INTRODUCTION TO SQL

(STRUCTURAL QUERY LANGUAGE)

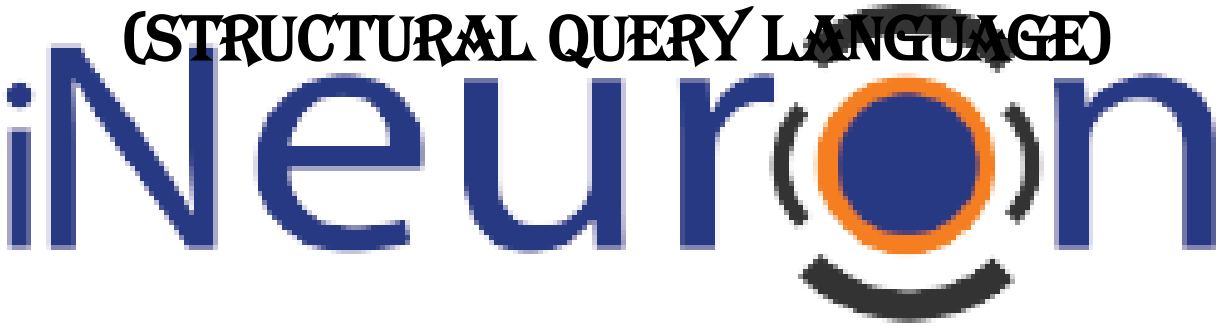


Table of Content

Contents

1. 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.

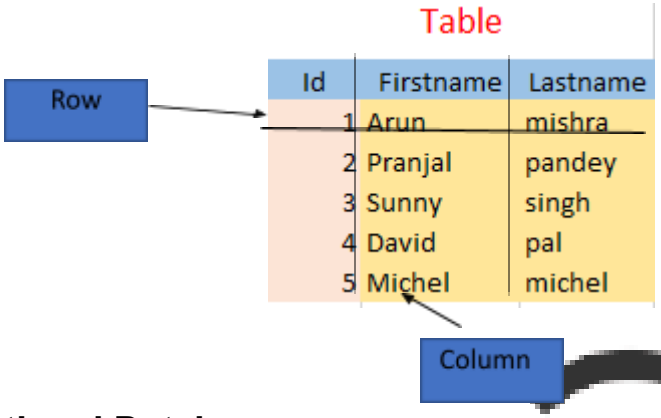
Since that time, the American National Standards Institute (ANSI) and the International Standards Organization have deemed the SQL language as the standard language in relational database communication. While major SQL vendors do modify the language to their desires, most base their SQL programs off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

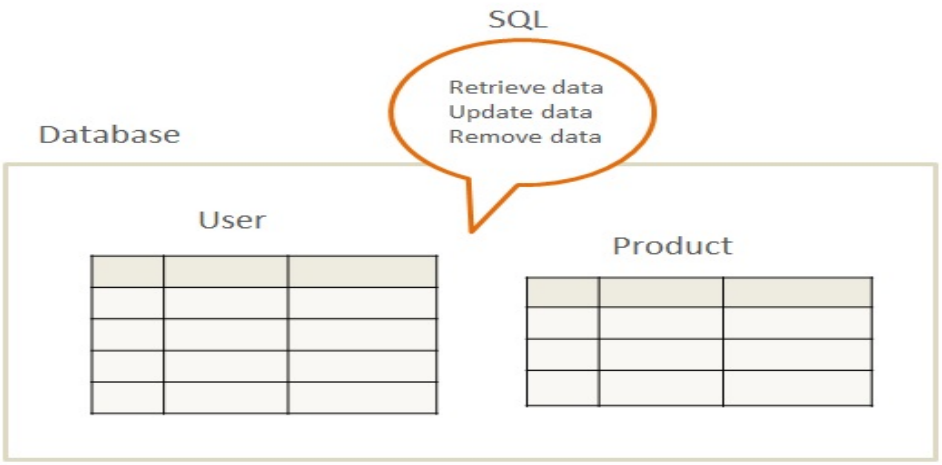
Relational Database

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**.



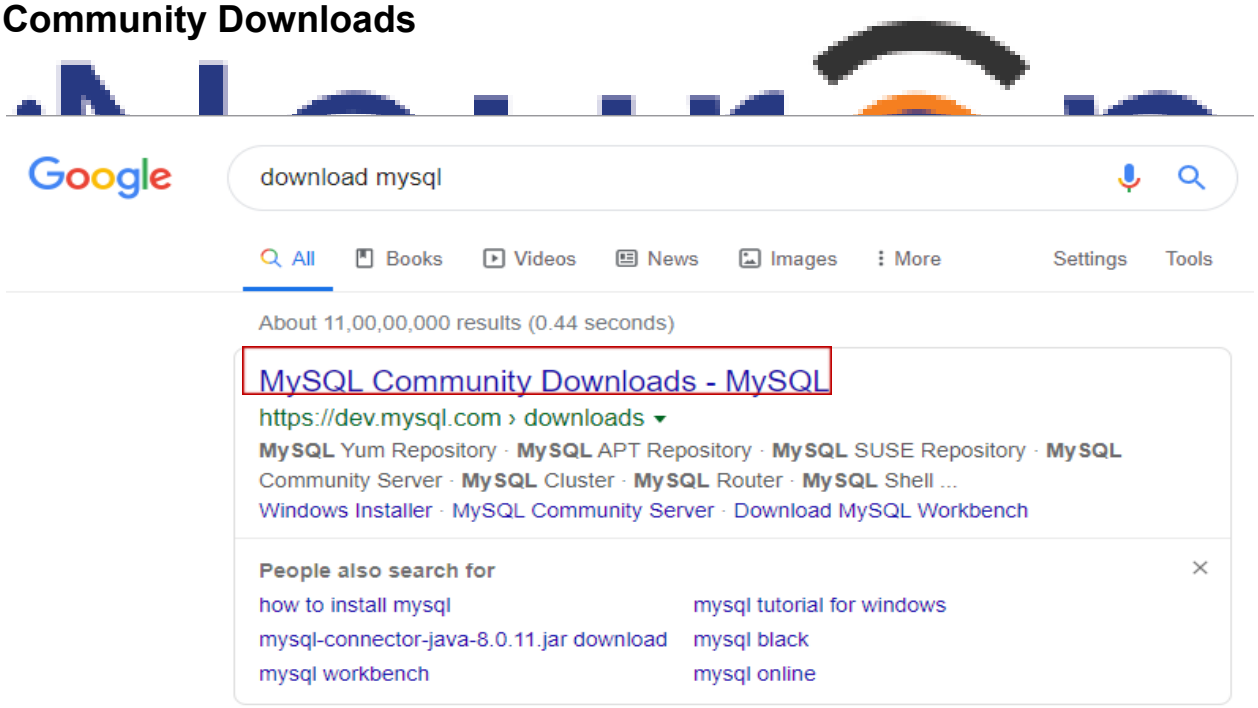
How to run SQL Query on the local system

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.

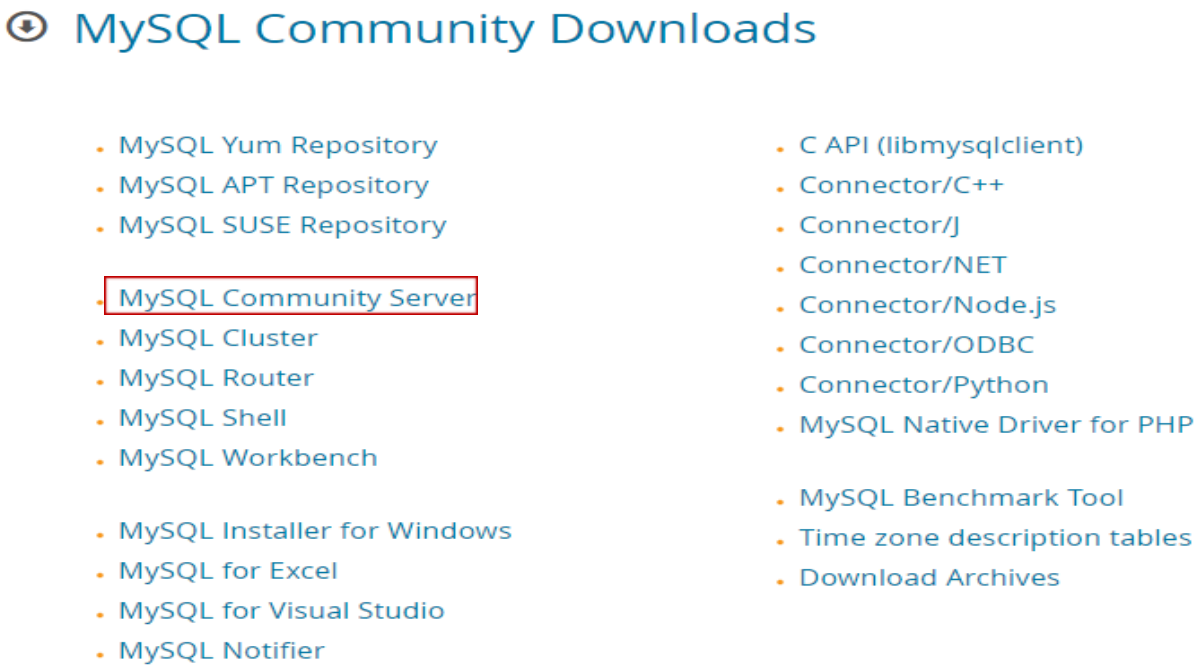
2. Downloading and Installing MySQL

2.1 Downloading MySQL

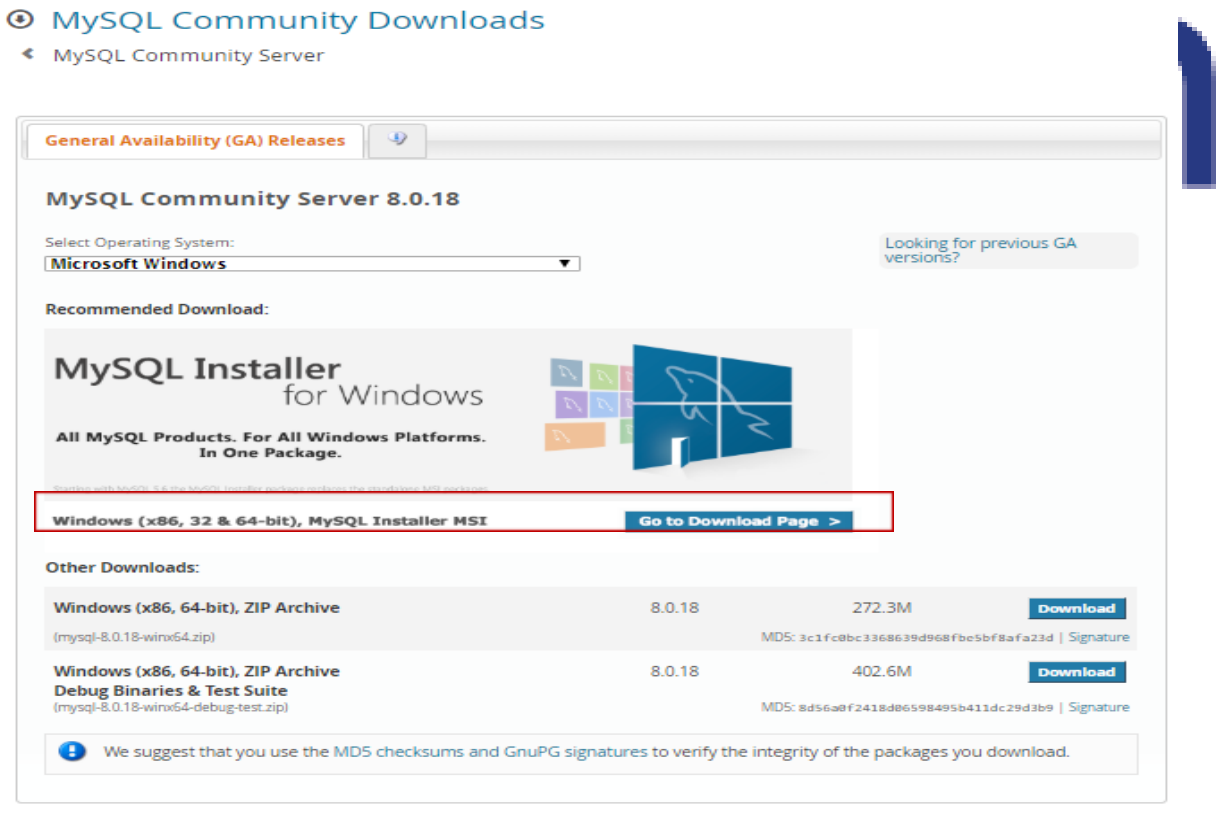
Step 1: Open Google and type **Download MySQL** and Click on **MySQL Community Downloads**



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



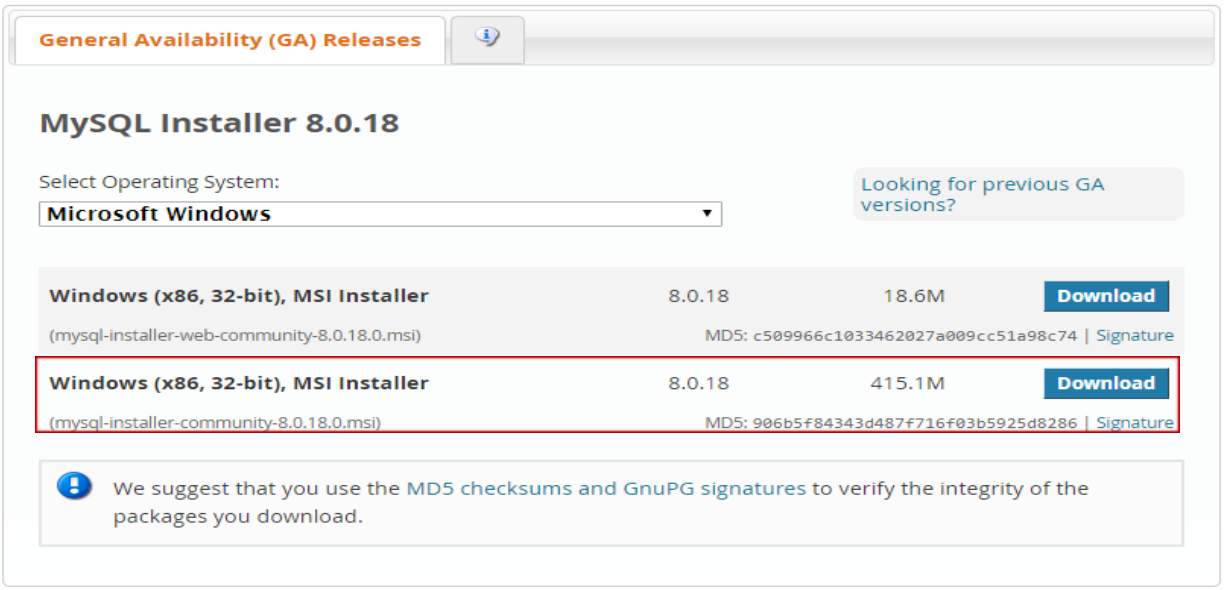
Step 3: Click on the MySQL installer MSI **Go to Download Page >**



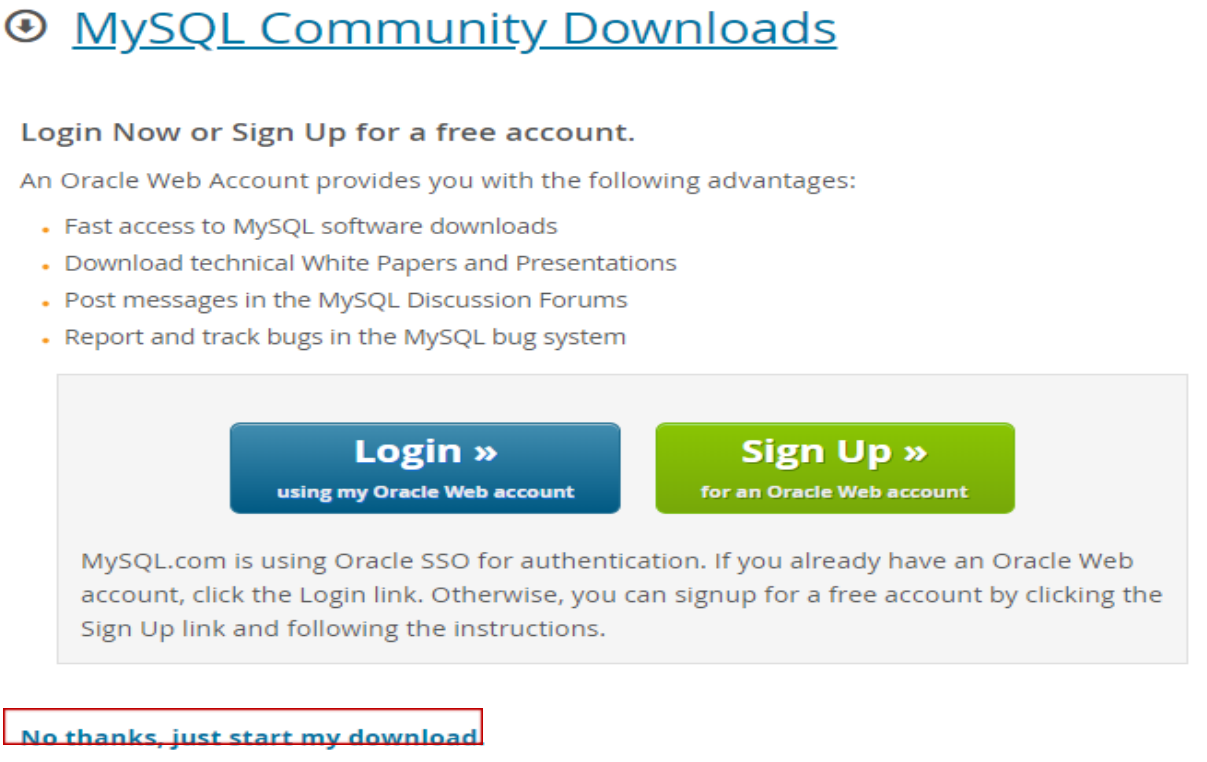
Step 4: Select the OS and click on **MSI Installer community**

MySQL Community Downloads

MySQL Installer



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

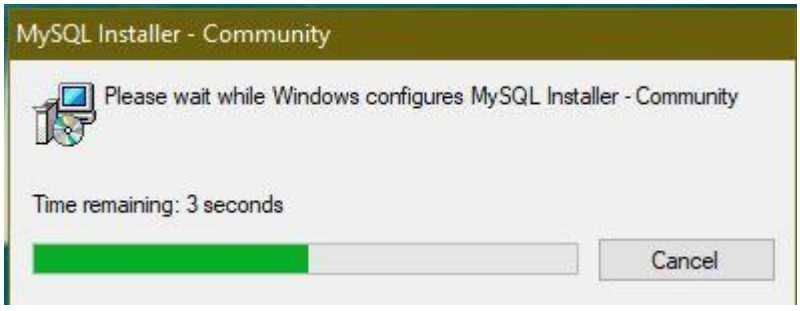


Note: Once the Downloading is completed, then double-click on that and install it on the local system.

