**INTRODUCTION TO SQL**

**(Structural Query Language)**

**Table of Content**

**Contents**

[1. Introduction to SQL 5](#_heading=h.gjdgxs)

[History of SQL 5](#_heading=h.30j0zll)

[What is Database? 5](#_heading=h.1fob9te)

[Relational Database 6](#_heading=h.3znysh7)

[SQL and Relational Databases 6](#_heading=h.2et92p0)

[How to run SQL Query on the local system 7](#_heading=h.tyjcwt)

[2. Downloading and Installing MySQL 7](#_heading=h.3dy6vkm)

[2.1 Downloading MySQL 7](#_heading=h.1t3h5sf)

[2.2. Installation of MySQL 10](#_heading=h.4d34og8)

[3. SQL QUERY 26](#_heading=h.2s8eyo1)

[The SQL SELECT DISTINCT 28](#_heading=h.17dp8vu)

[The SQL WHERE CLAUSE 28](#_heading=h.3rdcrjn)

[The SQL WHERE CLAUSE WITH AND, OR & NOT 30](#_heading=h.26in1rg)

[The SQL ORDER BY 31](#_heading=h.lnxbz9)

[The SQL SELECT TOP CLAUSE 32](#_heading=h.35nkun2)

[The SQL MIN() AND MAX() FUNCTION 33](#_heading=h.1ksv4uv)

[The SQL COUNT(), AVG() AND SUM() FUNCTION 34](#_heading=h.44sinio)

[The SQL LIKE-OPERATOR 35](#_heading=h.2jxsxqh)

[The SQL IN AND NOT IN OPERATORS 38](#_heading=h.z337ya)

[The SQL BETWEEN OPERATOR 39](#_heading=h.3j2qqm3)

[The SQL ALIAS 40](#_heading=h.1y810tw)

[The SQL GROUP BY STATEMENT 41](#_heading=h.4i7ojhp)

[The SQL HAVING CLAUSE 42](#_heading=h.2xcytpi)

[The SQL UNION 43](#_heading=h.1ci93xb)

[The SQL STORED PROCEDURE 44](#_heading=h.3whwml4)

[4. SQL JOIN 44](#_heading=h.2bn6wsx)

[INNER JOIN 45](#_heading=h.qsh70q)

[LEFT JOIN 46](#_heading=h.3as4poj)

[RIGHT JOIN 47](#_heading=h.1pxezwc)

[Full OUTER JOIN 48](#_heading=h.49x2ik5)

[SELF-JOIN 49](#_heading=h.2p2csry)

[5. SQL DATABASE 50](#_heading=h.147n2zr)

[The SQL CREATE DATABASE STATEMENT 50](#_heading=h.3o7alnk)

[The SQL DROP DATABASE STATEMENT 50](#_heading=h.23ckvvd)

[The SQL CREATE TABLE 51](#_heading=h.ihv636)

[The SQL DROP TABLE STATEMENT 52](#_heading=h.32hioqz)

[The SQL INSERT INTO STATEMENT 53](#_heading=h.1hmsyys)

[The SQL NULL VALUES 54](#_heading=h.41mghml)

[The SQL UPDATE STATEMENT 56](#_heading=h.2grqrue)

[The SQL DELETE STATEMENT 56](#_heading=h.vx1227)

[The SQL ALTER TABLE STATEMENT 57](#_heading=h.3fwokq0)

[5.1.1. ALTER TABLE - ADD COLUMN IN EXISTING TABLE 57](#_heading=h.1v1yuxt)

[5.1.2. ALTER TABLE – MODIFY/ALTER COLUMN 58](#_heading=h.4f1mdlm)

[5.1.3. ALTER TABLE - DROP COLUMN 58](#_heading=h.2u6wntf)

[6. The SQL CONSTRAINTS 59](#_heading=h.19c6y18)

[NOT NULL CONSTRAINTS 60](#_heading=h.3tbugp1)

[SQL UNIQUE CONSTRAINT 61](#_heading=h.28h4qwu)

[DROP A UNIQUE CONSTRAINT 63](#_heading=h.nmf14n)

[SQL PRIMARY KEY CONSTRAINTS 64](#_heading=h.37m2jsg)

[DROP PRIMARY KEY CONSTRAINTS 66](#_heading=h.1mrcu09)

[SQL FOREIGN KEY CONSTRAINT 67](#_heading=h.46r0co2)

[DROP A FOREIGN KEY CONSTRAINT 69](#_heading=h.2lwamvv)

[SQL CHECK CONSTRAINTS 69](#_heading=h.111kx3o)

[DROP A CHECK CONSTRAINT 71](#_heading=h.3l18frh)

[SQL DEFAULT CONSTRAINT 71](#_heading=h.206ipza)

[DROP A DEFAULT CONSTRAINT 73](#_heading=h.4k668n3)

[7. SQL CREATE INDEX STATEMENT 74](#_heading=h.2zbgiuw)

[DROP INDEX STATEMENT 75](#_heading=h.1egqt2p)

[8. SQL VIEWS STATEMENT 76](#_heading=h.3ygebqi)

[The WITH CHECK OPTION 77](#_heading=h.2dlolyb)

[DELETING ROWS INTO A VIEW 78](#_heading=h.sqyw64)

[DROPPING VIEWS 78](#_heading=h.3cqmetx)

[9. Advance MySQL 79](#_heading=h.2afmg28)

[9.1. MySQL Stored Procedure 79](#_heading=h.1rvwp1q)

[9.1.1. Creating the Stored Procedure 79](#_heading=h.4bvk7pj)

[9.1.2. EXECUTION OF STORE PROCEDURE 80](#_heading=h.2r0uhxc)

[9.1.3. DROP THE STORED PROCEDURE 81](#_heading=h.1664s55)

[9.1.4. STORED PROCEDURE PARAMETERS 82](#_heading=h.3q5sasy)

[9.1.5. STORED PROCEDURE VARIABLES 85](#_heading=h.25b2l0r)

[9.2. CONDITIONAL STATEMENT 86](#_heading=h.kgcv8k)

[9.2.1. IF-THEN STATEMENT 87](#_heading=h.34g0dwd)

[9.2.2. IF-THEN-ELSE STATEMENT 88](#_heading=h.1jlao46)

[9.2.3. IF THEN ELSEIF ELSE STATEMENT 89](#_heading=h.43ky6rz)

[9.3. CASE STATEMENT 90](#_heading=h.2iq8gzs)

[9.3.1. Simple CASE Statement 90](#_heading=h.xvir7l)

[9.3.2. Searched CASE Statement 92](#_heading=h.3hv69ve)

[9.4. LOOP STATEMENT 95](#_heading=h.1x0gk37)

[9.5. WHILE LOOP STATEMENT 96](#_heading=h.4h042r0)

[9.6. REPEAT LOOP STATEMENT 98](#_heading=h.2w5ecyt)

[9.7. CURSOR 99](#_heading=h.1baon6m)

# Introduction to SQL

SQL stands for Structural Query Language, and SQL is used for storing, manipulation, and retrieving data from the database.

## History of SQL

The SQL(Structural Query language) was first created in the 1970s by IBM researchers Raymond Boyce and Donald Chamberlin. The Query language, known then as **SEQUEL**, was created following the publishing of Edgar Frank Todd's paper, In 1970, A Relational Model of Data for Large Shared Data Banks.

In his paper, Todd proposed that all the data in a database be represented in the form of relations. It was based on this theory that Chamberlin and Boyce came up with SQL. The original SQL version was designed to retrieve and manipulate data stored in IBM's original RDBMS known as "**System R**." It wasn't until several years later, however, that the Structural Query language was made available publicly. In 1979, a company named as Relational Software, which later became Oracle, commercially released its version of the SQL language called Oracle V2.

Since that time, the American National Standards Institute (ANSI) and the International Standards Organization have deemed the SQL language as the standard language in relational database communication. While major SQL vendors do modify the language to their desires, most base their SQL programs off of the ANSI approved version.

## What is Database?

A database is a well-ordered collection of data. A database is an electronic system that permits data to be easily manipulated, accessed, and updated, or an organization uses a database as a method of managing, storing, and retrieving information. Modern databases are handled using a database management system (DBMS).

## Relational Database

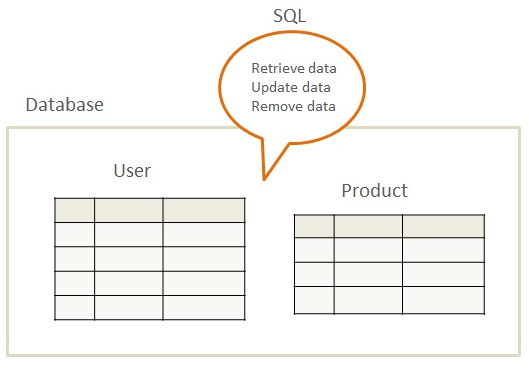
Relational Databases are used to store data in tables (rows and columns). Some common relational **database** management systems that use **SQL** are **Oracle**, **Sybase**, **Microsoft SQL Server**, **Access**, **Ingres**, etc.





## SQL and Relational Databases

A Relational Database contains tables that store the data that is related in some way. SQL is the query language that allows **retrieval and manipulation** of table data in the relational database. The database below has two tables: one with data on **Users** and another with data on **Products**.



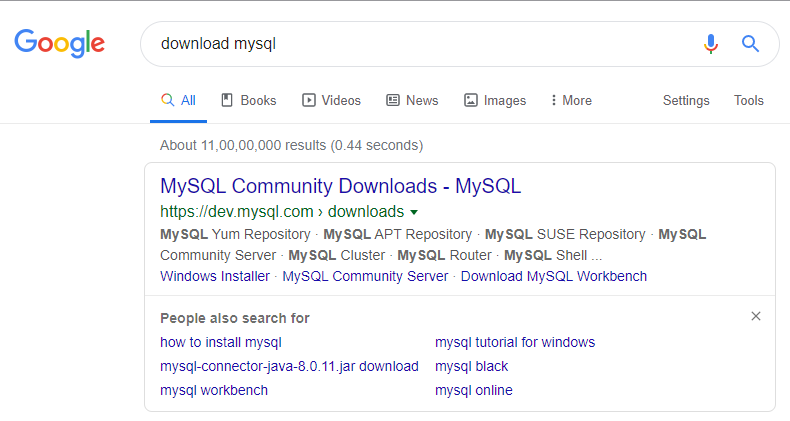
## How to run SQL Query on the local system

To run the SQL query on the local system, we need to install the MYSQL community server on the system. We have given step by step installation process below.

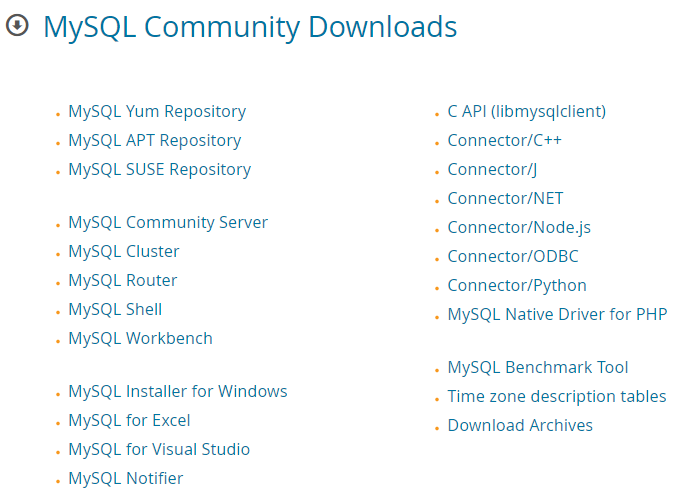
# Downloading and Installing MySQL

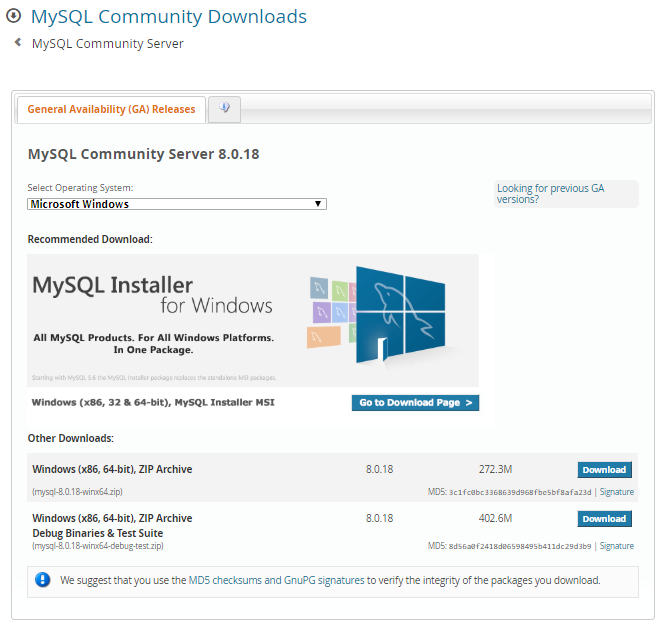
## 2.1 Downloading MySQL

**Step 1:**  Open Google and type **Download MySQL** and Click on **MySQL Community Downloads**

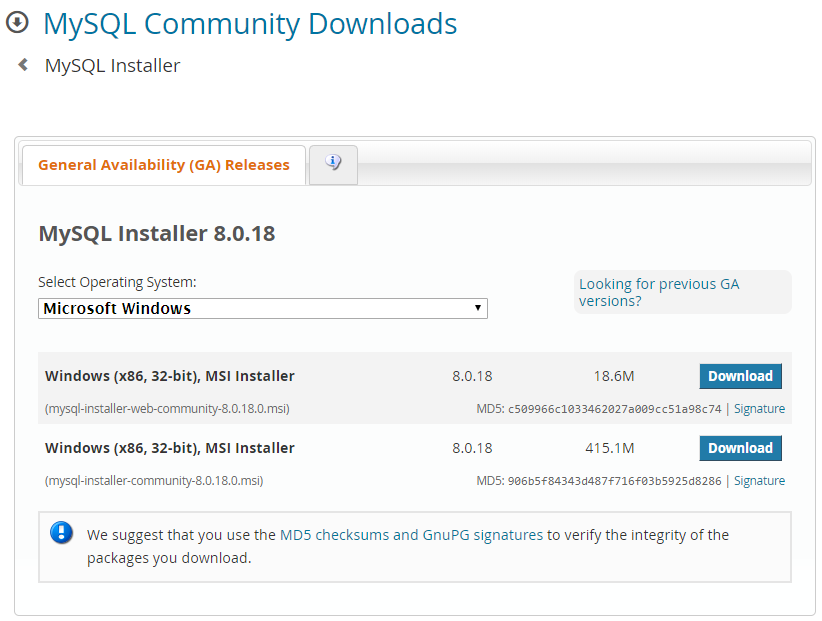


**Step 2:** Click on **MySQL Community Server**

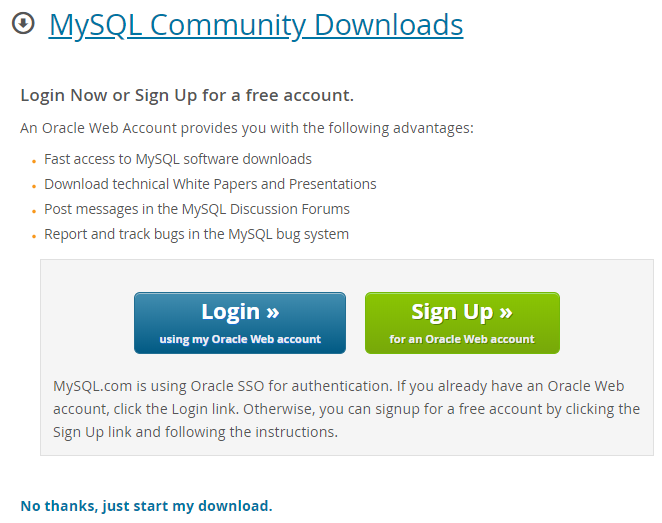


**Step 3:** Click on theMySQL installer MSI **Go to Download Page >**

**Step 4:** Select the OS and click on **MSI Installer community**



**Step 5:** Click on **start my download**

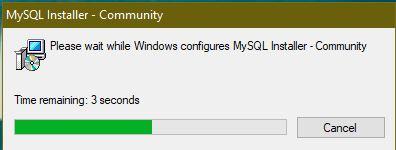


**Note:** Once the Downloading is completed, then double-click on that and install it on the local system.

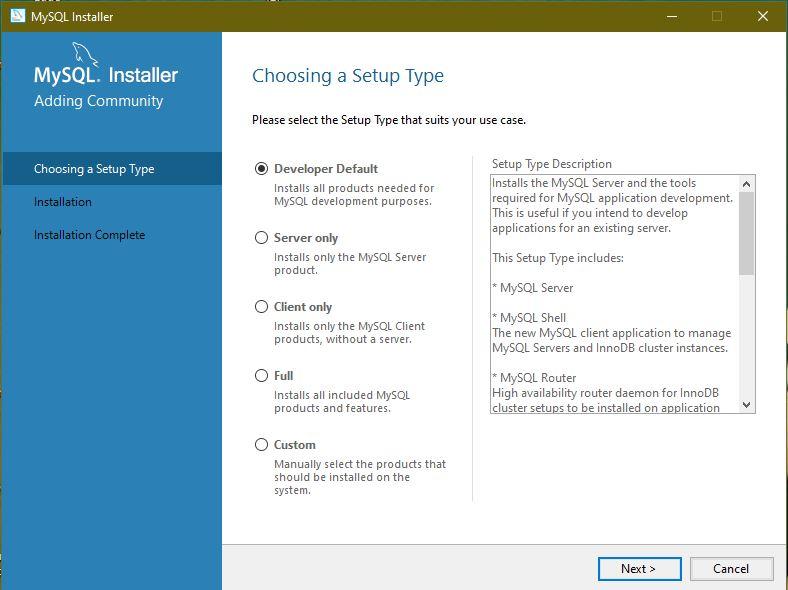
## 2.2. Installation of MySQL

**Step 1:** **Double-Click** on Downloaded Application.

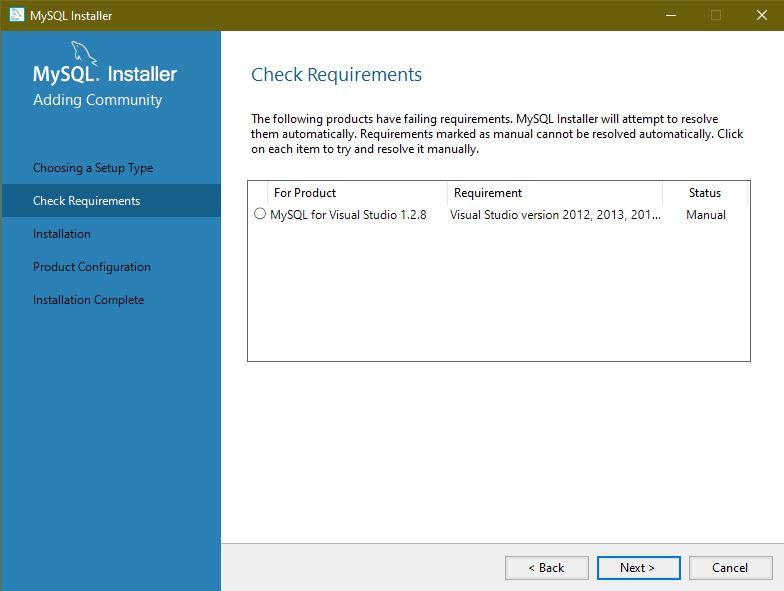
**Step 2:** After clicking on the application we will get a window like below



**Step 3:** Choosing the Setup type and click Next.



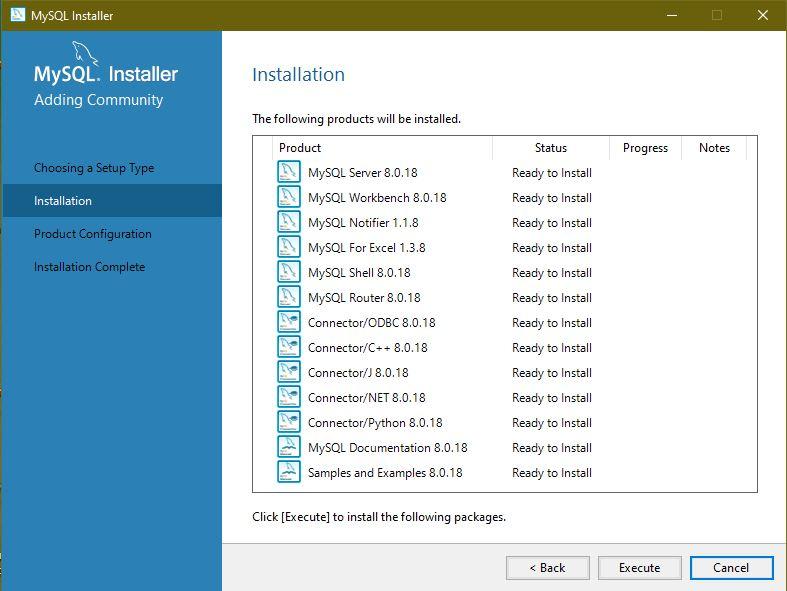
**Setup 4:** Click Next.

****

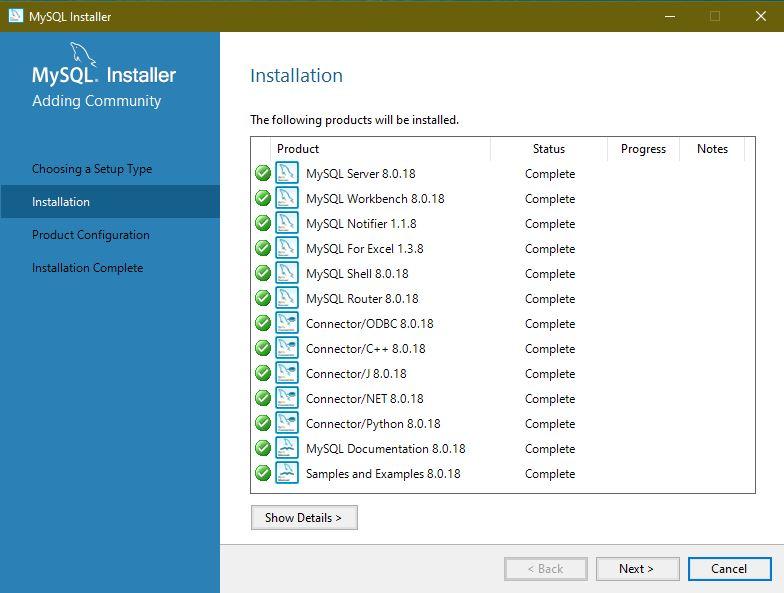
**Step 5:** Click Yes.

