 10248 | 12 | The quantity is under 30 |
| 10248 | 10 | The quantity is under 30 |
| 10248 | 5 | The quantity is under 30 |
| 10249 | 9 | The quantity is under 30 |
| 10249 | 40 | The quantity is greater than 30 |
| 10250 | 10 | The quantity is under 30 |
| 10250 | 35 | The quantity is greater than 30 |

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

EG:

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

O/P:

|  |  |  |
| --- | --- | --- |
| **CustomerName** | **City** | **Country** |
| Drachenblut Delikatessend | Aachen | Germany |
| Rattlesnake Canyon Grocery | Albuquerque | USA |
| Old World Delicatessen | Anchorage | USA |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **TCL Commands in SQL**  * In SQL, TCL stands for **Transaction control language**. * A single unit of work in a database is formed after the consecutive execution of commands is known as a transaction. * There are certain commands present in SQL known as TCL commands that help the user manage the transactions that take place in a database. * **COMMIT. ROLLBACK** and **SAVEPOINT** are the most commonly used TCL commands in SQL.   Now let us take a deeper dive into the TCL commands of SQL with the help of examples. All the queries in the examples will be written using the MySQL database. **1. COMMIT** COMMIT command in SQL is used to save all the transaction-related changes permanently to the disk. Whenever DDL commands such as INSERT, UPDATE and DELETE are used, the changes made by these commands are permanent only after closing the current session. So before closing the session, one can easily roll back the changes made by the DDL commands. Hence, if we want the changes to be saved permanently to the disk without closing the session, we will use the commit command.  **Syntax:**   1. **COMMIT**;   **Example:**  We will select an existing database, i.e., school.   1. mysql> USE school;   TCL Commands in SQL  To create a table named t\_school, we will execute the following query:   1. mysql> **CREATE** **TABLE** t\_school(ID **INT**, School\_Name **VARCHAR**(40), Number\_Of\_Students **INT**, Number\_Of\_Teachers **INT**, Number\_Of\_Classrooms **INT**, EmailID **VARCHAR**(40));   TCL Commands in SQL  BEGIN / START TRANSACTION command is used to start the transaction.   1. mysql> START **TRANSACTION**;   TCL Commands in SQL  Now, we will execute the following query to insert multiple records at the same time in the t\_school table.   1. mysql> **INSERT** **INTO** t\_school(ID, School\_Name, Number\_Of\_Students, Number\_Of\_Teachers, Number\_Of\_Classrooms, EmailID) **VALUES**(1, "Boys Town Public School", 1000, 80, 12, "btps15@gmail.com"), (2, "Guru Govind Singh Public School", 800, 35, 15, "ggps25@gmail.com"), (3, "Delhi Public School", 1200, 30, 10, "dps101@gmail.com"), (4, "Ashoka Universal School", 1110, 40, 40, "aus17@gmail.com"), (5, "Calibers English Medium School", 9000, 31, 50, "cems@gmail.com");   TCL Commands in SQL  We will now execute the SELECT query to verify the execution of the INSERT INTO query executed above.   1. mysql> **SELECT** \***FROM** t\_school;   After executing the SELECT query on the t\_school table, you will get the following output:   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | **ID** | **School\_Name** | **Number\_Of\_Students** | **Number\_Of\_Teachers** | **Number\_Of\_Classrooms** | **EmailID** | | 1 | Boys Town Public School | 1000 | 80 | 12 | btps15@gmail.com | | 2 | Guru Govind Singh Public School | 800 | 35 | 15 | ggps25@gmail.com | | 3 | Delhi Public School | 1200 | 30 | 10 | dps101@gmail.com | | 4 | Ashoka Universal School | 1110 | 40 | 40 | aus17@gmail.com | | 5 | Calibers English Medium School | 9000 | 31 | 50 | cems@gmail.com |   The output of the SELECT query shows that all the records are inserted successfully.  We will execute the COMMIT command to save the results of the operations carried on the t\_school table.   1. mysql> **COMMIT**;   TCL Commands in SQL  Autocommit is by default enabled in MySQL. To turn it off, we will set the value of autocommit as 0.   1. mysql> **SET** autocommit = 0;   TCL Commands in SQL  MySQL, by default, commits every query the user executes. But if the user wishes to commit only the specific queries instead of committing every query, then turning off the autocommit is useful. **2. SAVEPOINT** We can divide the database operations into parts. For example, we can consider all the insert related queries that we will execute consecutively as one part of the transaction and the delete command as the other part of the transaction. Using the SAVEPOINT command in SQL, we can save these different parts of the same transaction using different names. **For example**, we can save all the insert related queries with the savepoint named INS. To save all the insert related queries in one savepoint, we have to execute the SAVEPOINT query followed by the savepoint name after finishing the insert command execution.  **Syntax:**   1. SAVEPOINT savepoint\_name;  **3. ROLLBACK** While carrying a transaction, we must create savepoints to save different parts of the transaction. According to the user's changing requirements, he/she can roll back the transaction to different savepoints. Consider a scenario: We have initiated a transaction followed by the table creation and record insertion into the table. After inserting records, we have created a savepoint INS. Then we executed a delete query, but later we thought that mistakenly we had removed the useful record. Therefore in such situations, we have an option of rolling back our transaction. In this case, we have to roll back our transaction using the ROLLBACK command to the savepoint INS, which we have created before executing the DELETE query.  **Syntax:**   1. **ROLLBACK** **TO** savepoint\_name;   **Examples to understand the SAVEPOINT and ROLLBACK commands:**  **Example 1:**  We will select an existing database, i.e., school.   1. mysql> USE school;   TCL Commands in SQL  To create a table named t\_school, we will execute the following query:   1. mysql> **CREATE** **TABLE** t\_school(ID **INT**, School\_Name **VARCHAR**(40), Number\_Of\_Students **INT**, Number\_Of\_Teachers **INT**, Number\_Of\_Classrooms **INT**, EmailID **VARCHAR**(40));   TCL Commands in SQL  Now, we will execute the following query to insert multiple records at the same time in the t\_school table.   1. mysql> **INSERT** **INTO** t\_school(ID, School\_Name, Number\_Of\_Students, Number\_Of\_Teachers, Number\_Of\_Classrooms, EmailID) **VALUES**(1, "Boys Town Public School", 1000, 80, 12, "btps15@gmail.com"), (2, "Guru Govind Singh Public School", 800, 35, 15, "ggps25@gmail.com"), (3, "Delhi Public School", 1200, 30, 10, "dps101@gmail.com"), (4, "Ashoka Universal School", 1110, 40, 40, "aus17@gmail.com"), (5, "Calibers English Medium School", 9000, 31, 50, "cems@gmail.com");   TCL Commands in SQL  We will now execute the SELECT query to verify the execution of the INSERT INTO query executed above.   1. mysql> **SELECT** \***FROM** t\_school;   After executing the SELECT query on the t\_school table, you will get the following output:   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | **ID** | **School\_Name** | **Number\_Of\_Students** | **Number\_Of\_Teachers** | **Number\_Of\_Classrooms** | **EmailID** | | 1 | Boys Town Public School 1000 | 80 | 12 | btps15@gmail.com |  | | 2 | Guru Govind Singh Public School | 800 | 35 | 15 | ggps25@gmail.com | | 3 | Delhi Public School | 1200 | 30 | 10 | dps101@gmail.com | | 4 | Ashoka Universal School | 1110 | 40 | 40 | aus17@gmail.com | | 5 | Calibers English Medium School | 9000 | 31 | 50 | cems@gmail.com |   The output of the SELECT query shows that all the records are inserted successfully.  BEGIN / START TRANSACTION command is used to start the transaction.   1. mysql> START **TRANSACTION**;   TCL Commands in SQL  As we know, the SAVEPOINT command in SQL is used to save the different parts of the same transaction using different names. Consider till this point as one part of our transaction. We will save this part using a savepoint named Insertion.   1. mysql> SAVEPOINT Insertion;   TCL Commands in SQL  Now, we will execute the update command on the t\_school table to set the Number\_Of\_Students as 9050 for the record with ID 5.   