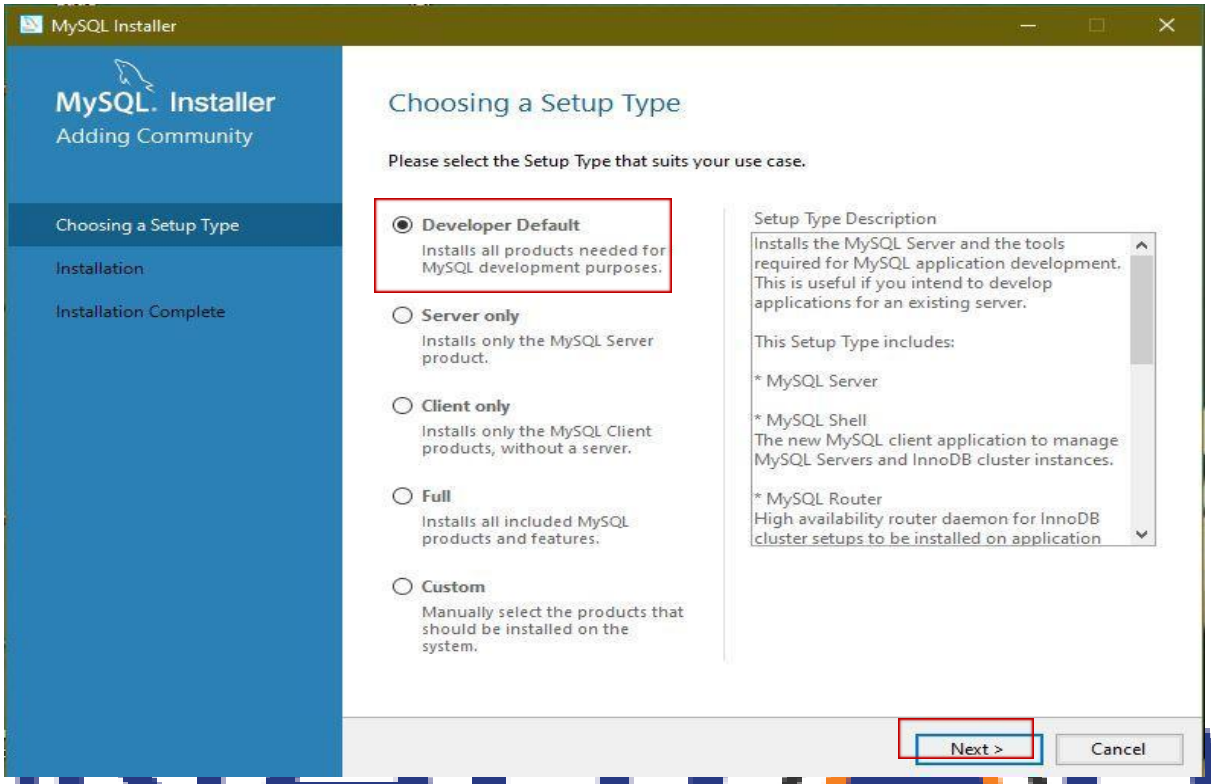
2.2. Installation of MySQL

Step 1: Double-Click on Downloaded Application.

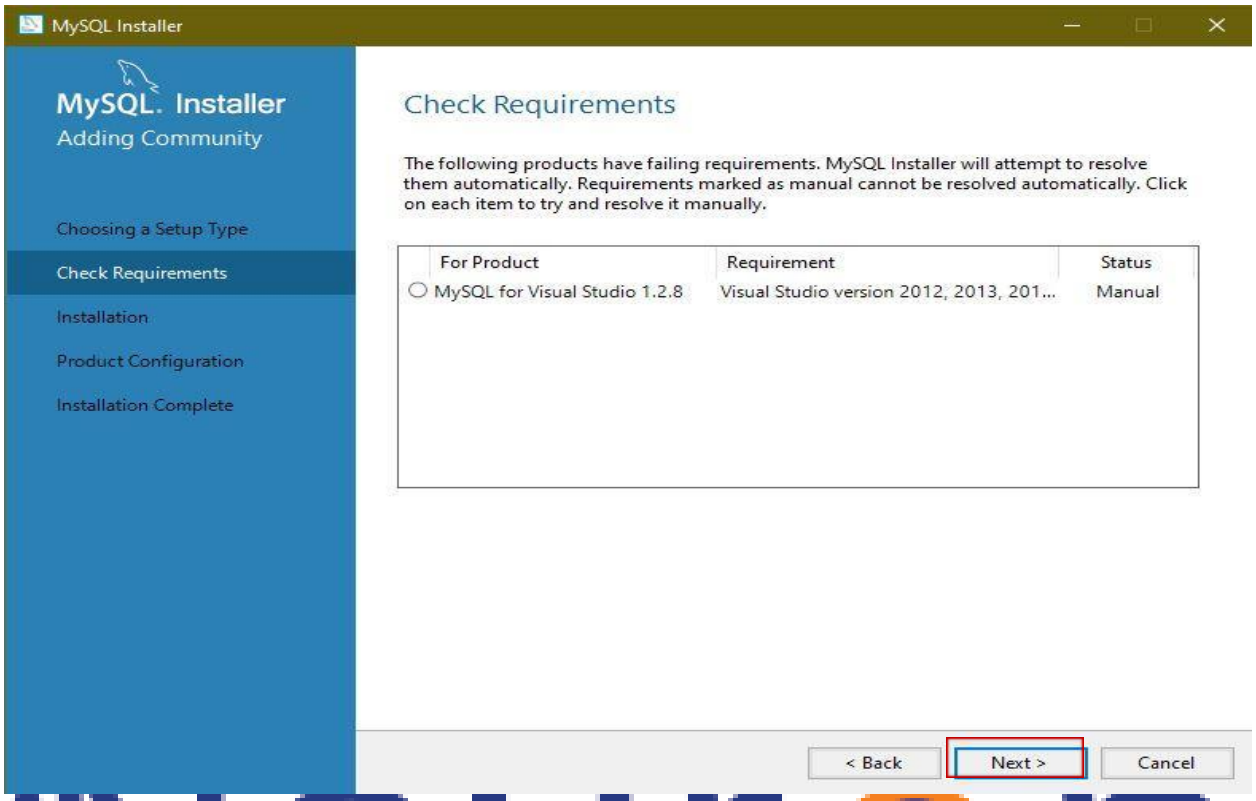
Step 2: After clicking on the application we will get a window like below



Step 3: Choosing the Setup type and click Next.

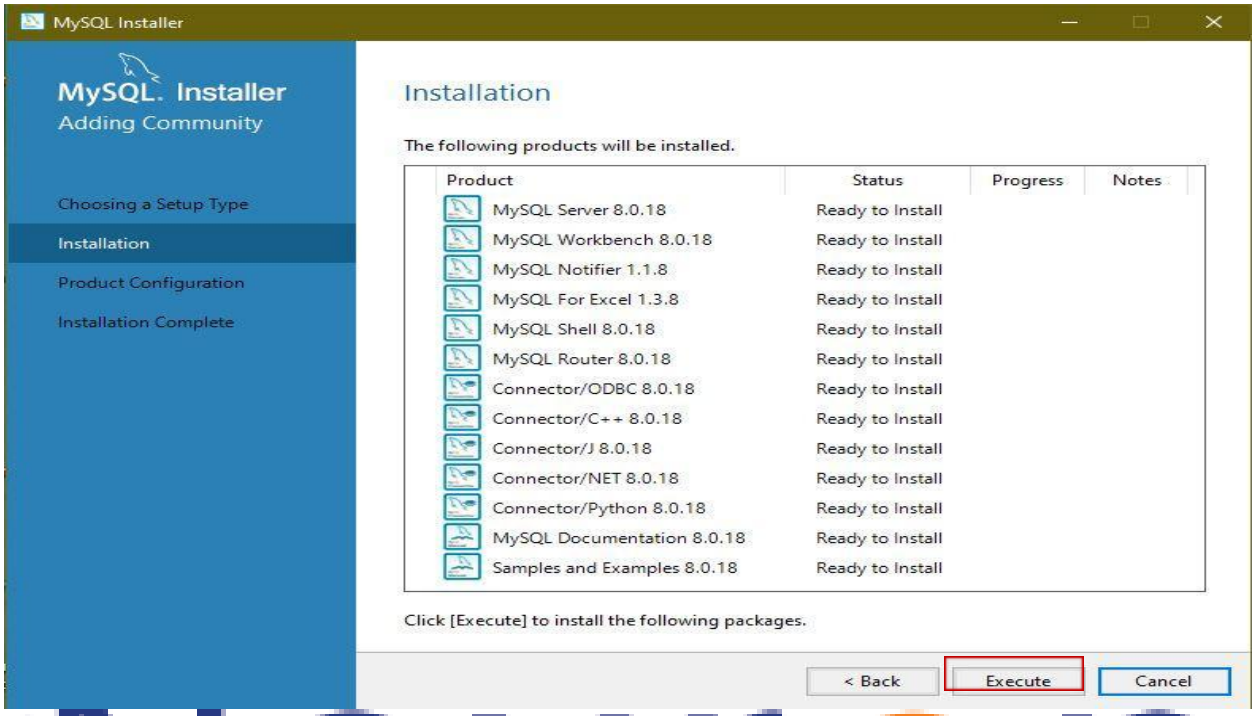


Setup 4: Click Next.

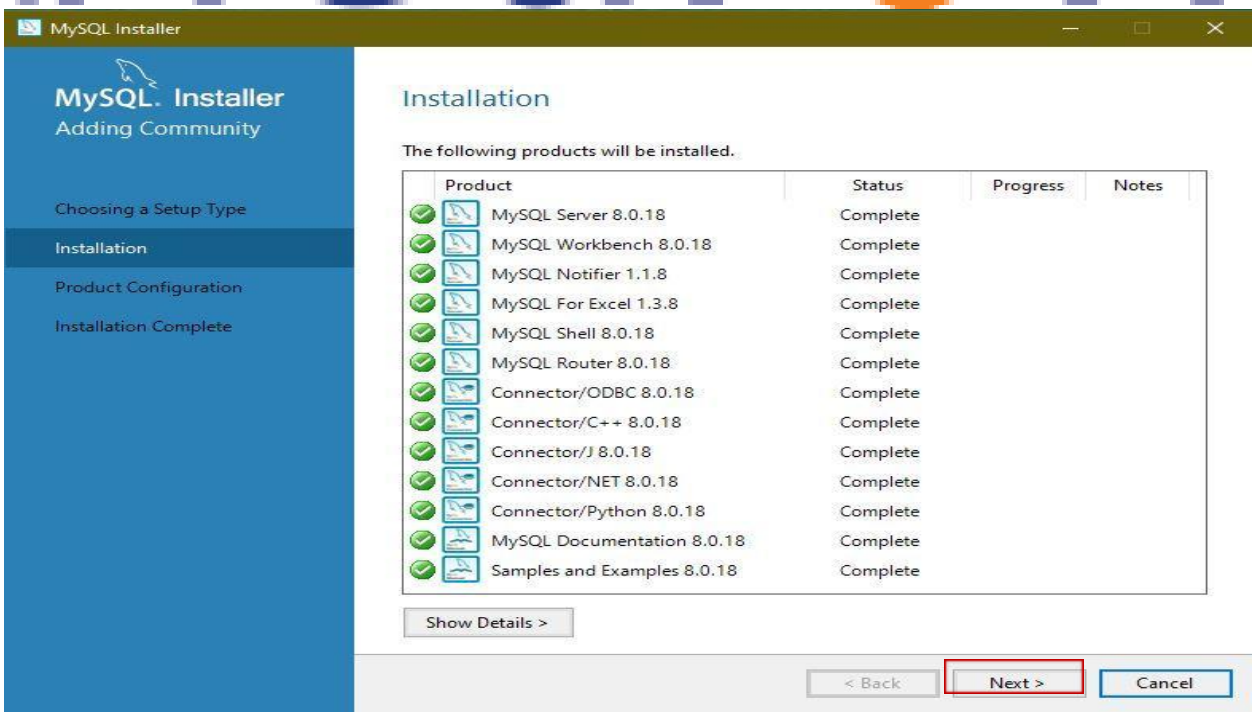


Step 5: Click Yes.

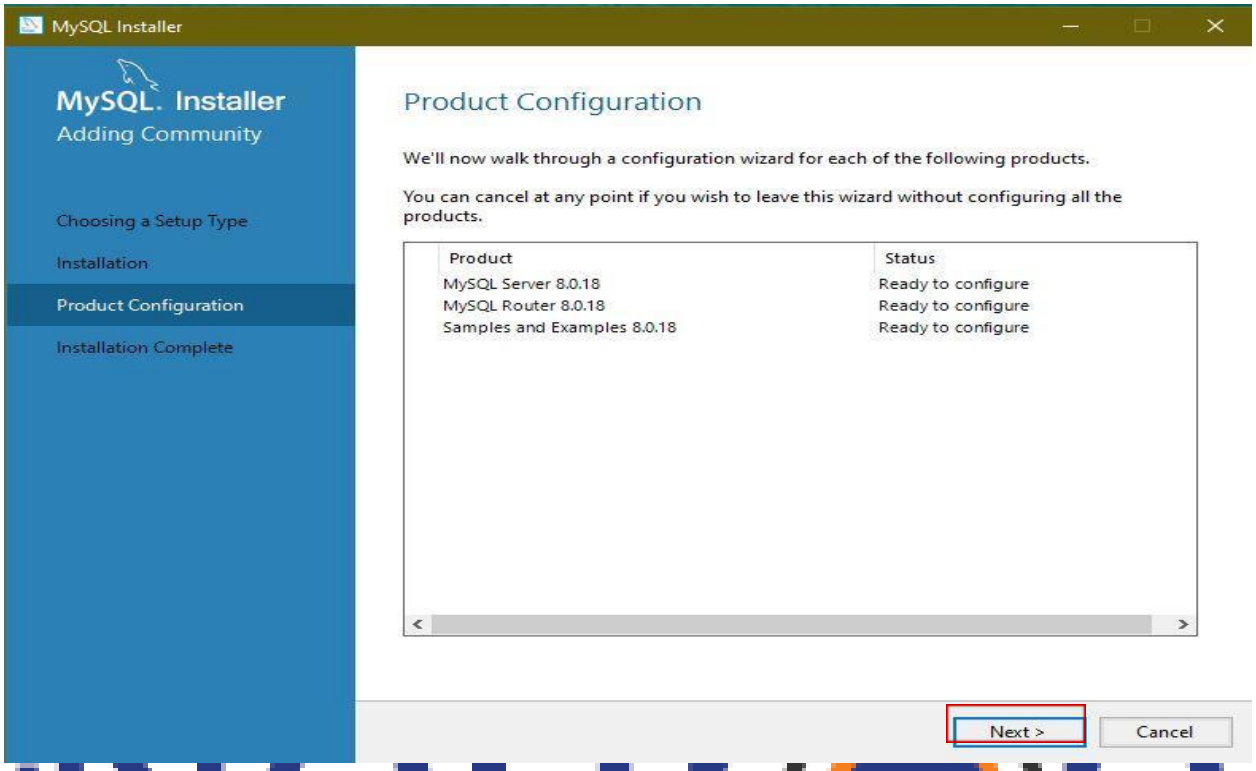
Step 6: Click Execute.



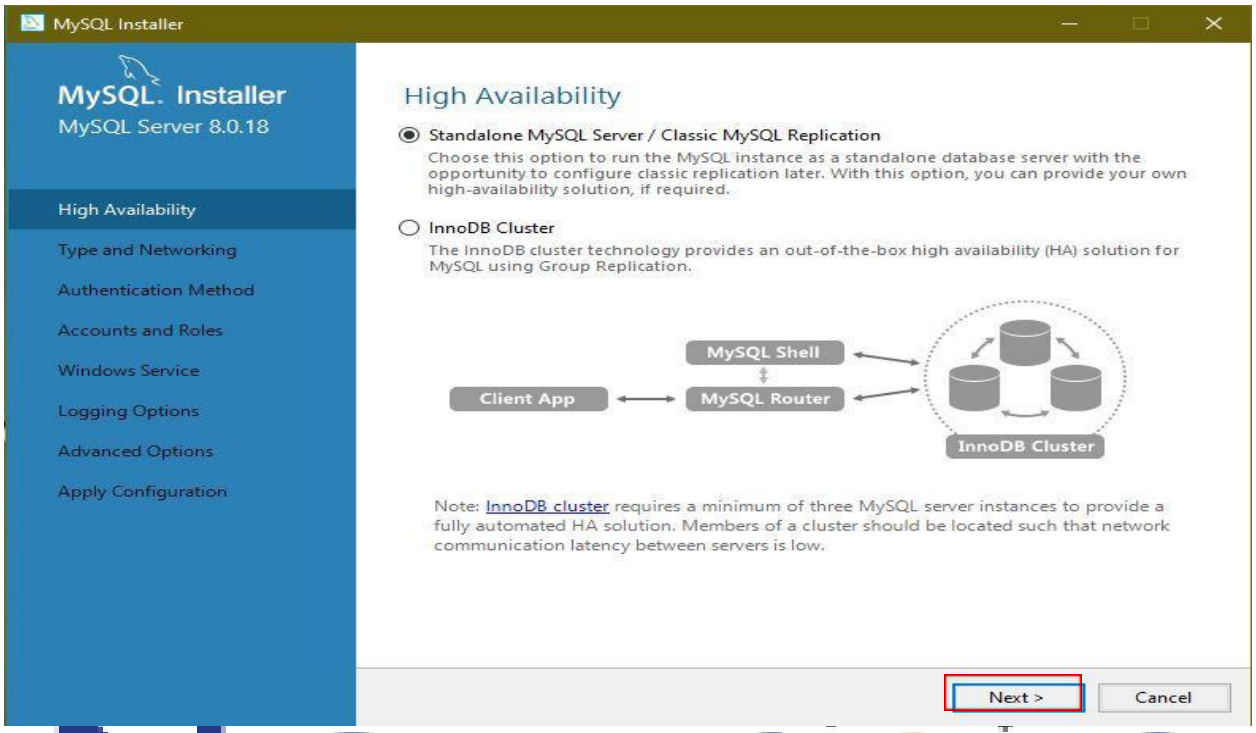
Step 7: After Execution, click on the Next.



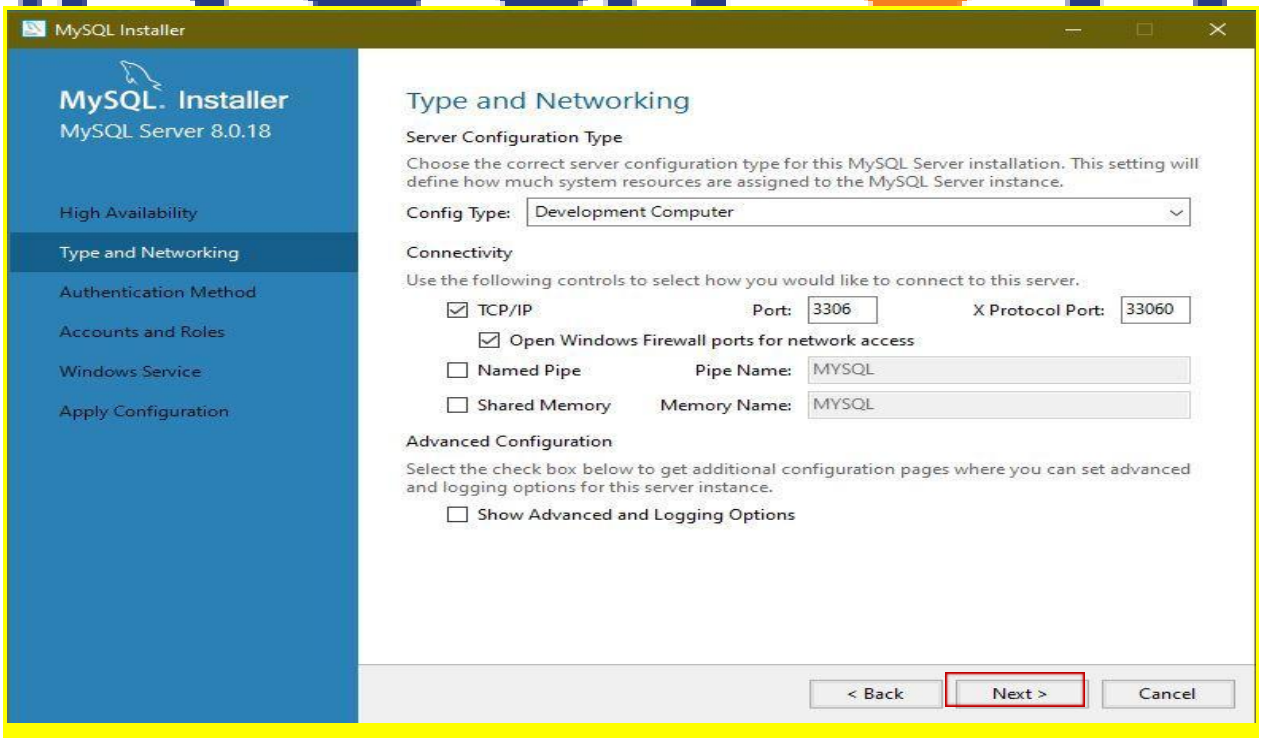
Step 8: Click Next.