****

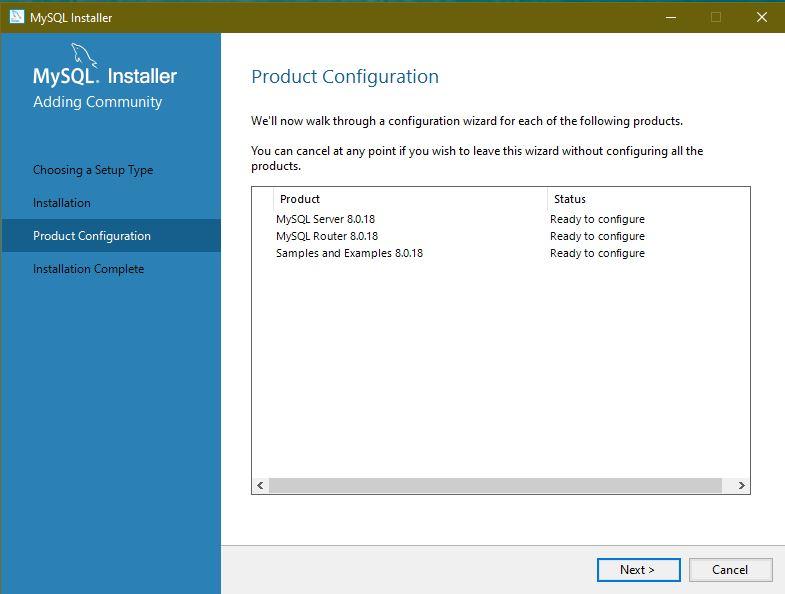
**Step 6:** Click Execute.

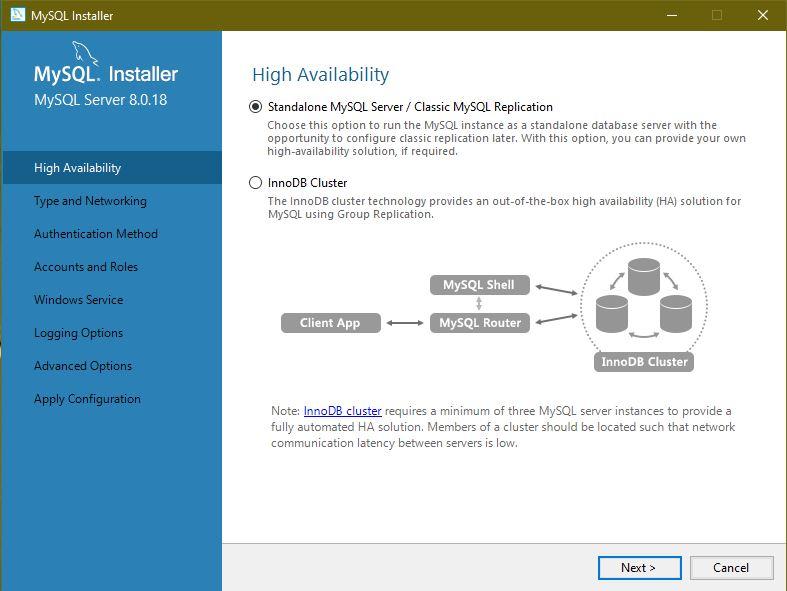
****

**Step 7:** After Execution, click on the Next.

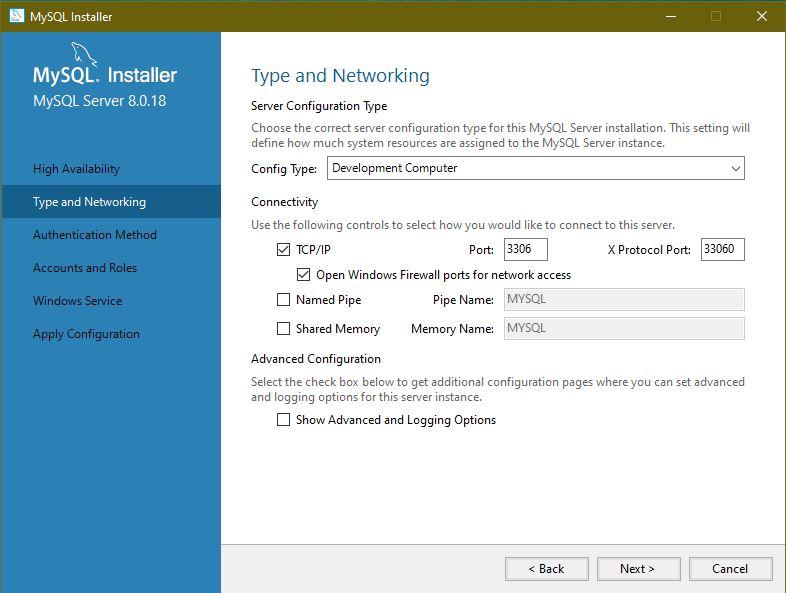
****

**Step 8:** Click Next.

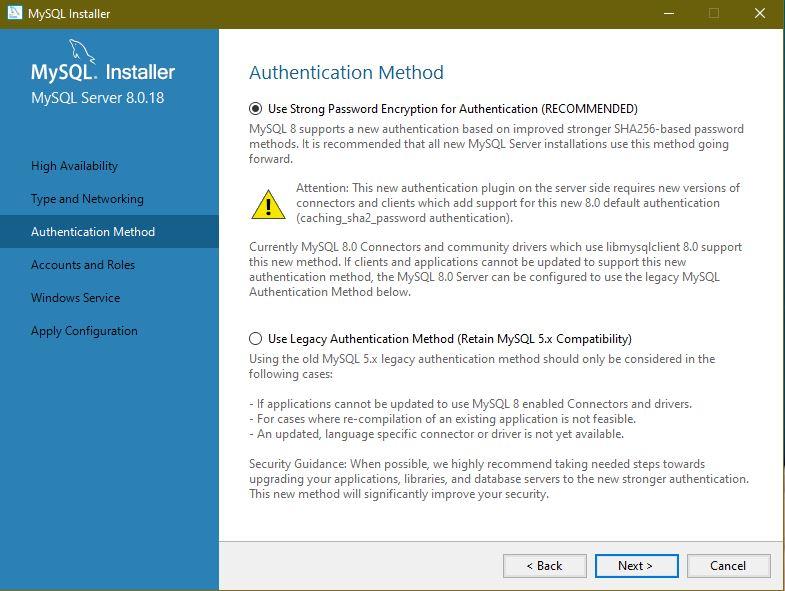
****

**Step 9:** Click Next.****

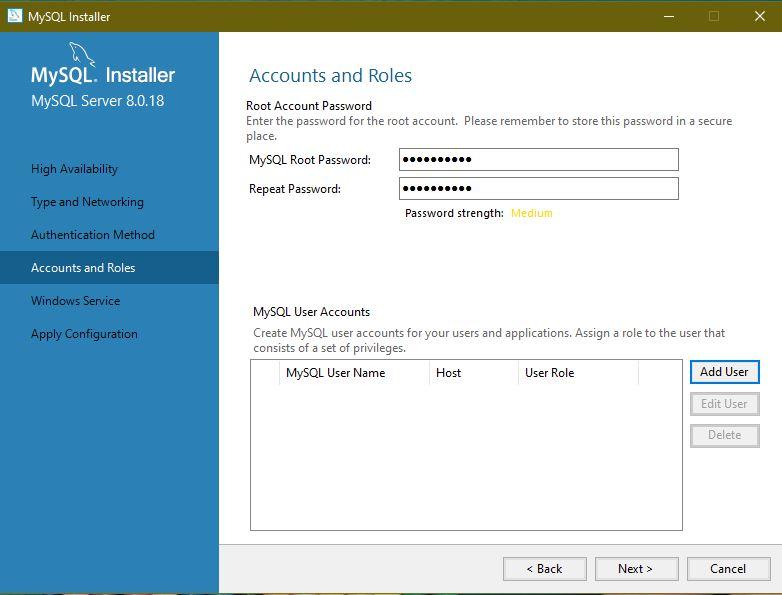
**Step 10:** Leave it as default and click Next.



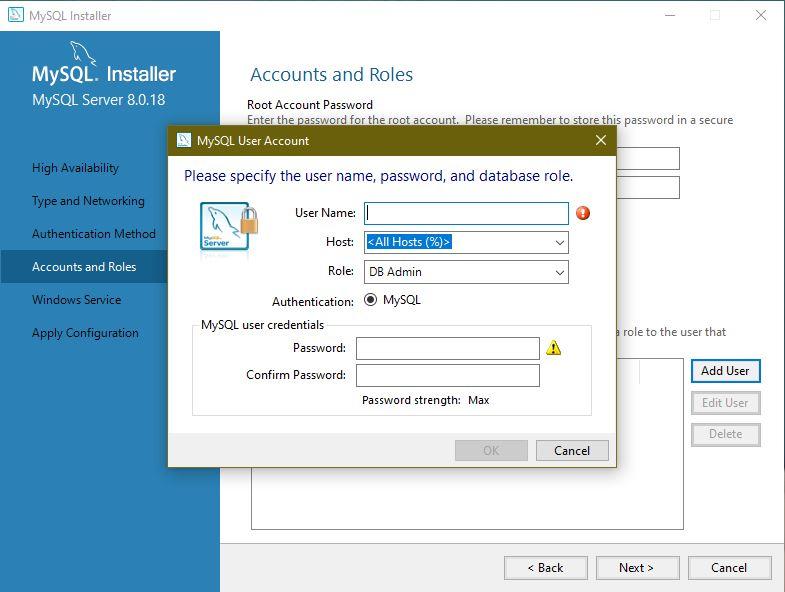
**Step 11:** Click Next

****

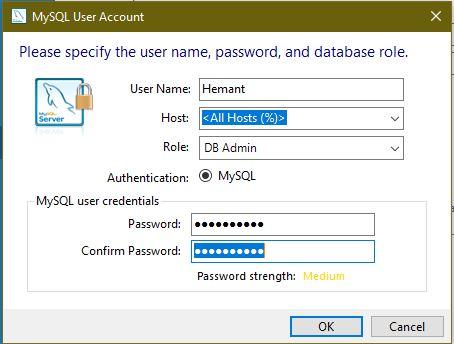
**Step 12:** Choose the root password

****

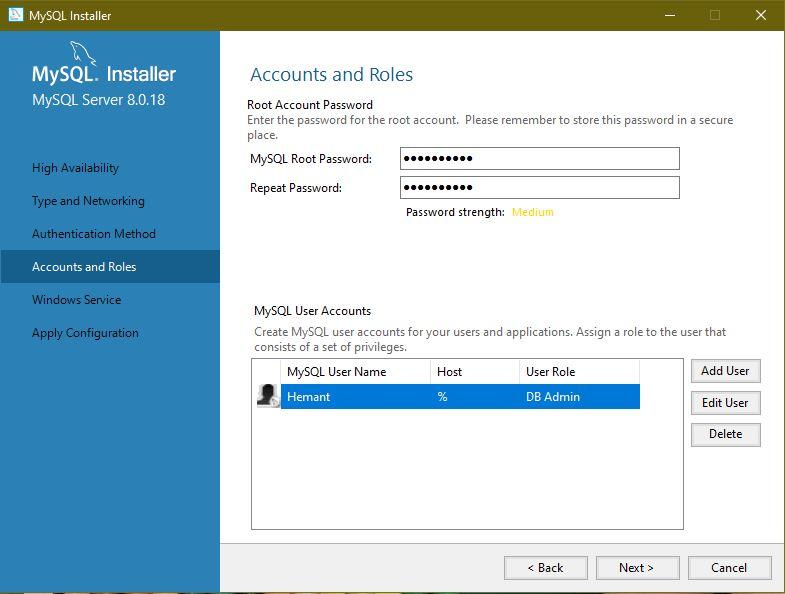
**Step 13:** Click on Add User and give the username and password.

****

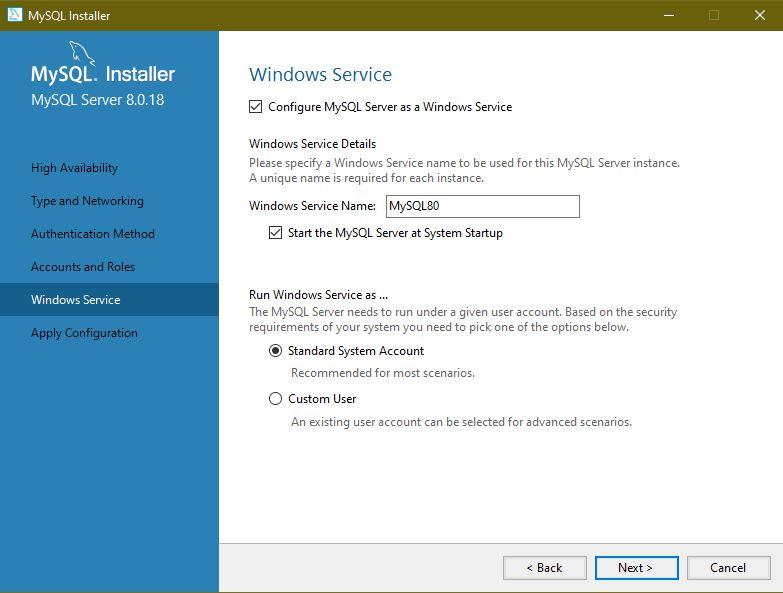
**Step 14:** After inserting the name and password click OK

****

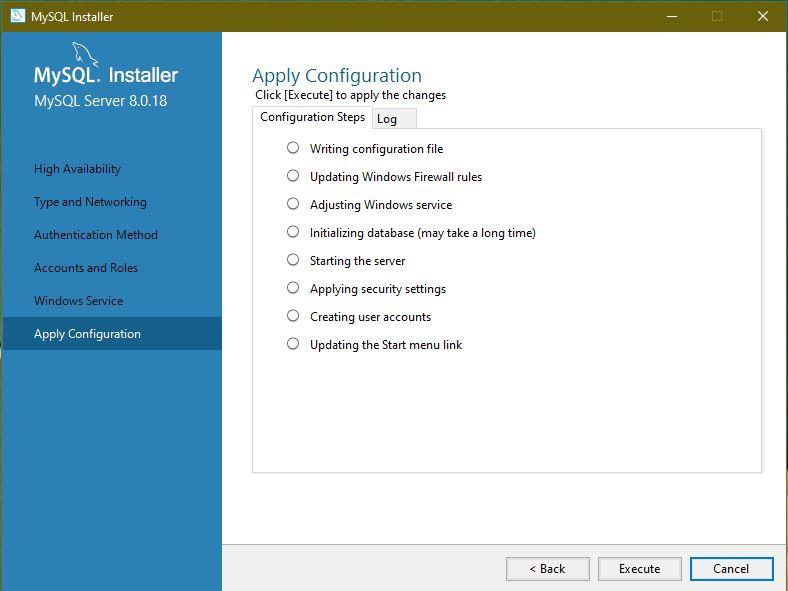
**Step 15:** After adding the user click Next

****

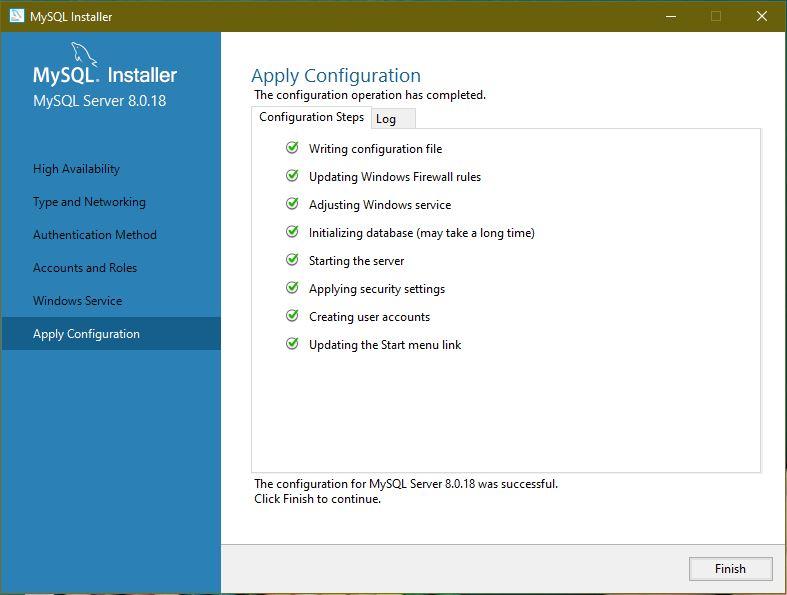
**Step 16:** Click Next

****

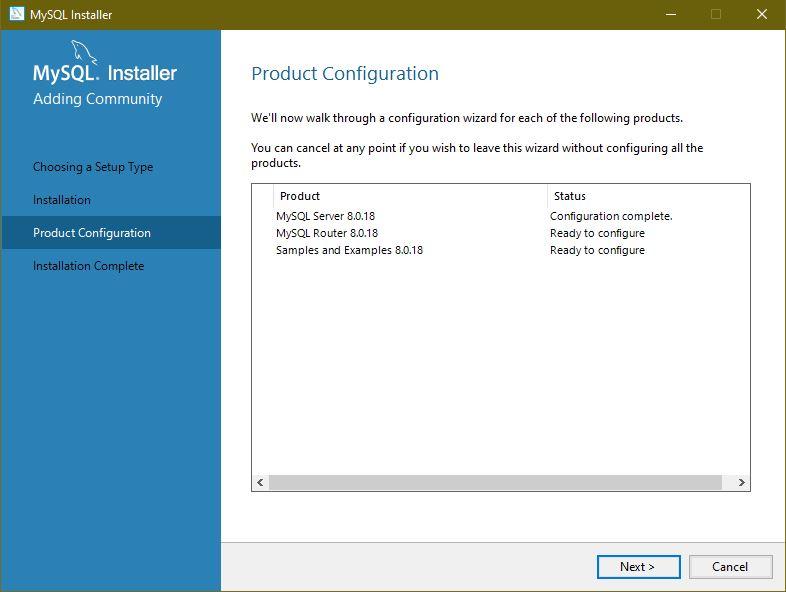
**Step 17:** Click on Execute

****

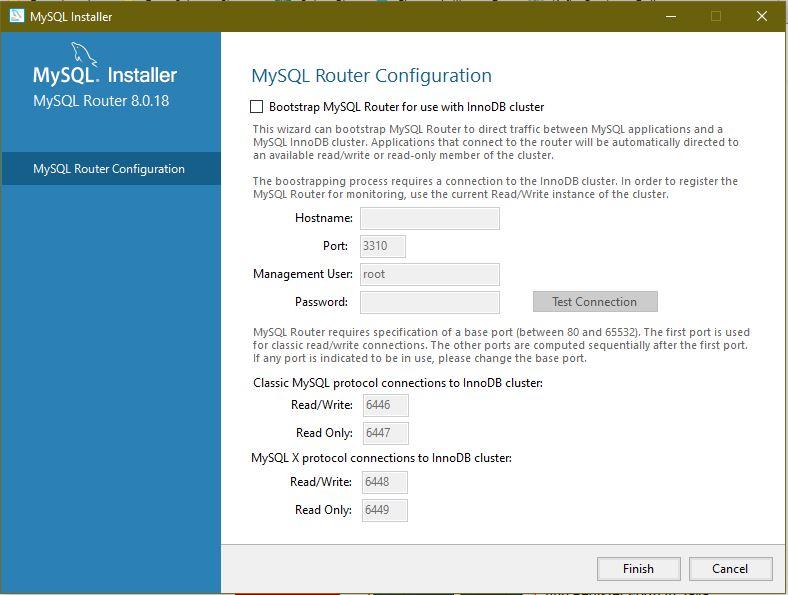
**Step 18:** After Clicking on execute,click Finish

****

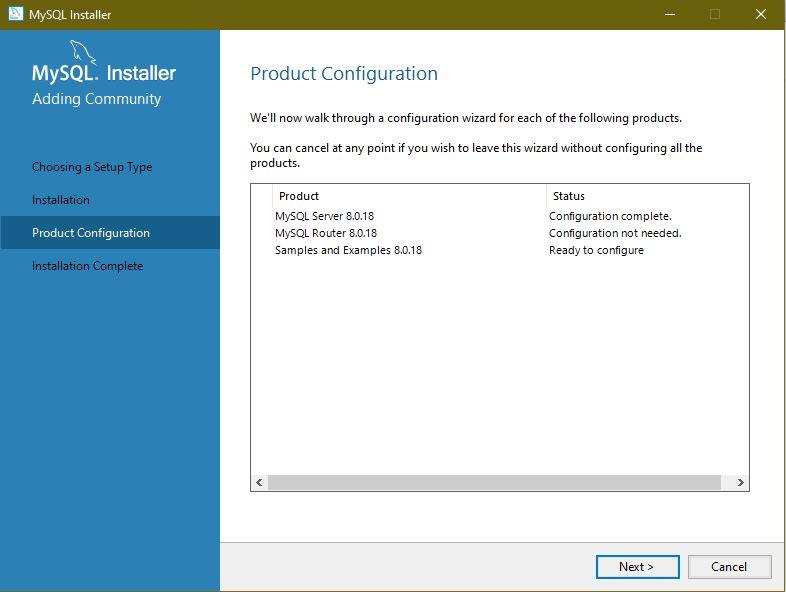
**Step 19:** Click Next

****

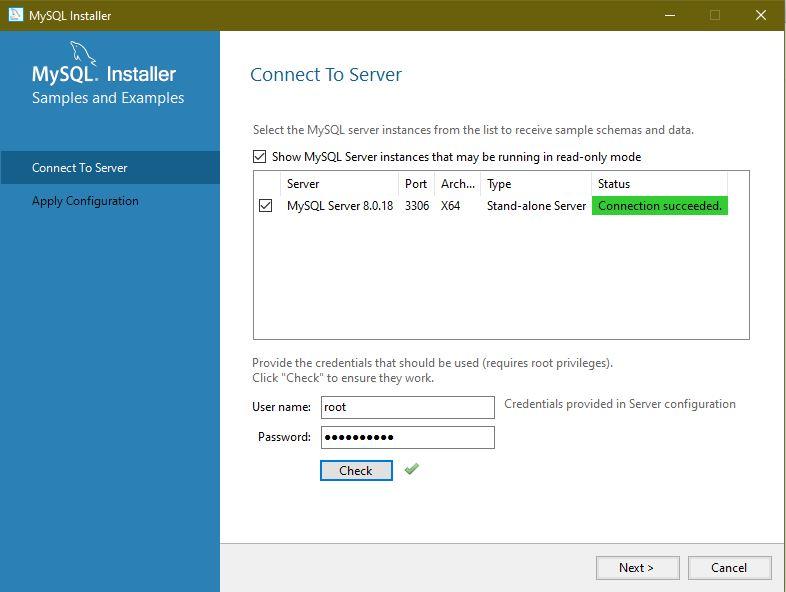
**Step 20:** Click on Finish

****

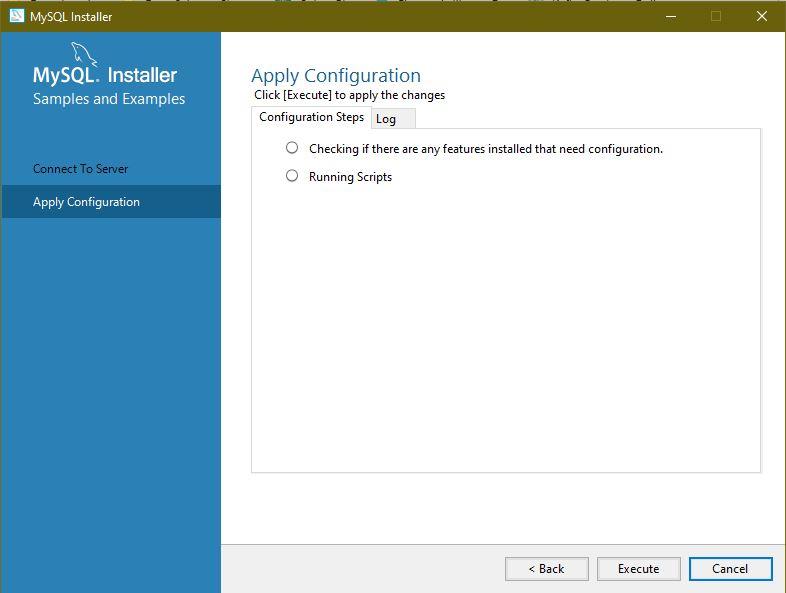
**Step 21:** Click Next.