1. mysql> **UPDATE** t\_school **SET** Number\_Of\_Students = 9050 **WHERE** ID = 5;   TCL Commands in SQL  To verify that the record with ID 5 now has the Number\_Of\_Students as 9050, we will execute the SELECT query.   1. mysql> **SELECT** \***FROM** t\_school;   After executing the SELECT query on the t\_school table, you will get the following output:   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | **ID** | **School\_Name** | **Number\_Of\_Students** | **Number\_Of\_Teachers** | **Number\_Of\_Classrooms** | **EmailID** | | 1 | Boys Town Public School | 1000 | 80 | 12 | btps15@gmail.com | | 2 | Guru Govind Singh Public School | 800 | 35 | 15 | ggps25@gmail.com | | 3 | Delhi Public School | 1200 | 30 | 10 | dps101@gmail.com | | 4 | Ashoka Universal School | 1110 | 40 | 40 | aus17@gmail.com | | 5 | Calibers English Medium School | 9050 | 31 | 50 | cems@gmail.com |   The output of the SELECT query shows that the record with ID 5 is updated successfully.  Consider the update operation as one part of our transaction. We will save this part using a savepoint named Updation.   1. mysql> SAVEPOINT Updation;   TCL Commands in SQL  Suddenly, our requirement changed, and we realized that we had updated a record that was not supposed to be. In such a scenario, we need to roll back our transaction to the savepoint, which was created prior to the execution of the UPDATE command.   1. mysql> **ROLLBACK** **TO** Insertion;   TCL Commands in SQL  We didn't need the updation carried on the record. Hence, we have rolled back to the savepoint named Insertion.  For confirming that we have got the same t\_school table that we had before carrying out the updation operation, we will again execute the SELECT query.   1. mysql> **SELECT** \***FROM** t\_school;  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | **ID** | **School\_Name** | **Number\_Of\_Students** | **Number\_Of\_Teachers** | **Number\_Of\_Classrooms** | **EmailID** | | 1 | Boys Town Public School | 1000 | 80 | 12 | btps15@gmail.comm | | 2 | Guru Govind Singh Public School | 800 | 35 | 15 | ggps25@gmail.comm | | 3 | Delhi Public School | 1200 | 30 | 10 | dps101@gmail.comm | | 4 | Ashoka Universal School | 1110 | 40 | 40 | aus17@gmail.comm | | 5 | Calibers English Medium School | 9000 | 31 | 50 | cems@gmail.com |   The SELECT query output confirms that the transaction is now successfully rolled back to the savepoint 'Insertion'.  **Example 2:**  We will select an existing database, i.e., bank.   1. mysql> USE bank;   TCL Commands in SQL  To create a table named customer, we will execute the following query:   1. mysql> **CREATE** **TABLE** customer(Customer\_ID **INT** **PRIMARY** **KEY**, **Name** **VARCHAR**(20), Age **INT**, Salary **INT**, Salary\_BankAccount **VARCHAR**(20));   TCL Commands in SQL  Now, we will execute the following query to insert multiple records at the same time in the customer table.   1. mysql> **INSERT** **INTO** customer(Customer\_ID, **Name**, Age, Salary, Salary\_BankAccount) **VALUES**(1, "Aryan Jain", 51, 56000, "SBI"), (2, "Arohi Dixit", 21, 25000, "Axis"), (3, "Vineet Garg", 24, 31000, "ICICI"), (4, "Anuja Sharma", 26, 49000, "HDFC"), (5, "Deepak Kohli", 28, 65000, "SBI");   TCL Commands in SQL  We will now execute the SELECT query to verify the execution of the INSERT INTO query executed above.   1. mysql> **SELECT** \***FROM** customer;   After executing the SELECT query on the t\_school table, you will get the following output:   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Customer\_ID** | **Name** | **Age** | **Salary** | **Salary\_BankAccount** | | 1 | Aryan Jain | 51 | 56000 | SBI | | 2 | Arohi Dixit | 21 | 25000 | Axis | | 3 | Vineet Garg | 24 | 31000 | ICICI | | 4 | Anuja Sharma | 26 | 49000 | HDFC | | 5 | Deepak Kohli | 28 | 65000 | SBI |   The output of the SELECT query shows that all the records are inserted successfully.  BEGIN / START TRANSACTION command is used to start the transaction.   1. mysql> START **TRANSACTION**;   TCL Commands in SQL  As we know, the SAVEPOINT command in SQL is used to save the different parts of the same transaction using different names. Consider till this point as one part of our transaction. We will save this part using a savepoint named Insertion.   1. mysql> SAVEPOINT Insertion;   TCL Commands in SQL  We will execute the delete command on the customer table to remove the record with ID 5.   1. mysql> **DELETE** **FROM** customer **WHERE** Customer\_ID = 5;   TCL Commands in SQL  We will execute the SELECT query to verify that the record with ID 5 has been removed.   1. mysql> **SELECT** \***FROM** customer;  |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Customer\_ID** | **Name** | **Age** | **Salary** | **Salary\_BankAccount** | | 1 | Aryan Jain | 51 | 56000 | SBI | | 2 | Arohi Dixit | 21 | 25000 | Axis | | 3 | Vineet Garg | 24 | 31000 | ICICI | | 4 | Anuja Sharma | 26 | 49000 | HDFC |   The output of the SELECT query shows that the record with ID 5 is removed successfully.  Consider the delete operation as one part of our transaction. We will save this part using a savepoint named Deletion.   1. mysql> SAVEPOINT Deletion;   TCL Commands in SQL  Suddenly, our requirement changed, and we realized that we had deleted a record that was not supposed to be. In such a scenario, we need to roll back our transaction to the savepoint, which was created prior to the execution of the DELETE command.   1. mysql> **ROLLBACK** **TO** Insertion;   We didn't need the deletion carried on the record. Hence, we have rolled back to the savepoint named Insertion.  For confirming that we have got the same customer table that we had before carrying out the deletion operation, we will again execute the SELECT query.   1. mysql> **SELECT** \***FROM** customer;  |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Customer\_ID** | **Name** | **Age** | **Salary** | **Salary\_BankAccount** | | 1 | Aryan Jain | 51 | 56000 | SBI | | 2 | Arohi Dixit | 21 | 25000 | Axis | | 3 | Vineet Garg | 24 | 31000 | ICICI | | 4 | Anuja Sharma | 26 | 49000 | HDFC | | 5 | Deepak Kohli | 28 | 65000 | SBI |   The SELECT query output confirms that the transaction is now successfully rolled back to the savepoint 'Insertion'. **SQL Subquery** The Subquery or Inner query is an SQL query placed inside another SQL query. It is embedded in the HAVING or WHERE clause of the SQL statements.  **Following are the important rules which must be followed by the SQL Subquery:**  1. The SQL subqueries can be used with the following statements along with the SQL expression operators:   * SELECT statement, * UPDATE statement, * INSERT statement, and * DELETE statement.   2. The subqueries in SQL are always enclosed in the parenthesis and placed on the right side of the SQL operators.  3. We cannot use the ORDER BY clause in the subquery. But, we can use the GROUP BY clause, which performs the same function as the ORDER BY clause.  4. If the subquery returns more than one record, we have to use the multiple value operators before the Subquery.  5. We can use the BETWEEN operator within the subquery but not with the subquery. Subquery with SELECT statement In SQL, inner queries or nested queries are used most frequently with the SELECT statement. The syntax of Subquery with the SELECT statement is described in the following block:   1. SELECT Column\_Name1, Column\_Name2, ...., Column\_NameN 2. FROM Table\_Name WHERE Column\_Name Comparison\_Operator 3. ( SELECT Column\_Name1, Column\_Name2, ...., Column\_NameN 4. FROM Table\_Name WHERE condition;   **Examples of Subquery with the SELECT Statement**  **Example 1:** This example uses the Greater than comparison operator with the Subquery.  Let's take the following table named Student\_Details, which contains Student\_RollNo., Stu\_Name, Stu\_Marks, and Stu\_City column.   |  |  |  |  | | --- | --- | --- | --- | | **Student\_RollNo.** | **Stu\_Name** | **Stu\_Marks** | **Stu\_City** | | 1001 | Akhil | 85 | Agra | | 1002 | Balram | 78 | Delhi | | 1003 | Bheem | 87 | Gurgaon | | 1004 | Chetan | 95 | Noida | | 1005 | Diksha | 99 | Agra | | 1006 | Raman | 90 | Ghaziabad | | 1007 | Sheetal | 68 | Delhi |   The following SQL query returns the record of those students whose marks are greater than the average of total marks:   1. SELECT \* FROM Student\_Details WHERE Stu\_Marks**>** ( SELECT AVG(Stu\_Marks ) FROM Student\_Details);   **Output:**   |  |  |  |  | | --- | --- | --- | --- | | **Student\_RollNo.