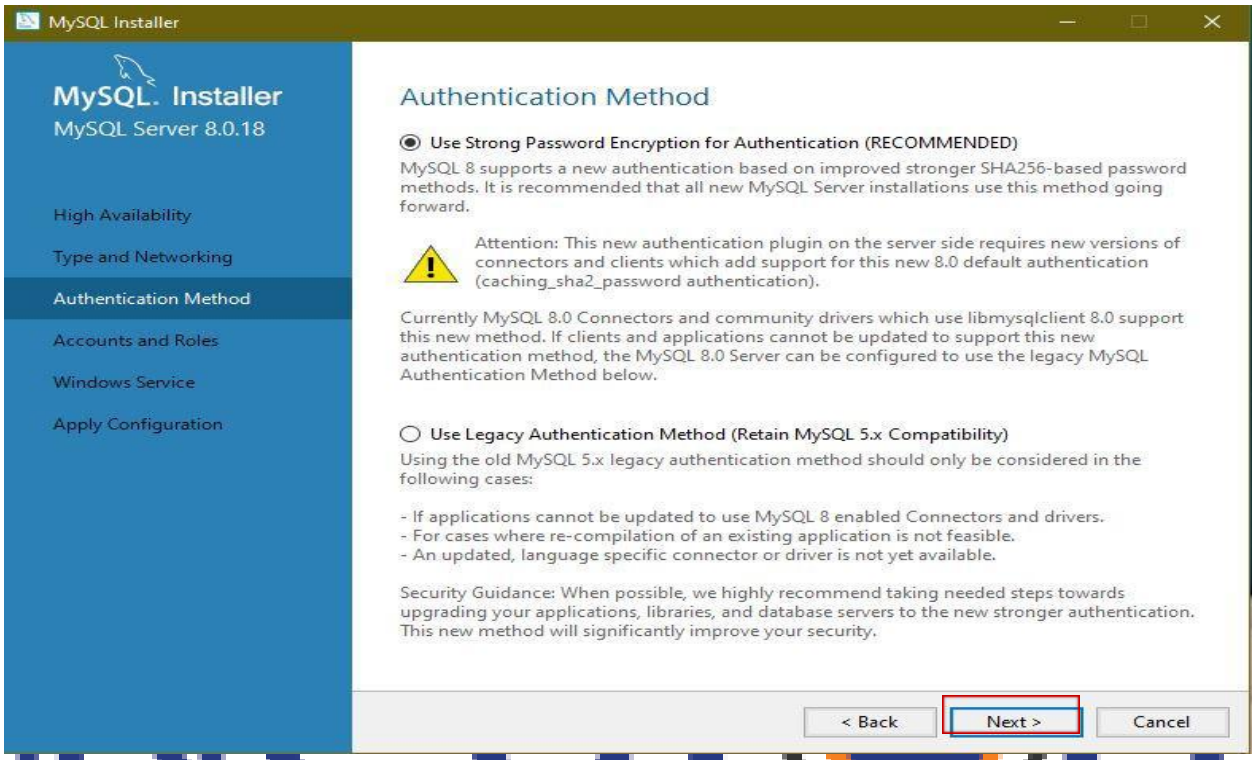
Step 9: Click Next.



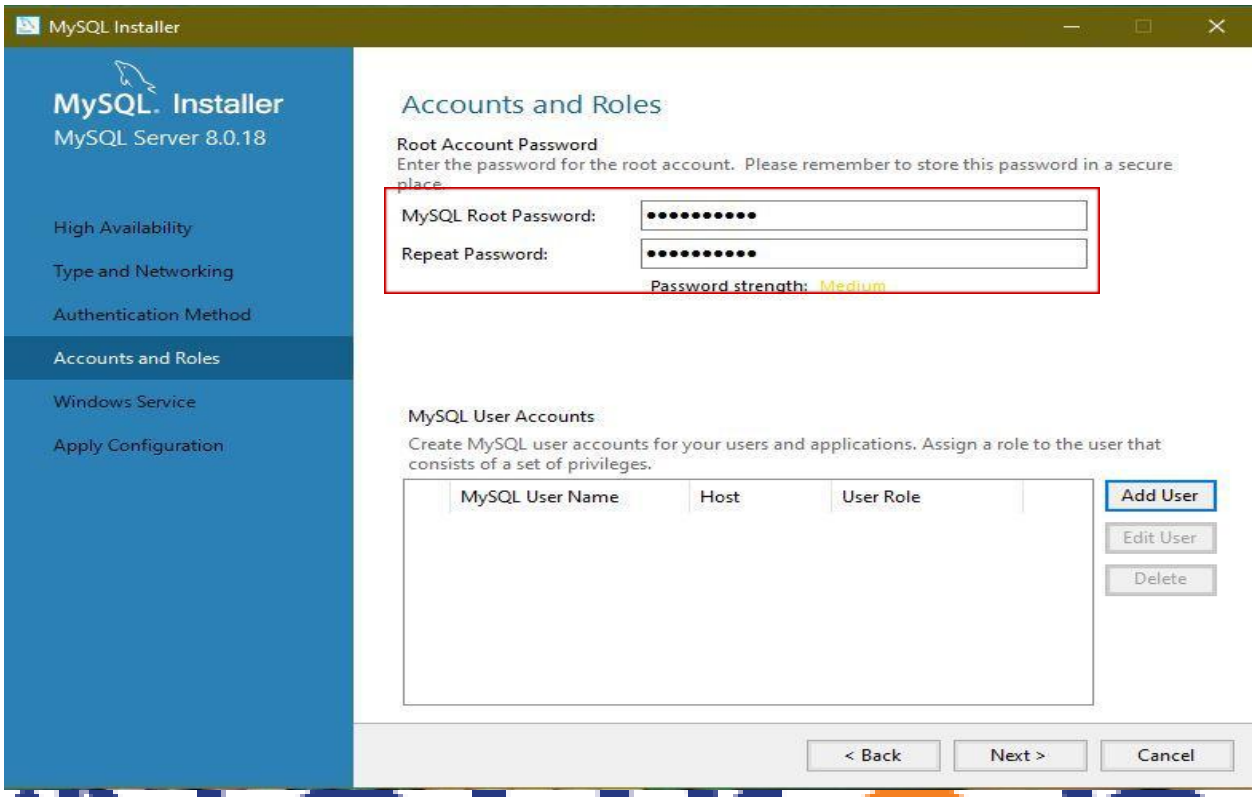
Step 10: Leave it as default and click Next.



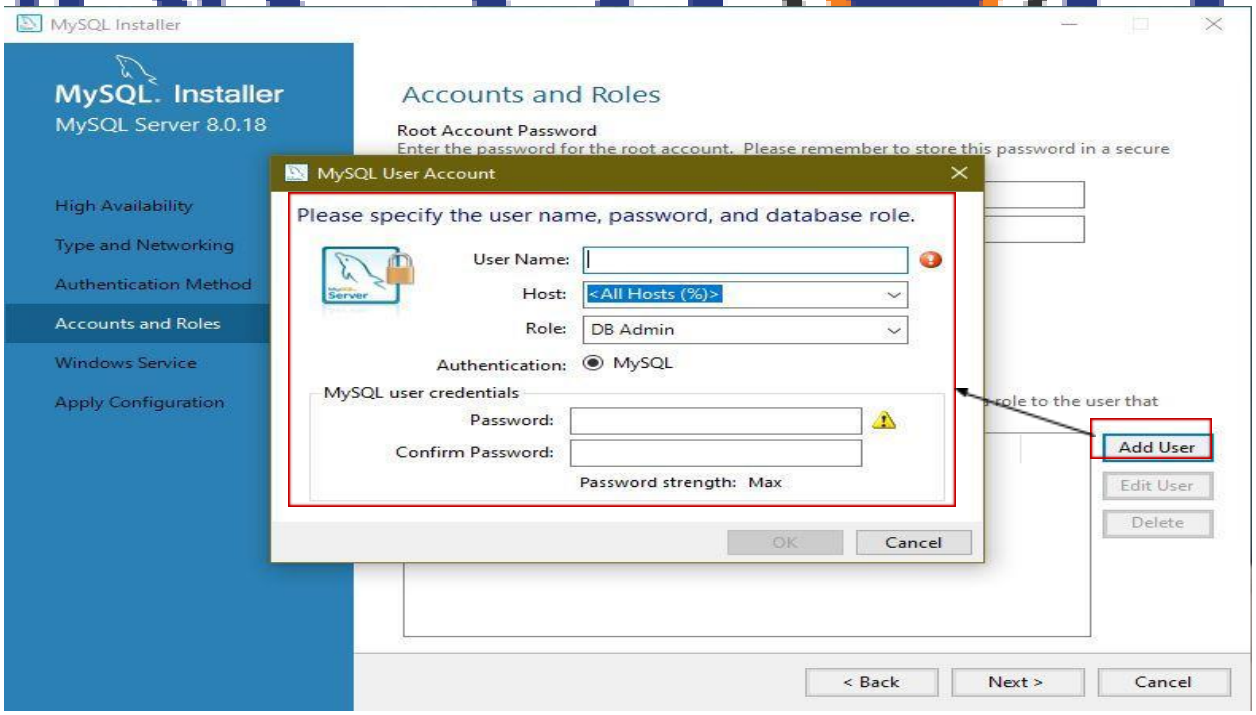
Step 11: Click Next



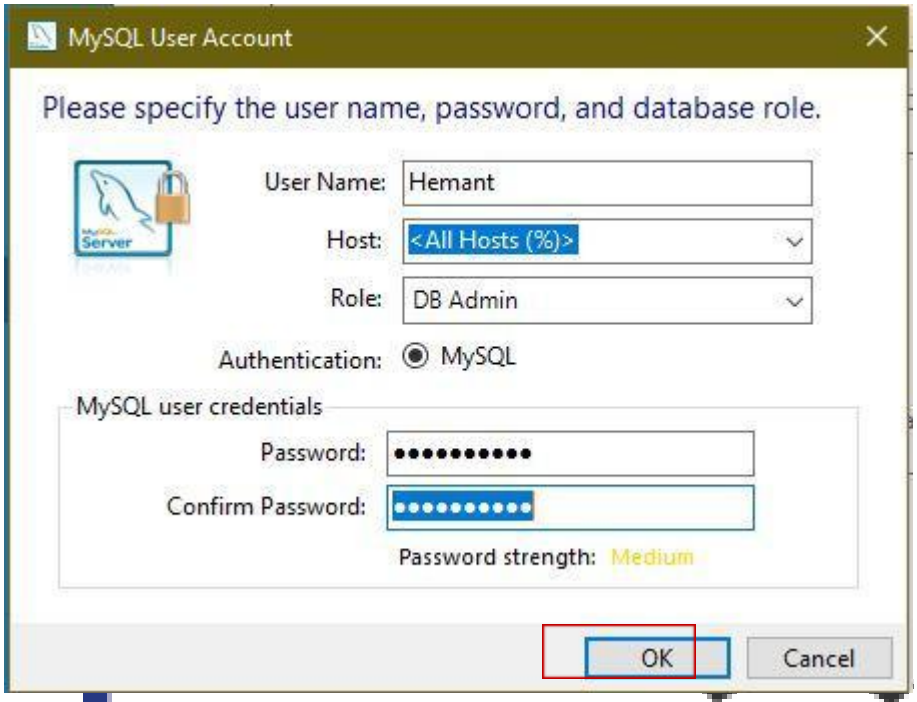
Step 12: Choose the root password



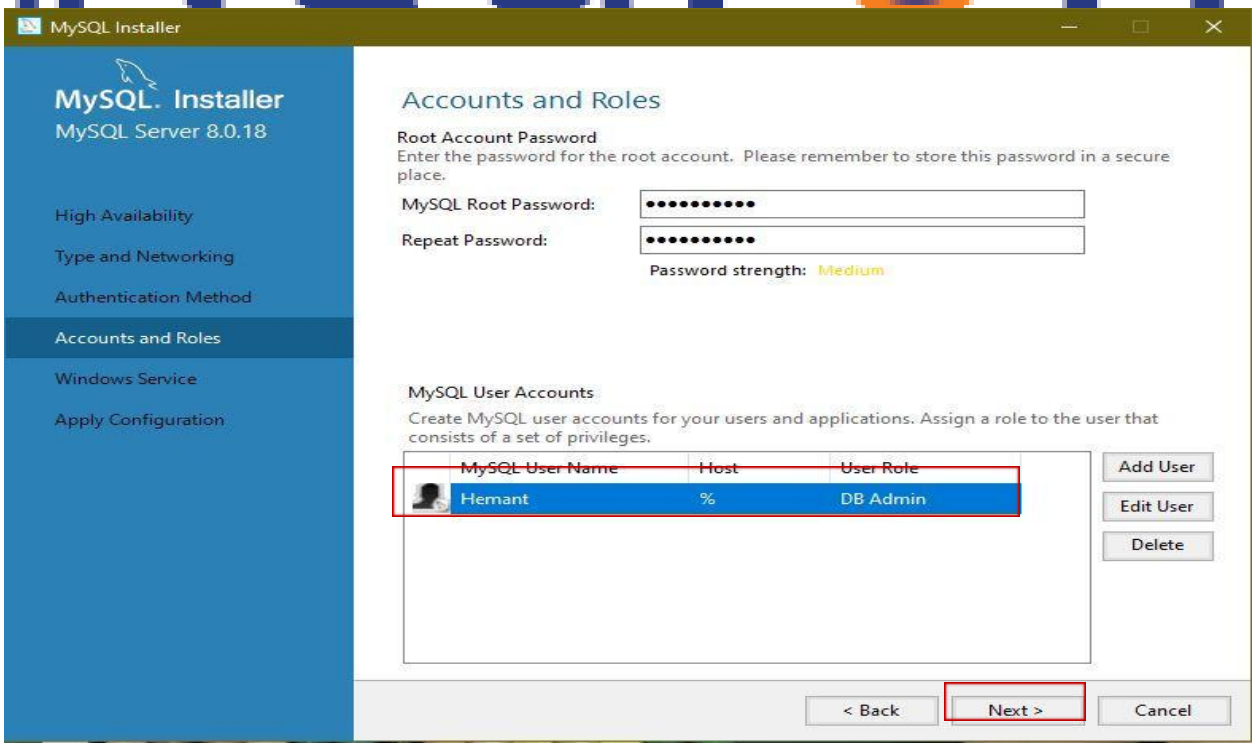
Step 13: Click on Add User and give the username and password.



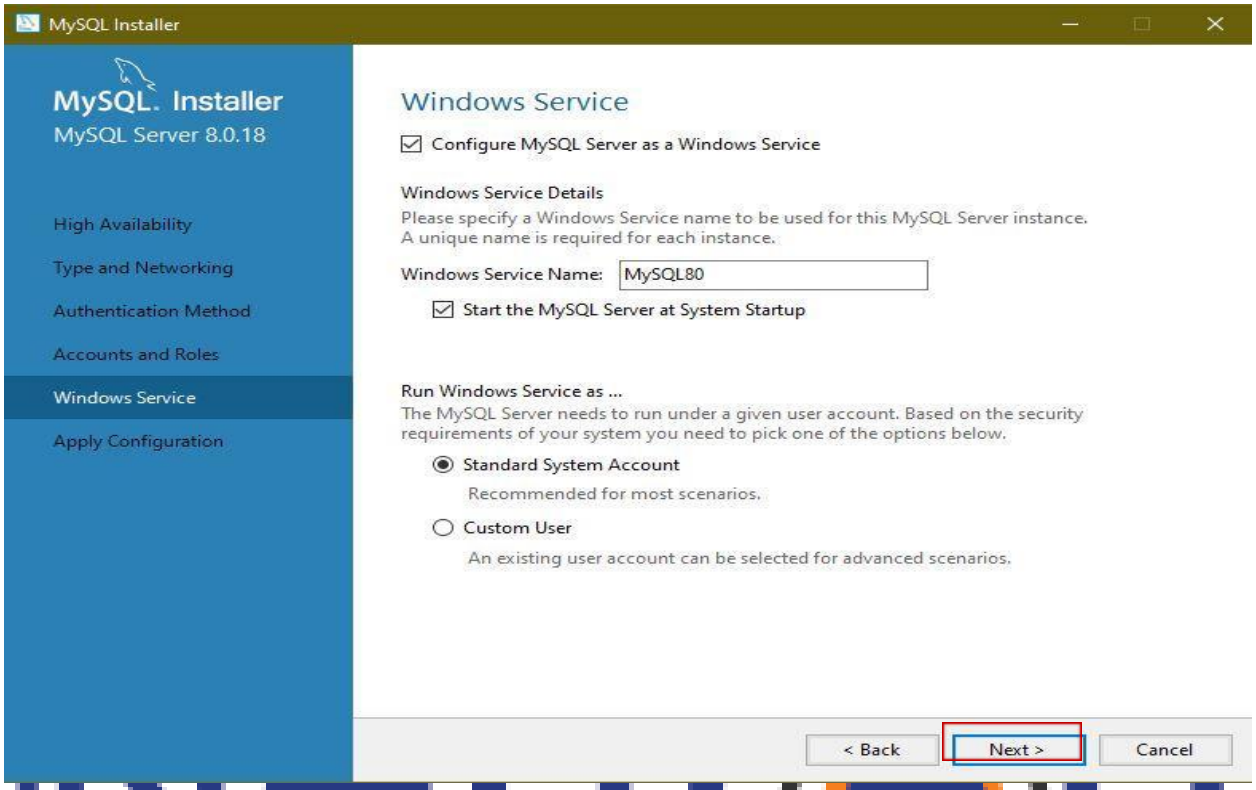
Step 14: After inserting the name and password click OK



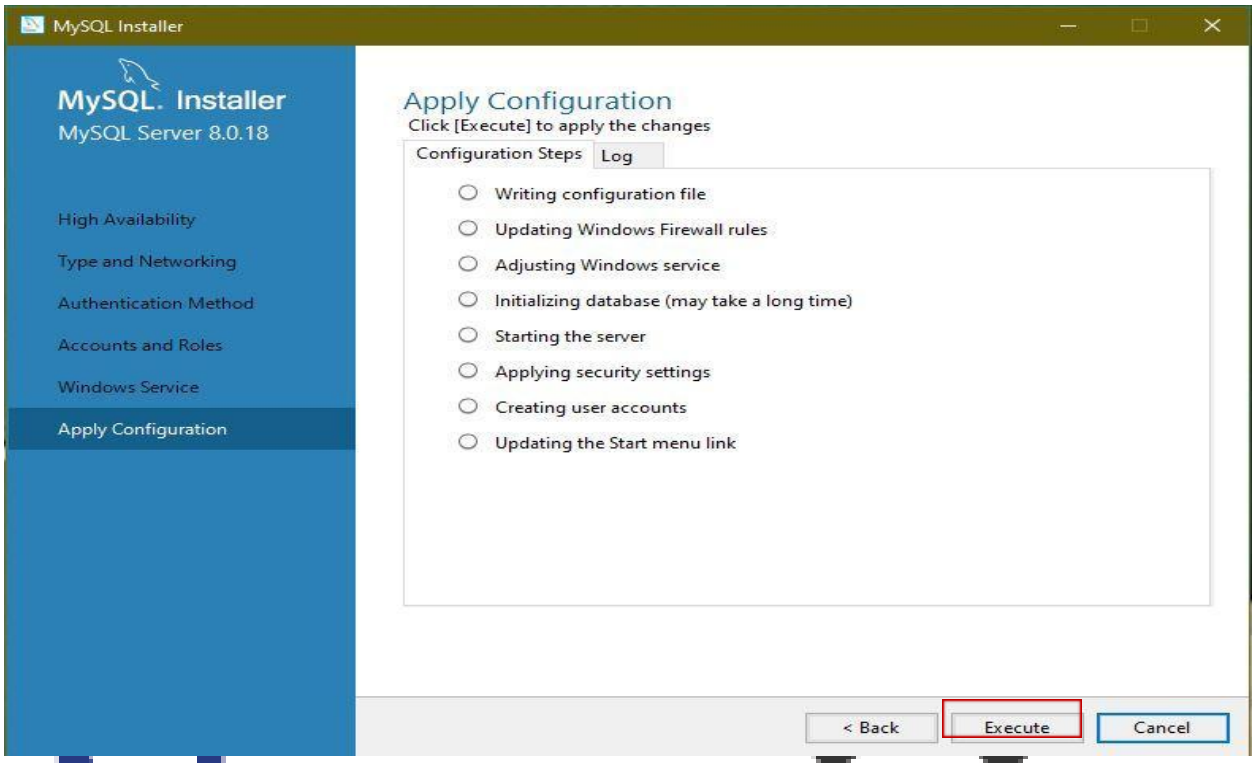
Step 15: After adding the user click Next