****

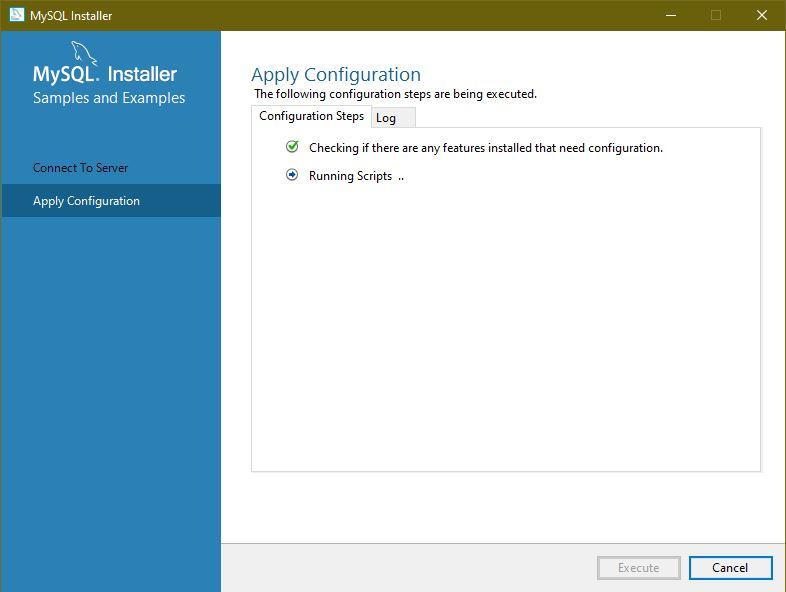
**Step 22:** Check the passwordand Click Next

****

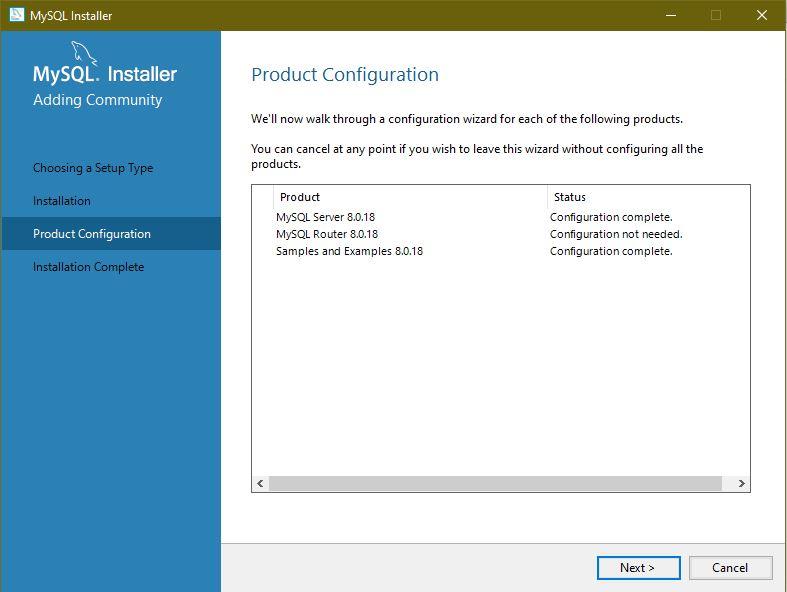
**Step 23:** Click Execute

****

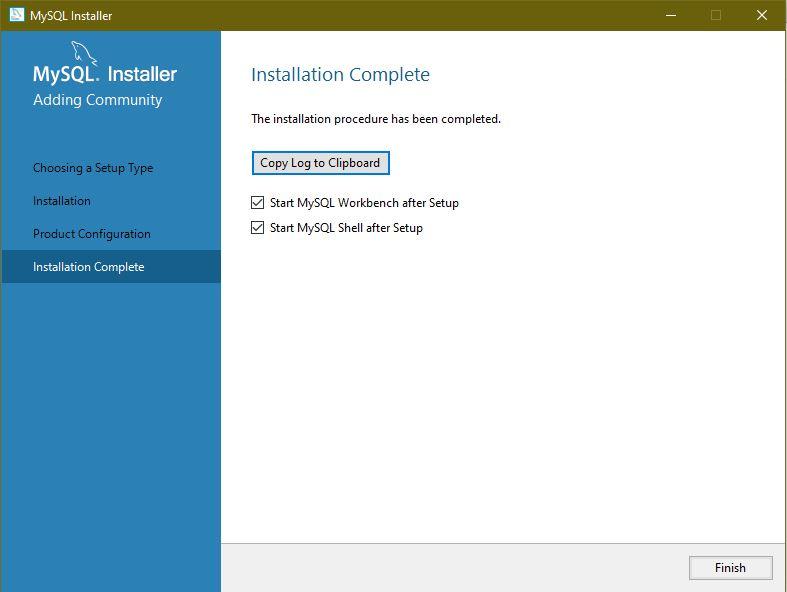
**Step 24:** After clicking on Execute

****

**Step 25:** Click Next

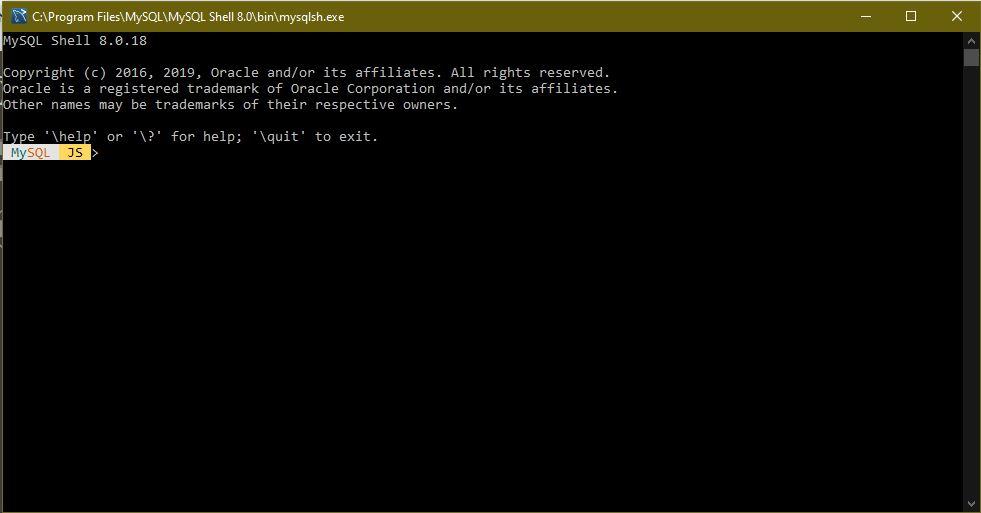


**Step 26:** Click Finish

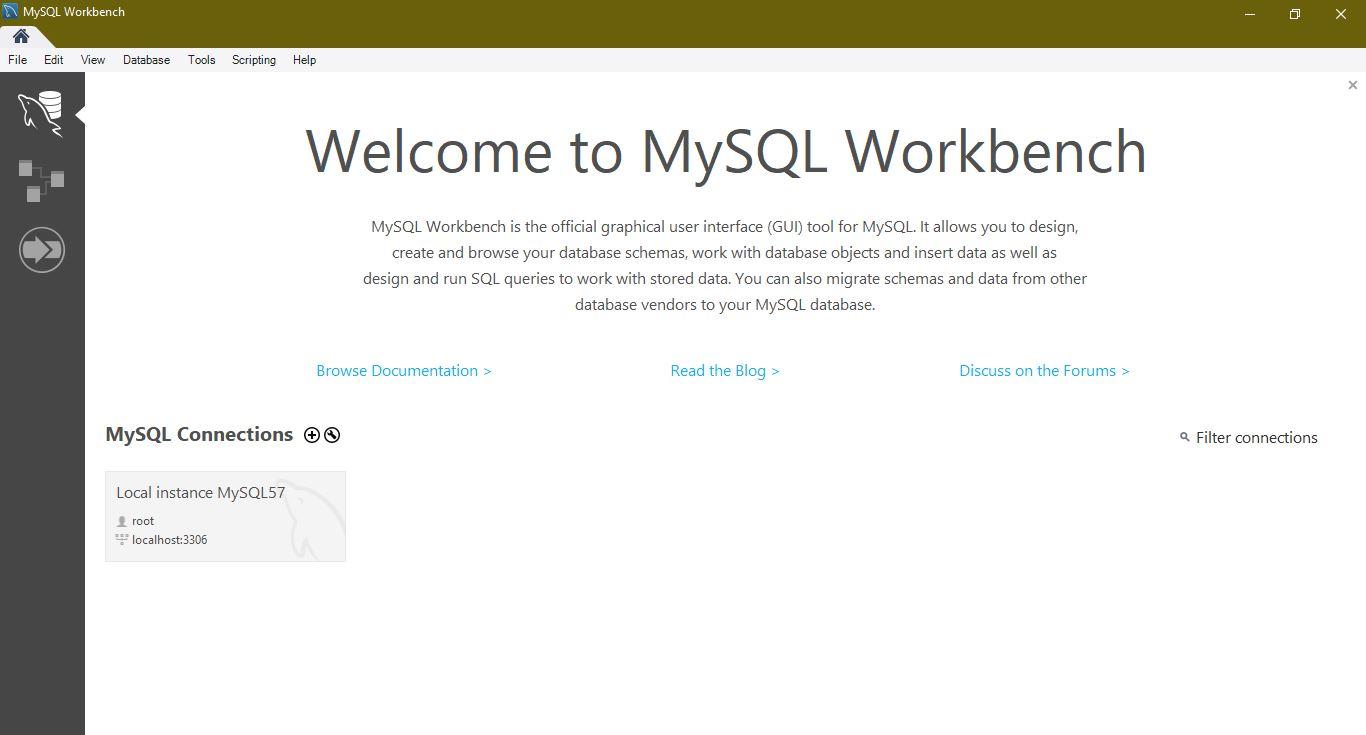
****

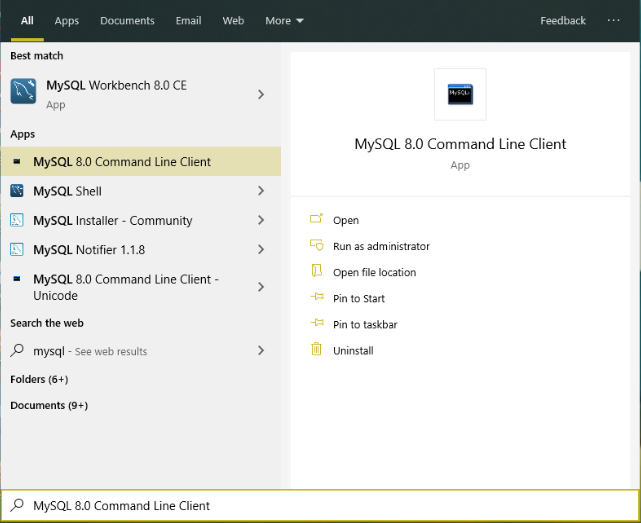
**Step 27:** After the successful installation of MySQL, two windows will open.

* MySQL Shell
* MySQL WorkBench



MySQL WorkBench will tell about database connectivity and other features of MySQL.

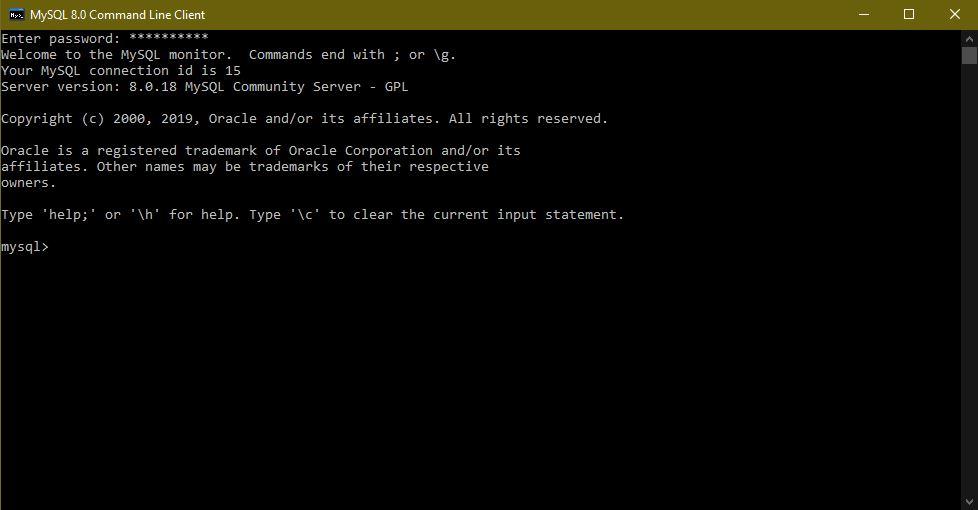


**Step 28:** Click on window Button and search forOpen MySQL Command. 

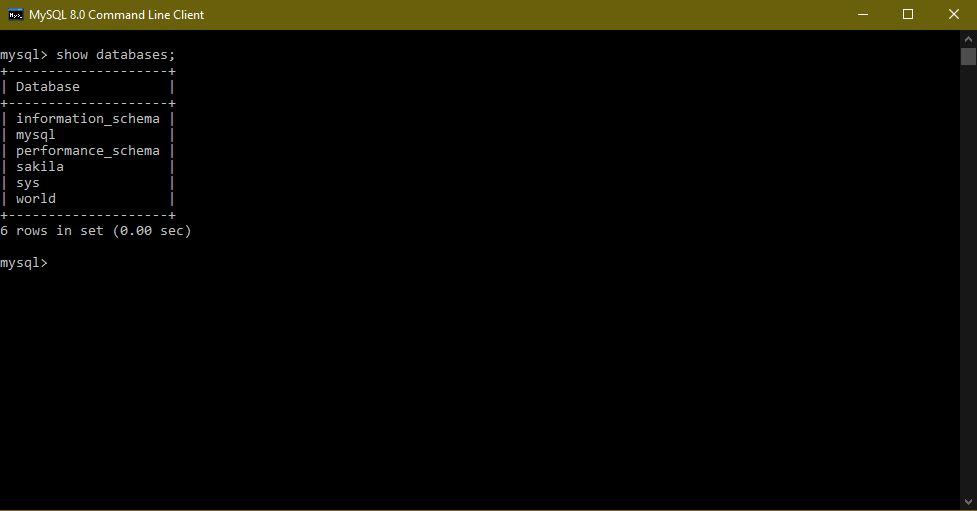
**Step 29:** Open MySQL Command-line Client and enter the password.



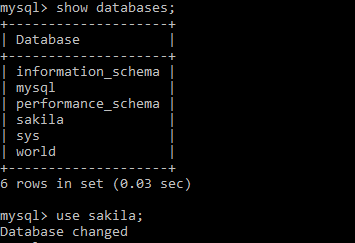
**Step 29:** After entering the password, your MySQL client will get connected with MySQL.



**Step 30:** There are many in-build Databases in MySQL; we can type **show database.**



**Step 31**: we can use any of the above databases by just typing **use database\_name**



# SQL QUERY

A database most often contains tables. Some name identifies each table. The table includes records(rows) with Data. To access those records, we need SQL Syntax. Most of the action you need to perform Database by using the SQL Statement.

Note: SQL keywords are not case-sensitive (e.g., select as SELECT)

* The syntax of the language describes the language element.
* SQL syntax is somewhat like simple English sentences.
* Keywords include SELECT, UPDATE, WHERE, ORDER BY ETC.

Four fundamental operations that can apply to any databases are:

* 1. Read the Data -- **SELECT**
  2. Insert the new Data -- **INSERT**
  3. Update existing Data -- **UPDATE**
  4. Remove Data –**DELETE**

These operations are referred to as the **CRUD** (Create, Read, Update, Delete).

**The SQL SELECT QUERY**

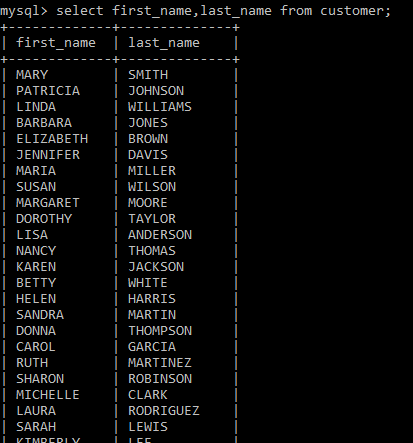
The SELECT statement permits you to read data from one or more tables.

The general syntax is:

SELECT first\_name, last\_name

FROM customer;

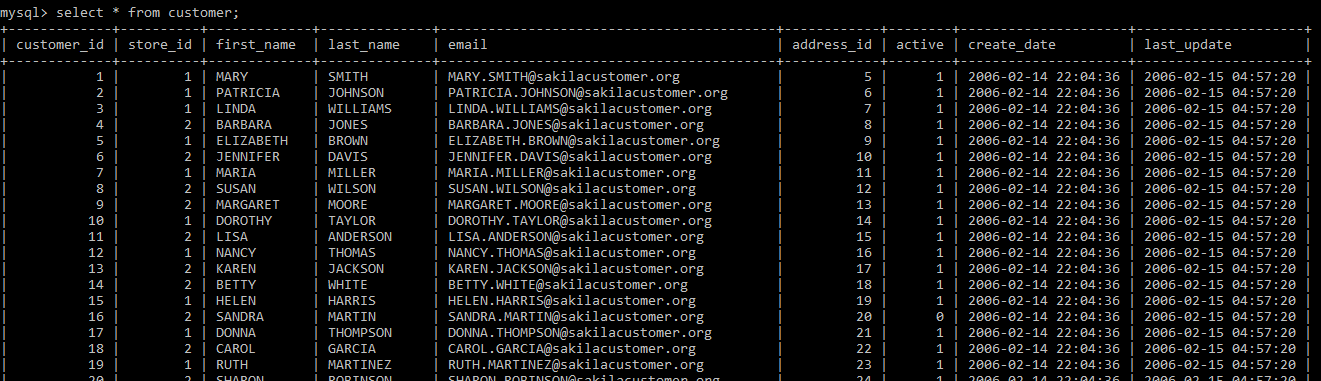
**Example**: Read the first\_name and last\_name from table **customer**.



To select all columns, use **\***

SELECT **\***

FROM customer;

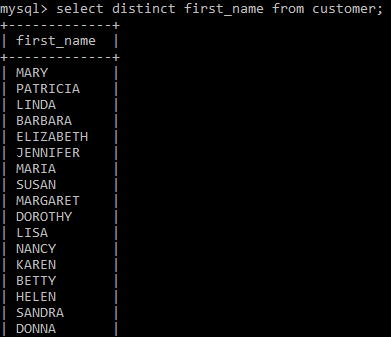


## The SQL SELECT DISTINCT

The SELECT DISTINCT statement is to return the different values.

SELECT DISTINCT first\_name

FROM customer;



## The SQL WHERE CLAUSE

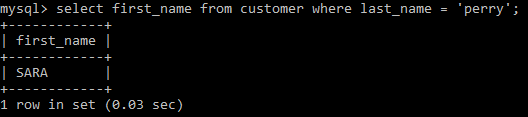
The WHERE clause allows the user to filter the data from the table. The WHERE clause allows the user to extract only those records that satisfy a specified condition.

**When we access, the Text value**

SQL requires single quotes around **text** **values** (many database systems will also use double quotes). And **numeric** **fields** should not be enclosed in quotes.