** | **Stu\_Name** | **Stu\_Marks** | **Stu\_City** | | 1003 | Bheem | 87 | Gurgaon | | 1004 | Chetan | 95 | Noida | | 1005 | Diksha | 99 | Agra | | 1006 | Raman | 90 | Ghaziabad |   **Example 2:** This example uses the IN operator with the subquery.  Let's take the following two tables named **Faculty\_Details** and **Department** tables. The **Faculty\_Details**table contains ID, Name, Dept\_ID, and address of faculties. And, the Department table contains the Dept\_ID, Faculty\_ID, and Dept\_Name.   |  |  |  |  | | --- | --- | --- | --- | | **Faculty\_ID** | **Name** | **Dept\_ID** | **Address** | | 101 | Bheem | 1 | Gurgaon | | 102 | Chetan | 2 | Noida | | 103 | Diksha | NULL | Agra | | 104 | Raman | 4 | Ghaziabad | | 105 | Yatin | 3 | Noida | | 106 | Anuj | NULL | Agra | | 107 | Rakes | 5 | Gurgaon |  |  |  |  | | --- | --- | --- | | **Dept\_ID** | **Faculty\_ID** | **Dept\_Name** | | 1 | 101 | BCA | | 2 | 102 | B.Tech | | 3 | 105 | BBA | | 4 | 104 | MBA | | 5 | 107 | MCA |  1. SELECT \* FROM Department WHERE Faculty\_ID IN ( 2. SELECT Faculty\_ID FROM Faculty WHERE City = 'Noida' OR City = 'Gurgaon' ) ;   **Output:**   |  |  |  | | --- | --- | --- | | **Dept\_ID** | **Faculty\_ID** | **Dept\_Name** | | 1 | 101 | BCA | | 2 | 102 | B.Tech | | 3 | 105 | BBA | | 5 | 107 | MCA |  Subquery with the INSERT statement We can also use the subqueries and nested queries with the INSERT statement in Structured Query Language. We can insert the results of the subquery into the table of the outer query. The syntax of Subquery with the INSERT statement is described in the following block:   1. INSERT INTO Table\_Name SELECT \* FROM Table\_Name WHERE Column\_Name Operator (Subquery);  **Examples of Subquery with the INSERT Statement** **Example1:** This example inserts the record of one table into another table using subquery with WHERE clause.  Let's take Old\_Employee and New\_Employee tables. The Old\_Employee and New\_Employee table contain the same number of columns. But, both the tables contain different records.   |  |  |  |  | | --- | --- | --- | --- | | **Emp\_ID** | **Emp\_Name** | **Emp\_Salary** | **Address** | | 1001 | Akhil | 50000 | Agra | | 1002 | Balram | 25000 | Delhi | | 1003 | Bheem | 45000 | Gurgaon | | 1004 | Chetan | 60000 | Noida | | 1005 | Diksha | 30000 | Agra | | 1006 | Raman | 50000 | Ghaziabad | | 1007 | Sheetal | 35000 | Delhi |   **Table: Old\_Employee**   |  |  |  |  | | --- | --- | --- | --- | | **Emp\_ID** | **Emp\_Name** | **Emp\_Salary** | **Address** | | 1008 | Sumit | 50000 | Agra | | 1009 | Akash | 55000 | Delhi | | 1010 | Devansh | 65000 | Gurgaon |   **Table: New\_Employee**  The New\_Employee contains the details of new employees. If you want to move the details of those employees whose salary is greater than 40000 from the Old\_Employee table to the New\_Employee table. Then for this issue, you have to type the following query in SQL:   1. INSERT INTO New\_Employee SELECT \* FROM Old\_Employee WHERE Emp\_Salary **>** 40000;   Now, you can check the details of the updated New\_Employee table by using the following SELECT query:   1. SELECT \* FROM New\_Employee;   **Output:**   |  |  |  |  | | --- | --- | --- | --- | | **Emp\_ID** | **Emp\_Name** | **Emp\_Salary** | **Address** | | 1008 | Sumit | 50000 | Agra | | 1009 | Akash | 55000 | Delhi | | 1010 | Devansh | 65000 | Gurgaon | | 1001 | Akhil | 50000 | Agra | | 1003 | Bheem | 45000 | Gurgaon | | 1004 | Chetan | 60000 | Noida | | 1006 | Raman | 50000 | Ghaziabad |   **Table: New\_Employee**  **Example 2:** This example describes how to use ANY operator with subquery in the INSERT Statement.  Here we have taken the New\_Employee, old\_Employee, and Department table.  The data of the New\_Employee table is shown in the following table:   |  |  |  |  | | --- | --- | --- | --- | | **Emp\_ID** | **Emp\_Name** | **Emp\_Salary** | **Dept\_ID** | | 1008 | Sumit | 50000 | 401 |   **Table: New\_Employee**  The data of the old\_Employee table is shown in the below table:   |  |  |  |  | | --- | --- | --- | --- | | **Emp\_ID** | **Emp\_Name** | **Emp\_Salary** | **Dept\_ID** | | 1001 | Akhil | 50000 | 404 | | 1002 | Balram | 25000 | 403 | | 1003 | Bheem | 45000 | 405 | | 1004 | Chetan | 60000 | 402 | | 1005 | Ram | 65000 | 407 | | 1006 | Shyam | 55500 | NULL | | 1007 | Shobhit | 60000 | NULL |   **Table: Old\_Employee**  The data of Department table is shown in the below table:   |  |  |  | | --- | --- | --- | | **Dept\_ID** | **Dept\_Name** | **Emp\_ID** | | 401 | Administration | 1008 | | 402 | HR | 1004 | | 403 | Testing | 1002 | | 404 | Coding | 1001 | | 405 | Sales | 1003 | | 406 | Marketing | NULL | | 407 | Accounting | 1005 |   INSERT INTO New\_Employee  SELECT \* FROM Old\_Employee  WHERE Emp\_ID = ANY( SELECT Emp\_ID FROM Department WHERE Dept\_ID = 407 OR Dept\_ID = 406 );  Now, check the details of the New\_Employee table by using the following SELECT statement:  **Output:**   |  |  |  |  | | --- | --- | --- | --- | | **Emp\_ID** | **Emp\_Name** | **Emp\_Salary** | **Dept\_ID** | | 1008 | Sumit | 50000 | 401 | | 1005 | Ram | 65000 | 407 |  Subquery with the UPDATE statement The subqueries and nested queries can be used with the UPDATE statement in Structured Query Language for updating the columns of the existing table. We can easily update one or more columns using a subquery with the UPDATE statement.  **Syntax of Subquery with the UPDATE statement**   1. UPDATE Table\_Name SET Column\_Name = New\_value WHERE Value OPERATOR (SELECT COLUMN\_NAME FROM TABLE\_NAME WHERE Condition) ;   **Example of Subquery with the UPDATE statement**  This example updates the record of one table using the IN operator with Subquery in the UPDATE statement.  Let's take an Employee\_Details and Department table.  The data of the Employee\_Details table is shown in the following table:   |  |  |  |  | | --- | --- | --- | --- | | **Emp\_ID** | **Emp\_Name** | **Emp\_Salary** | **Dept\_ID** | | 1001 | Akhil | 50000 | 404 | | 1002 | Balram | 25000 | 403 | | 1003 | Bheem | 45000 | 405 | | 1004 | Chetan | 60000 | 402 | | 1005 | Ram | 65000 | 407 | | 1006 | Shyam | 55500 | NULL | | 1007 | Shobhit | 60000 | NULL |   **Table: Employee\_Details**  The data of Department table is shown in the below table:   |  |  |  |  | | --- | --- | --- | --- | | **Dept\_ID** | **Dept\_Name** | **Emp\_ID** | **Dept\_Grade** | | 401 | Administration | 1008 | B | | 402 | HR | 1004 | A | | 403 | Testing | 1002 | A | | 404 | Coding | 1001 | B | | 405 | Sales | 1003 | A | | 406 | Marketing | NULL | C | | 407 | Accounting | 1005 | A |   The following updates the salary of those employees whose Department Grade is A:   1. UPDATE Employee\_Details SET Emp\_SalaryEmp\_Salary = Emp\_Salary + 5000 WHERE Emp\_ID IN ( SELECT Emp\_ID FROM Department WHERE Dept\_Grade = 'A' ) ;   The following query will show the updated data of the Employee\_Details table in the output:   1. SELECT \* FROM Employee\_Details ;   **Output:**   |  |  |  |  | | --- | --- | --- | --- | | **Emp\_ID** | **Emp\_Name** | **Emp\_Salary** | **Dept\_ID** | | 1001 | Akhil | 50000 | 404 | | 1002 | Balram | 30000 | 403 | | 1003 | Bheem | 50000 | 405 | | 1004 | Chetan | 65000 | 402 | | 1005 | Ram | 70000 | 407 | | 1006 | Shyam | 55500 | NULL | | 1007 | Shobhit | 60000 | NULL |   **Table: Employee\_Details** Subquery with the DELETE statement We can easily delete one or more records from the SQL table using Subquery with the DELETE statement in Structured Query Language.  **Syntax of Subquery with DELETE statement**   1. DELETE FROM Table\_Name WHERE Value OPERATOR (SELECT COLUMN\_NAME FROM TABLE\_NAME WHERE Condition) ;   **Example of Subquery with DELETE statement**  This example deletes the records from the table using the IN operator with Subquery in the DELETE statement.  Let's take an Employee\_Details and Department table.  