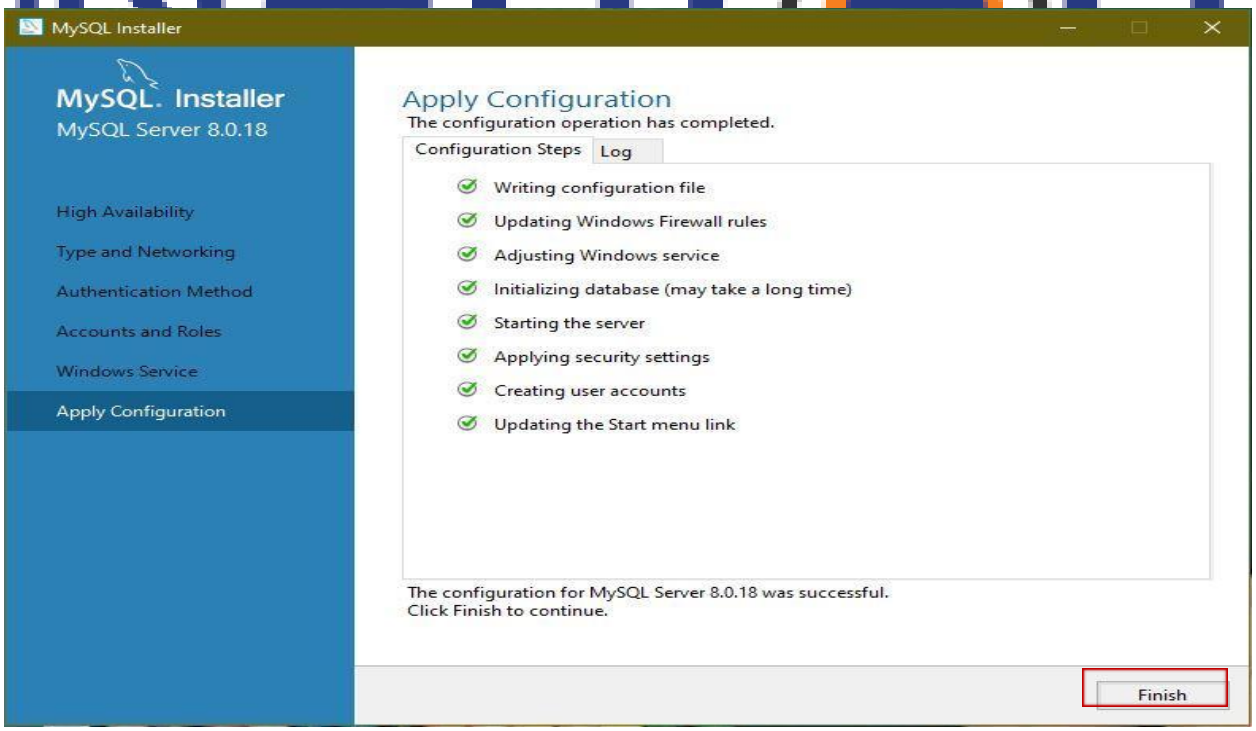
Step 16: Click Next



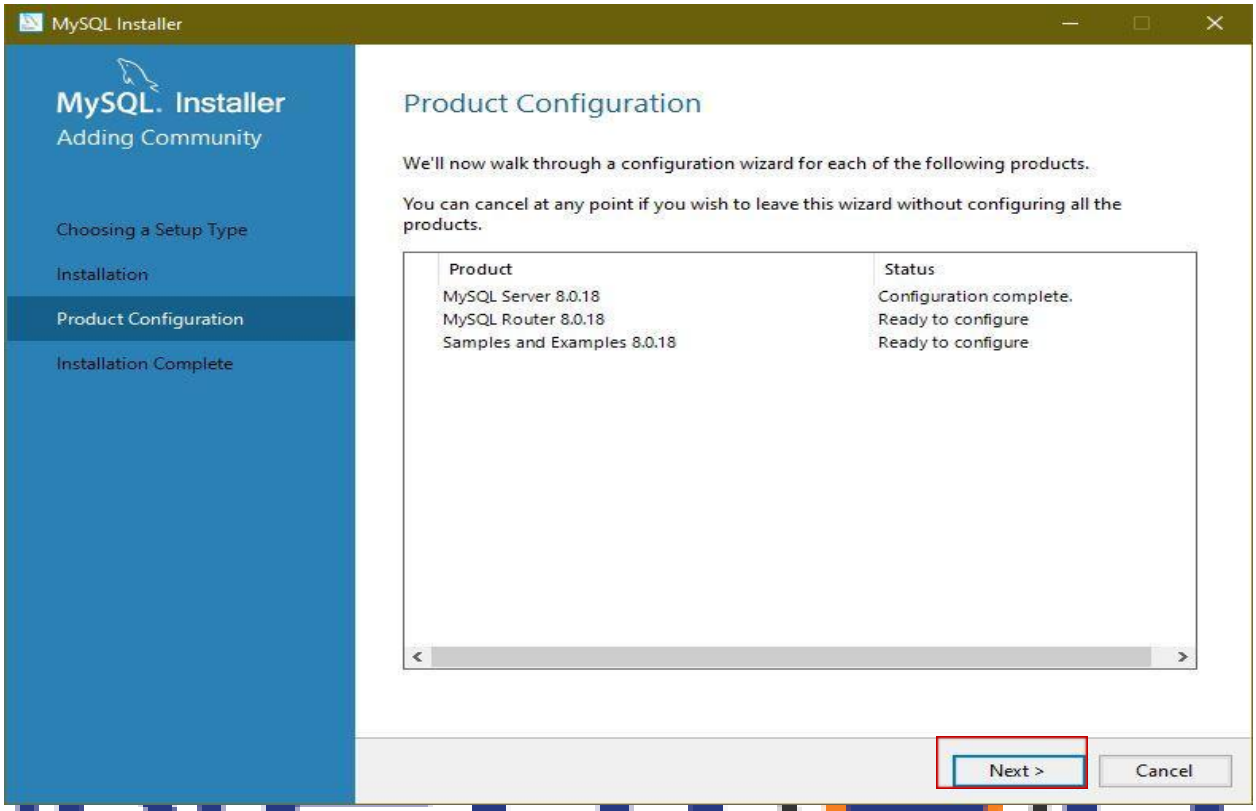
Step 17: Click on Execute



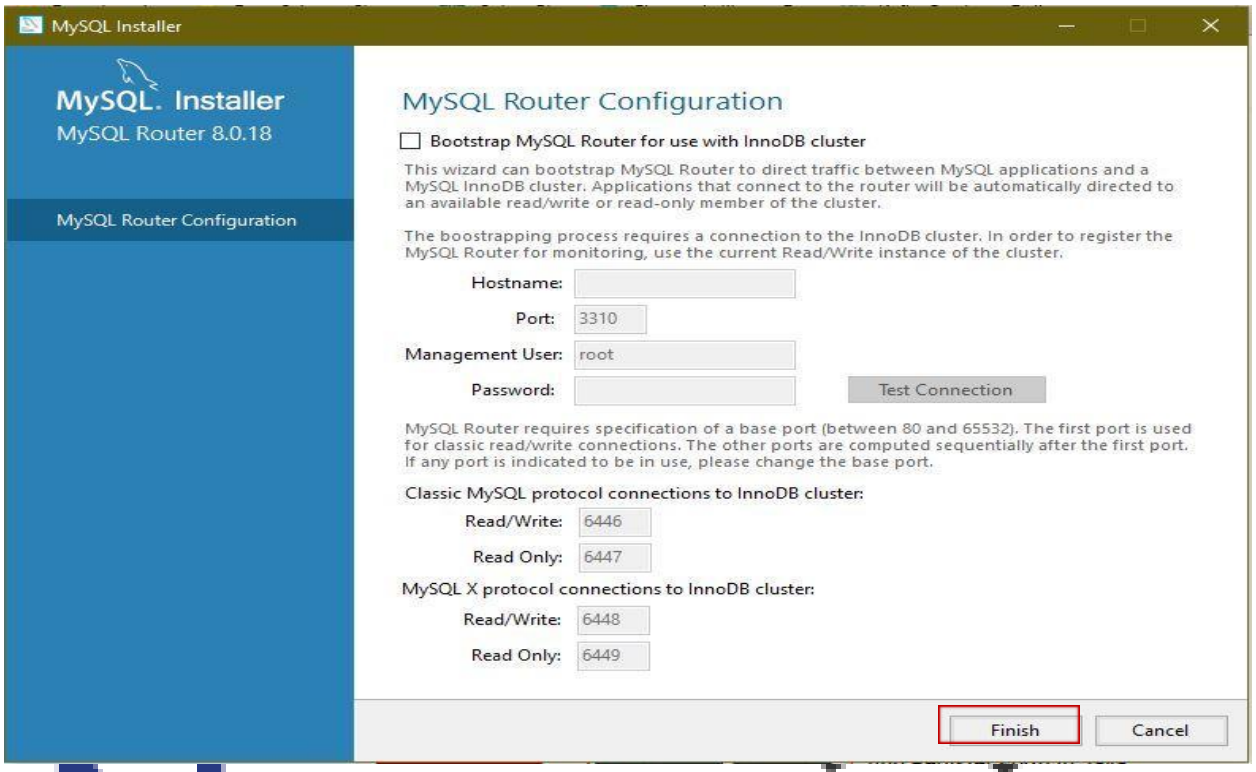
Step 18: After Clicking on execute, click Finish



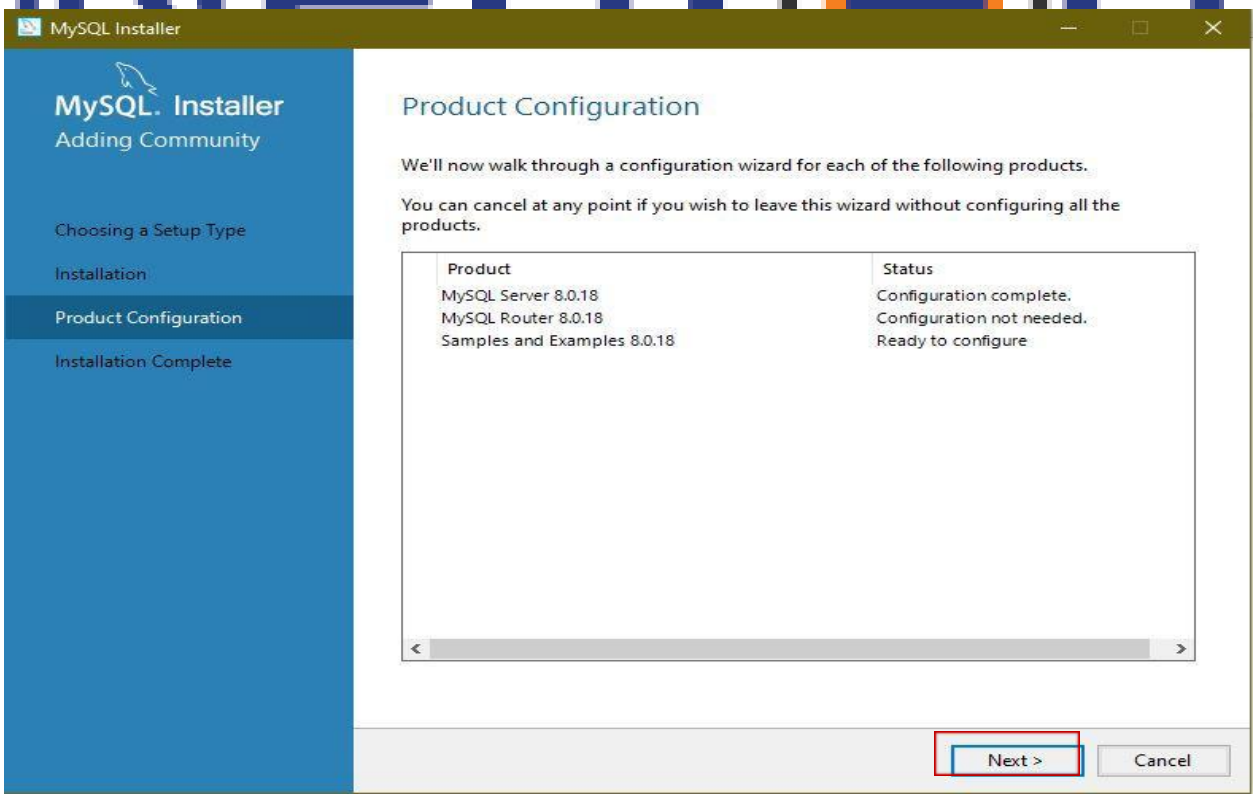
Step 19: Click Next



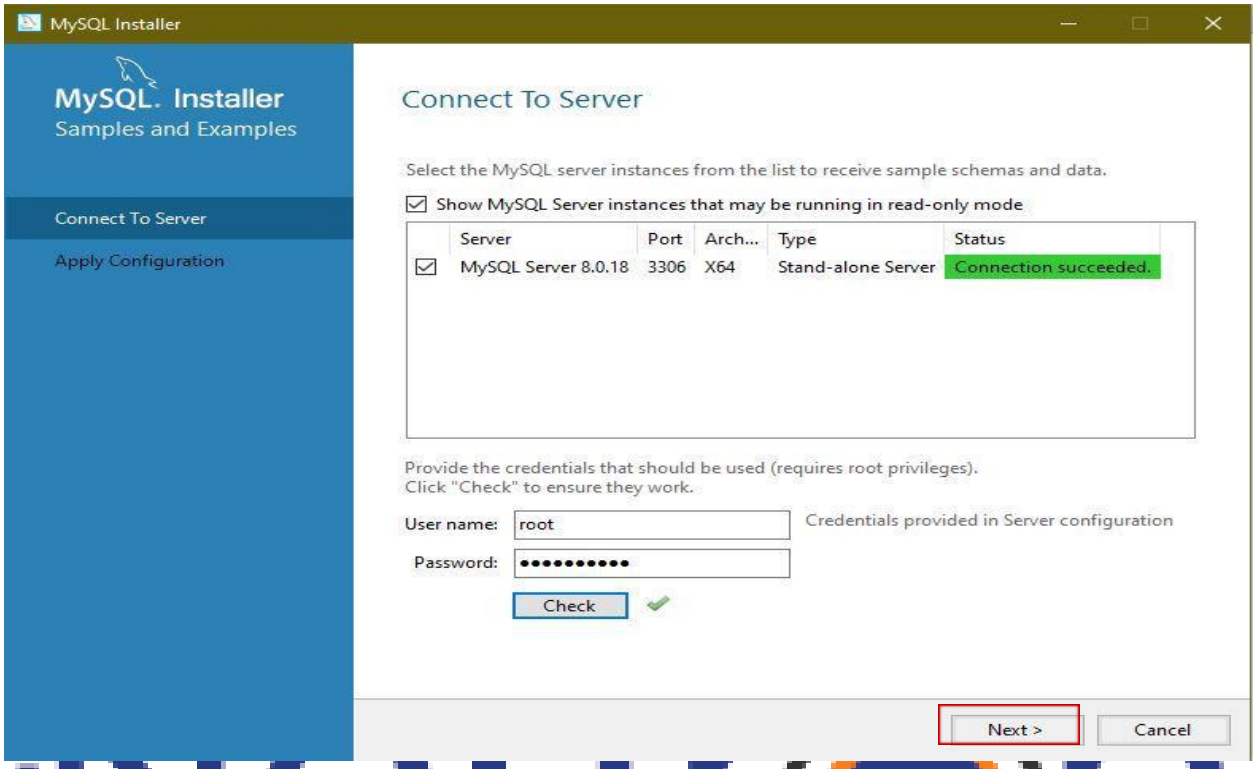
Step 20: Click on Finish



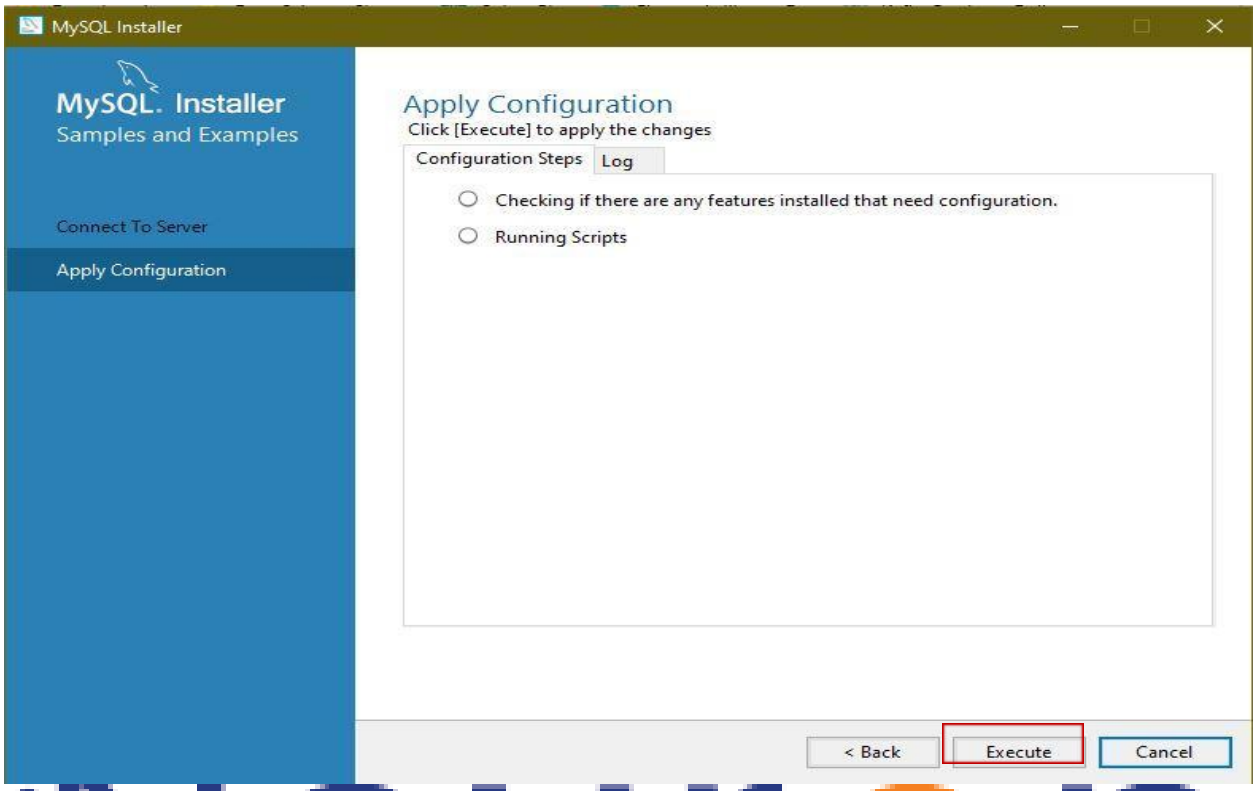
Step 21: Click Next.