SELECT first\_name FROM customer

WHERE last\_name = ‘perry’;



**When we access the Numeric field**

SELECT first\_name , last\_name FROM customer WHERE active = 0;



**Operators in where clause**

| **=** | Equal |
| --- | --- |
| **>** | Greater than |
| **<** | Less than |
| **>=** | Greater than equal |
| **<=** | Less than equal |
| **< >** | Not equal (also written as !=) |
| **BETWEEN** | Between a range |
| **LIKE** | Search for pattern |
| **IN** | Specify multiple possible values for a column |

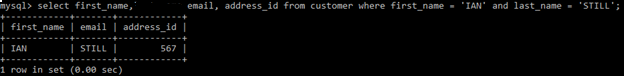
## The SQL WHERE CLAUSE WITH AND, OR & NOT

**A WHERE clause with AND**:

SELECT first\_name, email, address\_id

FROM customer

WHERE fisrt\_name = ‘IAN’ AND last\_name = ‘STILL’

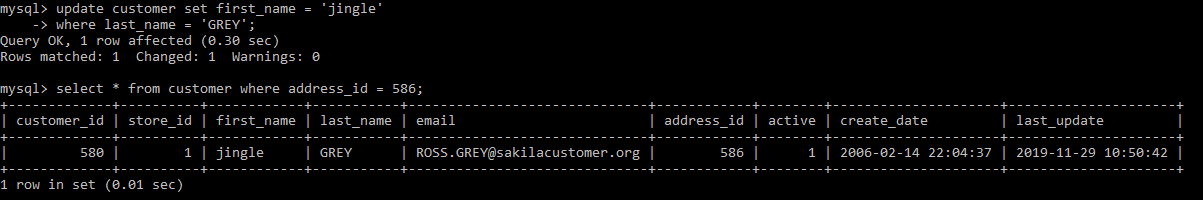


**A WHERE clause with OR:**

UPDATE customer

SET first\_name = ‘jingle’

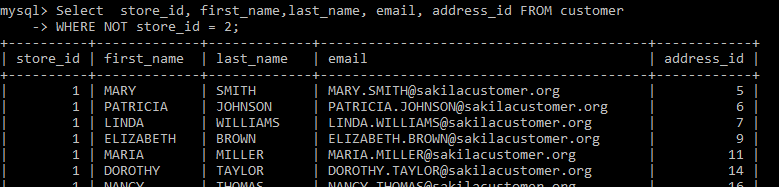
WHERE last\_name = ‘ GREY’;



**A WHERE clause with NOT:**

Select store\_id, first\_name,last\_name, email, address\_id FROM customer

WHERE NOT store\_id = 2;



## The SQL ORDER BY

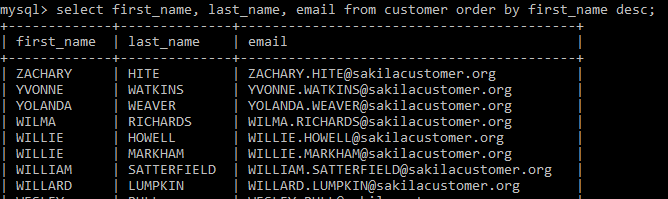
Order by is used to print the values from the table in order(ascending or descending)

**Order By in Descending order**

SELECT first\_name, last\_name,email

FROM customer

ORDER BY first\_name DESC;

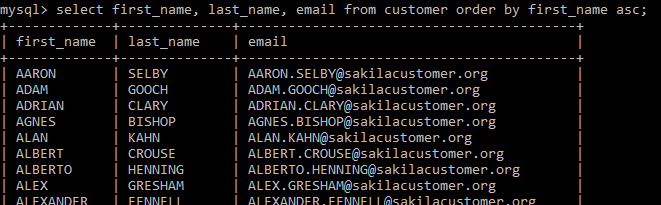


**Order By in Ascending order**

SELECT first\_name, last\_name,email

FROM customer

ORDER BY first\_name ASC;



## The SQL SELECT TOP CLAUSE

The **SELECT TOP** is used to specify the number of records from the to return. The SELECT TOP is useful on large tables with millions of records. It is returning a large number of records that can impact performance.

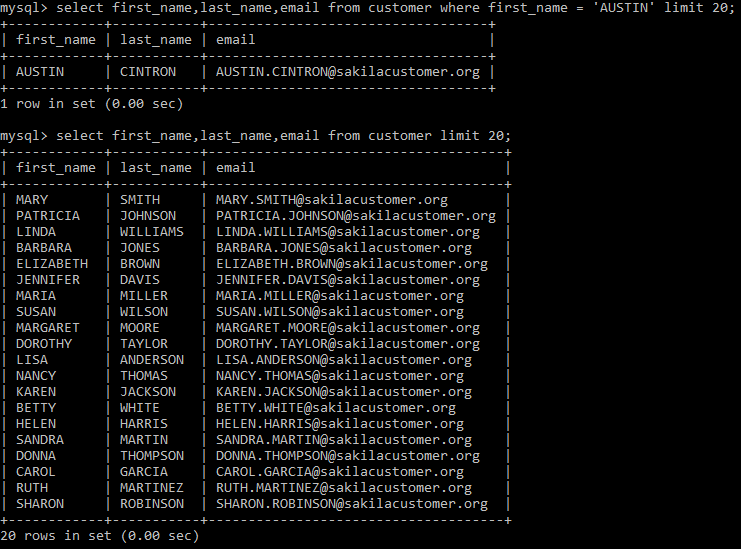
**Note**: Not all database systems support the SELECT TOP clause. MySQL supports the LIMIT clause to select a limited number of records, while Oracle uses ROWNUM.

**MySQL Syntax:**

SELECT first\_name, last\_name,email

FROM customer WHERE first\_name = ‘AUSTIN’

LIMIT 20;



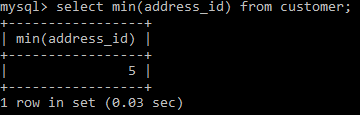
## The SQL MIN() AND MAX() FUNCTION

The MIN() function in SQL returns the smallest value of the selected column from the table. The MAX() function in SQL returns the largest value of the selected column from the table.

**MIN() Syntax**

SELECT MIN(address\_id)

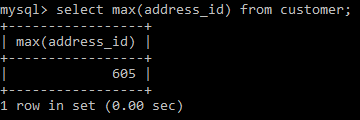
FROM customer;



**MAX() Syntax**

SELECT MAX(address\_id)

FROM customer;



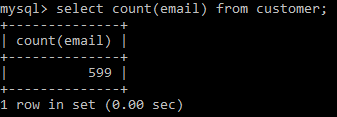
## The SQL COUNT(), AVG() AND SUM() FUNCTION

The **COUNT()**  function gives the number of rows that matches specified conditions. And the **AVG()** function in SQL returns the average value of a numeric column. The **SUM()** function in SQL returns the total sum of a numeric column.

**COUNT() Syntax**

SELECT COUNT(email)

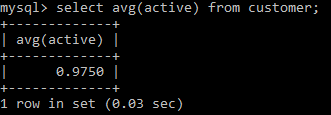
FROM customer;



**AVG() Syntax**

SELECT AVG(active)

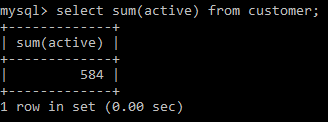
FROM customer;



**SUM() Syntax**

SELECT SUM(active)

FROM customer



## The SQL LIKE-OPERATOR

The **LIKE** operator is used with the WHERE clause to find for a specified pattern in an attribute. The two wildcards are used in conjunction with the LIKE operator:

* **%** - it represents zero, one, or multiple characters
* **\_** - it represents a single character

**Note**: MS Access uses an asterisk (\*) in place of the percent sign (%)and a question mark (?) in place of the underscore (\_).

The ‘%’ and the ‘\_’ can also be used in combinations.

**LIKE Syntax**

SELECT column1, column2, ...

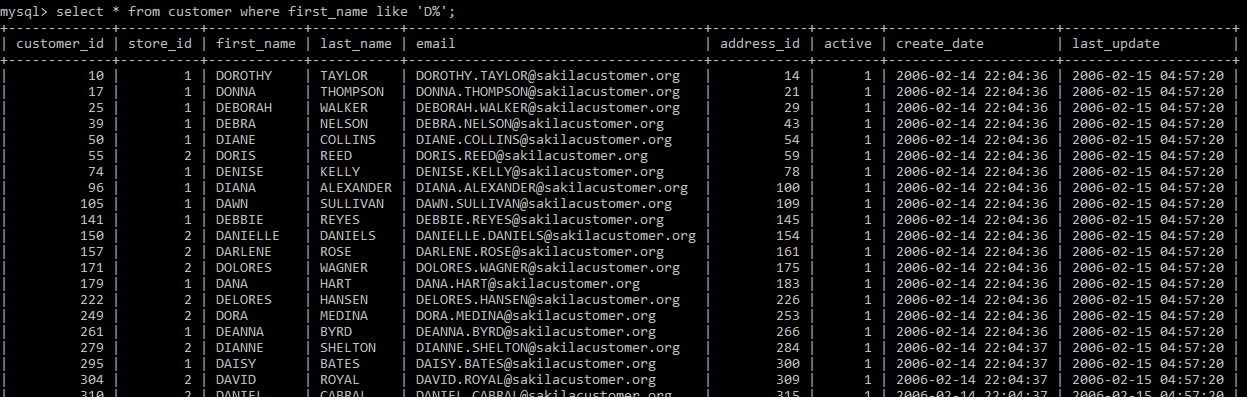
FROM table\_name

WHERE column LIKE pattern;

**Selects all columns of the customer with a first\_name starting with "D".**

SELECT \* FROM customer

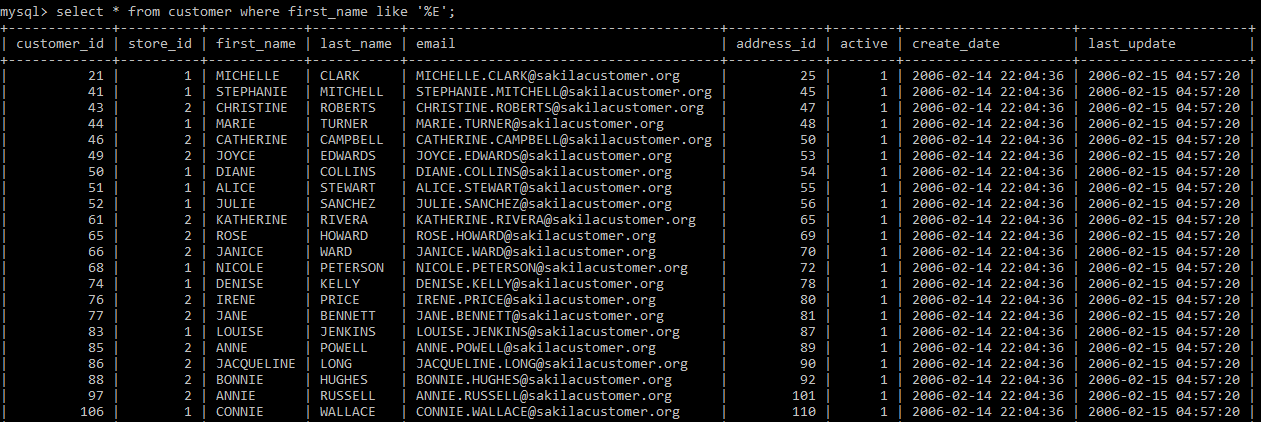
WHERE first\_name LIKE 'D%';



**Selects all columns of the customer with a first\_name Ending with "E":**

SELECT \* FROM customer

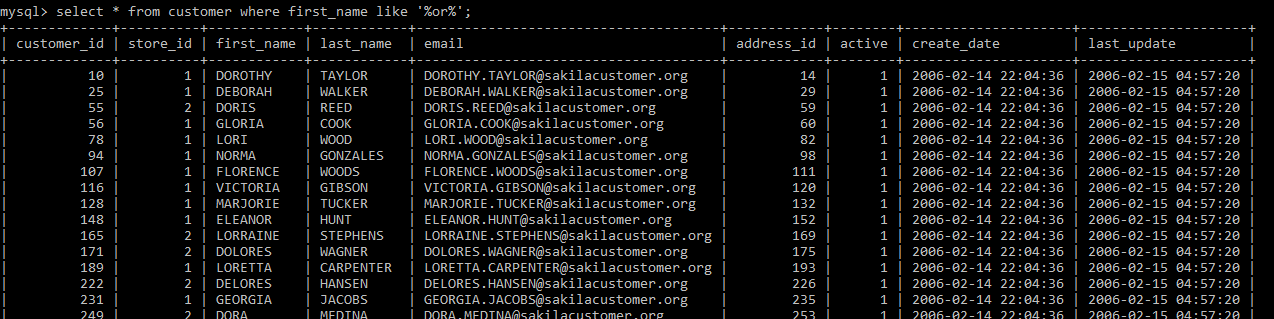
WHERE first\_name LIKE '%E';



**Selects all columns of the customer with a first\_name that have "or" in any position.**

SELECT \* FROM customer

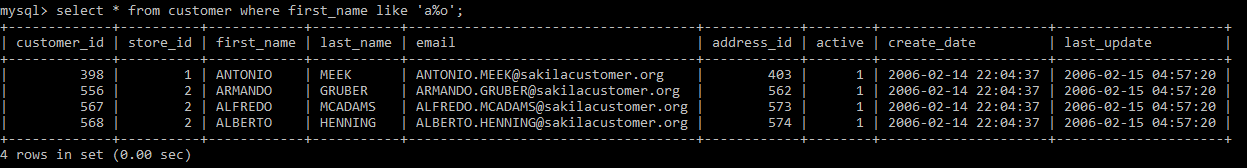
WHERE first\_name LIKE '%or%';



**Selects all columns of the customer with a first\_name that starts with "a" and ends with "o":**

SELECT \* FROM customer

WHERE first\_name LIKE 'a%o';



**Selects all columns of the customer with a first\_name that starts with "a" and are at least six characters in length:**

SELECT \* FROM customer

WHERE first\_name LIKE 'a\_\_%';



## The SQL IN AND NOT IN OPERATORS

The **IN** operator allows users to specify multiple values in a WHERE clause. The IN operator is a shorthand for various **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);

**Selects all the columns of customer whose customer\_id in (1,2,3):**

SELECT \* FROM customer

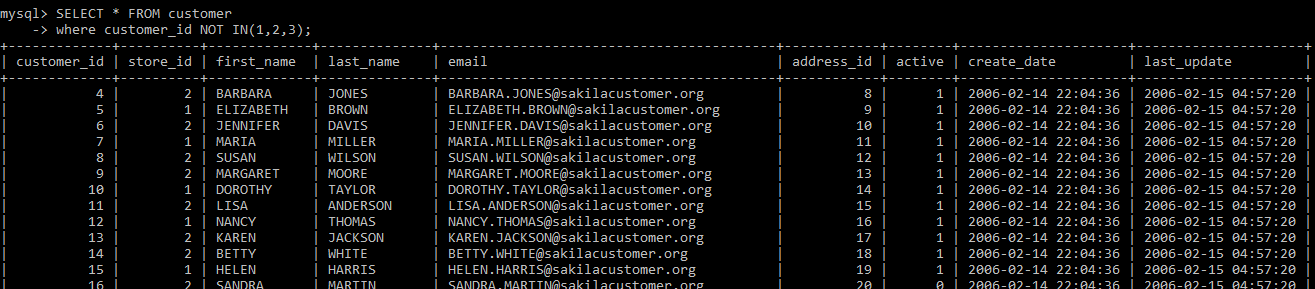
WHERE cutomer\_id IN (1,2,3);



**Selects all the columns of customer whose customer\_id in (1,2,3):**

SELECT \* FROM customer

WHERE cutomer\_id NOT IN (1,2,3);



## The SQL BETWEEN OPERATOR

The **BETWEEN** operator retrieves values within the given range. The values can be texts, numbers, or dates. The **BETWEEN** operator is inclusive: begin and end values are included.

**BETWEEN Syntax**

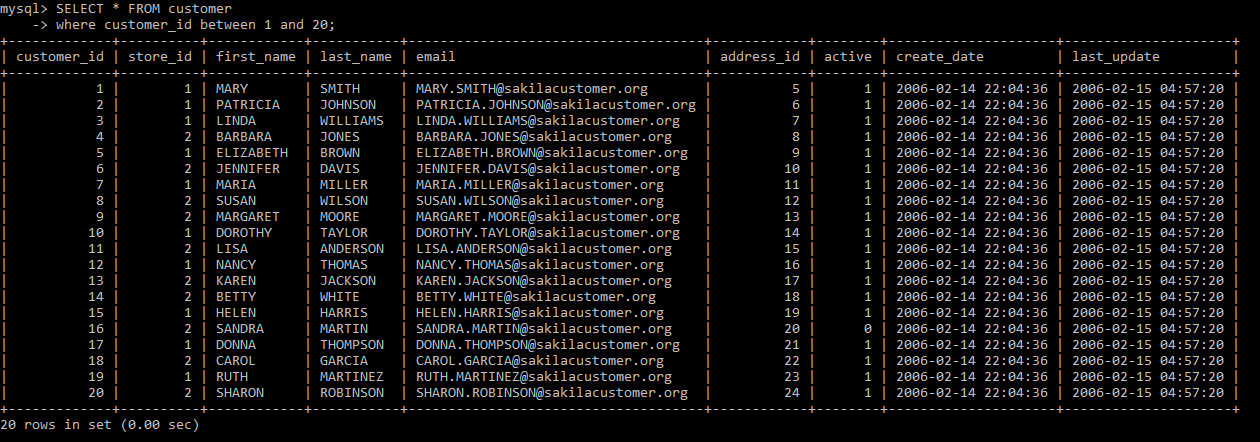
SELECT column\_name(s)

FROM table\_name

WHERE column\_name BETWEEN value1 AND value2;

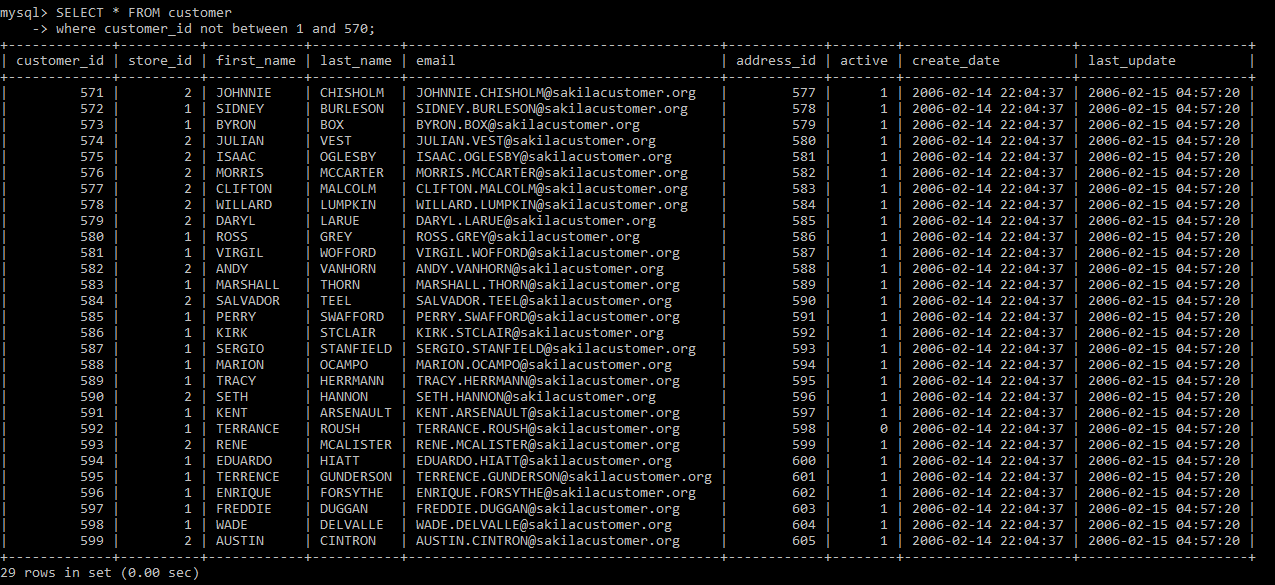
**Select all the columns from the customer with customer\_id between 1 to 20.**

SELECT \* FROM customer WHERE customer\_id BETWEEN 1 AND 20;



**Select all the columns from the customer with customer\_id, not between 1 to 570.**

SELECT \* FROM customer WHERE customer\_id NOT BETWEEN 1 AND 570;



## The SQL ALIAS

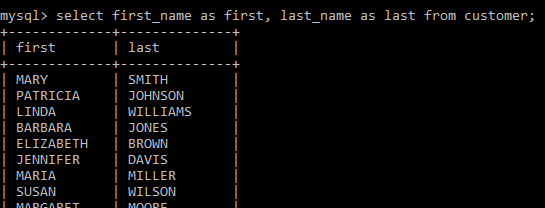
Aliases are used to give a nickname to a column in a table, a temporary name. Aliases are used to make column names more readable to the user.

**Alias Column Syntax**

SELECT first\_name AS first, last\_name AS last

FROM customer;

**Creates two aliases, one for the first\_name column and one for the last\_name column:**

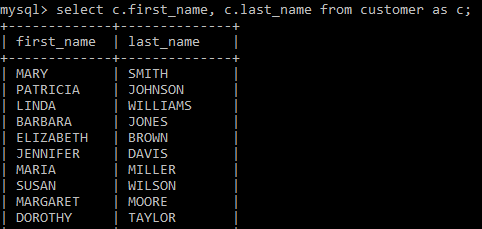


**Alias Table Syntax**

SELECT c.first\_name, c.last\_name

FROM customer AS c

**Create an alias for the customer table**



## The SQL GROUP BY STATEMENT

The **GROUP BY** used to group rows from the table. And it has the same values as summary rows. For example, find the number of customers in each country, The **GROUP BY** is often used with aggregate functions like (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

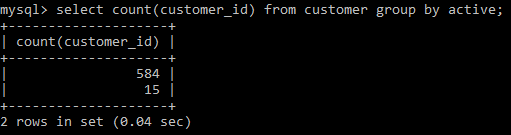
WHERE condition

GROUP BY column\_name(s)

ORDER BY column\_name(s);

**Count the number of active and non-active customers**

SELECT COUNT(customer\_id) FROM customer GROUP BY active



## The SQL HAVING CLAUSE

The **HAVING** clause is added to SQL because the WHERE keyword can not be used with aggregate functions.