The data of the Employee\_Details table is shown in the following table:   |  |  |  |  | | --- | --- | --- | --- | | **Emp\_ID** | **Emp\_Name** | **Emp\_Salary** | **Dept\_ID** | | 1001 | Akhil | 50000 | 404 | | 1002 | Balram | 25000 | 403 | | 1003 | Bheem | 45000 | 405 | | 1004 | Chetan | 60000 | 402 | | 1005 | Ram | 65000 | 407 | | 1006 | Shyam | 55500 | NULL | | 1007 | Shobhit | 60000 | NULL | | 1008 | Ankit | 48000 | 401 |   **Table: Employee\_Details**  The data of Department table is shown in the below table:   |  |  |  |  | | --- | --- | --- | --- | | **Dept\_ID** | **Dept\_Name** | **Emp\_ID** | **Dept\_Grade** | | 401 | Administration | 1008 | C | | 402 | HR | 1004 | A | | 403 | Testing | 1002 | C | | 404 | Coding | 1001 | B | | 405 | Sales | 1003 | A | | 406 | Marketing | NULL | C | | 407 | Accounting | 1005 | C |   The following query deletes the record of those employees from the Employee\_Details whose Department Grade is C:   1. DELETE FROM Employee\_Details WHERE Emp\_ID IN ( SELECT Emp\_ID FROM Department WHERE Dept\_Grade = 'C' ) ;   The following query will show the updated data of the Employee\_Details table in the output:   1. SELECT \* FROM Employee\_Details ;   **Output:**   |  |  |  |  | | --- | --- | --- | --- | | **Emp\_ID** | **Emp\_Name** | **Emp\_Salary** | **Dept\_ID** | | 1001 | Akhil | 50000 | 404 | | 1003 | Bheem | 45000 | 405 | | 1004 | Chetan | 60000 | 402 | | 1006 | Shyam | 55500 | NULL | | 1007 | Shobhit | 60000 | NULL |  **MySQL Trigger** A trigger in MySQL is a set of SQL statements that reside in a system catalog. **It is a special type of stored procedure that is invoked automatically in response to an event**. Each trigger is associated with a table, which is activated on any DML statement such as **INSERT, UPDATE**, or **DELETE**.  A trigger is called a special procedure because it cannot be called directly like a stored procedure. The main difference between the trigger and procedure is that a trigger is called automatically when a data modification event is made against a table. In contrast, a stored procedure must be called explicitly.  Generally, **triggers are of two types** according to the [SQL](https://www.javatpoint.com/sql-tutorial) standard: row-level triggers and statement-level triggers.  **Row-Level Trigger:** It is a trigger, which is activated for each row by a triggering statement such as insert, update, or delete. For example, if a table has inserted, updated, or deleted multiple rows, the row trigger is fired automatically for each row affected by the [insert](https://www.javatpoint.com/mysql-insert), [update](https://www.javatpoint.com/mysql-update), or [delete statement](https://www.javatpoint.com/mysql-delete).  **Statement-Level Trigger:** It is a trigger, which is fired once for each event that occurs on a table regardless of how many rows are inserted, updated, or deleted. **NOTE: We should know that MySQL doesn't support statement-level triggers. It provides supports for row-level triggers only.****Why we need/use triggers in MySQL?** We need/use triggers in MySQL due to the following features:   * Triggers help us to enforce business rules. * Triggers help us to validate data even before they are inserted or updated. * Triggers help us to keep a log of records like maintaining audit trails in tables. * SQL triggers provide an alternative way to check the integrity of data. * Triggers provide an alternative way to run the scheduled task. * Triggers increases the performance of SQL queries because it does not need to compile each time the query is executed. * Triggers reduce the client-side code that saves time and effort. * Triggers help us to scale our application across different platforms. * Triggers are easy to maintain.  **Limitations of Using Triggers in MySQL**  * MySQL triggers do not allow to use of all validations; they only provide extended validations. **For example**, we can use the NOT NULL, UNIQUE, CHECK and FOREIGN KEY constraints for simple validations. * Triggers are invoked and executed invisibly from the client application. Therefore, it isn't easy to troubleshoot what happens in the database layer. * Triggers may increase the overhead of the database server.  **Types of Triggers in MySQL?** We can define the maximum six types of actions or events in the form of triggers:   1. [**Before Insert**](https://www.javatpoint.com/mysql-before-insert-trigger)**:** It is activated before the insertion of data into the table. 2. [**After Insert**](https://www.javatpoint.com/mysql-after-insert-trigger)**:** It is activated after the insertion of data into the table. 3. [**Before Update**](https://www.javatpoint.com/mysql-before-update-trigger)**:** It is activated before the update of data in the table. 4. [**After Update**](https://www.javatpoint.com/mysql-after-update-trigger)**:** It is activated after the update of the data in the table. 5. [**Before Delete**](https://www.javatpoint.com/mysql-before-delete-trigger)**:** It is activated before the data is removed from the table. 6. [**After Delete**](https://www.javatpoint.com/mysql-after-delete-trigger)**:** It is activated after the deletion of data from the table.   When we use a statement that does not use INSERT, UPDATE or DELETE query to change the data in a table, the triggers associated with the trigger will not be invoked. **Naming Conventions** Naming conventions are the set of rules that we follow to give appropriate unique names. It saves our time to keep the work organize and understandable. Therefore, **we must use a unique name for each trigger associated with a table**. However, it is a good practice to have the same trigger name defined for different tables.  The following naming convention should be used to name the trigger in [MySQL](https://www.javatpoint.com/mysql-tutorial):   1. (BEFOR | **AFTER**) table\_name (**INSERT** | **UPDATE** | **DELETE**)   Thus,  **Trigger Activation Time:** BEFORE | AFTER  **Trigger Event:** INSERT | UPDATE | DELETE **How to create triggers in MySQL?** We can use the **CREATE TRIGGER** statement for creating a new trigger in MySQL. Below is the syntax of creating a trigger in MySQL:   1. **CREATE** **TRIGGER** trigger\_name 2. (**AFTER** | BEFORE) (**INSERT** | **UPDATE** | **DELETE**) 3. **ON** table\_name **FOR** EACH ROW 4. **BEGIN** 5. --variable declarations 6. --trigger code 7. **END**;  **MySQL Stored Procedure** A procedure (often called a stored procedure) is a **collection of pre-compiled SQL statements** stored inside the database. It is a subroutine or a subprogram in the regular computing language. **A procedure always contains a name, parameter lists, and SQL statements**. We can invoke the procedures by using triggers, other procedures and applications such as [Java](https://www.javatpoint.com/java-tutorial), [Python](https://www.javatpoint.com/python-tutorial), [PHP](https://www.javatpoint.com/php-tutorial), etc. It was first introduced in MySQL **version 5**. Presently, it can be supported by almost all relational database systems.  If we consider the enterprise application, we always need to perform specific tasks such as database cleanup, processing payroll, and many more on the database regularly. Such tasks involve multiple [SQL](https://www.javatpoint.com/sql-tutorial) statements for executing each task. This process might easy if we group these tasks into a single task. We can fulfill this requirement in [MySQL](https://www.javatpoint.com/mysql-tutorial) by creating a stored procedure in our database.  A procedure is called a **recursive stored procedure** when it calls itself. Most database systems support recursive stored procedures. But, it is not supported well in MySQL. **Stored Procedure Features**  * Stored Procedure increases the performance of the applications. Once stored procedures are created, they are compiled and stored in the database. * Stored procedure reduces the traffic between application and database server. Because the application has to send only the stored procedure's name and parameters instead of sending multiple SQL statements. * Stored procedures are reusable and transparent to any applications. * A procedure is always secure. The database administrator can grant permissions to applications that access stored procedures in the database without giving any permissions on the database tables.  **How to create a procedure?