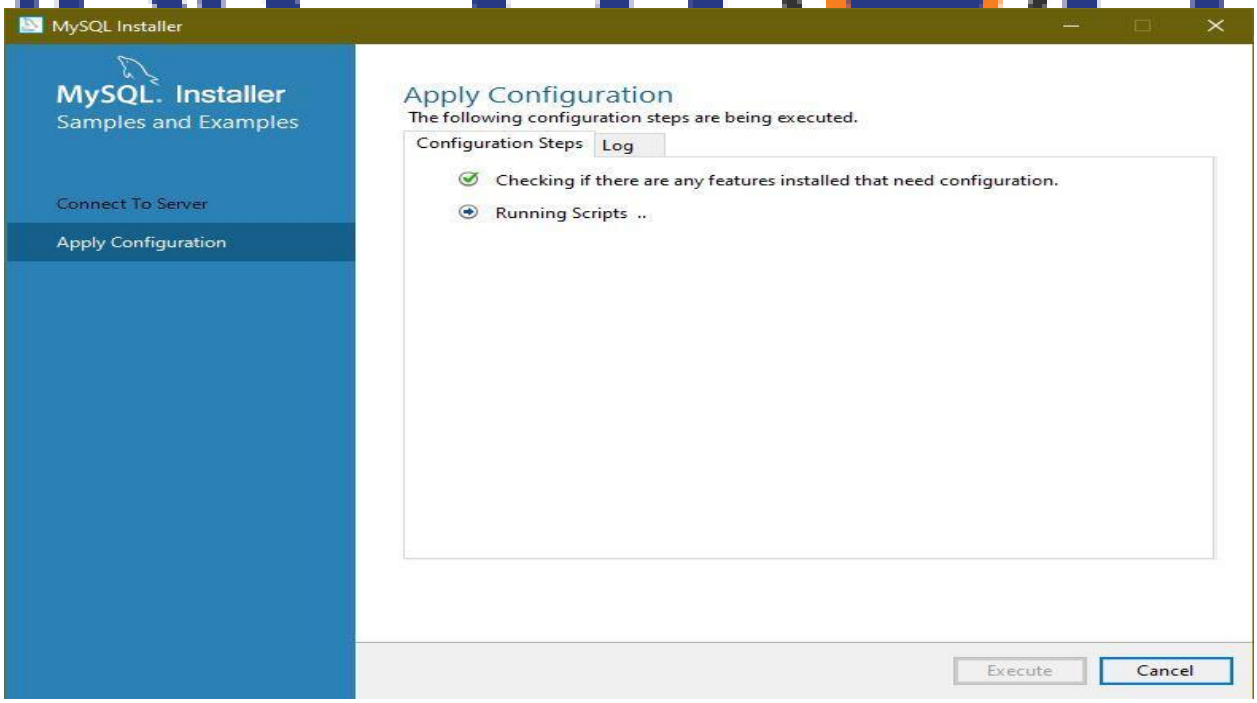
Step 22: Check the password and Click Next



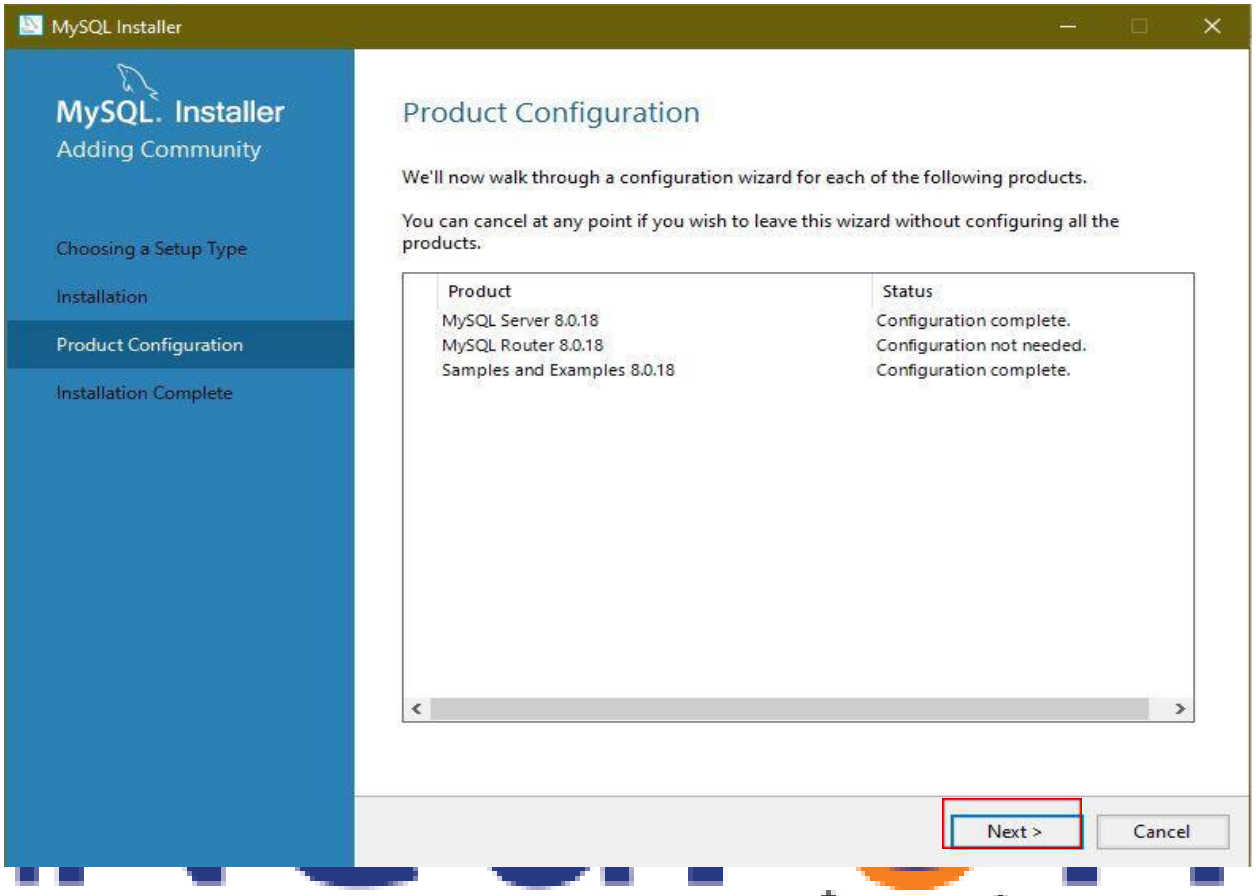
Step 23: Click Execute



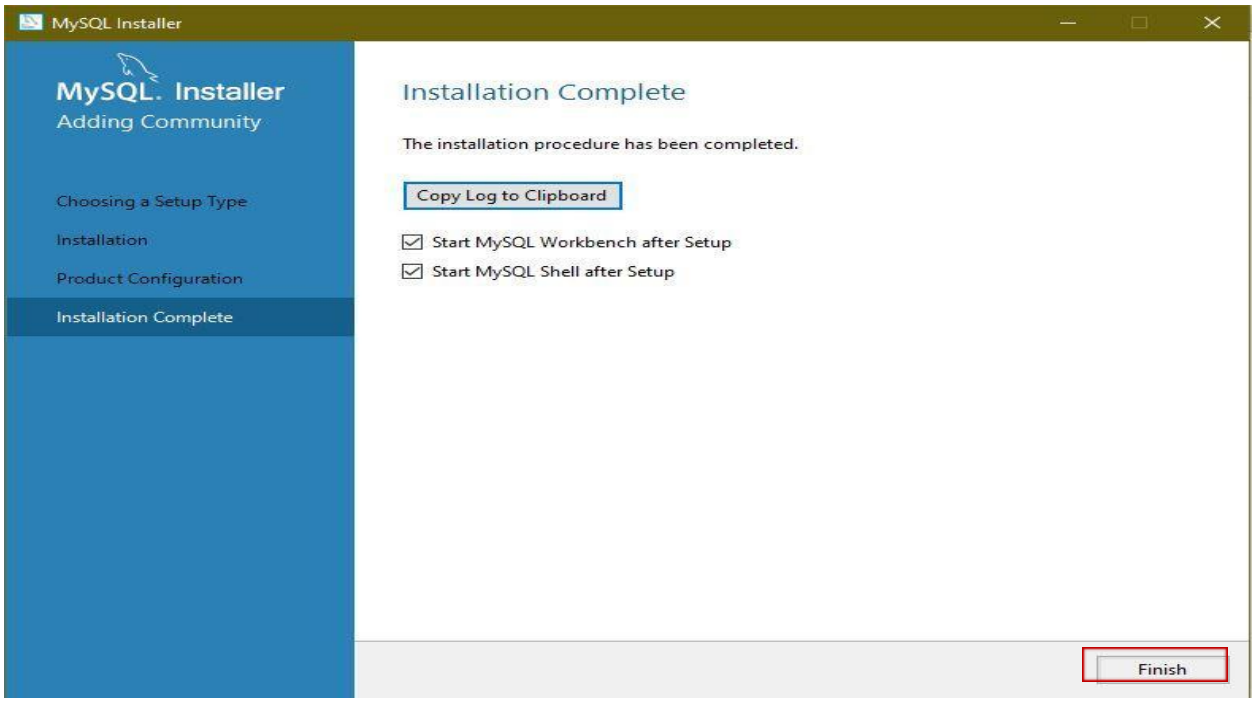
Step 24: After clicking on Execute



Step 25: Click Next

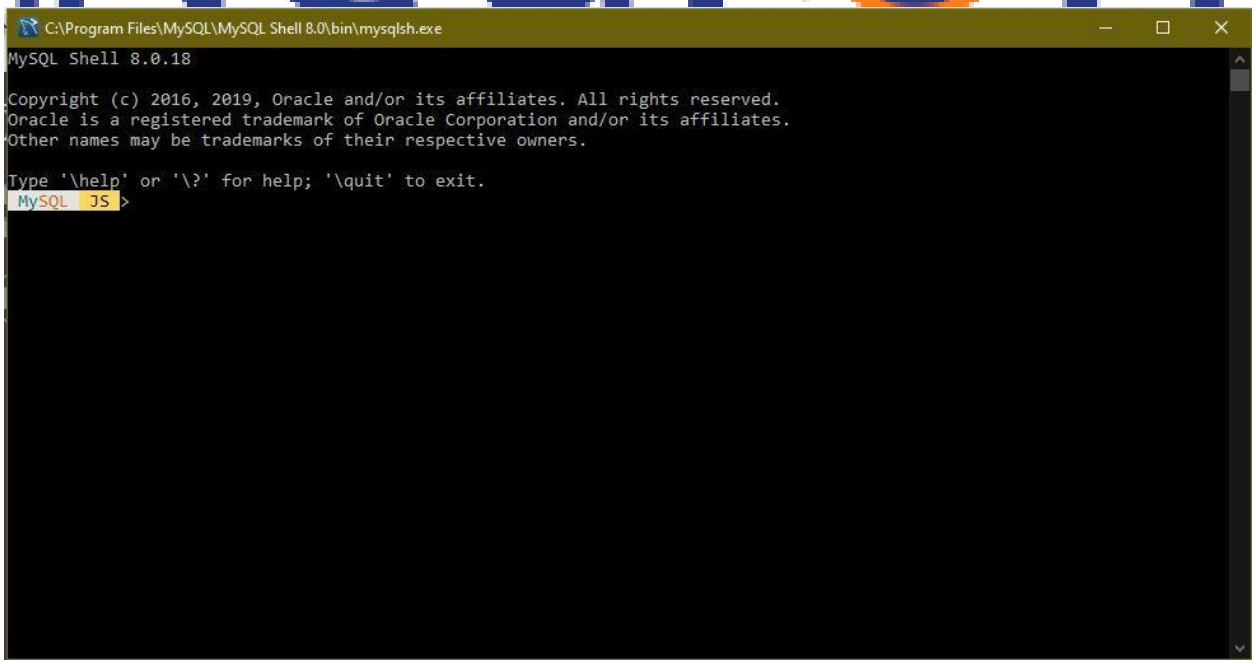


Step 26: Click Finish

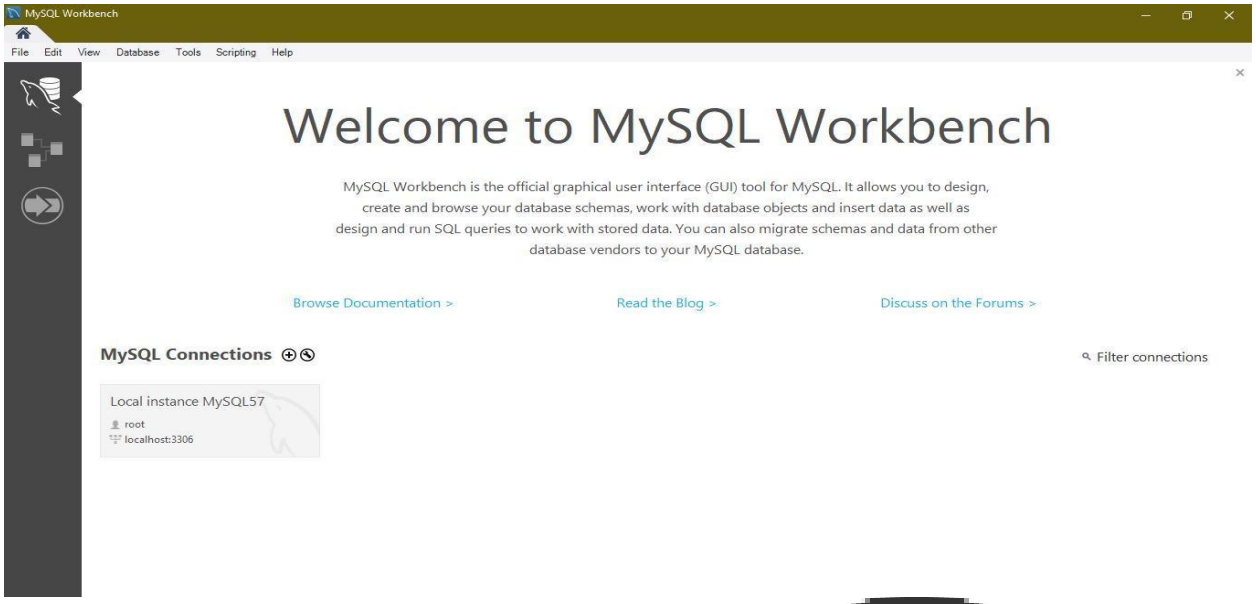


Step 27: After the successful installation of MySQL, two windows will open.

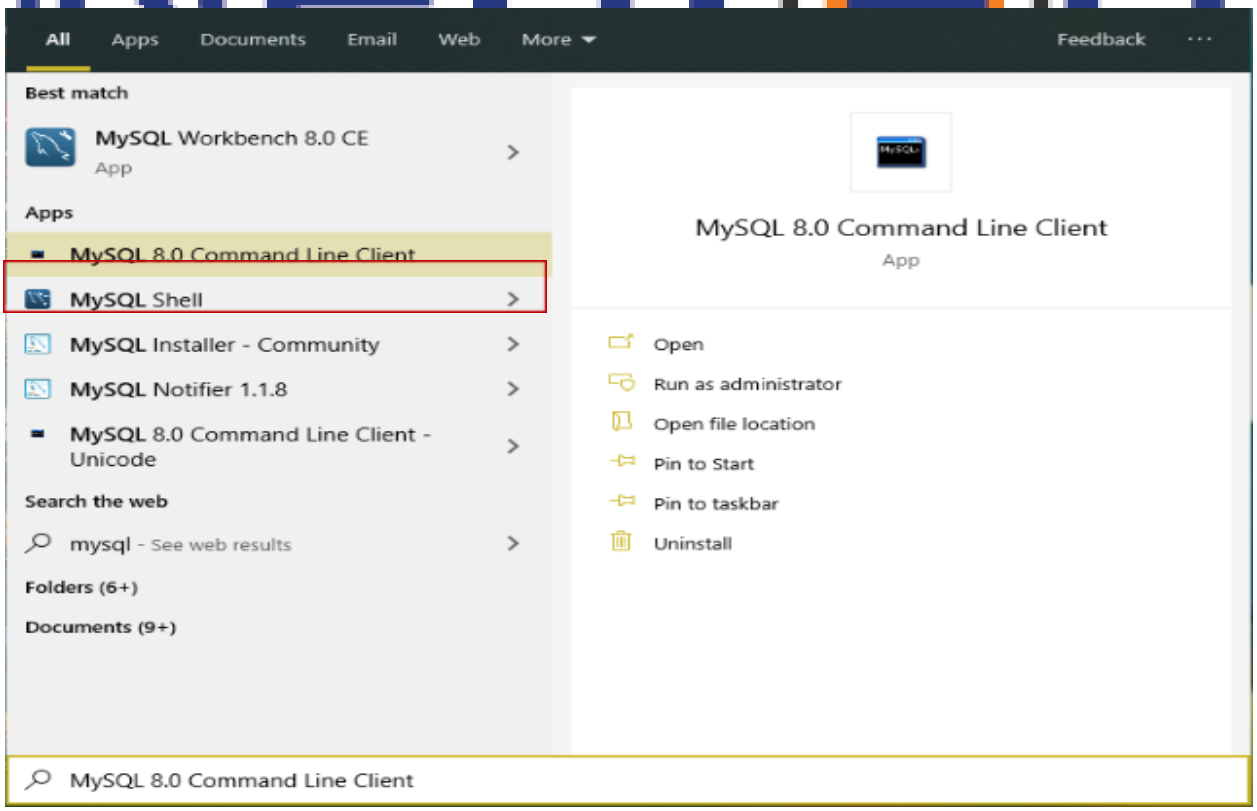
- MySQL Shell
- MySQL WorkBench



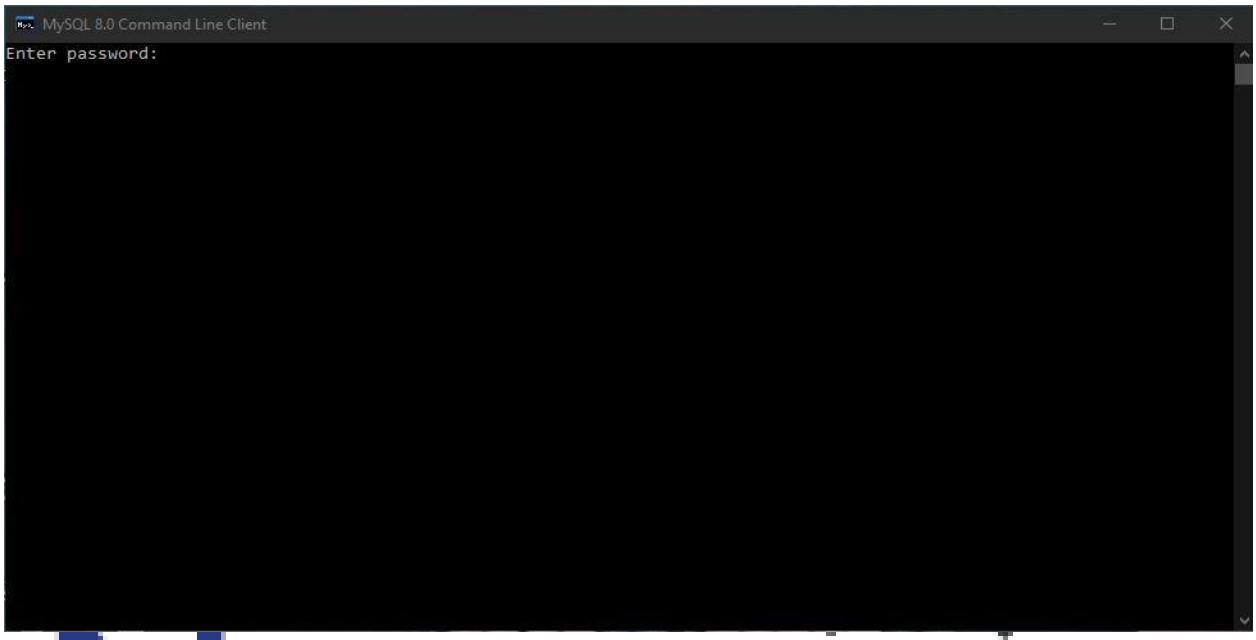
MySQL WorkBench will tell about database connectivity and other features of MySQL.



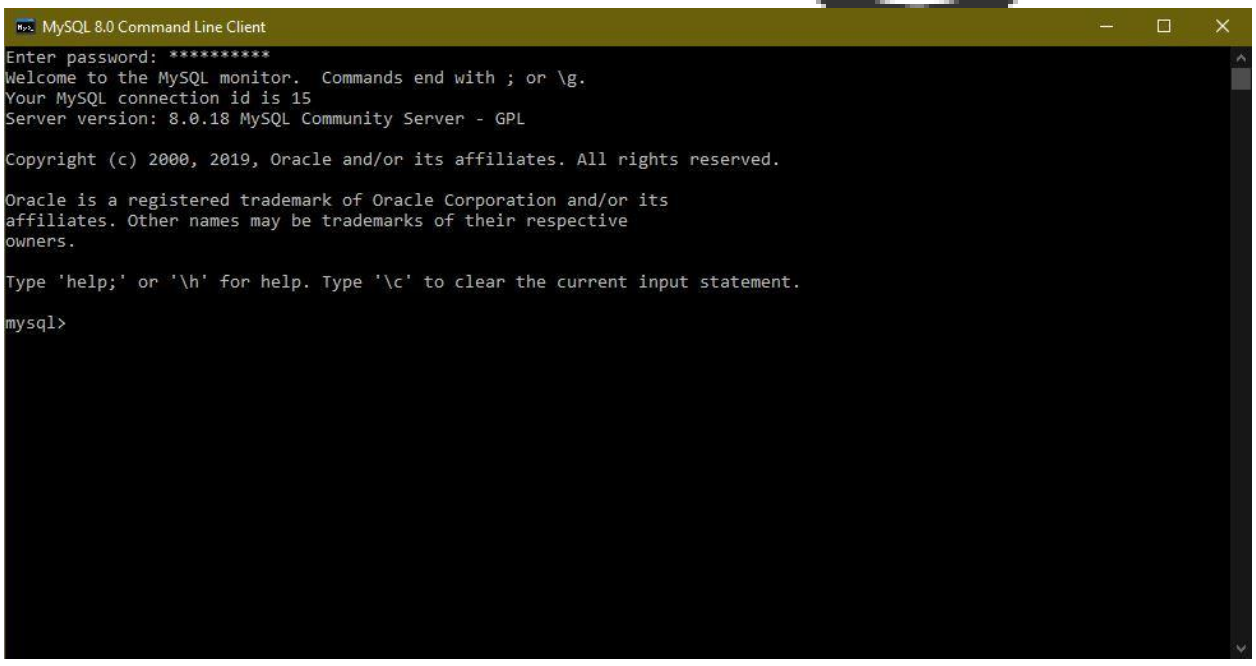
Step 28: Click on window Button and search for Open MySQL Command.