**HAVING Syntax**

SELECT column\_name(s)

FROM table\_name

WHERE condition

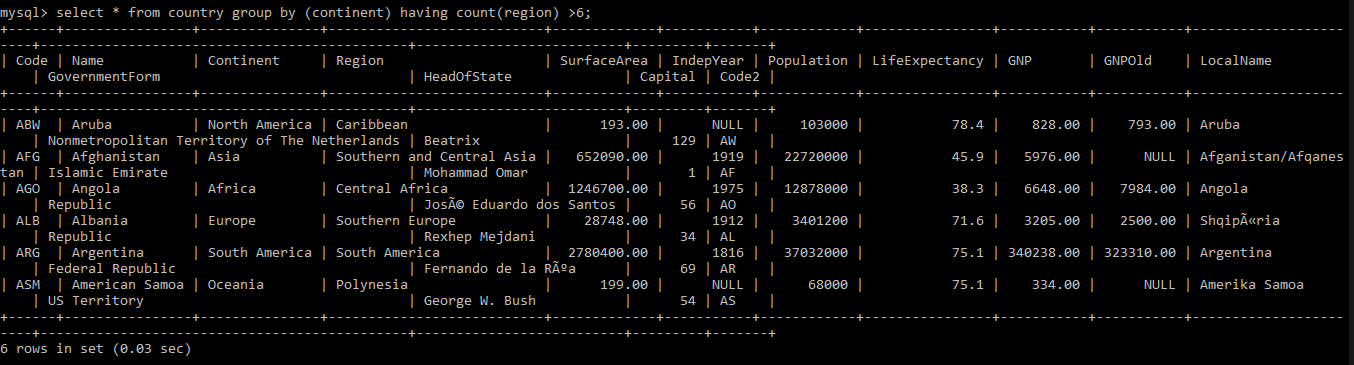
GROUP BY column\_name(s)

HAVING condition

ORDER BY column\_name(s);

**List the number of continents which has a region more than 6.**

SELECT \* from country group by(continent) having count(region) >6;



## The SQL UNION

The UNION operator allows the user to combine the result-set of two or more SELECT statements in SQL. Each SELECT statement within UNION should have the same number of columns. The columns in each SELECT statement should also be in the same order. The columns should also have similar data types.

**The SQL UNION**

Select column\_name(s) from table1

UNION

Select column\_name(s) from table2;

**UNION ALL Query**

The UNION operator selects only different values by default. To allow duplicate values, the user can use UNION ALL operator.

SELECT column\_name(s) FROM table1

UNION ALL

SELECT column\_name(s) FROM table2;

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

## The SQL STORED PROCEDURE

**What is a SQL Stored Procedure?**

The **stored procedure** is a prepared SQL query that you can save so that the query can be **reused** over and over again. So, if the user has an SQL query that you write over and over again, keep it as a stored procedure and execute it. Users can also pass parameters to a stored procedure so that the stored procedure can act based on the parameter value that is given.

**Stored Procedure Syntax**

CREATE PROCEDURE procedure\_name

AS

sql\_statement

GO;

**Execute a Stored Procedure**

EXEC procedure\_name;

# SQL JOIN

The SQL Join help in retrieving data from two or more database tables. The tables are mutually related using primary keys and foreign keys.

**Type of Join**

## INNER JOIN

The **INNER JOIN** is used to print rows from both tables that satisfy the given condition. For example, the user wants to get a list of users who have rented movies together with titles of movies rented by them. Users can use an INNER JOIN for that, which returns rows from both tables that satisfy with given conditions.

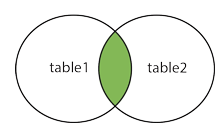


Fig. INNER JOIN

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

**INNER JOIN Syntax**

SELECT column\_name(s)

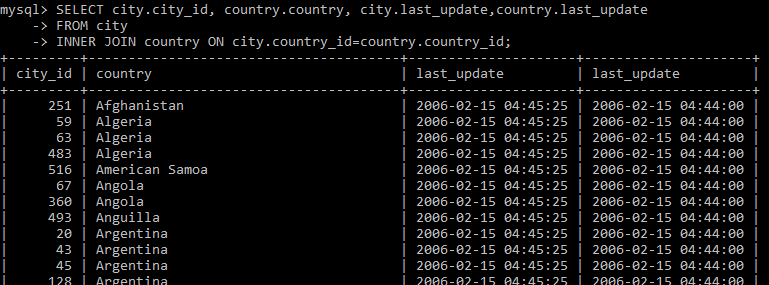
FROM table1

INNER JOIN table2

ON table1.column\_name = table2.column\_name;

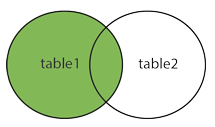
SELECT city.city\_id, country.country, city.last\_update, country.last\_update FROM city

INNER JOIN country ON city.country\_id = country.country\_id



## LEFT JOIN

The **LEFT JOIN** returns all the records from the table1 (left table) and the matched records from the table2 (right table). The output is NULL from the right side if there is no match.



Left Join

**LEFT JOIN Syntax**

SELECT column\_name(s)

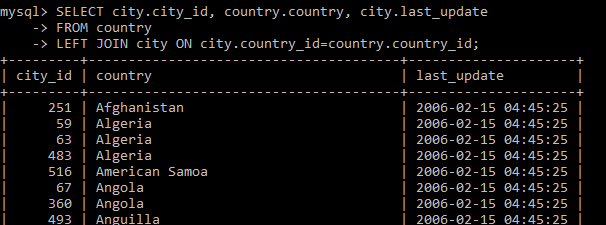
FROM table1

LEFT JOIN table2

ON table1.column\_name = table2.column\_name;

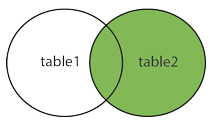
SELECT city.city\_id, country.country, city.last\_update, country.last\_update FROM city

LEFT JOIN country ON city.country\_id = country.country\_id



## RIGHT JOIN

The RIGHT JOIN is the opposite of LEFT JOIN. The RIGHT JOIN prints all the columns from the table2(right table) even if there no matching rows have been found in the table1 (left table). If there no matches have been found in the table (left table), NULL is returned.



RIGHT JOIN

**RIGHT JOIN Syntax**

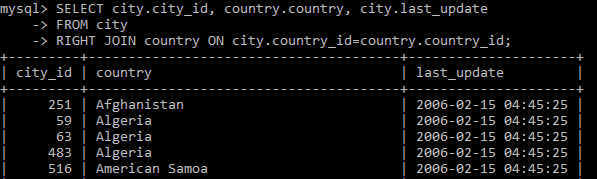
SELECT column\_name(s)

FROM table1

RIGHT JOIN table2 ON table1.column\_name = table2.column\_name;

SELECT city.city\_id, country.country, city.last\_update, country.last\_update FROM city

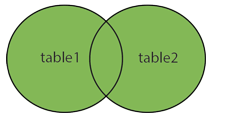
RIGHT JOIN country ON city.country\_id = country.country\_id



## Full OUTER JOIN

The FULL OUTER JOIN keyword returns all records when there are a match in left (table1) or right (table2) table records.

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



Full Join

Tip: FULL OUTER JOIN and FULL JOIN are the same.

**FULL OUTER JOIN Syntax**

SELECT column\_name(s)

FROM table1

FULL OUTER JOIN table2

ON table1.column\_name = table2.column\_name WHERE condition;

**Note**: MySQL does not support the Full Join, so we can perform left join and right join separately then take the union of them.

SELECT \* FROM t1

LEFT JOIN t2 ON t1.id = t2.id

UNION

SELECT \* FROM t1

RIGHT JOIN t2 ON t1.id = t2.id

## SELF-JOIN

A self-JOIN is a regular join, but the table is joined with itself.

**Self -JOIN Syntax**

SELECT column\_name(s)

FROM table1 T1, table1 T2

WHERE condition;

# SQL DATABASE

## The SQL CREATE DATABASE STATEMENT

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

**Syntax**

CREATE DATABASE database\_name;

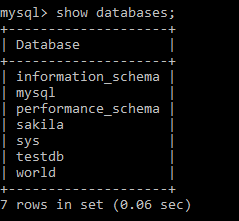
**Let’s create a database and give name as testdb**

CREATE database testdb;



Now, let’s check the databases in MySQL by using **show databases** query.

Show databases;



## The SQL DROP DATABASE STATEMENT

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

**Syntax**

DROP DATABASE database\_name;

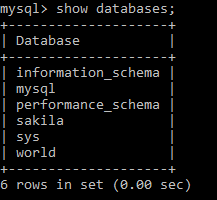
**Let’s drop the created database by using drop database testdb.**

DROP database testdb;



Now, let’s check the databases in MySQL by using **show databases** query after dropping the testdb.

SHOW databases;



The created database(testdb) has been dropped.

## The SQL CREATE TABLE

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

**Syntax**

CREATE TABLE table\_name (

column1 data\_type,

column2 data\_type,

column3 data\_type,

....

);

The column1, column2, ….., 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.)

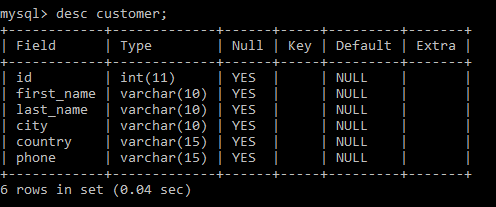
**Let’s create a customer table**

CREATE TABLE cutomer(id integer, first\_name varchar(10), last\_name varchar(10), city varchar(10), country varchar(15), phone varchar(15));



**To check the schema of the table, use desc table\_name.**

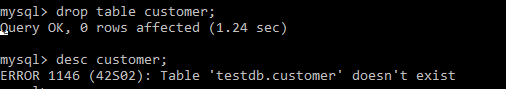
DESC customer;



## The SQL DROP TABLE STATEMENT

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

DROP TABLE customer;



The table has dropped after running the query drop table table\_name. As we can see, the table does not exist after dropped.

Now we are going to create the same table again to insert the values in that table.

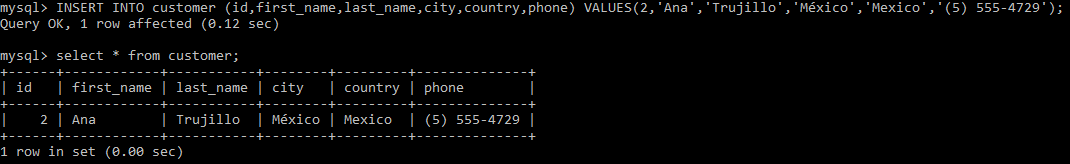
## The SQL INSERT INTO STATEMENT

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

**INSERT INTO query**

We can write the INSERT INTO statement in two ways. The first way is to specify both the column names and the values to be inserted:

INSERT INTO customer(id , first\_name, last\_name ,city ,country,phone)VALUES (2, ‘Ana’, ‘Trujillo’, ‘Mexico’, ‘Mexico’, (5) 555-4729);

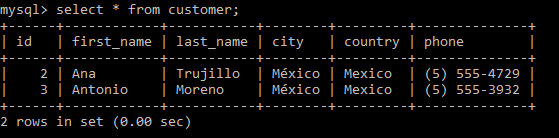


If users are adding values for all the columns of the table, you don’t need to specify the particular column names in the SQL query. However, ensure the order of the values is in the same order as the columns in the table.

**The INSERT INTO query would be as follows**:

INSERT INTO customer

VALUES (3, ‘Antonio, ‘Moreno, ‘Mexico’, ‘Mexico’, (5) 555-3932);



We have inserted two rows yet. Similarly, we can insert many rows in the table. Finally, we have added ten rows as we can see in the picture below.

SELECT \* FROM customer;



## The SQL NULL VALUES

**What is a NULL Value?**

The field with a NULL value is a field with no value. If the field in a table is optional, to insert new data or update data without adding a value to this field and Then, the field will be saved as a NULL value.

**Note**: A NULL value is not the same as a zero value, or we can say a field that holds spaces. The field with a NULL value is one that has been left blank during record creation!

**Insert the NULL values in tables**

INSERT INTO customer VALUES(11, ‘Victoria’, ‘Ashworth’, ‘London’, NULL, ‘(171) 555-1212’)



As we can able to see, the last row contains one NULL value.

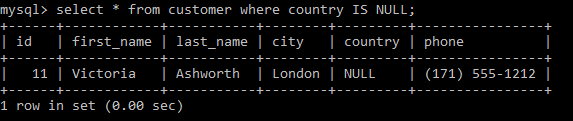
**How to check for NULL Values?**

To test for NULL values in the table has to use the **IS NULL** and **IS NOT NULL** operators instead.

**IS NULL Syntax**

SELECT \*

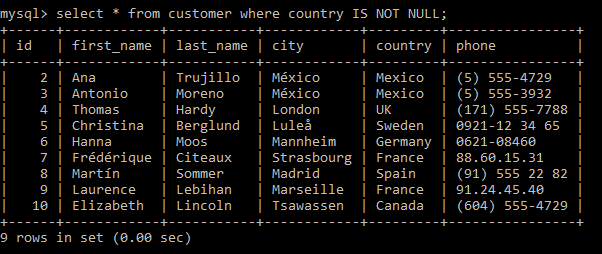
FROM customer WHERE country IS NULL;



**IS NOT NULL Syntax**

SELECT \* FROM customer

WHERE country IS NOT NULL;



It will return those countries which have some values(expect Null values).

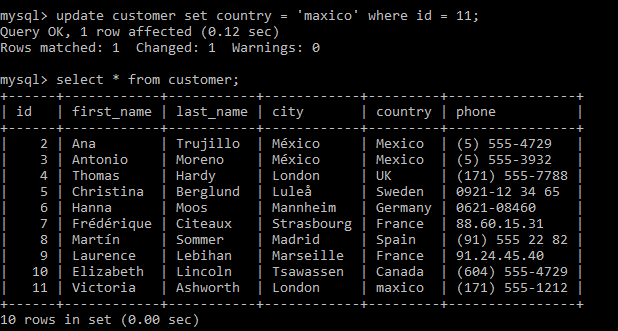
## The SQL UPDATE STATEMENT

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

**UPDATE Syntax**

UPDATE customer

SET country = ‘Mexico’ WHERE id = 11;



We have updated the null value of the country with Mexico.

## The SQL DELETE STATEMENT

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

**DELETE Syntax**

DELETE FROM customer WHERE id = 11;



We have deleted one row, which contains id = 11.

## The SQL ALTER TABLE STATEMENT

The ALTER TABLE statement in SQL is used to add, modify, or delete columns in an existing table. And it also used to add and drop various constraints on a current table.

### ALTER TABLE - ADD COLUMN IN EXISTING TABLE

To add a new column in a table, use the SQL query

ALTER TABLE customer

ADD email varchar(25);



### ALTER TABLE – MODIFY/ALTER COLUMN

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

ALTER TABLE customer ADD COLUMN dob date;



We have assigned the dob with the datatype date. But now we want to change the datatype from date to year.

ALTER TABLE customer MODIFY dob year;



### ALTER TABLE - DROP COLUMN

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

**Syntax:**

ALTER TABLE customer

DROP COLUMN email;



# The SQL CONSTRAINTS

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

**Syntax:**

CREATE TABLE table\_name (

column1 datatype constraint,

column2 datatype constraint,

column3 datatype constraint,

....

);

**SQL Constraints**

SQL constraints are used to specify any rules for the records in a table. Constraints can be used to limit the type of data that can go into a table. It ensures the accuracy and reliability of the records in the table, and if there is any violation between the constraint and the record 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 constraints are commonly used in SQL**

| **CONSTRAINTS** | **DESCRIPTION** |
| --- | --- |
| Not Null | It Ensures that a column cannot have a NULL value. |
| Unique | It Ensures that all the values in a column are unique. |
| Primary Key | It is a combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table. |
| Foreign Key | Uniquely identifies a record /row in another table |
| Check | It checks that all values in a column satisfy a specific condition |
| Default | It gives a default value for a column when no value is specified |
| Index | It is Used to create and retrieve data from the database quickly. |

## NOT NULL CONSTRAINTS

The NOT NULL constraint enforces a column NOT to accept NULL values. This imposes a field always to contain a value, which means that the user cannot insert a new record in a table or update a record without adding a value to this field.

**NOTE**: By default, a column can hold NULL values.

**Create a table using SQL not null constraints**

The following SQL ensures that the "id", "First\_name" and "Last\_name" columns will NOT accept NULL values when the "student" table is created:

**Example**

CREATE TABLE student(

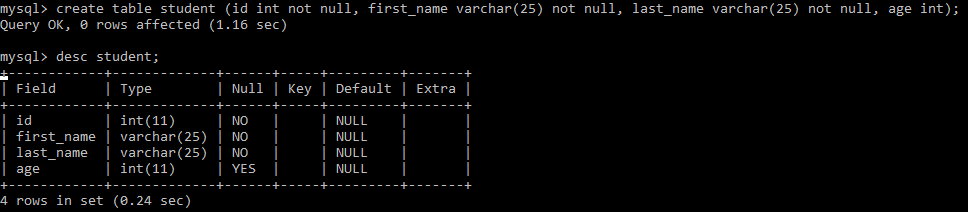
id int NOT NULL,

first\_name varchar(25) NOT NULL,

last\_name varchar(25) NOT NULL,

age int

);



In the above table, it has specified the id, first\_name, and last\_name as not null and age as null.

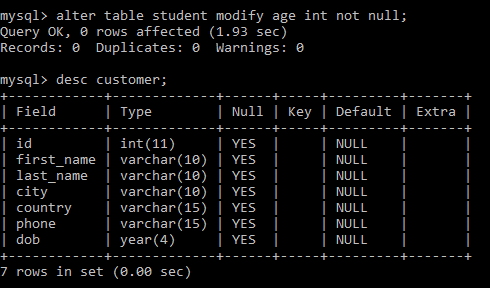
**SQL NOT NULL on ALTER table Statement**

To make a NOT NULL constraint on the "age" column when the "student" table is already created, use the following SQL:

**Example:**

ALTER TABLE student

MODIFY age int NOT NULL;



In the above table, it has specified the id, first\_name,last\_name, and age as not null.

## SQL UNIQUE CONSTRAINT

The **UNIQUE** constraint in SQL ensures that all values in a column are distinct. UNIQUE and PRIMARY KEY constraints both provides a guarantee for **uniqueness** for a column or group of columns. A PRIMARY KEY constraint, by default, has a UNIQUE constraint. However, the user can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.

**Creates UNIQUE constraint on the "id" column when the "person" table is created**

CREATE TABLE person (

id int NOT NULL,

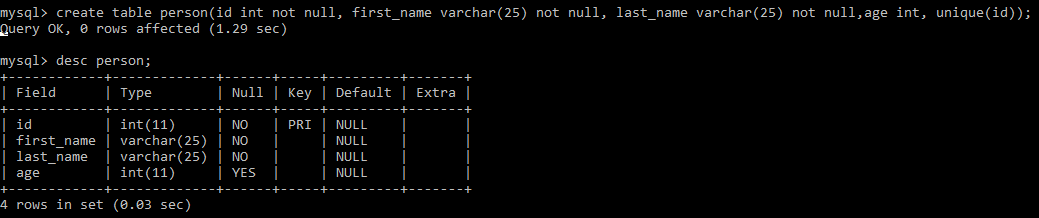
last\_name varchar(255) NOT NULL,

first\_name varchar(255),

age int,

UNIQUE (ID)

);

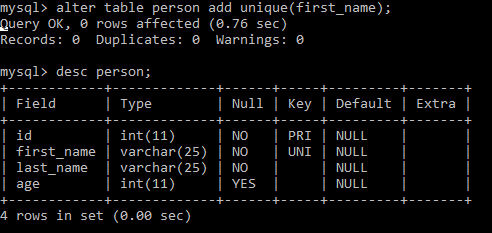


We have applied unique constraints on id, and as we can see, it is showing as the primary key.

**Create a UNIQUE constraint on the "first\_name" column when the "persons" table already exists.**

ALTER TABLE persons

ADD UNIQUE (first\_name);

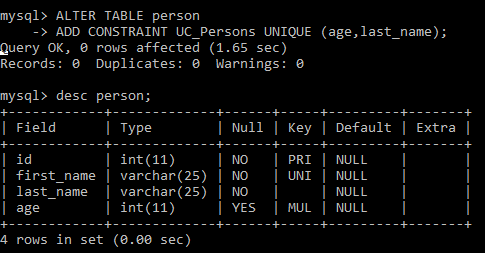


Now we have two unique constraints(id and first\_name) in the person table.

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

ALTER TABLE person

ADD CONSTRAINT UC\_person UNIQUE (age, last\_name);



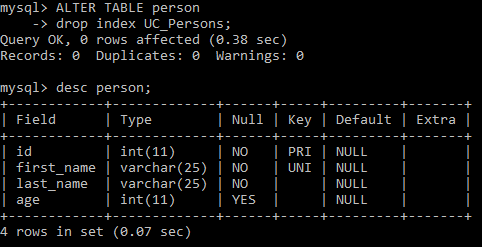
Here the age and last\_name are converted as unique constraints.

## DROP A UNIQUE CONSTRAINT

To drop a UNIQUE constraint, use the SQL query

ALTER TABLE person

DROP INDEX UC\_Person;

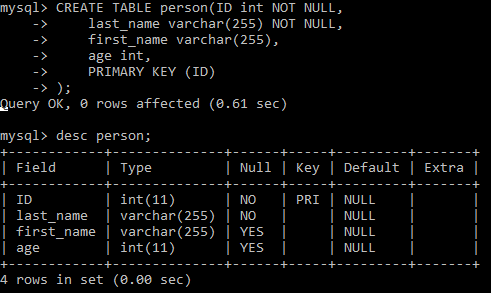


As we can see in the person table The unique constraint(UC\_Persons) has been dropped.

## SQL PRIMARY KEY CONSTRAINTS

The PRIMARY KEY constraint uniquely identifies each of the records in a table. Only ONE primary key can have in a table. And also, in the table, this primary key can consist of single or multiple columns (fields). Primary keys should contain UNIQUE values, and cannot contain **NULL** values.