** The following syntax is used for creating a stored procedure in MySQL. It can return one or more value through parameters or sometimes may not return at all. By default, a procedure is associated with our current database. But we can also create it into another database from the current database by specifying the name as **database\_name.procedure\_name**. See the complete syntax:   1. DELIMITER && 2. **CREATE** **PROCEDURE** procedure\_name [[IN | **OUT** | INOUT] parameter\_name datatype [, parameter datatype]) ] 3. **BEGIN** 4. Declaration\_section 5. Executable\_section 6. **END** && 7. DELIMITER ;  **Parameter Explanations** The procedure syntax has the following parameters:   |  |  | | --- | --- | | **Parameter Name** | **Descriptions** | | procedure\_name | It represents the name of the stored procedure. | | parameter | It represents the number of parameters. It can be one or more than one. | | Declaration\_section | It represents the declarations of all variables. | | Executable\_section | It represents the code for the function execution. |   **MySQL procedure parameter has one of three modes:**  **IN parameter**  It is the default mode. It takes a parameter as input, such as an attribute. When we define it, the calling program has to pass an argument to the stored procedure. This parameter's value is always protected.  **OUT parameters**  It is used to pass a parameter as output. Its value can be changed inside the stored procedure, and the changed (new) value is passed back to the calling program. It is noted that a procedure cannot access the OUT parameter's initial value when it starts.  **INOUT parameters**  It is a combination of IN and OUT parameters. It means the calling program can pass the argument, and the procedure can modify the INOUT parameter, and then passes the new value back to the calling program. **How to call a stored procedure?** We can use the **CALL statement** to call a stored procedure. This statement returns the values to its caller through its parameters (IN, OUT, or INOUT). The following syntax is used to call the stored procedure in MySQL:   1. CALL procedure\_name ( parameter(s))  **Example** Let us understand how to create a procedure in MySQL through example. First, we need to select a database that will store the newly created procedure. We can select the database using the below statement:   1. mysql> USE database\_name;   Suppose this database has a table named **student\_info** that contains the following data:  MySQL Procedure | MySQL Stored Procedure **Procedure without Parameter** Suppose we want **to display all records of this table whose marks are greater than 70** and count all the table rows. The following code creates a procedure named **get\_merit\_students**:   1. DELIMITER && 2. **CREATE** **PROCEDURE** get\_merit\_student () 3. **BEGIN** 4. **SELECT** \* **FROM** student\_info **WHERE** marks > 70; 5. **SELECT** COUNT(stud\_code) **AS** Total\_Student **FROM** student\_info; 6. **END** && 7. DELIMITER ;   If this code executed successfully, we would get the below output:  MySQL Procedure | MySQL Stored Procedure  Let us call the procedure to verify the output:   1. mysql> CALL get\_merit\_student();   It will give the output as follows:  MySQL Procedure | MySQL Stored Procedure **Procedures with IN Parameter** In this procedure, we have used the IN parameter as **'var1**' of integer type to accept a number from users. Its body part fetches the records from the table using a [**SELECT** statement](https://www.javatpoint.com/mysql-select) and returns only those rows that will be supplied by the user. It also returns the total number of rows of the specified table. See the procedure code:   1. DELIMITER && 2. **CREATE** **PROCEDURE** get\_student (IN var1 **INT**) 3. **BEGIN** 4. **SELECT** \* **FROM** student\_info LIMIT var1; 5. **SELECT** COUNT(stud\_code) **AS** Total\_Student **FROM** student\_info; 6. **END** && 7. DELIMITER ;   After successful execution, we can call the procedure as follows:   1. mysql> CALL get\_student(4);   We will get the below output:  MySQL Procedure | MySQL Stored Procedure **Procedures with OUT Parameter** In this procedure, we have used the OUT parameter as the **'highestmark'** of integer type. Its body part fetches the maximum marks from the table using a **MAX() function**. See the procedure code:   1. DELIMITER && 2. **CREATE** **PROCEDURE** display\_max\_mark (**OUT** highestmark **INT**) 3. **BEGIN** 4. **SELECT** **MAX**(marks) **INTO** highestmark **FROM** student\_info; 5. **END** && 6. DELIMITER ;   This procedure's parameter will get the highest marks from the **student\_info** table. When we call the procedure, the OUT parameter tells the database systems that its value goes out from the procedures. Now, we will pass its value to a session variable **@M** in the CALL statement as follows:   1. mysql> CALL display\_max\_mark(@M); 2. mysql> **SELECT** @M;   Here is the output:  MySQL Procedure | MySQL Stored Procedure **Procedures with INOUT Parameter** In this procedure, we have used the INOUT parameter as **'var1'** of integer type. Its body part first fetches the marks from the table with the specified **id** and then stores it into the same variable var1. The var1 first acts as the IN parameter and then OUT parameter. Therefore, we can call it the INOUT parameter mode. See the procedure code:   1. DELIMITER && 2. **CREATE** **PROCEDURE** display\_marks (INOUT var1 **INT**) 3. **BEGIN** 4. **SELECT** marks **INTO** var1 **FROM** student\_info **WHERE** stud\_id = var1; 5. **END** && 6. DELIMITER ;   After successful execution, we can call the procedure as follows:   1. mysql> **SET** @M = '3'; 2. mysql> CALL display\_marks(@M); 3. mysql> **SELECT** @M;   We will get the below output:  MySQL Procedure | MySQL Stored Procedure **How to show or list stored procedures in MySQL?** When we have several procedures in the MySQL server, it is very important to list all procedures. It is because sometimes the procedure names are the same in many databases. In that case, this query is very useful. We can list all procedure stored on the current MySQL server as follows:   1. SHOW **PROCEDURE** STATUS [LIKE 'pattern' | **WHERE** search\_condition]   **This statement displays all stored procedure names, including their characteristics**. If we want to display procedures in a particular database, we need to use the [**WHERE** clause](https://www.javatpoint.com/mysql-where). In case we want to list stored procedures with a specific word, we need to use the [**LIKE** clause](https://www.javatpoint.com/mysql-like).  We can list all stored procedure in the MySQL **mystudentsb** database using the below statement:   1. mysql> SHOW **PROCEDURE** STATUS **WHERE** db = 'mystudentdb';   It will give the below output where we can see that the mystudentdb database contains **four stored procedures**:  MySQL Procedure | MySQL Stored Procedure **How to delete/drop stored procedures in MySQL?** MySQL also allows a command to drop the procedure. When the procedure is dropped, it is removed from the database server also. The following statement is used to drop a stored procedure in MySQL:   1. **DROP** **PROCEDURE** [ IF EXISTS ] procedure\_name;   Suppose we want to remove the procedure named **display\_marks** from the mystudentdb database. We can do this by first selecting the database and then use the syntax as follows to remove the procedure:   1. mysql> **DROP** **PROCEDURE** display\_marks;   We can verify it by listing the procedure in the specified database using the **SHOW PROCEDURE STATUS** command. See the below output:  MySQL Procedure | MySQL Stored Procedure **How to create a procedure in MySQL workbench?** We first launch the tool and log in with the username and password to create procedures in the [MySQL Workbench](https://www.javatpoint.com/mysql-workbench). Now, we need to do the following steps for creating a stored procedure:  1. Go to the **Navigation tab** and click on the **Schema menu** where all the previously created databases available. Select your desired database (for example, **employeedb**). It will pop up the following options.  