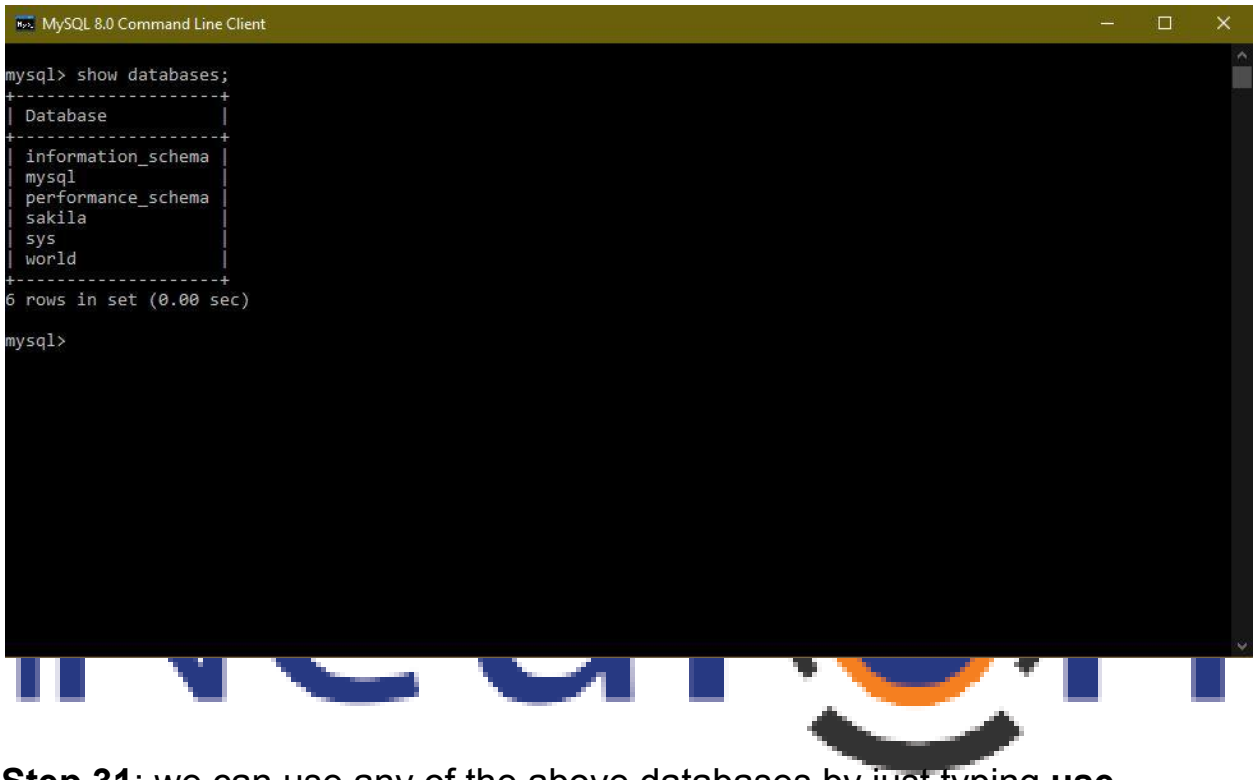
Step 29: Open MySQL Command-line Client and enter the password.



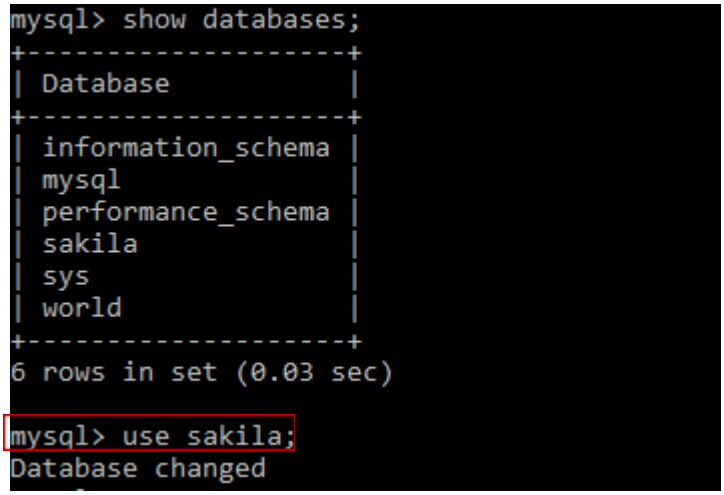
Step 29: After entering the password, your MySQL client will get connected with MySQL.



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**



3. 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**.

```
mysql> select first_name,last_name from customer;
+--------------+---------------+
| first_name   | last_name     |
+--------------+---------------+
| MARY         | SMITH         |
| PATRICIA     | JOHNSON       |
| LINDA        | WILLIAMS      |
| BARBARA      | JONES         |
| ELIZABETH    | BROWN         |
| JENNIFER     | DAVIS         |
| MARIA        | MILLER        |
| SUSAN        | WILSON        |
| MARGARET     | MOORE         |
| DOROTHY      | TAYLOR        |
| LISA         | ANDERSON      |
| NANCY        | THOMAS        |
| KAREN        | JACKSON       |
| BETTY        | WHITE         |
| HELEN        | HARRIS        |
| SANDRA       | MARTIN        |
| DONNA        | THOMPSON      |
| CAROL        | GARCIA        |
| RUTH         | MARTINEZ      |
| SHARON       | ROBINSON      |
| MICHELLE     | CLARK         |
| LAURA        | RODRIGUEZ     |
| SARAH        | LEWIS         |
| KIMBERLY     | LEE           |
```

To select all columns, use *

```
SELECT *
FROM customer;
```

```
mysql> select * from customer;
+-------------+----------+------------+-------------+----------------------------------------+------------+--------+---------------------+---------------------+
| customer_id | store_id | first_name | last_name   | email                                  | address_id | active | create_date         | last_update         |
+-------------+----------+------------+-------------+----------------------------------------+------------+--------+---------------------+---------------------+
|           1 |        1 | MARY       | SMITH       | MARY.SMITH@sakilacustomer.org          |          5 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|           2 |        1 | PATRICIA   | JOHNSON     | PATRICIA.JOHNSON@sakilacustomer.org    |          6 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|           3 |        1 | LINDA      | WILLIAMS    | LINDA.WILLIAMS@sakilacustomer.org      |          7 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|           4 |        2 | BARBARA    | JONES       | BARBARA.JONES@sakilacustomer.org       |          8 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|           5 |        1 | ELIZABETH  | BROWN       | ELIZABETH.BROWN@sakilacustomer.org     |          9 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|           6 |        2 | JENNIFER   | DAVIS       | JENNIFER.DAVIS@sakilacustomer.org      |         10 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|           7 |        1 | MARIA      | MILLER      | MARIA.MILLER@sakilacustomer.org        |         11 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|           8 |        2 | SUSAN      | WILSON      | SUSAN.WILSON@sakilacustomer.org        |         12 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|           9 |        2 | MARGARET   | MOORE       | MARGARET.MOORE@sakilacustomer.org      |         13 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          10 |        1 | DOROTHY    | TAYLOR      | DOROTHY.TAYLOR@sakilacustomer.org      |         14 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          11 |        2 | LISA       | ANDERSON    | LISA.ANDERSON@sakilacustomer.org       |         15 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          12 |        1 | NANCY      | THOMAS      | NANCY.THOMAS@sakilacustomer.org        |         16 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          13 |        2 | KAREN      | JACKSON     | KAREN.JACKSON@sakilacustomer.org       |         17 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          14 |        2 | BETTY      | WHITE       | BETTY.WHITE@sakilacustomer.org         |         18 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          15 |        1 | HELEN      | HARRIS      | HELEN.HARRIS@sakilacustomer.org        |         19 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          16 |        2 | SANDRA     | MARTIN      | SANDRA.MARTIN@sakilacustomer.org       |         20 |      0 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          17 |        1 | DONNA      | THOMPSON    | DONNA.THOMPSON@sakilacustomer.org      |         21 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          18 |        2 | CAROL      | GARCIA      | CAROL.GARCIA@sakilacustomer.org        |         22 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          19 |        1 | RUTH       | MARTINEZ    | RUTH.MARTINEZ@sakilacustomer.org       |         23 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          20 |        2 | SHARON     | ROBINSON    | SHARON.ROBINSON@sakilacustomer.org     |         24 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
```

The SQL SELECT DISTINCT

The SELECT DISTINCT statement is to return the different values.

```
SELECT DISTINCT first_name
FROM  customer;
```

```
mysql> select distinct first_name from customer;
+-------------+
| first_name  |
+-------------+
| MARY        |
| PATRICIA    |
| LINDA       |
| BARBARA     |
| ELIZABETH   |
| JENNIFER    |
| MARIA       |
| SUSAN       |
| MARGARET    |
| DOROTHY     |
| LISA        |
| NANCY       |
| KAREN       |
| BETTY       |
| HELEN       |
| SANDRA      |
| DONNA       |
```

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’;

```
mysql> select first_name from customer where last_name = 'perry';
+-------------+
| first_name |
+-------------+
| SARA       |
+-------------+
1 row in set (0.03 sec)
```

When we access the Numeric field

SELECT first_name , last_name FROM customer WHERE active = 0;

```
mysql> select first_name, last_name from customer where active = 0;
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| SANDRA     | MARTIN    |
| JUDITH     | COX       |
| SHEILA     | WELLS     |
| ERICA      | MATTHEWS  |
| HEIDI      | LARSON    |
| PENNY      | NEAL      |
| KENNETH    | GOODEN    |
| HARRY      | ARCE      |
| NATHAN     | RUNYON    |
| THEODORE   | CULP      |
| MAURICE    | CRAWLEY   |
| BEN        | EASTER    |
| CHRISTIAN  | JUNG      |
| JIMMIE     | EGGLESTON |
| TERRANCE   | ROUSH     |
+------------+-----------+
15 rows in set (0.00 sec)
```

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'
```

```
mysql> select first_name,          email, address_id from customer where first_name = 'IAN' and last_name = 'STILL';
+------------+-------+------------+
| first_name | email | address_id |
+------------+-------+------------+
| IAN        | STILL |        567 |
+------------+-------+------------+
1 row in set (0.00 sec)
```

A WHERE clause with OR:

```
UPDATE customer
SET first_name = 'jingle'
WHERE last_name = ' GREY';
```

```
mysql> update customer set first_name = 'jingle'
    -> where last_name = 'GREY';
Query OK, 1 row affected (0.30 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from customer where address_id = 586;
+-------------+----------+------------+-----------+-----------------------------+------------+--------+---------------------+---------------------+
| customer_id | store_id | first_name | last_name | email                       | address_id | active | create_date         | last_update         |
+-------------+----------+------------+-----------+-----------------------------+------------+--------+---------------------+---------------------+
|         580 |        1 | jingle     | GREY      | ROSS.GREY@sakilacustomer.org |        586 |      1 | 2006-02-14 22:04:37 | 2019-11-29 10:50:42 |
+-------------+----------+------------+-----------+-----------------------------+------------+--------+---------------------+---------------------+
1 row in set (0.01 sec)
```

A WHERE clause with NOT:

Select store_id, first_name,last_name, email, address_id FROM customer

WHERE NOT store_id = 2;

```
mysql> Select  store_id, first_name,last_name, email, address_id FROM customer
    -> WHERE NOT store_id = 2;
+----------+-------------+-------------+----------------------------------------+------------+
| store_id | first_name  | last_name   | email                                  | address_id |
+----------+-------------+-------------+----------------------------------------+------------+
|        1 | MARY        | SMITH       | MARY.SMITH@sakilacustomer.org          |          5 |
|        1 | PATRICIA    | JOHNSON     | PATRICIA.JOHNSON@sakilacustomer.org    |          6 |
|        1 | LINDA       | WILLIAMS    | LINDA.WILLIAMS@sakilacustomer.org      |          7 |
|        1 | ELIZABETH   | BROWN       | ELIZABETH.BROWN@sakilacustomer.org     |          9 |
|        1 | MARIA       | MILLER      | MARIA.MILLER@sakilacustomer.org        |         11 |
|        1 | DOROTHY     | TAYLOR      | DOROTHY.TAYLOR@sakilacustomer.org      |         14 |
|        1 | NANCY       | THOMAS      | NANCY.THOMAS@sakilacustomer.org        |         16 |
```

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;
```

```
mysql> select first_name, last_name, email from customer order by first_name desc;
+-------------+--------------+-----------------------------------------+
| first_name  | last_name    | email                                   |
+-------------+--------------+-----------------------------------------+
| ZACHARY     | HITE         | ZACHARY.HITE@sakilacustomer.org         |
| YVONNE      | WATKINS      | YVONNE.WATKINS@sakilacustomer.org       |
| YOLANDA     | WEAVER       | YOLANDA.WEAVER@sakilacustomer.org       |
| WILMA       | RICHARDS     | WILMA.RICHARDS@sakilacustomer.org       |
| WILLIE      | HOWELL       | WILLIE.HOWELL@sakilacustomer.org        |
| WILLIE      | MARKHAM      | WILLIE.MARKHAM@sakilacustomer.org       |
| WILLIAM     | SATTERFIELD  | WILLIAM.SATTERFIELD@sakilacustomer.org  |
| WILLARD     | LUMPKIN      | WILLARD.LUMPKIN@sakilacustomer.org      |
```

Order By in Ascending order

```
SELECT first_name, last_name,email
FROM customer
ORDER BY first_name ASC;
```

```
mysql> select first_name, last_name, email from customer order by first_name asc;
+-------------+--------------+-----------------------------------------+
| first_name  | last_name    | email                                   |
+-------------+--------------+-----------------------------------------+
| AARON       | SELBY        | AARON.SELBY@sakilacustomer.org          |
| ADAM        | GOOCH        | ADAM.GOOCH@sakilacustomer.org           |
| ADRIAN      | CLARY        | ADRIAN.CLARY@sakilacustomer.org         |
| AGNES       | BISHOP       | AGNES.BISHOP@sakilacustomer.org         |
| ALAN        | KAHN         | ALAN.KAHN@sakilacustomer.org            |
| ALBERT      | CROUSE       | ALBERT.CROUSE@sakilacustomer.org        |
| ALBERTO     | HENNING      | ALBERTO.HENNING@sakilacustomer.org      |
| ALEX        | GRESHAM      | ALEX.GRESHAM@sakilacustomer.org         |
| ALEXANDER   | FENNELL      | ALEXANDER.FENNELL@sakilacustomer.org    |
```

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;
```

```
mysql> select first_name,last_name,email from customer where first_name = 'AUSTIN' limit 20;
+------------+-----------+-----------------------------------+
| first_name | last_name | email                             |
+------------+-----------+-----------------------------------+
| AUSTIN     | CINTRON   | AUSTIN.CINTRON@sakilacustomer.org |
+------------+-----------+-----------------------------------+
1 row in set (0.00 sec)

mysql> select first_name,last_name,email from customer limit 20;
+------------+-----------+-----------------------------------+
| first_name | last_name | email                             |
+------------+-----------+-----------------------------------+
| MARY       | SMITH     | MARY.SMITH@sakilacustomer.org     |
| PATRICIA   | JOHNSON   | PATRICIA.JOHNSON@sakilacustomer.org |
| LINDA      | WILLIAMS  | LINDA.WILLIAMS@sakilacustomer.org |
| BARBARA    | JONES     | BARBARA.JONES@sakilacustomer.org  |
| ELIZABETH  | BROWN     | ELIZABETH.BROWN@sakilacustomer.org |
| JENNIFER   | DAVIS     | JENNIFER.DAVIS@sakilacustomer.org |
| MARIA      | MILLER    | MARIA.MILLER@sakilacustomer.org   |
| SUSAN      | WILSON    | SUSAN.WILSON@sakilacustomer.org   |
| MARGARET   | MOORE     | MARGARET.MOORE@sakilacustomer.org |
| DOROTHY    | TAYLOR    | DOROTHY.TAYLOR@sakilacustomer.org |
| LISA       | ANDERSON  | LISA.ANDERSON@sakilacustomer.org  |
| NANCY      | THOMAS    | NANCY.THOMAS@sakilacustomer.org   |
| KAREN      | JACKSON   | KAREN.JACKSON@sakilacustomer.org  |
| BETTY      | WHITE     | BETTY.WHITE@sakilacustomer.org    |
| HELEN      | HARRIS    | HELEN.HARRIS@sakilacustomer.org   |
| SANDRA     | MARTIN    | SANDRA.MARTIN@sakilacustomer.org  |
| DONNA      | THOMPSON  | DONNA.THOMPSON@sakilacustomer.org |
| CAROL      | GARCIA    | CAROL.GARCIA@sakilacustomer.org   |
| RUTH       | MARTINEZ  | RUTH.MARTINEZ@sakilacustomer.org  |
| SHARON     | ROBINSON  | SHARON.ROBINSON@sakilacustomer.org |
+------------+-----------+-----------------------------------+
20 rows in set (0.00 sec)
```

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;
```

```
mysql> select min(address_id) from customer;
+-----------------+
| min(address_id) |
+-----------------+
|               5 |
+-----------------+
1 row in set (0.03 sec)
```

MAX() Syntax

SELECT MAX(address_id)

FROM customer;

```
mysql> select max(address_id) from customer;
+-----------------+
| max(address_id) |
+-----------------+
|             605 |
+-----------------+
1 row in set (0.00 sec)
```

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;

```
mysql> select count(email) from customer;
+--------------+
| count(email) |
+--------------+
|          599 |
+--------------+
1 row in set (0.00 sec)
```

AVG() Syntax

SELECT AVG(active)

FROM customer;

```
mysql> select avg(active) from customer;
+-------------+
| avg(active) |
+-------------+
|      0.9750 |
+-------------+
1 row in set (0.03 sec)
```


SUM() Syntax

SELECT SUM(active)

FROM customer

```
mysql> select sum(active) from customer;
+-------------+
| sum(active) |
+-------------+
|         584 |
+-------------+
1 row in set (0.00 sec)
```

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%';


```
mysql> select * from customer where first_name like 'D%';
+-------------+----------+------------+-----------+--------------------------------+------------+--------+---------------------+---------------------+
| customer_id | store_id | first_name | last_name | email                          | address_id | active | create_date         | last_update         |
+-------------+----------+------------+-----------+--------------------------------+------------+--------+---------------------+---------------------+
|          10 |        1 | DOROTHY    | TAYLOR    | DOROTHY.TAYLOR@sakilacustomer.org |         14 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          17 |        1 | DONNA      | THOMPSON  | DONNA.THOMPSON@sakilacustomer.org |         21 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          25 |        1 | DEBORAH    | WALKER    | DEBORAH.WALKER@sakilacustomer.org |         29 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          39 |        1 | DEBRA      | NELSON    | DEBRA.NELSON@sakilacustomer.org |         43 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          50 |        1 | DIANE      | COLLINS   | DIANE.COLLINS@sakilacustomer.org |         54 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          55 |        2 | DORIS      | REED      | DORIS.REED@sakilacustomer.org |         59 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          74 |        1 | DENISE     | KELLY     | DENISE.KELLY@sakilacustomer.org |         78 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          96 |        1 | DIANA      | ALEXANDER | DIANA.ALEXANDER@sakilacustomer.org |        100 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|         105 |        1 | DAWN       | SULLIVAN  | DAWN.SULLIVAN@sakilacustomer.org |        109 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|         141 |        1 | DEBBIE     | REYES     | DEBBIE.REYES@sakilacustomer.org |        145 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|         150 |        2 | DANIELLE   | DANIELS   | DANIELLE.DANIELS@sakilacustomer.org |        154 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|         157 |        2 | DARLENE    | ROSE      | DARLENE.ROSE@sakilacustomer.org |        161 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|         171 |        2 | DOLORES    | WAGNER    | DOLORES.WAGNER@sakilacustomer.org |        175 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|         179 |        1 | DANA       | HART      | DANA.HART@sakilacustomer.org |        183 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|         222 |        2 | DELORES    | HANSEN    | DELORES.HANSEN@sakilacustomer.org |        226 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|         249 |        2 | DORA       | MEDINA    | DORA.MEDINA@sakilacustomer.org |        253 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|         261 |        1 | DEANNA     | BYRD      | DEANNA.BYRD@sakilacustomer.org |        266 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|         279 |        2 | DIANNE     | SHELTON   | DIANNE.SHELTON@sakilacustomer.org |        284 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         295 |        1 | DAISY      | BATES     | DAISY.BATES@sakilacustomer.org |        300 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         304 |        2 | DAVID      | ROYAL     | DAVID.ROYAL@sakilacustomer.org |        309 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         310 |        2 | DANIEL     | CABRAL    | DANIEL.CABRAL@sakilacustomer.org |        315 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
```

Selects all columns of the customer with a first_name Ending with "E":