CREATE TABLE person(ID int NOT NULL, last\_name varchar(255) NOT NULL, first\_name varchar(255), age int, PRIMARY KEY(ID));



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

CREATE TABLE person (

id int NOT NULL,

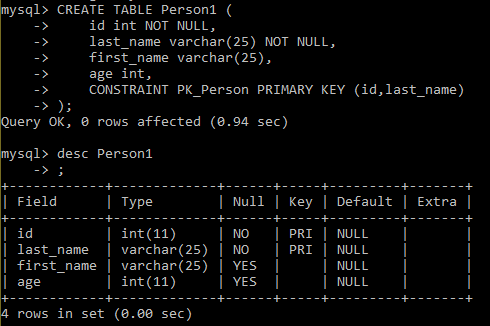
last\_name varchar(255) NOT NULL,

first\_name varchar(255),

age int,

CONSTRAINT PK\_person PRIMARY KEY (id,last\_name)

);



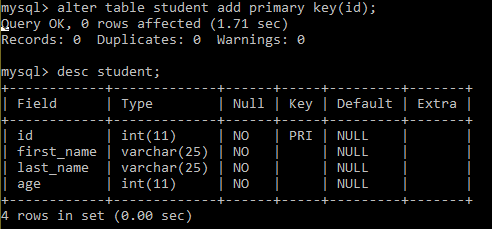
**Note**: In this example, there is only ONE PRIMARY KEY as PK\_Person. And the VALUE of the primary key is made up of **two columns** (id+ last\_name).

**SQL PRIMARY KEY on ALTER TABLE**

Create a PRIMARY KEY constraint on the column\_name "**id**" when the table\_name(student) is already created, use the following SQL:

ALTER TABLE student

ADD PRIMARY KEY (id);

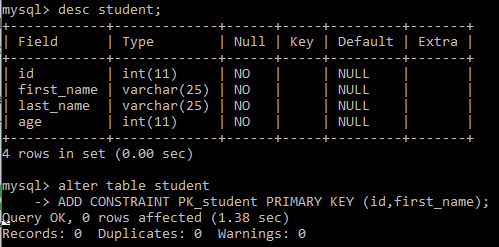


Here we have assigned the primary key as “id” on the student table.

Allow the naming of a PRIMARY KEY constraint, and for defining a PRIMARY KEY constraint on multiple columns, use the SQL query:

ALTER TABLE student

ADD CONSTRAINT PK\_student PRIMARY KEY (id,first\_name);

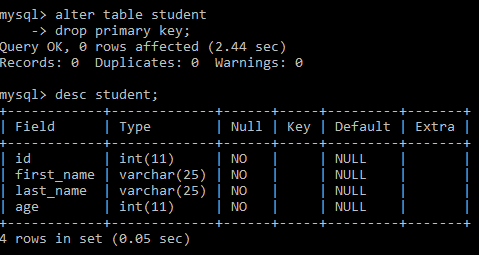


## DROP PRIMARY KEY CONSTRAINTS

To drop the PRIMARY KEY constraint from the table, use the SQL Query:

ALTER TABLE student

DROP PRIMARY KEY;



As we can see from the student table, the primary key has been dropped from the table.

## SQL FOREIGN KEY CONSTRAINT

A FOREIGN KEY is used to link two tables together. It is sometimes also called a referencing key. Foreign Key is a combination of columns (can be single column) whose value matches a Primary Key in the different tables. The relationship between two tables matches the Primary Key in one of the tables with a Foreign Key in the second table. If the table contains a primary key defined on any field, then the user should not have two records having the equal value of that field.

Let’s create two tables using the foreign key.

**CUSTOMER table**

CREATE TABLE customer(

Id int NOT NULL,

Name varchar(20) NOT NULL,

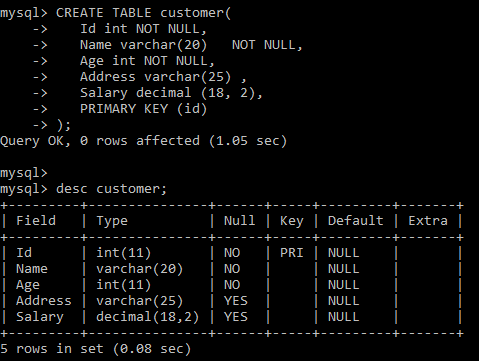
Age int NOT NULL,

Address varchar(25) ,

Salary decimal (18, 2),

PRIMARY KEY (id)

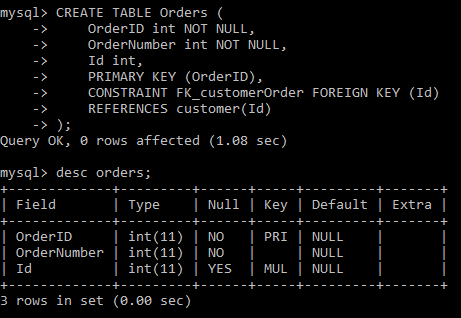
);



**Order Table with Foreign key**

CREATE TABLE Orders (OrderID int NOT NULL, OrderNumber int NOT NULL, Id int,

PRIMARY KEY(OrderID), CONSTRAINT FK\_customerOrder FOREIGN KEY(Id));



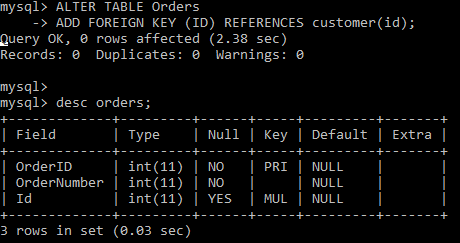
Here the Id is the primary key for the customer table and foreign key for orders table.

**FOREIGN KEY on ALTER TABLE**

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

ALTER TABLE Orders

ADD FOREIGN KEY (ID) REFERENCES customer(id);

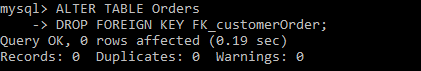


## DROP A FOREIGN KEY CONSTRAINT

To drop a FOREIGN KEY constraint from the table, use the SQL query:

ALTER TABLE Orders

DROP FOREIGN KEY FK\_PersonOrder;



## SQL CHECK CONSTRAINTS

The CHECK CONSTRAINTS is used to limit the range of value that can be placed in a column if the user defines a CHECK constraint on a single column, it allows only specific values for the column. If the user defines a CHECK constraint on a table, it can limit the values in particular columns based on values in another column in the row.

**SQL CHECK on CREATE TABLE**

SQL Query to creates a CHECK constraint on the column "Age" when the table "Persons" is created. The CHECK constraint makes sure that the user can not have any person below 18 years:

CREATE TABLE Persons (

ID int NOT NULL,

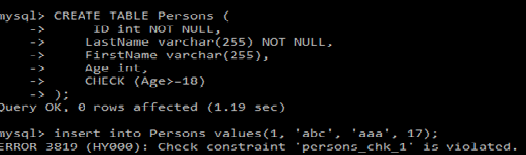
LastName varchar(255) NOT NULL,

FirstName varchar(255),

Age int,

CHECK (Age>=18)

);

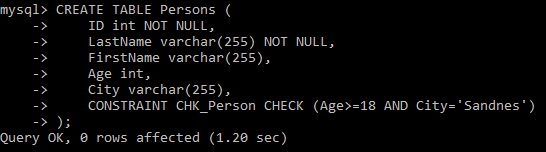


Here we have created the Persons table and given a check constraint on the Age column. If the Age<18, then it will throw an error, as shown below.

INSERT INTO Persons VALUES(1, ‘abc’, ‘aaa’, 17);



For creating a CHECK constraint on multiple columns in the table, use the SQL syntax:

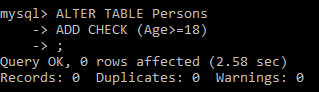


**CHECK on ALTER TABLE**

**C**reate a CHECK constraint on the column "Age" when the table is already created, use the following SQL:

ALTER TABLE Persons

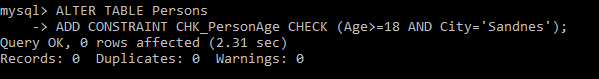
ADD CHECK (Age >= 18)



Defining CHECK constraint on multiple columns of a table, use the SQL query:

ALTER TABLE Persons

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

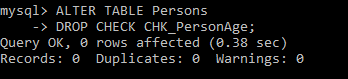


## DROP A CHECK CONSTRAINT

To drop a CHECK constraint from the table, use the following SQL:

ALTER TABLE Persons

DROP CHECK CHK\_PersonAge;



Here we have dropped the CHK\_PersonAge constraints by using the drop statement.

## SQL DEFAULT CONSTRAINT

The DEFAULT constraint in SQL is used to provide a default value for a column of the table. The default value will be added to every new record if no other value is mentioned.

**SQL DEFAULT on CREATE TABLE**

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

CREATE TABLE Persons (

ID int NOT NULL,

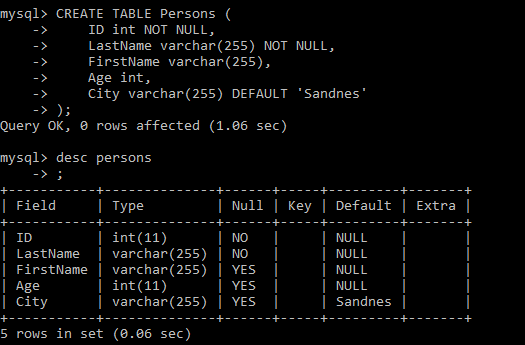
LastName varchar(255) NOT NULL,

FirstName varchar(255),

Age int,

City varchar(255) DEFAULT 'Sandnes'

);



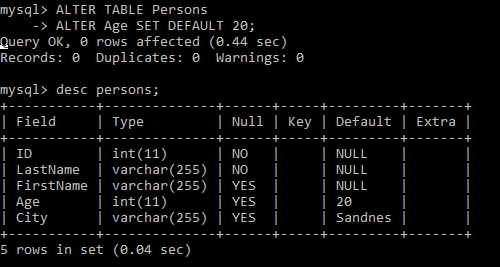
As we can see in the Persons table, the city name is written as Sandnes by Default.

**SQL DEFAULT on ALTER TABLE**

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

ALTER TABLE Persons

ALTER Age SET DEFAULT 20;

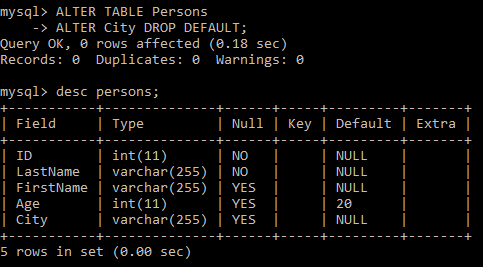


## DROP A DEFAULT CONSTRAINT

To drop a DEFAULT constraint from the table, use the SQL query:

ALTER TABLE Persons

ALTER City DROP DEFAULT;



As we can see in the Persons table, the default value of the city has been removed.

# SQL CREATE INDEX STATEMENT

CREATE INDEX statement in SQL is used to create indexes in tables. The indexes are used to retrieve data from the database more quickly than others. The user can not see the indexes, and they are just used to speed up queries /searches.

**Note**: Updating the table with indexes takes a lot of time than updating a table without indexes. It is because the indexes also need an update. So, only create indexes on those columns that will be frequently searched against.

**CREATE INDEX Syntax**

It creates an index on a table. Duplicate values are allowed:

CREATE INDEX index\_name

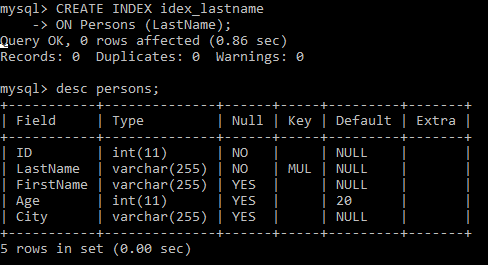
ON table\_name (column1, column2, ...);

**Example**:

Creates an index named "idex\_lastname" on the "LastName" column in the "Persons" table:

CREATE INDEX idex\_lastname

on Persons (LastName)



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

CREATE INDEX idex\_pname

ON Persons (LastName, FirstName);

**CREATE UNIQUE INDEX**

It creates a unique index on a table and Duplicate values are not allowed.

**Syntax:**

Create UNIQUE INDEX index\_name

on table\_name (column1, column2, ...);

**Note**: The query for creating indexes varies among different databases. Therefore, Check the query for creating indexes in your database.

## DROP INDEX STATEMENT

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

ALTER TABLE table\_name

DROP INDEX index\_name;

# SQL VIEWS STATEMENT

In SQL, the view is a virtual table based on the result-set of an SQL statement. A view holds rows and columns, similar to a real table. The fields in a view are fields from one or more real tables in the database. You can add SQL functions, WHERE, and JOIN statements to a view and present the data as if the data were coming from one single table.

**CREATE VIEW Syntax**

CREATE VIEW view\_name AS

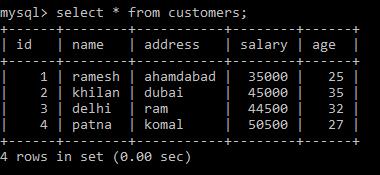
SELECT column1, column2, ...

FROM table\_name

WHERE condition;

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

**Create a table customer**

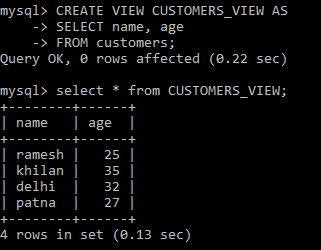


Create a view on the table **customers**. Here, the view would be used to have a customer name and age from the **customers** table.

CREATE VIEW CUSTOMERS\_VIEW AS

SELECT name, age

FROM customers;



## The WITH CHECK OPTION

The **WITH CHECK OPTION** in SQL is a CREATE VIEW statement option. The objective of the WITH CHECK OPTION is to make sure that all UPDATE and INSERTs satisfy the condition(s) in the view definition.

If they do not satisfy the condition(s), the UPDATE or INSERT returns an error.

The following code block has an example of creating the same view CUSTOMERS\_VIEW with the WITH CHECK OPTION.

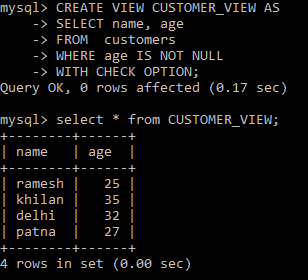
CREATE VIEW CUSTOMER\_VIEW AS

SELECT name, age

FROM customers

WHERE age IS NOT NULL

WITH CHECK OPTION;



Here we have created a view(CUSTOMER\_VIEW) **with the check** option.

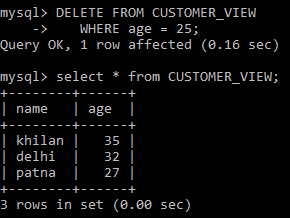
## DELETING ROWS INTO A VIEW

Rows of data can be deleted from a view. The same rules that apply to the UPDATE and INSERT commands apply to the DELETE command.

**Example** Delete a record having AGE = 25.

DELETE FROM CUSTOMER\_VIEW

WHERE age = 25;

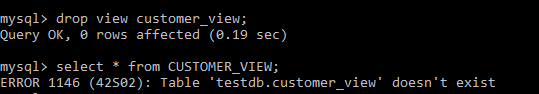


Here we have deleted the row, which contains the age = 25.

## DROPPING VIEWS

Where the user has a view, you need a method to drop the view if it is no longer needed. The query is straightforward and is given below:

DROP VIEW view\_name;



It’s similar to the other dropping option, as we have done yet for tables. As we can see, the view is not available in the database after dropping the view.

# STORED PROCEDURE AND FUNCTIONS

Advance MySQL provides better understanding for Stored Procedure, View, Triggers, Events and Indexes. In this chapter, we are going to understand all of the above terminology one by one in details with the help of MySQL workbench.

## MySQL Stored Procedure

**What is a SQL Stored Procedure?**

The **stored procedure** is a prepared SQL query that you can save so that the query can be **reused** over and over again. So, if the user has an SQL query that you write over and over again, keep it as a stored procedure and execute it. Users can also pass parameters to a stored procedure so that the stored procedure can act based on the parameter value that is given.

### Creating the Stored Procedure

**Syntax for creating a Stored Procedure**

DELIMITER $$

CREATE PROCEDURE PROCEDURE\_NAME()

BEGIN

SELECT Column\_name1, Column\_name2,………..

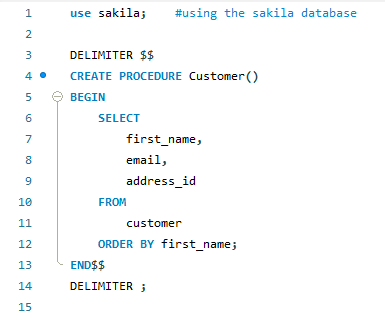
FROM Table\_name

END$$

DELIMITER

Here, the DELIMITER is not the part of Query, the first Delimiter is change the default delimiter to **//** and the second delimiter is change the delimiter to the default. The Stored procedure is saved automatically in the database while creation.

To execute the query in MySQL, use the MySQL workbench for better user-interface, and use inbuilt databases to perform the advance MySQL queries.



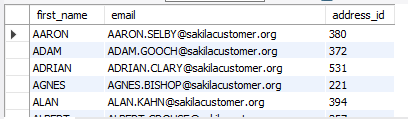
Here we have create a procedure called **Customer,** and we have mentioned few column names in it. And in last we have closed the procdure. If we want to know the output of the above query, then need to run the procedure by clicking on the execution button on workbench display.

### EXECUTION OF STORE PROCEDURE

Execution of the Stored Procedure is very simple by using the **CALL procedure\_name,** Execute the below query to get the result of the defined stored procedure.



After calling the procedure, we are able to see the selected columns which are mentioned in the procedure. The output is as follow:



Stored Procedue can have parameters,so while execution we can pass the argument and get the result. We can use control flow (like: IF, LOOP, CASE, etc.)in the stored procedure to make dynamic queries and also we can pass one stored procedure inside the other which will help to modulize the queries.

### DROP THE STORED PROCEDURE

Drop procedure use to delete the stored procedure from the databases. The following query used to delete the stored procedure for the Database:

DROP Stored\_procedure\_name





The below syntax used for conditionally drop of stored\_procedure and first it check the procedure\_name & if it exist then drop the stored procedure from the database.

DROP PROCEDURE [IF EXIST] Stored\_procedure\_name;



If the stored procedure is not available then it throw an error like mentioned below.



### STORED PROCEDURE PARAMETERS

We can create a stored procedure with parameters. In Stored procedure the parameters are like IN, OUT and INPUT. The parameters make the Stored Procedure more flexible and useful.

#### DEFINING A PARAMETERS

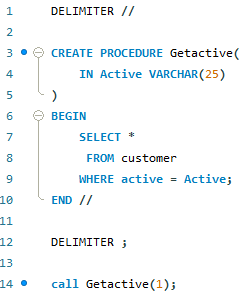
To define the parameter inside the stored procedure, run the below query:

[ IN | OUT | INPUT ] PARAMETER\_NAME datatype[(length)]

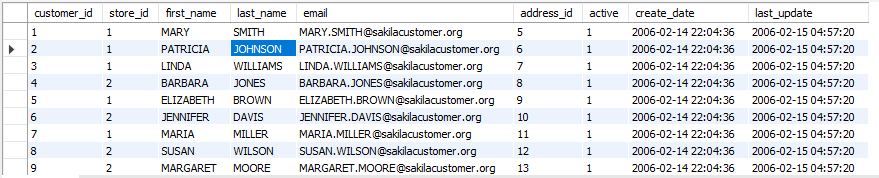
#### IN Parameter

It is the default parameter in Stored Procedure and the calling program should pass an argument to stored Procedure. The value of **IN** is protected that means even the **IN** value is changed inside the stored procedure the original value will retained after end of the Stored Procedure.

**Example for *IN:*** Create a Stored Procedure that find all the active customers by the input parameter **as Active.**



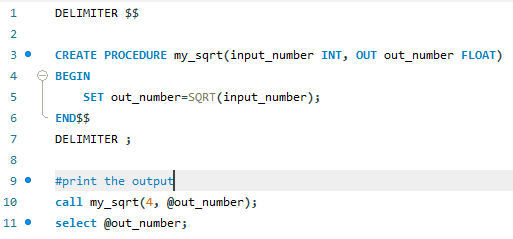
Output:



#### Out Parameter

The value of the Output Parameter can be changed inside the Store Procedure and pass the new value while calling the Stored Procedure.

**Example for *OUT:*** write a stored procedure to print the square root of a number.



Output:

The my\_sqrt stored procedure has two parameters.

**input\_number** : it takes input from user in interger format.

**out\_number** : it store the output of the function.

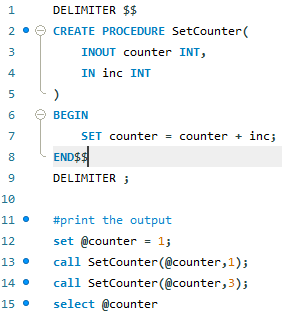
To print the output, we have call the my\_sqrt() and pass the arguments, the first argument is user input and second is output, and to show the output use select @out\_number.



#### INOUT

It is the combination of the IN and OUT Parameters.

**Example for *INOUT* :** Here we are just counting the numbers between a region using **INOUT** stored procedure.



Output:

It will print the query in sequence so if we call the stored procedure for many time,so it will count all the sum and in last it will print the counter values.



### STORED PROCEDURE VARIABLES

In this unit we will learn about the variables, and also how to declare variables? How to use the variables?. Basically a variable is a called as data object whose value can be change while execution of Stored Procedure.

#### DECLARING THE VARIABLE

To declare the variable inside a stored procedure, use the below query:

DECLARE Variable\_name datatype(size) [DEFAULT Default\_value];

Here,

DECLARE – It is a keyword and it is use to declare the variable. First write DECLARE keyword and then variable\_name.

Datatype(size) – it us use to define the variable datatype (like: IN, Varchar, or char)and size use to define the length of the variable.

Default – it assign variable with default value option. If we declare the variable without specifying any default values, then it’s values will be NULL.