MySQL Procedure | MySQL Stored Procedure  2. Right-click on the **Stored Procedure**, and we will get the default procedure code. See the below screen:  MySQL Procedure | MySQL Stored Procedure  3. Complete the procedure code and click on the **Apply button**. In the next window, we will review the procedure code once again, and if no error was found, click on the Apply button.  MySQL Procedure | MySQL Stored Procedure  4. After clicking on the Apply button, click on the **Finish** button for completion.  MySQL Procedure | MySQL Stored Procedure  5. We can navigate to the schema menu again to verify this newly created procedure. It means first select your database and expand it to display its sub-menu. In the **sub-menu**, expanding the stored procedure option will show the newly created procedure. See the below image:  MySQL Procedure | MySQL Stored Procedure  6. We can call the procedure by tapping on the red rectangle box or simply execute the CALL statement.  MySQL Procedure | MySQL Stored Procedure **How to alter the procedure in MySQL?** MySQL does not allow any command to alter the procedure in MySQL. However, it provides a command that is used to change the characteristics of a stored procedure. This command may alter more than one change in the procedure but does not modify the stored procedure's parameters or body. If we want to make such changes, we must **drop and re-create the procedure** using the DROP PROCEDURE and CREATE PROCEDURE statement.  **The below statement is used to change the characteristics of a procedure but not the actual procedure**:   1. **ALTER** **PROCEDURE** procedure\_name [characteristics ...] 3. characteristics: { 4. COMMENT 'string' 5. | LANGUAGE SQL 6. | { **CONTAINS** SQL | **NO** SQL | READS SQL DATA | MODIFIES SQL DATA } 7. | SQL SECURITY { DEFINER | INVOKER } 8. }   Suppose we want to **add a comment** to the existing procedure. In such a case, we can use the **ALTER** statement as follows to accomplish this task:   1. mysql> **ALTER** **PROCEDURE** get\_merit\_student 2. COMMENT 'It displays all records';   After executing this statement, we can verify it by using the below statement:   1. mysql> SHOW **CREATE** **PROCEDURE** get\_merit\_student \G;   It will display the below output where we can see that the **comment** is added successfully.  MySQL Procedure | MySQL Stored Procedure  It is to note that **we can alter the body of the stored procedure in MySQL using the workbench tool**. So open this tool, navigate to the schema menu, and expand the database that contains stored procedures. Now, select your procedure, right-click on it and choose **ALTER STORED PROCEDURE** option. See the below screen:  MySQL Procedure | MySQL Stored Procedure  After clicking this option, we will get a window that contains a procedure code. See the below screen that contains procedure code to display all employee:  MySQL Procedure | MySQL Stored Procedure  Now, we will modify this code. Suppose we want to display only male employees. To do this, we can change this code from the below code and click on the **Apply button**:   1. **SELECT** \* **FROM** employee **WHERE** gender = 'M';   MySQL Procedure | MySQL Stored Procedure  In this window, we will review the procedure code once again, and if no error was found, click on the **Apply -> Apply -> Finish** button to complete the process. **Drawbacks of Using Stored Procedures**  * If we use stored procedures, the memory uses of every connection that uses those stored procedures will increase substantially. Also, if we overuse many logical applications inside stored procedures, the CPU usage will increase. It is because the database server is not well designed for logical operations. * Stored procedure's constructs are not designed to develop complex and flexible business logic. * It is difficult to debug stored procedures. Only a few database management systems allow us to debug stored procedures. Unfortunately, MySQL does not provide facilities for debugging stored procedures. * It is not easy to develop and maintain stored procedures. Developing and maintaining stored procedures are often required a specialized skill set that not all application developers possess. It may lead to problems in both application development and maintenance phases.  **MySQL Grant Privilege** MySQL has a feature that provides many control options to the administrators and users on the database. We have already learned how to create a new user using [CREATE USER](https://www.javatpoint.com/mysql-create-user) statement in MySQL server. Now, we are going to learn about grant privileges to a user account. MySQL provides GRANT statements to give access rights to a user account. **GRANT Statement** The grant statement enables system administrators to **assign privileges and roles** to the [MySQL](https://www.javatpoint.com/mysql-tutorial) user accounts so that they can use the assigned permission on the database whenever required. **Syntax** The following are the basic syntax of using the GRANT statement:   1. **GRANT** privilege\_name(s) 2. **ON** object 3. **TO** user\_account\_name;  **Parameter Explanation** In the above syntax, we can have the following parameters:   |  |  | | --- | --- | | **Parameter Name** | **Descriptions** | | privilege\_name(s) | It specifies the access rights or grant privilege to user accounts. If we want to give multiple privileges, then use a comma operator to separate them. | | object | It determines the privilege level on which the access rights are being granted. It means granting privilege to the table; then the object should be the name of the table. | | user\_account\_name | It determines the account name of the user to whom the access rights would be granted. |  **Privilege Levels** MySQL supports the following privilege levels:   |  |  |  | | --- | --- | --- | | **Privilege Level** | **Syntax** | **Descriptions** | | Global | GRANT ALL ON \*.\* TO john@localhost; | It applies to all databases on MySQL server. We need to use \*.\* syntax for applying global privileges. Here, the user can query data from all databases and tables of the current server. | | Database | GRANT ALL ON mydb.\* TO john@localhost; | It applies to all objects in the current database. We need to use the db\_name.\* syntax for applying this privilege. Here, a user can query data from all tables in the given database. | | Table | GRANT DELETE ON mydb.employees TO john@localhsot; | It applies on all columns in a specified table. We need to use db\_name.table\_name syntax for assigning this privilege. Here, a user can query data from the given table of the specified database. | | Column | GRANT SELECT (col1), INSERT (col1, col2), UPDATE (col2) ON mydb.mytable TO john@localhost; | It applies on a single column of a table. Here, we must have to specify the column(s) name enclosed with parenthesis for each privilege. The user can select one column, insert values in two columns, and update only one column in the given table. | | Stored Routine | GRANT EXECUTE ON PROCEDURE mydb.myprocedure TO john@localhost; | It applies to stored routines (procedure and functions). It contains CREATE ROUTINE, ALTER ROUTINE, EXECUTE, and GRANT OPTION privileges. Here, a user can execute the stored procedure in the current database. | | Proxy | GRANT PROXY ON root TO peter@localhost; | It enables one user to be a proxy for other users. |  **GRANT Statement Example** Let us understand the GRANT privileges through the example. First, we need to create a new user named "**john@localhost**" using the following statement:   1. mysql> **CREATE** USER john@localhost IDENTIFIED **BY** 'jtp12345';   Next, execute the SHOW GRANT statement to check the privileges assigned to john@localhost using the following query:   1. mysql> SHOW GRANTS **FOR** john@localhost;   It will give the below output. Here, the **USAGE** means a user can log in to the database but does not have any privileges.  MySQL Grant Privilege  If we want to **assign all privileges** to all databases in the current server to john@localhost, execute the below statement:   1. mysql> **GRANT** ALL **ON** mystudentdb.\* **TO** john@localhost;   Again, execute the SHOW GRANT statement to verify the privileges. After the successful execution, we will get the below output. Here all privileges are assigned to all databases in the current server to john@localhost.  