SELECT * FROM customer

WHERE first_name LIKE '%E';

```
mysql> select * from customer where first_name like '%E';
```

customer_id	store_id	first_name	last_name	email	address_id	active	create_date	last_update
21	1	MICHELLE	CLARK	MICHELLE.CLARK@sakilacustomer.org	25	1	2006-02-14 22:04:36	2006-02-15 04:57:20
41	1	STEPHANIE	MITCHELL	STEPHANIE.MITCHELL@sakilacustomer.org	45	1	2006-02-14 22:04:36	2006-02-15 04:57:20
43	2	CHRISTINE	ROBERTS	CHRISTINE.ROBERTS@sakilacustomer.org	47	1	2006-02-14 22:04:36	2006-02-15 04:57:20
44	1	MARIE	TURNER	MARIE.TURNER@sakilacustomer.org	48	1	2006-02-14 22:04:36	2006-02-15 04:57:20
46	2	CATHERINE	CAMPBELL	CATHERINE.CAMPBELL@sakilacustomer.org	50	1	2006-02-14 22:04:36	2006-02-15 04:57:20
49	2	JOYCE	EDWARDS	JOYCE.EDWARDS@sakilacustomer.org	53	1	2006-02-14 22:04:36	2006-02-15 04:57:20
50	1	DIANE	COLLINS	DIANE.COLLINS@sakilacustomer.org	54	1	2006-02-14 22:04:36	2006-02-15 04:57:20
51	1	ALICE	STEWART	ALICE.STEWART@sakilacustomer.org	55	1	2006-02-14 22:04:36	2006-02-15 04:57:20
52	1	JULIE	SANCHEZ	JULIE.SANCHEZ@sakilacustomer.org	56	1	2006-02-14 22:04:36	2006-02-15 04:57:20
61	2	KATHERINE	RIVERA	KATHERINE.RIVERA@sakilacustomer.org	65	1	2006-02-14 22:04:36	2006-02-15 04:57:20
65	2	ROSE	HOWARD	ROSE.HOWARD@sakilacustomer.org	69	1	2006-02-14 22:04:36	2006-02-15 04:57:20
66	2	JANICE	WARD	JANICE.WARD@sakilacustomer.org	70	1	2006-02-14 22:04:36	2006-02-15 04:57:20
68	1	NICOLE	PETERSON	NICOLE.PETERSON@sakilacustomer.org	72	1	2006-02-14 22:04:36	2006-02-15 04:57:20
74	1	DENISE	KELLY	DENISE.KELLY@sakilacustomer.org	78	1	2006-02-14 22:04:36	2006-02-15 04:57:20
76	2	IRENE	PRICE	IRENE.PRICE@sakilacustomer.org	80	1	2006-02-14 22:04:36	2006-02-15 04:57:20
77	2	JANE	BENNETT	JANE.BENNETT@sakilacustomer.org	81	1	2006-02-14 22:04:36	2006-02-15 04:57:20
83	1	LOUISE	JENKINS	LOUISE.JENKINS@sakilacustomer.org	87	1	2006-02-14 22:04:36	2006-02-15 04:57:20
85	2	ANNE	POWELL	ANNE.POWELL@sakilacustomer.org	89	1	2006-02-14 22:04:36	2006-02-15 04:57:20
86	2	JACQUELINE	LONG	JACQUELINE.LONG@sakilacustomer.org	90	1	2006-02-14 22:04:36	2006-02-15 04:57:20
88	2	BONNIE	HUGHES	BONNIE.HUGHES@sakilacustomer.org	92	1	2006-02-14 22:04:36	2006-02-15 04:57:20
97	2	ANNIE	RUSSELL	ANNIE.RUSSELL@sakilacustomer.org	101	1	2006-02-14 22:04:36	2006-02-15 04:57:20
106	1	CONNIE	WALLACE	CONNIE.WALLACE@sakilacustomer.org	110	1	2006-02-14 22:04:36	2006-02-15 04:57:20

Selects all columns of the customer with a first_name that have "or" in any position.

```
SELECT * FROM customer
WHERE first_name LIKE '%or%';
```

```
mysql> select * from customer where first_name like '%or%';
```

customer_id	store_id	first_name	last_name	email	address_id	active	create_date	last_update
10	1	DOROTHY	TAYLOR	DOROTHY.TAYLOR@sakilacustomer.org	14	1	2006-02-14 22:04:36	2006-02-15 04:57:20
25	1	DEBORAH	WALKER	DEBORAH.WALKER@sakilacustomer.org	29	1	2006-02-14 22:04:36	2006-02-15 04:57:20
55	2	DORIS	REED	DORIS.REED@sakilacustomer.org	59	1	2006-02-14 22:04:36	2006-02-15 04:57:20
56	1	GLORIA	COOK	GLORIA.COOK@sakilacustomer.org	60	1	2006-02-14 22:04:36	2006-02-15 04:57:20
78	1	LORI	WOOD	LORI.WOOD@sakilacustomer.org	82	1	2006-02-14 22:04:36	2006-02-15 04:57:20
94	1	NORMA	GONZALES	NORMA.GONZALES@sakilacustomer.org	98	1	2006-02-14 22:04:36	2006-02-15 04:57:20
107	1	FLORENCE	WOODS	FLORENCE.WOODS@sakilacustomer.org	111	1	2006-02-14 22:04:36	2006-02-15 04:57:20
116	1	VICTORIA	GIBSON	VICTORIA.GIBSON@sakilacustomer.org	120	1	2006-02-14 22:04:36	2006-02-15 04:57:20
128	1	MARJORIE	TUCKER	MARJORIE.TUCKER@sakilacustomer.org	132	1	2006-02-14 22:04:36	2006-02-15 04:57:20
148	1	ELEANOR	HUNT	ELEANOR.HUNT@sakilacustomer.org	152	1	2006-02-14 22:04:36	2006-02-15 04:57:20
165	2	LORRAINE	STEPHENS	LORRAINE.STEPHENS@sakilacustomer.org	169	1	2006-02-14 22:04:36	2006-02-15 04:57:20
171	2	DOLORES	WAGNER	DOLORES.WAGNER@sakilacustomer.org	175	1	2006-02-14 22:04:36	2006-02-15 04:57:20
189	1	LORETTA	CARPENTER	LORETTA.CARPENTER@sakilacustomer.org	193	1	2006-02-14 22:04:36	2006-02-15 04:57:20
222	2	DELORES	HANSEN	DELORES.HANSEN@sakilacustomer.org	226	1	2006-02-14 22:04:36	2006-02-15 04:57:20
231	1	GEORGIA	JACOBS	GEORGIA.JACOBS@sakilacustomer.org	235	1	2006-02-14 22:04:36	2006-02-15 04:57:20
249	2	DORA	MEDINA	DORA.MEDINA@sakilacustomer.org	253	1	2006-02-14 22:04:36	2006-02-15 04:57:20

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';
```

```
mysql> select * from customer where first_name like 'a%o';
+-------------+----------+------------+-----------+---------------------------------+------------+--------+---------------------+---------------------+
| customer_id | store_id | first_name | last_name | email                           | address_id | active | create_date         | last_update         |
+-------------+----------+------------+-----------+---------------------------------+------------+--------+---------------------+---------------------+
|         398 |        1 | ANTONIO    | MEEK      | ANTONIO.MEEK@sakilacustomer.org |        403 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         556 |        2 | ARMANDO    | GRUBER    | ARMANDO.GRUBER@sakilacustomer.org |        562 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         567 |        2 | ALFREDO    | MCADAMS   | ALFREDO.MCADAMS@sakilacustomer.org |        573 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         568 |        2 | ALBERTO    | HENNING   | ALBERTO.HENNING@sakilacustomer.org |        574 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
+-------------+----------+------------+-----------+---------------------------------+------------+--------+---------------------+---------------------+
4 rows in set (0.00 sec)
```

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__%';
```

```
mysql> select * from customer where first_name like 'a_____%';
+-------------+----------+------------+-----------+---------------------------------+------------+--------+---------------------+---------------------+
| customer_id | store_id | first_name | last_name | email                           | address_id | active | create_date         | last_update         |
+-------------+----------+------------+-----------+---------------------------------+------------+--------+---------------------+---------------------+
|         175 |        1 | ANNETTE    | OLSON     | ANNETTE.OLSON@sakilacustomer.org |        179 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|         228 |        2 | ALLISON    | STANLEY   | ALLISON.STANLEY@sakilacustomer.org |        232 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|         320 |        2 | ANTHONY    | SCHWAB    | ANTHONY.SCHWAB@sakilacustomer.org |        325 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         398 |        1 | ANTONIO    | MEEK      | ANTONIO.MEEK@sakilacustomer.org |        403 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         439 |        2 | ALEXANDER  | FENNELL   | ALEXANDER.FENNELL@sakilacustomer.org |        444 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         556 |        2 | ARMANDO    | GRUBER    | ARMANDO.GRUBER@sakilacustomer.org |        562 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         567 |        2 | ALFREDO    | MCADAMS   | ALFREDO.MCADAMS@sakilacustomer.org |        573 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         568 |        2 | ALBERTO    | HENNING   | ALBERTO.HENNING@sakilacustomer.org |        574 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
+-------------+----------+------------+-----------+---------------------------------+------------+--------+---------------------+---------------------+
8 rows in set (0.00 sec)
```

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);

```
mysql> SELECT * FROM customer
    -> where customer_id IN(1,2,3);
+-------------+----------+------------+-----------+----------------------------------+------------+--------+---------------------+---------------------+
| customer_id | store_id | first_name | last_name | email                            | address_id | active | create_date         | last_update         |
+-------------+----------+------------+-----------+----------------------------------+------------+--------+---------------------+---------------------+
|           1 |        1 | MARY       | SMITH     | MARY.SMITH@sakilacustomer.org    |          5 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|           2 |        1 | PATRICIA   | JOHNSON   | PATRICIA.JOHNSON@sakilacustomer.org |          6 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|           3 |        1 | LINDA      | WILLIAMS  | LINDA.WILLIAMS@sakilacustomer.org |          7 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
+-------------+----------+------------+-----------+----------------------------------+------------+--------+---------------------+---------------------+
3 rows in set (0.00 sec)
```

Selects all the columns of customer whose customer_id in (1,2,3):

SELECT * FROM customer

WHERE cutomer_id NOT IN (1,2,3);

```
mysql> SELECT * FROM customer
    -> where customer_id NOT IN(1,2,3);
+-------------+----------+------------+-----------+----------------------------------+------------+--------+---------------------+---------------------+
| customer_id | store_id | first_name | last_name | email                            | address_id | active | create_date         | last_update         |
+-------------+----------+------------+-----------+----------------------------------+------------+--------+---------------------+---------------------+
|           4 |        2 | BARBARA    | JONES     | BARBARA.JONES@sakilacustomer.org |          8 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|           5 |        1 | ELIZABETH  | BROWN     | ELIZABETH.BROWN@sakilacustomer.org |          9 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|           6 |        2 | JENNIFER   | DAVIS     | JENNIFER.DAVIS@sakilacustomer.org |         10 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|           7 |        1 | MARIA      | MILLER    | MARIA.MILLER@sakilacustomer.org  |         11 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|           8 |        2 | SUSAN      | WILSON    | SUSAN.WILSON@sakilacustomer.org  |         12 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|           9 |        2 | MARGARET   | MOORE     | MARGARET.MOORE@sakilacustomer.org |         13 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          10 |        1 | DOROTHY    | TAYLOR    | DOROTHY.TAYLOR@sakilacustomer.org |         14 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          11 |        2 | LISA       | ANDERSON  | LISA.ANDERSON@sakilacustomer.org |         15 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          12 |        1 | NANCY      | THOMAS    | NANCY.THOMAS@sakilacustomer.org  |         16 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          13 |        2 | KAREN      | JACKSON   | KAREN.JACKSON@sakilacustomer.org |         17 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          14 |        2 | BETTY      | WHITE     | BETTY.WHITE@sakilacustomer.org   |         18 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          15 |        1 | HELEN      | HARRIS    | HELEN.HARRIS@sakilacustomer.org  |         19 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          16 |        2 | SANDRA     | MARTIN    | SANDRA.MARTIN@sakilacustomer.org |         20 |      0 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
```

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;

```
mysql> SELECT * FROM customer
    -> where customer_id between 1 and 20;
+-------------+----------+------------+-----------+----------------------------------+------------+--------+---------------------+---------------------+
| customer_id | store_id | first_name | last_name | email                            | address_id | active | create_date         | last_update         |
+-------------+----------+------------+-----------+----------------------------------+------------+--------+---------------------+---------------------+
|           1 |        1 | MARY       | SMITH     | MARY.SMITH@sakilacustomer.org    |          5 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|           2 |        1 | PATRICIA   | JOHNSON   | PATRICIA.JOHNSON@sakilacustomer.org |          6 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|           3 |        1 | LINDA      | WILLIAMS  | LINDA.WILLIAMS@sakilacustomer.org |          7 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|           4 |        2 | BARBARA    | JONES     | BARBARA.JONES@sakilacustomer.org |          8 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|           5 |        1 | ELIZABETH  | BROWN     | ELIZABETH.BROWN@sakilacustomer.org |          9 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|           6 |        2 | JENNIFER   | DAVIS     | JENNIFER.DAVIS@sakilacustomer.org |         10 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|           7 |        1 | MARIA      | MILLER    | MARIA.MILLER@sakilacustomer.org  |         11 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|           8 |        2 | SUSAN      | WILSON    | SUSAN.WILSON@sakilacustomer.org  |         12 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|           9 |        2 | MARGARET   | MOORE     | MARGARET.MOORE@sakilacustomer.org |         13 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          10 |        1 | DOROTHY    | TAYLOR    | DOROTHY.TAYLOR@sakilacustomer.org |         14 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          11 |        2 | LISA       | ANDERSON  | LISA.ANDERSON@sakilacustomer.org |         15 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          12 |        1 | NANCY      | THOMAS    | NANCY.THOMAS@sakilacustomer.org  |         16 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          13 |        2 | KAREN      | JACKSON   | KAREN.JACKSON@sakilacustomer.org |         17 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          14 |        2 | BETTY      | WHITE     | BETTY.WHITE@sakilacustomer.org   |         18 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          15 |        1 | HELEN      | HARRIS    | HELEN.HARRIS@sakilacustomer.org  |         19 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          16 |        2 | SANDRA     | MARTIN    | SANDRA.MARTIN@sakilacustomer.org |         20 |      0 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          17 |        1 | DONNA      | THOMPSON  | DONNA.THOMPSON@sakilacustomer.org |         21 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          18 |        2 | CAROL      | GARCIA    | CAROL.GARCIA@sakilacustomer.org  |         22 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          19 |        1 | RUTH       | MARTINEZ  | RUTH.MARTINEZ@sakilacustomer.org |         23 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
|          20 |        2 | SHARON     | ROBINSON  | SHARON.ROBINSON@sakilacustomer.org |         24 |      1 | 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
+-------------+----------+------------+-----------+----------------------------------+------------+--------+---------------------+---------------------+
20 rows in set (0.00 sec)
```

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;

```
mysql> SELECT * FROM customer
    -> where customer_id not between 1 and 570;
+-------------+----------+------------+-----------+----------------------------------------+------------+--------+---------------------+---------------------+
| customer_id | store_id | first_name | last_name | email                                  | address_id | active | create_date         | last_update         |
+-------------+----------+------------+-----------+----------------------------------------+------------+--------+---------------------+---------------------+
|         571 |        2 | JOHNNIE    | CHISHOLM  | JOHNNIE.CHISHOLM@sakilacustomer.org    |        577 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         572 |        1 | SIDNEY     | BURLESON  | SIDNEY.BURLESON@sakilacustomer.org     |        578 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         573 |        1 | BYRON      | BOX       | BYRON.BOX@sakilacustomer.org           |        579 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         574 |        2 | JULIAN     | VEST      | JULIAN.VEST@sakilacustomer.org         |        580 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         575 |        2 | ISAAC      | OGLESBY   | ISAAC.OGLESBY@sakilacustomer.org       |        581 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         576 |        2 | MORRIS     | MCCARTER  | MORRIS.MCCARTER@sakilacustomer.org     |        582 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         577 |        2 | CLIFTON    | MALCOLM   | CLIFTON.MALCOLM@sakilacustomer.org     |        583 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         578 |        2 | WILLARD    | LUMPKIN   | WILLARD.LUMPKIN@sakilacustomer.org     |        584 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         579 |        2 | DARYL      | LARUE     | DARYL.LARUE@sakilacustomer.org         |        585 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         580 |        1 | ROSS       | GREY      | ROSS.GREY@sakilacustomer.org           |        586 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         581 |        1 | VIRGIL     | WOFFORD   | VIRGIL.WOFFORD@sakilacustomer.org      |        587 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         582 |        2 | ANDY       | VANHORN   | ANDY.VANHORN@sakilacustomer.org        |        588 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         583 |        1 | MARSHALL   | THORN     | MARSHALL.THORN@sakilacustomer.org      |        589 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         584 |        2 | SALVADOR   | TEEL      | SALVADOR.TEEL@sakilacustomer.org       |        590 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         585 |        1 | PERRY      | SWAFFORD  | PERRY.SWAFFORD@sakilacustomer.org      |        591 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         586 |        1 | KIRK       | STCLAIR   | KIRK.STCLAIR@sakilacustomer.org        |        592 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         587 |        1 | SERGIO     | STANFIELD | SERGIO.STANFIELD@sakilacustomer.org    |        593 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         588 |        1 | MARION     | OCAMPO    | MARION.OCAMPO@sakilacustomer.org       |        594 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         589 |        1 | TRACY      | HERRMANN  | TRACY.HERRMANN@sakilacustomer.org      |        595 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         590 |        2 | SETH       | HANNON    | SETH.HANNON@sakilacustomer.org         |        596 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         591 |        1 | KENT       | ARSENAULT | KENT.ARSENAULT@sakilacustomer.org      |        597 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         592 |        1 | TERRANCE   | ROUSH     | TERRANCE.ROUSH@sakilacustomer.org      |        598 |      0 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         593 |        2 | RENE       | MCALISTER | RENE.MCALISTER@sakilacustomer.org      |        599 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         594 |        1 | EDUARDO    | HIATT     | EDUARDO.HIATT@sakilacustomer.org       |        600 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         595 |        1 | TERRENCE   | GUNDERSON | TERRENCE.GUNDERSON@sakilacustomer.org  |        601 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         596 |        1 | ENRIQUE    | FORSYTHE  | ENRIQUE.FORSYTHE@sakilacustomer.org    |        602 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         597 |        1 | FREDDIE    | DUGGAN    | FREDDIE.DUGGAN@sakilacustomer.org      |        603 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         598 |        1 | WADE       | DELVALLE  | WADE.DELVALLE@sakilacustomer.org       |        604 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
|         599 |        2 | AUSTIN     | CINTRON   | AUSTIN.CINTRON@sakilacustomer.org      |        605 |      1 | 2006-02-14 22:04:37 | 2006-02-15 04:57:20 |
+-------------+----------+------------+-----------+----------------------------------------+------------+--------+---------------------+---------------------+
29 rows in set (0.00 sec)
```

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:

```
mysql> select first_name as first, last_name as last from customer;
+-------------+--------------+
| first       | last         |
+-------------+--------------+
| MARY        | SMITH        |
| PATRICIA    | JOHNSON      |
| LINDA       | WILLIAMS     |
| BARBARA     | JONES        |
| ELIZABETH   | BROWN        |
| JENNIFER    | DAVIS        |
| MARIA       | MILLER       |
| SUSAN       | WILSON       |
| MARGARET    | MOORE        |
```

Alias Table Syntax

```
SELECT c.first_name, c.last_name
```

FROM customer AS c

Create an alias for the customer table

```
mysql> select c.first_name, c.last_name from customer as c;
+-------------+--------------+
| first_name  | last_name    |
+-------------+--------------+
| MARY        | SMITH        |
| PATRICIA    | JOHNSON      |
| LINDA       | WILLIAMS     |
| BARBARA     | JONES        |
| ELIZABETH   | BROWN        |
| JENNIFER    | DAVIS        |
| MARIA       | MILLER       |
| SUSAN       | WILSON       |
| MARGARET    | MOORE        |
| DOROTHY     | TAYLOR       |
```

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

```
mysql> select count(customer_id) from customer group by active;
+--------------------+
| count(customer_id) |
+--------------------+
|                584 |
|                 15 |
+--------------------+
2 rows in set (0.04 sec)
```

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;
```

```
mysql> select * from country group by (continent) having count(region) >6;
+------+---------------+---------------+---------------------------+-------------+------------+------------+----------------+----------+----------+------------------
---+-----------------------------------------+---------------------------+------------+---------+-------+
| Code | Name          | Continent     | Region                    | SurfaceArea | IndepYear | Population | LifeExpectancy | GNP      | GNPOld   | LocalName
   | GovernmentForm                          | HeadOfState               | Capital | Code2 |
+------+---------------+---------------+---------------------------+-------------+------------+------------+----------------+----------+----------+------------------
---+-----------------------------------------+---------------------------+------------+---------+-------+
| ABW  | Aruba         | North America | Caribbean                 |      193.00 |       NULL |     103000 |           78.4 |   828.00 |   793.00 | Aruba
   | Nonmetropolitan Territory of The Netherlands | Beatrix                   |        129 | AW      |
| AFG  | Afghanistan   | Asia          | Southern and Central Asia |   652090.00 |       1919 |   22720000 |           45.9 |  5976.00 |     NULL | Afganistan/Afqanes
tan | Islamic Emirate                         | Mohammad Omar             |          1 | AF      |
| AGO  | Angola        | Africa        | Central Africa            |  1246700.00 |       1975 |   12878000 |           38.3 |  6648.00 |  7984.00 | Angola
   | Republic                                | JosÃ© Eduardo dos Santos |         56 | AO      |
| ALB  | Albania       | Europe        | Southern Europe           |    28748.00 |       1912 |    3401200 |           71.6 |  3205.00 |  2500.00 | ShqipÃ«ria
   | Republic                                | Rexhep Mejdani            |         34 | AL      |
| ARG  | Argentina     | South America | South America             |  2780400.00 |       1816 |   37032000 |           75.1 | 340238.00 | 323310.00 | Argentina
   | Federal Republic                        | Fernando de la RÃºa       |         69 | AR      |
| ASM  | American Samoa | Oceania      | Polynesia                 |      199.00 |       NULL |      68000 |           75.1 |   334.00 |     NULL | Amerika Samoa
   | US Territory                            | George W. Bush            |         54 | AS      |
+------+---------------+---------------+---------------------------+-------------+------------+------------+----------------+----------+----------+------------------
---+-----------------------------------------+---------------------------+------------+---------+-------+
6 rows in set (0.03 sec)
```

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;
```

4. 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.