Using MySQL Stored Procedure, we can declare more than one variable.

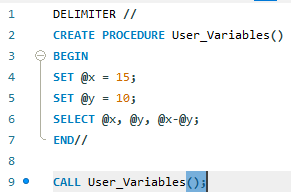


#### ASSIGNING VARIABLE

Once we declare the variable, now it is ready to use. To assign a value to the variable use **SET** statement:



The value of the variable Total is assigned as 10. We can declare and create the variable in stored procedure as follow:



Here we have assigned two variables and using under stored procedure to print the difference between two numbers(x and y). Let’s call the stored Procedure to check the output of the above code. And the output is below:



#### VARIABLE SCOPE

Variable scope is for limited time period. It is defined inside the stored procedure within **BEGIN** and it will be out of scope once the **END** statement reaches.

When we declare any variable inside the **BEGIN END** Statement, it will be out of scope once the **END** statement reached, there after we cannot use it.

## CONDITIONAL STATEMENT

In this unit we will learn about the IF statement in MySQL and we will also learn about how to write the Conditional-Statement in MySQL.

In MySQL, Conditional- Statement has three forms: **IF-THEN**, **IF-THEN-ELSE** and **IF-THEN-ELSEIF-ELSE.** Here we are going to learn about conditional statement in details one by one.

### IF-THEN STATEMENT

**IF-THEN** statement allow user to execute the block of SQL Query based on the specific condition. Here is the syntax for IF-THEN:

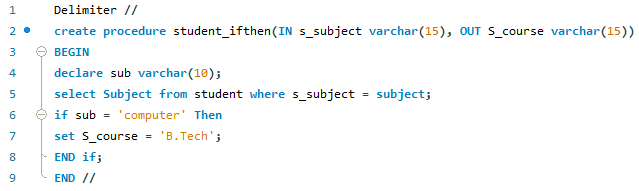
IF condition THEN

Statement;

END IF;

Here,

First, it will check the condition to execute the statement between the **IF-THEN** and **END IF** andif the condition is TRUE, otherwise it will go to the next END IF.



In the above query, we are checking the condition as if the subject is as ‘computer’ then set the course as ‘B.Tech’. Let’s check the output of the stored procedure ‘**student\_ifthen’.**



As we can see here, the course is coming as B.Tech, because we set the condition as like that.



### IF-THEN-ELSE STATEMENT

**IF-THEN-ELSE** is similar to the **IF-THEN.** Where we will execute a block of code on a particular condition and if the condition will not satisfy then it will go for **else** block. The syntax as follow:

IF Condition THEN;

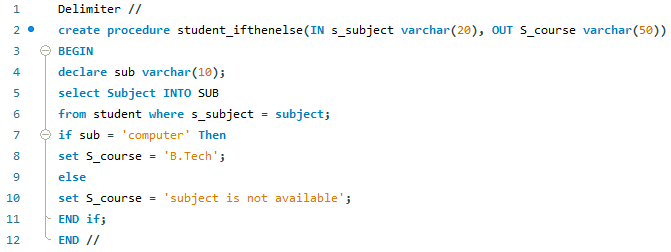
Statement;

ELSE

Else-Statement;

END IF;

Here we are going to check the subject name as ‘computer’ and if the condition is not satisfied then the else part will execute “subject is not available”.



So ler’s call the procedure to check the output of the above query.





As we can see here, we have given the subject name as ‘History’, so that the else part is executed and given output as the ‘subject is not available’.

### IF THEN ELSEIF ELSE STATEMENT

**IF-THEN-ELSE** is similar to the **IF-THEN.** Where we will execute a block of code on a particular condition in **IF block** and if the condition will not satisfy then it will go for **ELSEIF** block and again if the condition is not satisfied then it will execute the **else** block. The syntax as follow:

IF Condition THEN;

IF-Statement;

ELSEIF

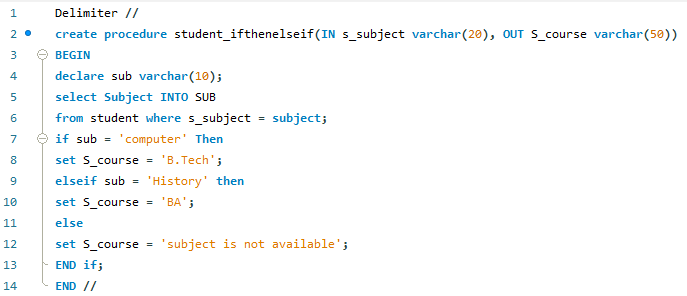
ElSEIF-Statement;

ELSE

ElSE-statement

END IF;

Now we are going to perform the same operation as above, where we are going to check the subject name in **if** block as ‘computer’ and if the condition is not satisfied then the **else-if** part will execute and if the condition is not satisfied then the **else** block will get executed.



So ler’s call the procedure to check the output of the above query.



The else-if block is get executed and print the course name as **BA.**



let’s call the else part,





The conditional statement explained here one-by-one. We saw the three conditional statements as IF-THEN, IF-THEN-ELSE, IF-THEN-ELSEIF-ELSE.

## CASE STATEMENT

In this unit, we are going to learn about the **CASE statement**, it is an alternative conditional statement for the IF-Statement and CASE statement makes the code more efficient and readable. CASE statement has two forms: **Simple CASE** and **Searched CASE**.

So, let’s learn about the CASE statement and their use using the Stored Procedure.

### Simple CASE Statement

The simple CASE statement sequentially compare the case\_values in with the when\_values until it finds as equal. The basic syntax for the Simple CASE statement:

CASE case\_value

When when\_values THEN statement

…..

[ELSE else-statements]

END CASE;

If the case\_values will not be equal to the when\_values then it will execute the else statement. And if the else is also not satisfied then it will throw an error as CASE not found for CASE Statement.

To avoid the error when the case\_value will not equal to when\_values then we can use an empty **BEGIN END** block in the else block as follows:

CASE case\_value

When when\_values THEN statement

…..

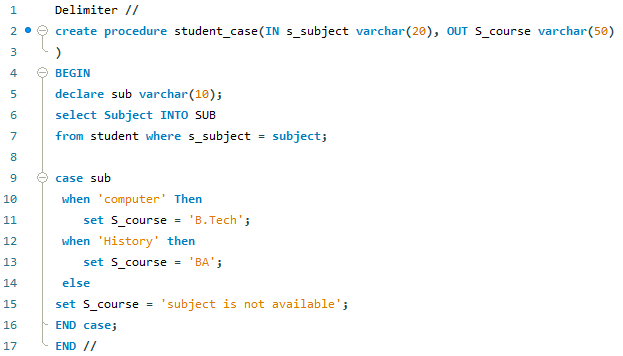
[ELSE else-statements]

BEGIN

END;

END CASE;

Here we are going to do same operation as IF conditional statement. We are going to check the subjects are lying under which course using the CASE statement.



Here instead of using IF-ELSE we have used the CASE Statement, under that we are checking the condition as when “condition” is true then set values. If the condition is not satisfied under the CASE statement then the ELSE block will get executed.

Let’s run the query and call the Stored Procedure. Now we will print the Case statement as follow:





And also,





Now, Let’s see the else block, if the case\_values are not satisfied then else block will executed.





As we can see, the else part is executed ans showing “subject is not available”.

### Searched CASE Statement

Searched CASE Statement is similar to Simple Case Statement but it only allows you to compare a value with a set of distinct values. It is equivalent to the IF Statement but it is more readable than IF Statement. To perform the more complex matches (like ranges), we use the Searched CASE Statement.

The syntax is as follow:

CASE case\_value

When when\_values THEN statement

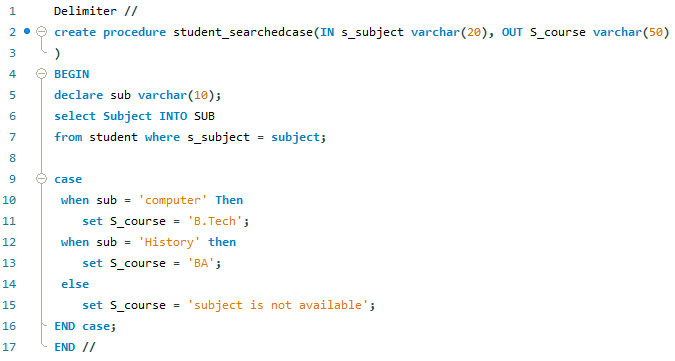
……..

[ELSE else-statements]

END CASE;

The searched **CASE** statement check each **Seach\_condition** inside the WHEN clause until it finds as TRUE then it will execute the corresponding THEN statement. If the **search\_condition** is not satisfied then the **CASE** evaluates the **ELSE** Statement.

We are going to search the subject names and checking them that under which course they existing. If the condition is true then it will execute and return the course name but if the condition is not satisfied then it will execute the else block.



Here we are searching, when the subject name as ‘computer’ then return course as ‘B.Tech’. And if the condition is not satisfied then it will execute the else block and return as ‘subject is not available’.

Let’s call the Stored Procedure & check the satisfied condition.



Here we have passed the subject name as ‘computer’ and it is belongs in B.Tech. Let’s have a look on Output.



Similarly, let’s execute the **else** part:



As we can see here, it executed the else part of the Stored Procedure. And showing output as the ‘subject is not available’.



#### CASE Vs IF STATEMENT

IF and CASE are allow you to execute the block of code on specific condition. We can use both IF and CASE statement inside stored procedure, it’s completely depends on our choice.

* Simple CASE statement is more efficient and readable than the IF Statement while comparing a single expression.
* IF statement if better when we are executing complex expressions.
* If we are using CASE statement, then make sure that at least one condition should be satisfied otherwise we need to add error handler inside the stored procedure.

## LOOP STATEMENT

In MySQL, the **LOOP** statement is used to execute one or more than one statement repeatedly.

The syntax for LOOP statement:

[begin loop: ] loop

Statement\_list

END LOOP [end\_label]

The loop executes the **statement\_list** repeatedly one by one and the statement\_list can be one or more and separated by the **semicolon (;)**. To terminate the loop we use the **LEAVE** statement after the condition is successfully satisfied.

The syntax for LOOP Statement with LEAVE Statement:

[label]: LOOP

IF Condition THEN

LEAVE[label];

END IF;

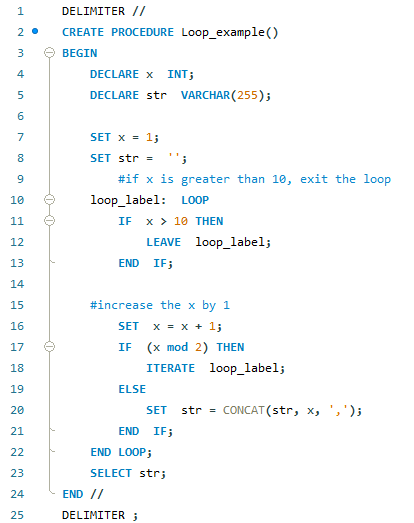
END LOOP;

The **LEAVE** statement is exactly work as the **break** in other programming language. It will immediately exit from the loop.

We are going to use the LOOP inside the Stored Procedure to print the even numbers from 1 to 10. To print the even number we need to implement two conditions:

* The number should be less than 10.
* The number should be divisible by 2.

So let’s implement it using SQL query inside the Stored Procedure using the LOOP and IF-THEN Statement. The query is written below to print the even numbers from the 1 to 10.



Let’s call the stored procedure to check the result.





As we can see, the output is the collection of the even numbers between 1 and 10.

## WHILE LOOP STATEMENT

WHILE loop is execute the block of codes until the condition TRUE. The syntax is written below:

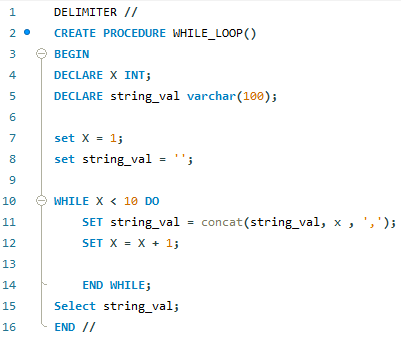
[begin loop:] WHILE condition DO

Statement

END WHILE [end\_label]

The WHILE statement will check the condition at the beginning of the each iteration and if the condition satisfied, then it execute the Statement until the condition is true. We can have one or more than one statements inside the DO and END WHILE.

We are going to print the numbers from 1 to 10 using while loop inside the stored procedure.



Let’s call the stored procedure and check the output of the above query.





We can see here, the output of the query is the collection of the numbers from 1 to 10 (excluding 10). Because we have given the condition in a while loop as <10, so till that, it will print all the numbers from 1 to till 9.

## REPEAT LOOP STATEMENT

REPEAT statement is used to execute one or more statements until the condition satisfied. It is similar to the DO WHILE LOOP in C. The syntax for the REPEAT loop statement as follows:

REPEAT

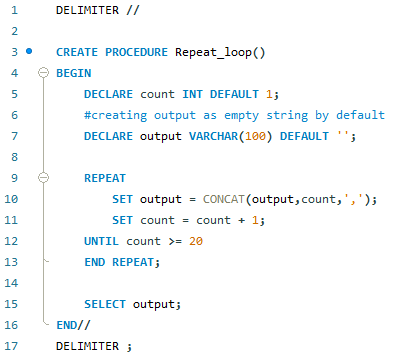
Statement

UNTIL condition

END REPEAT

Here, the REPEAT LOOP executes the statement before checking the condition. Therefore the statement will always execute at least once. It is also known as the post-test loop.

Let’s print the numbers from 1 to 20(excluding 20) by using a REPEAT loop inside the stored procedure.



Let's check the output of the stored procedure by calling it.





We have successfully printed the numbers from 1 to 20 by using the Repeat loop inside the stored procedure.

## CURSOR

MySQL cursor in Stored Procedure is used to iterate through a result set returned by a select statement. We use the cursor to handle a result set inside a stored procedure. A cursor allows you to iterate a set of rows returned by a select query and process through each row separately.

MySQL cursor is read-only, Non-scrollable, and Asensitive.

Syntax to write the cursor in the stored procedure:

* 1. Declare Cursor\_name CURSOR from SELECT\_Statement.

The cursor is declared after the variable declaration; if you say before, then it will throw an error.

* 1. OPEN Cursor\_name.

We are open the cursor by using the OPEN statement and also OPEN initialize the result set for the cursor.

* 1. Declare CONTINUE HANDLER FOR NOT FOUND (Termination Statement)

To declare not found a handler, we use the above query. The finished is a variable that indicates the cursor has reached the end of the result list.

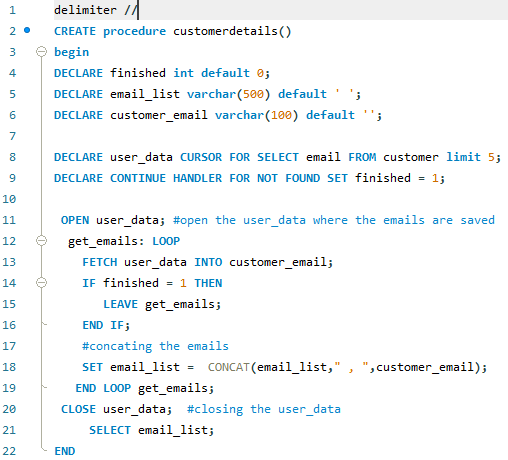
* 1. FETCH cursor name INTO variable\_list.

By using the FETCH statement, retrieve the next row pointed by the cursor and move to the next row in the result list.

* 1. Close cursor\_name.

By using the CLOSE, we deactivate the cursor and release the memory associated with it.

Let’s understand cursor by implementing it on the tables. So, we are going to get emails from customers from the customer table. To implement it, we are going to use the above five steps.



Let’s check the output by calling the stored procedure



In the output set, we have finally extracted the top 5 emails and stored them in the email\_list by using the cursor inside the stored procedure.



## ERROR HANDLING

MySQL ERROR HANDLING uses to encounter Errors in the stored procedure. Whenever any error occurs inside a stored procedure, it is very important to handle it. To handle that, MySQL is providing an easy way to define handlers that handle the errors (such as warnings or exceptions to specific conditions).

To declare the handler, we use the following syntax:

DECLARE action HANDLER FOR condition statement;

If the condition matches, then the MySQL will execute the statement and continue or exit from the code block based on the action.

Action accepts one of the following values:

**CONTINUE**: the execution of the code block is continuing.

**EXIT**: the execution of the enclosing code block, where the handler is declared or terminated.

**Declaring the Error Handling for CONTINUE**

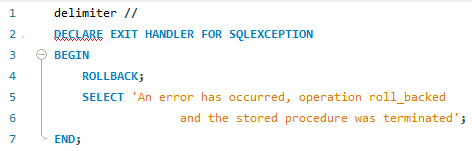
The following handler set the value of error variable equal to 1 and continues the execution if any error occurs;

DECLARE CONTINUE HANDLER FOR SQLEXCEEPTION

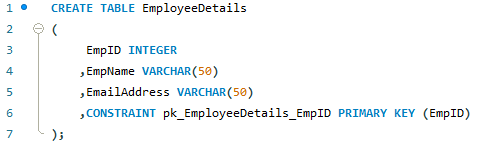
SET error\_variable = 1

**Declaring the Error Handling for EXIT**

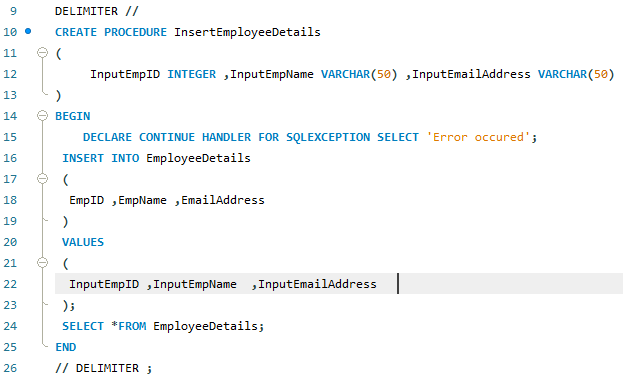
The following handler rolls back the previous operation, issues an error message and exit the current code block in case of error occurs. If we declare it inside the BEGIN END block of a stored procedure, it will terminate the stored procedure immediately.



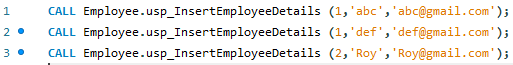
Let’s understand with an example of the error handler; so we are creating an employeedetails table and we are trying to pass the error handler (continue and exit).



We have created a table with EmployeeDetails and given a constraint as pk\_EmployeeDetails with primary key (EmpID). Let’s create a procedure to handle to continue error handler.

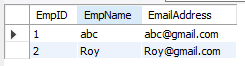


Here we have created a procedure, that will detect the duplicasy in data and throw an error message as error occurred. We are going to pass the values inside the table.

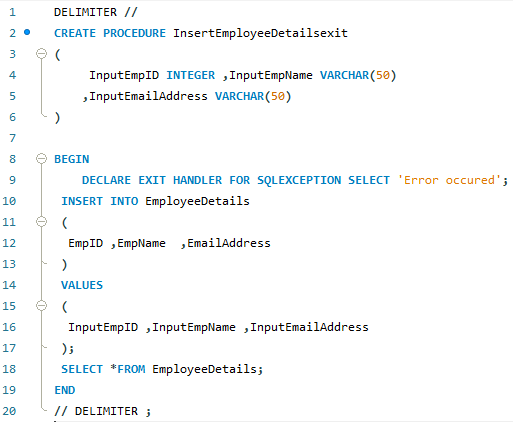




It is giving this message because one error is occurred while inserting the data into the table, and the handler is found the duplicate data. But along with this, it will print the remaining values which will not have any duplicate data.



We have inserted two rows successfully and third is not inserted because of duplicacy of data. As we have seen the CONTINUE error handler in the above example. Now let’s do an example with EXIT error handler using stored procedure. We are going to use the same table and same procedure except the error handler. In the below example, we are going to perform the error handler using the exit error handler, which is use to do the execution of the enclosing code block, where the handler is declared or terminated.



Here we are inserting the two rows as follow:



But the result of the error handler using the EXIT error handler will print the error message alone. It will not show the values which are inserted inside the table.



When we call the stored procedur in EXIT handler, it will just give the error message as above.

### ERROR CONDITIONS WITH MySQL SIGNAL/ RESIGNAL STATEMENT

**SIGNAL STATEMENT**

Signal statement is used to return an error or warning conditions to the caller using the stored procedure. It is provide and easy way to get the message and the values based on our need.

The syntax for SIGNAL statement:

SIGNAL SQLSTATE | condition\_name;

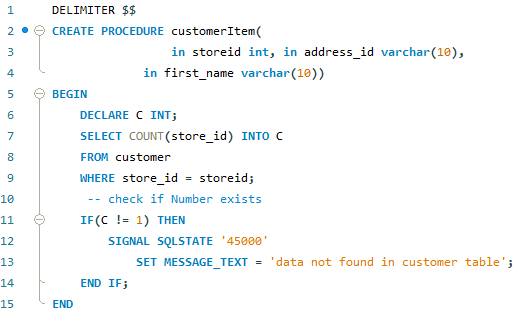
SET condition\_information\_item\_name = value1

SET condition\_information\_item\_name = value2

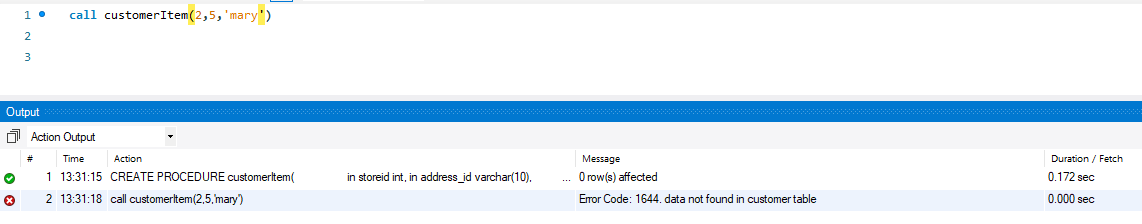
SIGNAL keyword is a SQLSTATE values or condition name declare by using DECLARE CONDITION. The condition\_information\_item\_name can beMESSAGE\_TEXT, MYSQL\_ERROR, CURSOR\_NAME etc.