MySQL Grant Privilege **Stored Routine Example** Here, the grant privileges are applied to **procedures and functions** where a user can execute the stored procedure in the current MySQL database. The EXECUTE privilege provides the ability to execute a function and procedure.  Let us understand it with the example. Suppose we have a function **calculatesalary** and want to grant **EXECUTE** privilege to a user john, run the following query:   1. mysql> **GRANT** **EXECUTE** **ON** **FUNCTION** calculatesalary **TO** john@localhost;   If there is a need to provide the EXECUTE privilege to all users, we must run the below command:   1. mysql> **GRANT** **EXECUTE** **ON** **FUNCTION** calculatesalary **TO** \*@localhost;   We can choose access right from the below list on which privileges can be applied.   1. **SELECT:** It enables us to view the result set from a specified table. 2. **INSERT:** It enables us to add records in a given table. 3. **DELETE:** It enables us to remove rows from a table. 4. **CREATE:** It enables us to create tables/schemas. 5. **ALTER:** It enables us to modify tables/schemas. 6. **UPDATE:** It enables us to modify a table. 7. **DROP:** It enables us to drop a table. 8. **INDEX:** It enables us to create indexes on a table. 9. **ALL:** It enables us to give ALL permissions except GRANT privilege. 10. **GRANT:** It enables us to change or add access rights.  **MySQL Cursor** In MySQL, Cursor can also be created. Following are the steps for creating a cursor. 1. Declare Cursor A cursor is a select statement, defined in the declaration section in [MySQL](https://www.javatpoint.com/mysql-tutorial). Syntax  1. **DECLARE** cursor\_name **CURSOR** **FOR** 2. **Select** statement;  Parameter: **cursor\_name:** name of the cursor  **select\_statement:** select query associated with the cursor 2. Open Cursor After declaring the cursor the next step is to open the cursor using open statement. Syntax  1. **Open** cursor\_name;  Parameter: **cursor\_name:** name of the cursor which is already declared. 3. Fetch Cursor After declaring and opening the cursor, the next step is to fetch the cursor. It is used to fetch the row or the column. Syntax  1. **FETCH** [ **NEXT** [ **FROM** ] ] cursor\_name **INTO** variable\_list;  Parameter: **cursor\_name:** name of the cursor  **variable\_list:** variables, comma separated, etc. is stored in a cursor for the result set 4. Close Cursor The final step is to close the cursor. Syntax  1. **Close** cursor\_name;  Parameter: **Cursor\_name:** name of the cursor Example for the cursor: **Step 1:** Open the database and table.  MySQL Cursor  **Step 2:** Now create the cursor.  **Query:**  MySQL Cursor  **Step 3:** Now call the cursor.  **Query:**   1. **SET** @name\_list =""; 2. CALL list\_name(@name\_list); 3. **SELECT** @name\_list;   MySQL Cursor **MySQL Functions**Creating a function In MySQL, Function can also be created. A function always returns a value using the return statement. The function can be used in SQL queries. Syntax  1. **CREATE** **FUNCTION** function\_name [ (parameter datatype [, parameter datatype]) ] 2. **RETURNS** return\_datatype 3. **BEGIN** 4. Declaration\_section 5. Executable\_section 6. **END**;  Parameter: **Function\_name:** name of the function  **Parameter:** number of parameter. It can be one or more than one.  **return\_datatype:** return value datatype of the function  **declaration\_section:** all variables are declared.  **executable\_section:** code for the function is written here. Example 1 **Step 1:** Create database and table.  **Database:** employee  MySQL Functions  Table 1 : designation  MySQL Functions  Table 2 : staff  MySQL Functions  **Step 2:** Create a function  **Function query:**   1. DELIMITER $$ 2. **CREATE** **FUNCTION** get\_designation\_name(d\_id **INT**) **RETURNS** **VARCHAR**( 20 ) 3. **BEGIN** 4. **DECLARE** de\_name **VARCHAR**( 20 ) **DEFAULT** ""; 5. **SELECT** **name** **INTO** de\_name **FROM** designation **WHERE** id = d\_id; 6. **RETURN** de\_name; 7. **END** $$   MySQL Functions  **Step 3:** Execute the function  **Query :**  SELECT id, get\_designation1(`d\_id`) as DESIGNATION, name FROM 'staff'  MySQL Functions Drop a function In MySQL Function can also be dropped. When A function id dropped, it is removed from the database. Syntax:  1. **Drop** **function** [ IF EXISTS ] function\_name;  Parameter **function\_name:** name of the function to be dropped. Example 1: drop function get\_designation\_name;  MySQL Functions Loops in MySQL The **MySQL LOOP** statement could be used to run a block of code or set of statements, again and again, depending on the condition. Stored procedures are a subset of SQL statements that are kept in the [SQL](https://www.geeksforgeeks.org/sql-tutorial/)catalog as subroutines. These procedures may have both IN and OUT parameters. If you use [SELECT](https://www.geeksforgeeks.org/sql-select-query/)statements, they might return result sets or multiple result sets. In MYSQL, functions can also be made.  IF, CASE, ITERATE, LEAVE LOOP, WHILE, and REPEAT are examples of flow control statements supported by [MySQL](https://www.geeksforgeeks.org/mysql-common-mysql-queries/), much like in other programming languages. These statements can be used in stored programs (procedures), and stored functions can use RETURN. One Flow Control Statement may be used inside another.  **Syntax:**  *[labelname:] LOOP*  *statements*  *END LOOP [labelname]*  **Parameters:**   * **Label name:** It is an optional label at the start and end. * **Statements:** They could have one or multiple statements, each ended by a semicolon (;) and executed by LOOP.   **Syntax of the LOOP statement with LEAVE Statement :**  *[labelname]: LOOP*  *— terminate the loop*  *IF condition THEN*  *LEAVE [labelname];*  *END IF;*  *END LOOP;*  **Example 1**  Let’s see this is a stored procedure written in MySQL that creates a procedure called “GeekLoop”.This procedure uses a loop to output the numbers from 1 to 5 and then outputs the final value of the variable “no”.  We should also be well versed with this that the code sets the delimiter to “$$” instead of the default “;” delimiter, and that it resets the delimiter back to “;” at the end of the procedure. Additionally, the user should have the necessary permissions to create and execute stored procedures in the MySQL database.  **Query:**  DROP PROCEDURE IF EXISTS GeekLoop();  DELIMITER $$  CREATE PROCEDURE GeekLoop()  BEGIN  DECLARE no INT;  SET no = 0;  loop: LOOP  SET no = no +1;  select no ;  IF no =5 THEN  LEAVE loop;  END IF;  END LOOP loop;  SELECT no;  END $$  DELIMITER ;  **Statement to check the output :**  CALL GeekLoop();  **Output –**      **Example 2**  This is a MySQL user-defined function called “Geekdemo”. The function takes one integer parameter “value1” and returns an integer value.  Here’s a breakdown of what each line does:   1. **DELIMITER**: “*Changesthedelimiterto*” instead of the default “;” delimiter. 2. **CREATE FUNCTION** Geekdemo (value1 INT): Creates a user-defined function called “Geekdemo” that takes one integer parameter called “value1”. 3. **RETURNS INT**: Indicates that the function returns an integer value. 4. **BEGIN**: Begins the block of code for the function. 5. **DECLARE**value2 INT: Declares a local variable called “value2” of type integer and initializes it to 0. 6. label: LOOP: Defines a loop called “label”. 7. **SET**value 2= value2 + value1 Calculates the sum of “value2” and “value1” and stores the result in “income”. However, the “income” variable is not defined before this line, so it should be “value2”. 8. **IF**value2 < 4000 THEN: If “value2” is less than 4000, the loop will continue. 9. **ITERATE**label: Skips to the next iteration of the loop. 10. **END**IF: Ends the conditional statement. 11. **LEAVE**label: Exits the loop labeled “label” when “value2” is greater than or equal to 4000. 12. **END**LOOP label: Ends the loop. 13. **RETURN**value2: Returns the final value of “value2”. 14. **END**$$: Ends the block of code for the function. 15. **DELIMITER:** Changes the delimiter back to “;”.   **Query:**  DELIMITER $$  CREATE FUNCTION Geekdemo (value1 INT)  RETURNS INT  BEGIN  DECLARE value2 INT;  SET value2 = 0;  label: LOOP  SET value2 = value2 + value1;  IF value2 < 4000 THEN  ITERATE label;  END IF;  LEAVE label;  END LOOP label;  RETURN value2 ;  END $$  DELIMITER ;  Queries to check the output :  CALL Geekdemo();  **Input –**  value1: 3500  **Output –**  value2: 3500 MySQL WHILE Loop **Introduction :** [MySQL](https://www.geeksforgeeks.org/mysql-common-mysql-queries/)WHILE loop statement is used to execute one or more statements again and again, as long as a condition is true. We can use the loop when we need to execute the task with repetition while condition is true. **Note –** Use a WHILE LOOP statement in case you are unsure of what number of times you would like the loop body to execute. Since the WHILE condition is evaluated before entering the loop, it’s possible that the loop body might not execute even once.  **Syntax :**  [label\_name:] WHILE  condition DO  statements\_list  END WHILE [label\_name]  **Syntax label meaning –**   * **Label\_name –** label\_nameis optional, it’s a name related to the WHILE loop. * **Condition –** conditionis tested each undergoes through the WHILE loop. If the condition results in TRUE, the statements\_list is executed, or if the condition results in FALSE, the WHILE loop is terminated. * **Statements\_list –** Statements\_list is that the list of statements to be executed withstand the WHILE loop.   **Block diagram of While loop :**    *Block diagram of WHILE loop*    **Examples of MySQL WHILE Loop :**  **Example-1 :** Lets us create a function using a while loop.  DELIMITER $$  CREATE FUNCTION GeekInc ( value INT )  RETURNS INT  BEGIN  DECLARE inc INT;  SET inc = 0;  label:  WHILE inc <= 30000 DO  SET inc = inc + value;  END  WHILE label;  RETURN inc;  END; $$  DELIMITER ;  **Analysis –**   * Value is the input for the GeekInc function. * inc is declared and set to 0. * While inc is less than and equal to 3000, it will set inc to inc + value.   To check output used the following command given below.  CALL GeekInc(10000);  **Output –**  0, 10000, 20000, 30000  **Example-2 :** Let us create a procedure using a while loop.  DELIMITER $$  CREATE procedure while\_ex()  block: BEGIN  declare value VARCHAR(20) default ' ' ;  declare num INT default 0;  SET num = 1;  WHILE num <= 5 DO  SET value = CONCAT(value, num ,',' );  SET num = num + 1;  END  WHILE block;  select value ;  END $$  DELIMITER ;  **Analysis –**   * create procedure while\_ex and declare value and num. * set num at 1, while num is equal to or less than 5 do * set value equal to concatenation of value and num.   To check output used the following command given below.  call while\_ex();  **Output –**   | **value** | | --- | | 1,2,3,4,5 |   **Example-3 :** Let us create a table “Test\_Cal” which has dates as follows.  CREATE TABLE Test\_Cal(  t\_in INT AUTO\_INCREMENT,  fulldate DATE UNIQUE,  day TINYINT NOT NULL,  month TINYINT NOT NULL,  PRIMARY KEY(id)  );  Now, create a stored procedure to insert data into the table as follows.  DELIMITER $$  CREATE PROCEDURE InsertCal(dt DATE)  BEGIN  INSERT INTO Test\_Cal(  fulldate,  day,  month )  VALUES(dt,  EXTRACT(DAY FROM dt),  EXTRACT(MONTH FROM dt)  );  END$$  DELIMITER ;  Now create stored procedure LoadCal() that updates the number of days starting from a start date into the table.  DELIMITER $$  CREATE PROCEDURE LoadCal(  startDate DATE,  day INT  )  BEGIN  DECLARE counter INT DEFAULT 1;  DECLARE dt DATE DEFAULT startDate;  WHILE counter <= day DO  CALL InsertCal(dt);  SET counter = counter + 1;  SET dt = DATE\_ADD(dt,INTERVAL 1 day);  END WHILE;  END$$  DELIMITER ;  **Analysis –**   * The stored procedure LoadCal() has two parameters: startDate and day. * First, declare a counter and dt variables for saving values. * Then, check if the counter is less than or equal day, if yes: * Run stored procedure Inertial() to insert a row into the Test\_Cal table. * An increase in counter by 1 increases the dt by 1 day using the DATE\_ADD(). * The WHILE loop inserts date into the table till the counter is the same as the day.   To check output used the following command given below.  CALL LoadCal('2021-01-01',31);  select \* from Test\_Cal where tid < 10 ;  **Output –**   | **t\_id** | **fulldate** | **day** | **month** | | --- | --- | --- | --- | | 1 | 2021-01-01 | 1 | 1 | | 2 | 2021-01-02 | 2 | 1 | | 3 | 2021-01-03 | 3 | 1 | | 4 | 2021-01-04 | 4 | 1 | | 5 | 2021-01-05 | 5 | 1 | | 6 | 2021-01-06 | 6 | 1 | | 7 | 2021-01-07 | 7 | 1 | | 8 | 2021-01-08 | 8 | 1 | | 9 | 2021-01-09 | 9 | 1 |  **PHP Connect to MySQL** PHP 5 and later can work with a MySQL database using:   * **MySQLi extension** (the "i" stands for improved) * **PDO (PHP Data Objects)**   Earlier versions of PHP used the MySQL extension. However, this extension was deprecated in 2012. Should I Use MySQLi or PDO? If you need a short answer, it would be "Whatever you like".  Both MySQLi and PDO have their advantages:  PDO will work on 12 different database systems, whereas MySQLi will only work with MySQL databases.  So, if you have to switch your project to use another database, PDO makes the process easy. You only have to change the connection string and a few queries. With MySQLi, you will need to rewrite the entire code - queries included.  Both are object-oriented, but MySQLi also offers a procedural API.  Both support Prepared Statements. Prepared Statements protect from SQL injection, and are very important for web application security. MySQL Examples in Both MySQLi and PDO Syntax In this, and in the following chapters we demonstrate three ways of working with PHP and MySQL:   * MySQLi (object-oriented) * MySQLi (procedural) * PDO  MySQLi Installation For Linux and Windows: The MySQLi extension is automatically installed in most cases, when php5 mysql package is installed.  For installation details, go to: <http://php.net/manual/en/mysqli.installation.php> PDO Installation For installation details, go to: <http://php.net/manual/en/pdo.installation.php> Open a Connection to MySQL Before we can access data in the MySQL database, we need to be able to connect to the server: **Example (MySQLi Object-Oriented)**[**Get your own PHP Server**](https://www.w3schools.com/spaces/) <?php $servername = "localhost"; $username = "username"; $password = "password";  // Create connection $conn = new mysqli($servername, $username, $password);  // Check connection if ($conn->connect\_error) {   die("Connection failed: " . $conn->connect\_error); } echo "Connected successfully"; ?>  **Note on the object-oriented example above:**  $connect\_error was broken until PHP 5.2.9 and 5.3.0. If you need to ensure compatibility with PHP versions prior to 5.2.9 and 5.3.0, use the following code instead:  // Check connection if (mysqli\_connect\_error()) {   die("Database connection failed: " . mysqli\_connect\_error()); } **Example (MySQLi Procedural)** <?php $servername = "localhost"; $username = "username"; $password = "password";  // Create connection $conn = mysqli\_connect($servername, $username, $password);  // Check connection if (!$conn) {   die("Connection failed: " . mysqli\_connect\_error()); } echo "Connected successfully"; ?> **Example (PDO)** <?php $servername = "localhost"; $username = "username"; $password = "password";  try {   $conn = new PDO("mysql:host=$servername;dbname=myDB", $username, $password);   // set the PDO error mode to exception   $conn->setAttribute(PDO::ATTR\_ERRMODE, PDO::ERRMODE\_EXCEPTION);   echo "Connected successfully"; } catch(PDOException $e) {   echo "Connection failed: " . $e->getMessage(); } ?>  **Note:** In the PDO example above we have also **specified a database (myDB)**. PDO require a valid database to connect to. If no database is specified, an exception is thrown.  **Tip:** A great benefit of PDO is that it has an exception class to handle any problems that may occur in our database queries. If an exception is thrown within the try{ } block, the script stops executing and flows directly to the first catch(){ } block. Close the Connection The connection will be closed automatically when the script ends. To close the connection before, use the following: **MySQLi Object-Oriented:** $conn->close(); **MySQLi Procedural:** mysqli\_close($conn); **PDO:** $conn = null; |

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Add |  |
| - | Subtract |  |
| \* | Multiply |  |
| / | Divide |  |
| % | Modulo |  |