Let’s create a stored procedure with customer table, where we are going to check the store\_id and customer details. If the record is not found then it will give an error message as the “data not found in custmer table”.

Note: **45000** is a generic **SQLSTATE** value that illustrates an unhandled user-defined exception.



Output:



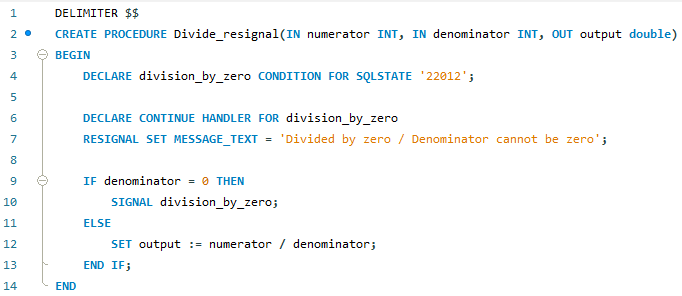
As we can see here, the error message is giving as the “data not found in the customer table”.

**RESIGNAL STATEMENT**

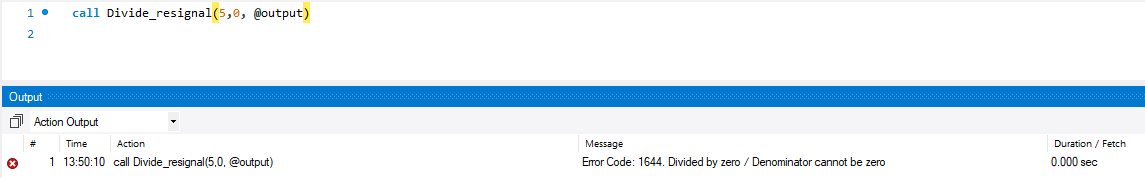
RESIGNAL STATEMENT is used to raise a warning or error messages. It is similar to the SIGNAL statement in terms of functionality and the syntax,except:

* We must have to use RESIGNAL statement inside the error handler, otherwise it will throw an error as “RESIGNAL when handler is not active”.
* It can omit all the attributes of RESIGNAL statement, even the SQLSTATE values.

We are creating a stored procedure to division of a number with any number, but if any number is divided by zero then it will throw an error message as the “divided by zero”.



Let’s call the stored procedure to check the output, when we divide any number with zero.



Here, we are dividing a number with zero, so it’s throwing an error message as “divided by zero/ denominator can not be zero”.

## STORED FUNCTION

STORED FUNCTION is a special type of the store program where we store the functions to encapsulate the common formulas and rule, and that are reusable.

### DECLARING STORED FUNCTION

DELIMITER $$

CREATE FUNCTION function\_name(

param1,

param2,…)

RETURNS datatype

[NOT] DETERMINISTIC

BEGIN

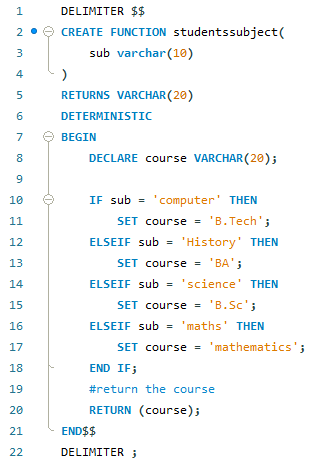
statements

END $$

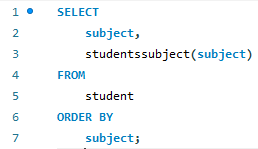
DELIMITER ;

The syntax is easy to understand. It is similar to other programming languages. First we use to create the function with the CREATE FUNCTION function\_name (pass the parameters in it). RETURN type is used to return the values/ statement. Then specify the function as DETERNMINISTIC OR NON-DETERMINISTIC. MySQL uses NON-DETERMINISTIC by default.

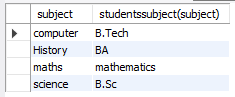
Now we are going to create a stored function on the student table. We are going to define if the subject\_name is coming then it should print as their respective course\_name.



Let’s check the function output by calling it under the SELECT statement.



We have selected the column as subject and applying the function at the subject. Let’s check the output of the function.



As we can see, the function is giving the subject with their respective courses.

### DROP FUNCTION

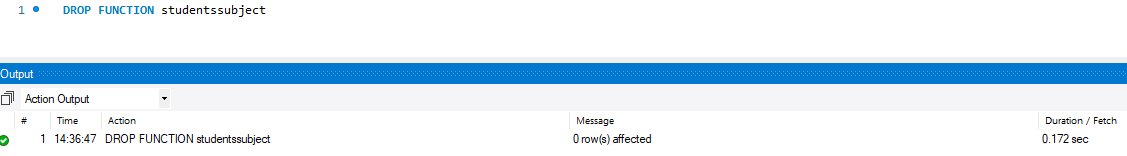
The DROP FUNCTION is use to drop the created stored function from the database. By using the Syntax as:

DROP FUNCTION function\_name;

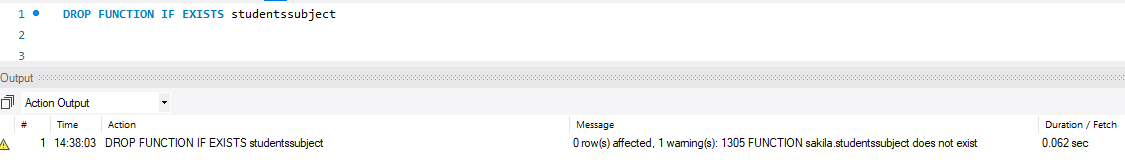
If the FUNCTION is not exit, In that case check the condition as:

DROP FUNCTION IF EXISTS function\_name;

Let’s do an example to show case the above function. For that, we are going to drop the studentssubject() function.



Now, let’s check the second command to drop the table.



It throwing a warning message as function does not exists. That means the function is already deleted.

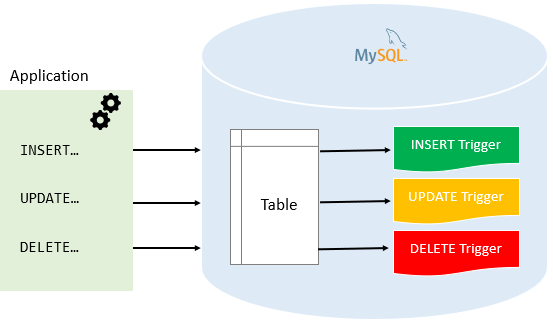
# TRIGGERS

Trigger is a stored program that invoked automatically in response to an event such as insert, delete or update that occurs in the table. Suppose, you defined a trigger and you insert a row inside the table, then it will automatically invoked before or after the insertion of row.

There are two types of the TRIGGERS:

1. **Row-level-Triggers**: it is activated for each row that is inserted, deleted or updated.
2. **Statement-level-Triggers**: it is executed for each transaction.

**Note**: It supports only **Row-level-Triggers**.



**Advantage of Triggers**

* It provides a way to check the integrity in data.
* It can be useful for auditing the data changes in tables
* It handles the errors from the database layer.

## CREATING TRIGGERS

CREATE TRIGGER statement is used to create the triggers. The syntax is following:

CREATE TRIGGER trigger\_name

{BEFORE | AFTER} {INSERT | DELETE | UPDATE }

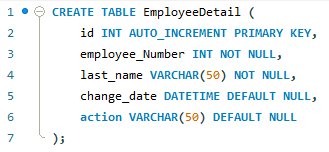
ON table\_name FOR EACH ROW

trigger\_body;

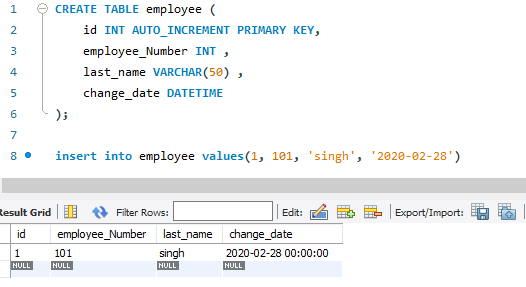
Here,

* The first line is for creating the trigger with trigger\_name.
* It will make the condition that the trigger invokes before or after any modification in row.
* The operation we can choose as INSERT, DELETE OR UPDATE on the table\_name at any row.

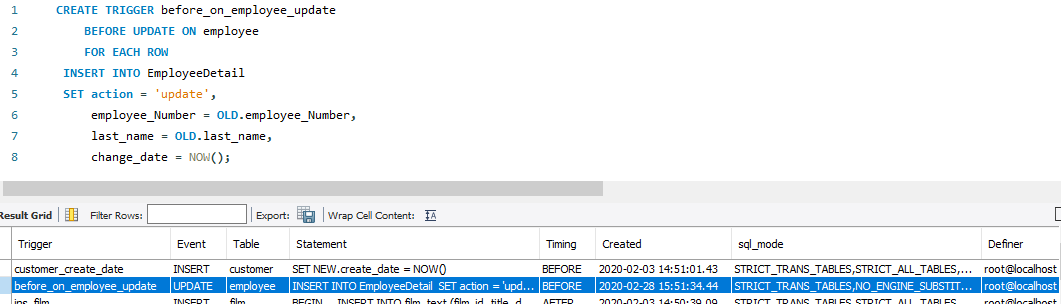
Let’s understand the triggers by using an example. Now, we are going to create a table names as EmployeeDetail. And defined a primary key as id;



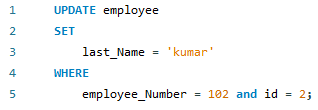
Create another table and insert some rows into that table, now we are going to create one more as “employee”. On the “employee” table, we are going to perform all the trigger operation on it and the operations log will be stored in “EmployeeDetail”.



Let’s create a trigger using the before update operation on employee table. As we can see, the trigger is created as name “before\_on\_employee\_update”.

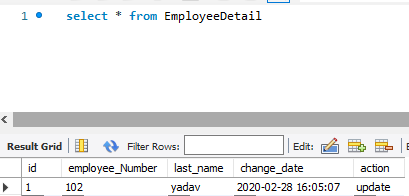


Now, let’s use the trigger and update the row values of employee,



Let’s check the table EmployeeDetail and check the action on it.

As we can see, the trigger is automatically invoked and inserted a new row inside the EmployeeDetail table and the row is updated.



## DROP TRIGGER

To delete the TRIGGER we use the DROP TRIGGER statement, and it will delete the trigger from the database. The syntax is as follow:

DROP TRIGGER [IF EXISTS] trigger\_name;

Here,

* Firstly, it will check the triggers\_name and if it exist then delete that particular trigger.
* To delete any trigger, the trigger\_name should be written after the DROP TRIGGER.

OR

DROP TRIGGER trigger\_name;

It will delete the trigger without checking their existence in the database.



The trigger **before\_on\_employee\_update** has been deleted from the database.

## BEFORE INSERT TRIGGER

The before insert trigger are automatically fired before an insert occurs on the table. The syntax for before insert trigger as follow:

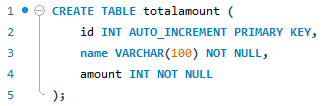
CREATE TRIGGER trigger\_name

BEFORE INSERT

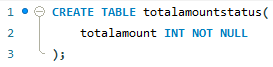
ON table\_name FOR EACH ROW

trigger\_body;

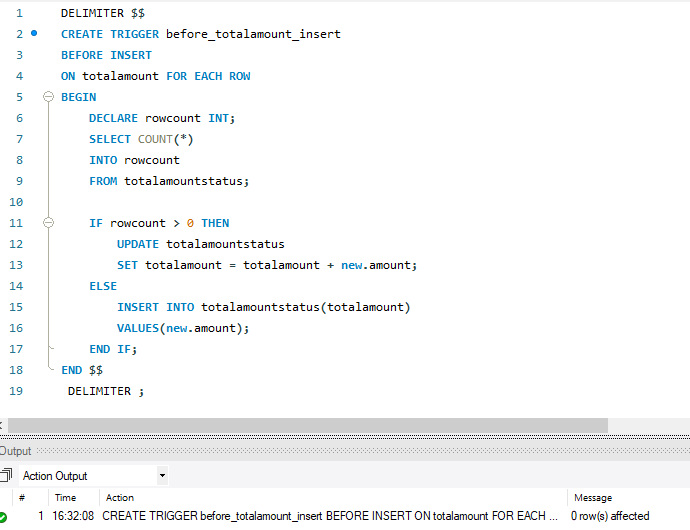
let’s understand through an example. We are creating a table as **totalamount;**



Now, let’s create another table as **totalamountstatus** to store the summary of the triggers.



Lets create a before insert trigger to get the totalamount in the **totalamountstatus** table before a new work center is inserted into the **totalamount** table.



The trigger is created successfully for updating before insert into the **totalamount** table. Let’s test the trigger by inserting the value in it.



We have successfully insderted the value in the totalamount table. But the value is invoked in the totalamountstatus table. Let’s call the **totalamountstatus** table to check the total amount.





The trigger is invoked and inserted a new row into the totalamountstatus. If we insert another value that will automatically added into the present amount and return the totalamount.

## AFTER INSERT TRIGGER

The after insert trigger are automatically fired after an insert occurs on the table. The syntax for after insert trigger as follow:

CREATE TRIGGER trigger\_name

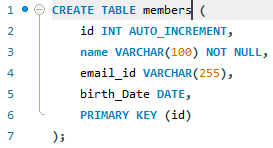
AFTER INSERT

ON table\_name FOR EACH ROW

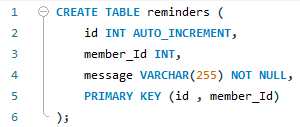
trigger\_body

Let’s understand the after insert trigger using an example;

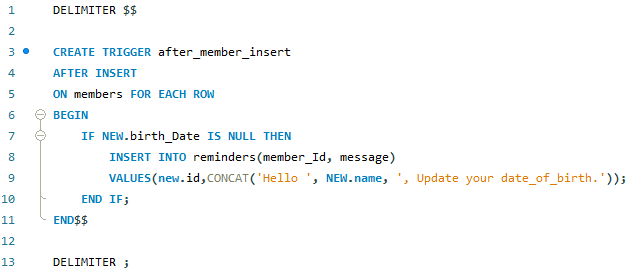
Create a table named as **members**.



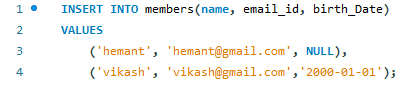
Create another table as **remembers.**



Now, create a after insert trigger as **after\_members\_insert** and that trigger insert into reminders table if the birth\_date of any person is null.



Let’s test the alter insert trigger.



We have inserted the two rows inside the members table and the members table is shown below;



As we can see here, the two rows are inserted but the birthdate of Hemant is null and as we mentioned the condition in trigger, it will invoke a message if birth date is as null. Let’s check the **reminders** table.



As we have made the condition inside the trigger, it has invoked automatically when the birth day found as null. And the message showing as Hello Hemant, update your date\_of\_birth.

## BEFORE UPDATE TRIGGER

The BEFORE UPDATE TRIGGER is invoked automatically before an update event occurs on the table which associated with the trigger.

CREATE TRIGGER trigger\_name

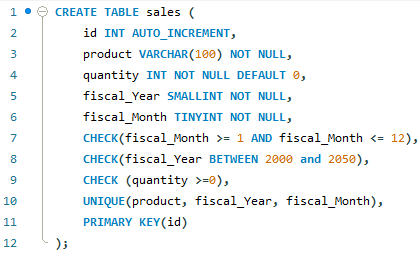
BEFORE UPDATE

ON table\_name FOR EACH ROW

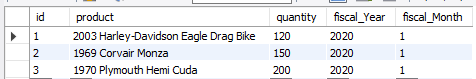
trigger\_body

Let’s understand through an example;

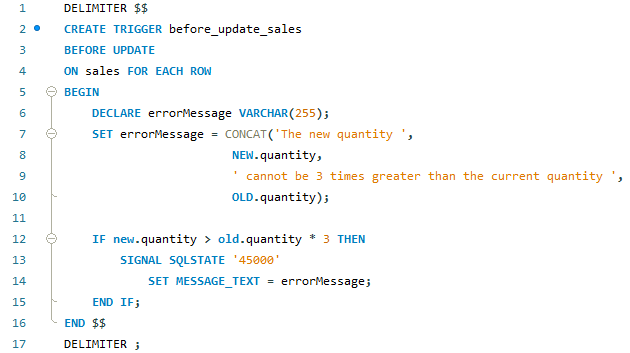
Create a table as sales;



Insert few rows into the sales table;



Creating the BEFORE UPDATE TRIGGER, and assigning the error message as the new quantity cannot be greater than 3-times of previous.

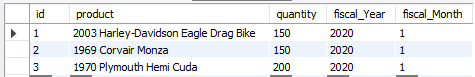


The trigger will automatically invoke and fire before updating any values in any row.

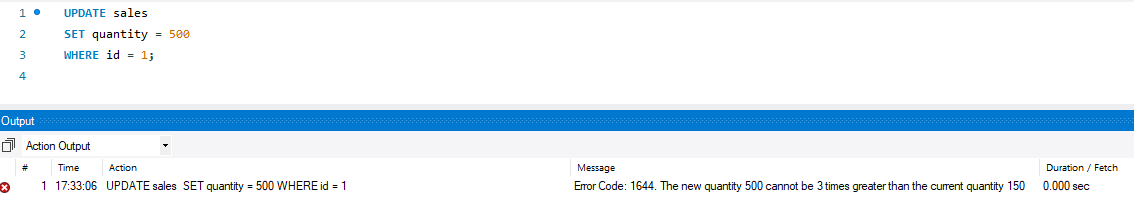
Let’s update the values in row of sales table;



We have updated a value of quantity where the id = 1 but it will not satisfied the condition so it will not give the error message, see the table;



Let’s update the quantity as some other value which are 3-times greater than the quantity150.



As we have increased the quantity as 3-times higher than previous, it’s showing message as “the new quantity cannot be 3times greater than the current quantity”.

## AFTER UPDATE TRIGGER

The ALTER UPDATE TRIGGER invoke automatically after updating the events in the associated table. The syntax for AFTER update triggers as follow:

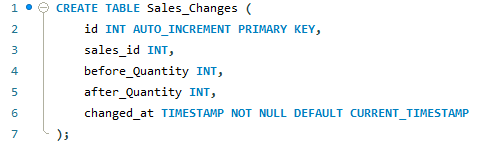
CREATE TRIGGER trigger\_name

AFTER UPDATE

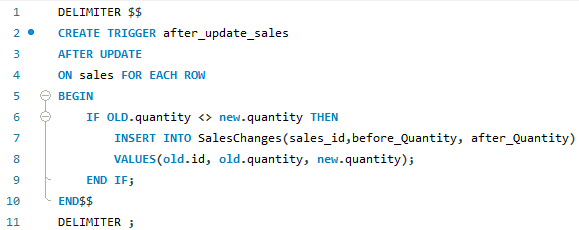
ON table\_name FOR EACH ROW

trigger\_body

Let’s understand the after update trigger with an example; we are going to use the first table as **sales** table and the second table as **sales\_changes.** So let’s create the second table sales\_changes.

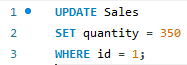


Now, let’s create the AFTER UPDATE TRIGGER;



The after\_update\_sales trigger automatically invoked after updating any row of the **sales** table.

Updating the quantity column in sales table, where id = 1



Let’s check the **sales\_changes** table;



As we can see the value is updated automatically in the sales\_changes.

## BEFORE DELETE TRIGGER

The BEFORE DELETE TRIGGER are fired automatically before a delete event occurs in table. The syntax for before delete trigger as follow:

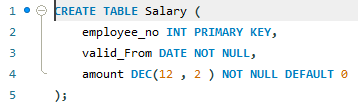
TRIGGER trigger\_name

BEFORE DELETE

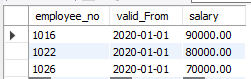
ON table\_name FOR EACH ROW

trigger\_body

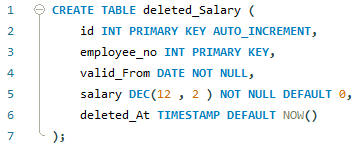
Let’s create a table as **salary;**



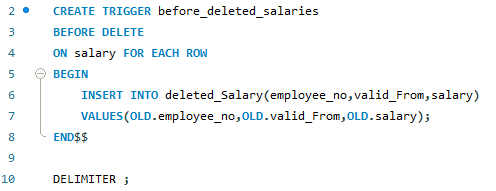
Insert few rows into salary table;



Create another table as deleted\_salary to store the deleted salaries;



Now let’s create a stored procedure, which contains the before delete triggers. Before delete trigger store the deleted value into the deleted\_salary table.



Let’s delete a row from the **salary** table;



Now, check the deleted\_Salary table to check whether the data is stored or not.



As we can see here, the BEFORE DELETE TRIGGER is automatically invoked the row before event occurs on the **salary** table.

## AFTER DELETE TRIGGER

AFTER DELETE TRIGGERS are invoke automatically after deleting the event occurs on the table. The syntax for AFTER DELETE TRIGGERS as follow:

CREATE TRIGGER trigger\_name

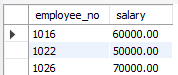
AFTER DELETE

ON table\_name FOR EACH ROW

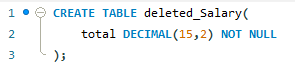
trigger\_body;

Create a table **salary** and insert few rows into the table;

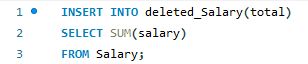




Create another table to store the deleted row into that,we are creating another table as deleted\_salary;



Now,let’s store the value of total into the deleted\_salary table by using the below command. Here,we are using the SUM() function to add the salaries from the salary table and store it into the deleted\_salary as total.

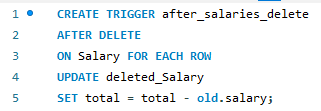


So the total amount is 180000.

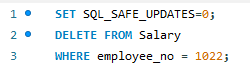


Now, Let’s create AFTER DELETE TRIGGER;

We are creating a trigger which update the total salary into the deleted\_Salary table after deleting from the salary table.



Let’s delete a row where the employee\_no = 1022 inside the salary table;



Check the deleted\_Salary;



As we can the value of total is decresed by 50000, because it is substracted from the total amount.