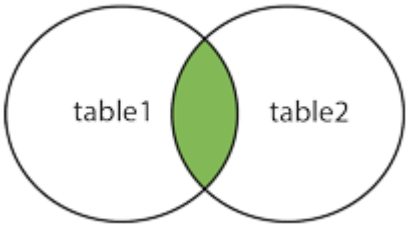


Fig. INNER JOIN

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;
```

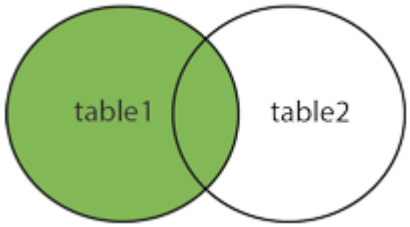
SELECT city.city_id, country.country, city.last_update, country.last_update FROM city

INNER JOIN country ON city.country_id = country.country_id

```
mysql> SELECT city.city_id, country.country, city.last_update,country.last_update
    -> FROM city
    -> INNER JOIN country ON city.country_id=country.country_id;
+---------+-------------------------------------+---------------------+---------------------+
| city_id | country                             | last_update         | last_update         |
+---------+-------------------------------------+---------------------+---------------------+
|     251 | Afghanistan                         | 2006-02-15 04:45:25 | 2006-02-15 04:44:00 |
|      59 | Algeria                             | 2006-02-15 04:45:25 | 2006-02-15 04:44:00 |
|      63 | Algeria                             | 2006-02-15 04:45:25 | 2006-02-15 04:44:00 |
|     483 | Algeria                             | 2006-02-15 04:45:25 | 2006-02-15 04:44:00 |
|     516 | American Samoa                      | 2006-02-15 04:45:25 | 2006-02-15 04:44:00 |
|      67 | Angola                              | 2006-02-15 04:45:25 | 2006-02-15 04:44:00 |
|     360 | Angola                              | 2006-02-15 04:45:25 | 2006-02-15 04:44:00 |
|     493 | Anguilla                            | 2006-02-15 04:45:25 | 2006-02-15 04:44:00 |
|      20 | Argentina                           | 2006-02-15 04:45:25 | 2006-02-15 04:44:00 |
|      43 | Argentina                           | 2006-02-15 04:45:25 | 2006-02-15 04:44:00 |
|      45 | Argentina                           | 2006-02-15 04:45:25 | 2006-02-15 04:44:00 |
|     128 | Argentina                           | 2006-02-15 04:45:25 | 2006-02-15 04:44:00 |
```

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;
```

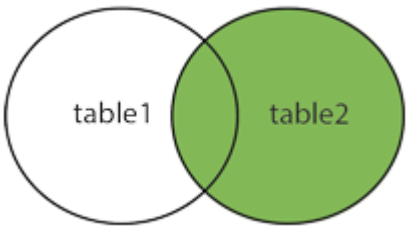
SELECT city.city_id, country.country, city.last_update, country.last_update FROM city

LEFT JOIN country ON city.country_id = country.country_id

```
mysql> SELECT city.city_id, country.country, city.last_update
    -> FROM country
    -> LEFT JOIN city ON city.country_id=country.country_id;
+---------+-----------------------------------------+---------------------+
| city_id | country                                 | last_update         |
+---------+-----------------------------------------+---------------------+
|     251 | Afghanistan                             | 2006-02-15 04:45:25 |
|      59 | Algeria                                 | 2006-02-15 04:45:25 |
|      63 | Algeria                                 | 2006-02-15 04:45:25 |
|     483 | Algeria                                 | 2006-02-15 04:45:25 |
|     516 | American Samoa                          | 2006-02-15 04:45:25 |
|      67 | Angola                                  | 2006-02-15 04:45:25 |
|     360 | Angola                                  | 2006-02-15 04:45:25 |
|     493 | Anguilla                                | 2006-02-15 04:45:25 |
```

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

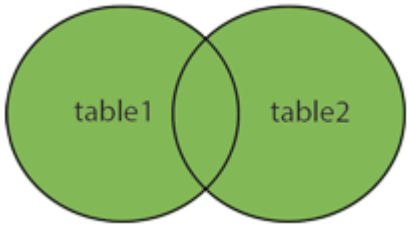
RIGHT JOIN country ON city.country_id = country.country_id

```
mysql> SELECT city.city_id, country.country, city.last_update
    -> FROM city
    -> RIGHT JOIN country ON city.country_id=country.country_id;
+---------+-------------------------------------+---------------------+
| city_id | country                             | last_update         |
+---------+-------------------------------------+---------------------+
|     251 | Afghanistan                         | 2006-02-15 04:45:25 |
|      59 | Algeria                             | 2006-02-15 04:45:25 |
|      63 | Algeria                             | 2006-02-15 04:45:25 |
|     483 | Algeria                             | 2006-02-15 04:45:25 |
|     516 | American Samoa                      | 2006-02-15 04:45:25 |
```

- **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;
```

5. 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;

```
mysql> create database testdb;
Query OK, 1 row affected (0.28 sec)
```

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

Show databases;

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sakila             |
| sys                |
| testdb             |
| world              |
+--------------------+
7 rows in set (0.06 sec)
```

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;
```

```
mysql> drop database testdb;
Query OK, 0 rows affected (0.78 sec)
```

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

```
SHOW databases;
```

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sakila             |
| sys                |
| world              |
+--------------------+
6 rows in set (0.00 sec)
```

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.

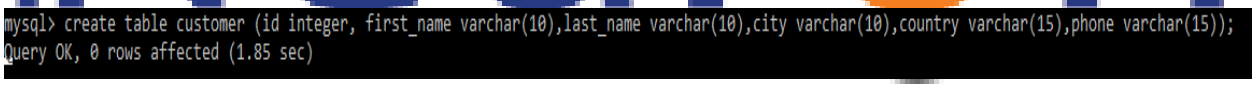
Syntax

```
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.)

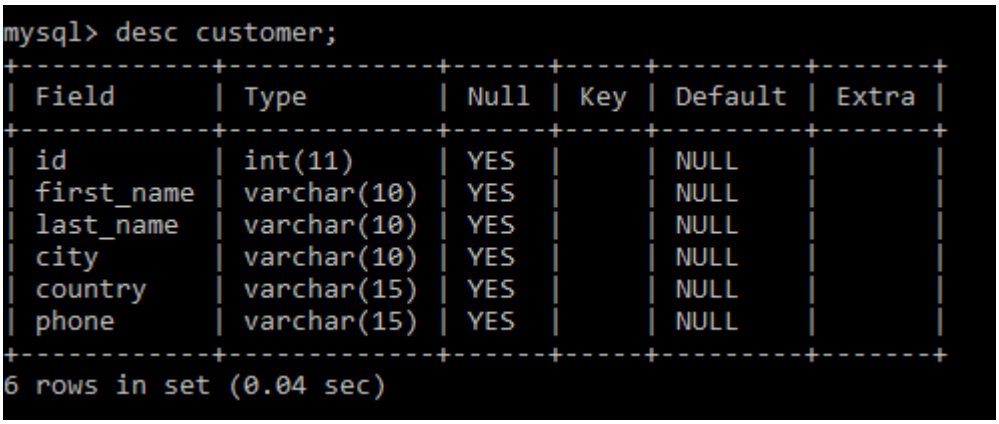
Let's create a customer table

- 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;

```
mysql> drop table customer;
Query OK, 0 rows affected (1.24 sec)

mysql> desc customer;
ERROR 1146 (42S02): Table 'testdb.customer' doesn't exist
```

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);

```
mysql> INSERT INTO customer (id,first_name,last_name,city,country,phone) VALUES(2,'Ana','Trujillo','México','Mexico','(5) 555-4729');
Query OK, 1 row affected (0.12 sec)

mysql> select * from customer;
+------+------------+-----------+--------+---------+--------------+
| id   | first_name | last_name | city   | country | phone        |
+------+------------+-----------+--------+---------+--------------+
|    2 | Ana        | Trujillo  | México | Mexico  | (5) 555-4729 |
+------+------------+-----------+--------+---------+--------------+
1 row in set (0.00 sec)
```

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);

```
mysql> select * from customer;
+------+------------+-----------+--------+---------+--------------+
| id   | first_name | last_name | city   | country | phone        |
+------+------------+-----------+--------+---------+--------------+
|    2 | Ana        | Trujillo  | México | Mexico  | (5) 555-4729 |
|    3 | Antonio    | Moreno    | México | Mexico  | (5) 555-3932 |
+------+------------+-----------+--------+---------+--------------+
2 rows in set (0.00 sec)
```

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;

```
mysql> select * from customer;
+------+------------+-----------+------------+---------+----------------+
| id   | first_name | last_name | city       | country | phone          |
+------+------------+-----------+------------+---------+----------------+
|    2 | Ana        | Trujillo  | México     | Mexico  | (5) 555-4729   |
|    3 | Antonio    | Moreno    | México     | Mexico  | (5) 555-3932   |
|    4 | Thomas     | Hardy     | London     | UK      | (171) 555-7788 |
|    5 | Christina  | Berglund  | Luleå      | Sweden  | 0921-12 34 65  |
|    6 | Hanna      | Moos      | Mannheim   | Germany | 0621-08460     |
|    7 | Frédérique | Citeaux   | Strasbourg | France  | 88.60.15.31    |
|    8 | Martín     | Sommer    | Madrid     | Spain   | (91) 555 22 82 |
|    9 | Laurence   | Lebihan   | Marseille  | France  | 91.24.45.40    |
|   10 | Elizabeth  | Lincoln   | Tsawassen  | Canada  | (604) 555-4729 |
|   11 | Victoria   | Ashworth  | London     | UK      | (171) 555-1212 |
+------+------------+-----------+------------+---------+----------------+
10 rows in set (0.00 sec)
```

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’)

```
mysql> INSERT INTO customer VALUES(11,'Victoria','Ashworth','London',NULL,'(171) 555-1212');
Query OK, 1 row affected (0.16 sec)

mysql> select * from customer;
+------+-------------+-----------+-------------+---------+-----------------+
| id   | first_name  | last_name | city        | country | phone           |
+------+-------------+-----------+-------------+---------+-----------------+
|    2 | Ana         | Trujillo  | México      | Mexico  | (5) 555-4729    |
|    3 | Antonio     | Moreno    | México      | Mexico  | (5) 555-3932    |
|    4 | Thomas      | Hardy     | London      | UK      | (171) 555-7788  |
|    5 | Christina   | Berglund  | Luleå       | Sweden  | 0921-12 34 65   |
|    6 | Hanna       | Moos      | Mannheim    | Germany | 0621-08460      |
|    7 | Frédérique  | Citeaux   | Strasbourg  | France  | 88.60.15.31     |
|    8 | Martín      | Sommer    | Madrid      | Spain   | (91) 555 22 82  |
|    9 | Laurence    | Lebihan   | Marseille   | France  | 91.24.45.40     |
|   10 | Elizabeth   | Lincoln   | Tsawassen   | Canada  | (604) 555-4729  |
|   11 | Victoria    | Ashworth  | London      | NULL    | (171) 555-1212  |
+------+-------------+-----------+-------------+---------+-----------------+
10 rows in set (0.00 sec)
```

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;
```

```
mysql> select * from customer where country IS NULL;
+------+------------+-----------+--------+---------+----------------+
| id   | first_name | last_name | city   | country | phone          |
+------+------------+-----------+--------+---------+----------------+
|   11 | Victoria   | Ashworth  | London | NULL    | (171) 555-1212 |
+------+------------+-----------+--------+---------+----------------+
1 row in set (0.00 sec)
```

IS NOT NULL Syntax

SELECT * FROM customer

WHERE country IS NOT NULL;

```
mysql> select * from customer where country IS NOT NULL;
+------+------------+-----------+------------+---------+----------------+
| id   | first_name | last_name | city       | country | phone          |
+------+------------+-----------+------------+---------+----------------+
|    2 | Ana        | Trujillo  | México     | Mexico  | (5) 555-4729   |
|    3 | Antonio    | Moreno    | México     | Mexico  | (5) 555-3932   |
|    4 | Thomas     | Hardy     | London     | UK      | (171) 555-7788 |
|    5 | Christina  | Berglund  | Luleå      | Sweden  | 0921-12 34 65  |
|    6 | Hanna      | Moos      | Mannheim   | Germany | 0621-08460     |
|    7 | Frédérique | Citeaux   | Strasbourg | France  | 88.60.15.31    |
|    8 | Martín     | Sommer    | Madrid     | Spain   | (91) 555 22 82 |
|    9 | Laurence   | Lebihan   | Marseille  | France  | 91.24.45.40    |
|   10 | Elizabeth  | Lincoln   | Tsawassen  | Canada  | (604) 555-4729 |
+------+------------+-----------+------------+---------+----------------+
9 rows in set (0.00 sec)
```

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;

```
mysql> update customer set country = 'maxico' where id = 11;
Query OK, 1 row affected (0.12 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from customer;
+------+------------+-----------+------------+---------+----------------+
| id   | first_name | last_name | city       | country | phone          |
+------+------------+-----------+------------+---------+----------------+
|    2 | Ana        | Trujillo  | México     | Mexico  | (5) 555-4729   |
|    3 | Antonio    | Moreno    | México     | Mexico  | (5) 555-3932   |
|    4 | Thomas     | Hardy     | London     | UK      | (171) 555-7788 |
|    5 | Christina  | Berglund  | Luleå      | Sweden  | 0921-12 34 65  |
|    6 | Hanna      | Moos      | Mannheim   | Germany | 0621-08460     |
|    7 | Frédérique | Citeaux   | Strasbourg | France  | 88.60.15.31    |
|    8 | Martín     | Sommer    | Madrid     | Spain   | (91) 555 22 82 |
|    9 | Laurence   | Lebihan   | Marseille  | France  | 91.24.45.40    |
|   10 | Elizabeth  | Lincoln   | Tsawassen  | Canada  | (604) 555-4729 |
|   11 | Victoria   | Ashworth  | London     | maxico  | (171) 555-1212 |
+------+------------+-----------+------------+---------+----------------+
10 rows in set (0.00 sec)
```

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;

```
mysql> delete from customer where id = 11;
Query OK, 1 row affected (0.15 sec)

mysql> select * from customer;
+------+------------+-----------+------------+---------+----------------+
| id   | first_name | last_name | city       | country | phone          |
+------+------------+-----------+------------+---------+----------------+
|    2 | Ana        | Trujillo  | México     | Mexico  | (5) 555-4729   |
|    3 | Antonio    | Moreno    | México     | Mexico  | (5) 555-3932   |
|    4 | Thomas     | Hardy     | London     | UK      | (171) 555-7788 |
|    5 | Christina  | Berglund  | Luleå      | Sweden  | 0921-12 34 65  |
|    6 | Hanna      | Moos      | Mannheim   | Germany | 0621-08460     |
|    7 | Frédérique | Citeaux   | Strasbourg | France  | 88.60.15.31    |
|    8 | Martín     | Sommer    | Madrid     | Spain   | (91) 555 22 82 |
|    9 | Laurence   | Lebihan   | Marseille  | France  | 91.24.45.40    |
|   10 | Elizabeth  | Lincoln   | Tsawassen  | Canada  | (604) 555-4729 |
+------+------------+-----------+------------+---------+----------------+
9 rows in set (0.00 sec)
```

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.

5.1.1. 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);
```

```
mysql> alter table customer add email varchar(25);
Query OK, 0 rows affected (2.12 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> select * from customer;
+------+------------+-----------+------------+---------+----------------+-------+
| id   | first_name | last_name | city       | country | phone          | email |
+------+------------+-----------+------------+---------+----------------+-------+
|    2 | Ana        | Trujillo  | México     | Mexico  | (5) 555-4729   | NULL  |
|    3 | Antonio    | Moreno    | México     | Mexico  | (5) 555-3932   | NULL  |
|    4 | Thomas     | Hardy     | London     | UK      | (171) 555-7788 | NULL  |
|    5 | Christina  | Berglund  | Luleå      | Sweden  | 0921-12 34 65  | NULL  |
|    6 | Hanna      | Moos      | Mannheim   | Germany | 0621-08460     | NULL  |
|    7 | Frédérique | Citeaux   | Strasbourg | France  | 88.60.15.31    | NULL  |
|    8 | Martín     | Sommer    | Madrid     | Spain   | (91) 555 22 82 | NULL  |
|    9 | Laurence   | Lebihan   | Marseille  | France  | 91.24.45.40    | NULL  |
|   10 | Elizabeth  | Lincoln   | Tsawassen  | Canada  | (604) 555-4729 | NULL  |
+------+------------+-----------+------------+---------+----------------+-------+
9 rows in set (0.00 sec)
```

5.1.2. 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;

```
mysql> alter table customer add dob date;
Query OK, 0 rows affected (1.83 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

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;

```
mysql> alter table customer modify dob year;
Query OK, 9 rows affected (3.68 sec)
Records: 9  Duplicates: 0  Warnings: 0
```

5.1.3. 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;

```
mysql> alter table customer drop column email;
Query OK, 0 rows affected (2.40 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> select * from customer;
+------+------------+-----------+------------+---------+----------------+
| id   | first_name | last_name | city       | country | phone          |
+------+------------+-----------+------------+---------+----------------+
|    2 | Ana        | Trujillo  | México     | Mexico  | (5) 555-4729   |
|    3 | Antonio    | Moreno    | México     | Mexico  | (5) 555-3932   |
|    4 | Thomas     | Hardy     | London     | UK      | (171) 555-7788 |
|    5 | Christina  | Berglund  | Luleå      | Sweden  | 0921-12 34 65  |
|    6 | Hanna      | Moos      | Mannheim   | Germany | 0621-08460     |
|    7 | Frédérique | Citeaux   | Strasbourg | France  | 88.60.15.31    |
|    8 | Martín     | Sommer    | Madrid     | Spain   | (91) 555 22 82 |
|    9 | Laurence   | Lebihan   | Marseille  | France  | 91.24.45.40    |
|   10 | Elizabeth  | Lincoln   | Tsawassen  | Canada  | (604) 555-4729 |
+------+------------+-----------+------------+---------+----------------+
9 rows in set (0.00 sec)
```

6. 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
);
```

```
mysql> create table student (id int not null, first_name varchar(25) not null, last_name varchar(25) not null, age int);
Query OK, 0 rows affected (1.16 sec)

mysql> desc student;
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| id         | int(11)     | NO   |     | NULL    |       |
| first_name | varchar(25) | NO   |     | NULL    |       |
| last_name  | varchar(25) | NO   |     | NULL    |       |
| age        | int(11)     | YES  |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+
4 rows in set (0.24 sec)
```

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;
```

```
mysql> alter table student modify age int not null;
Query OK, 0 rows affected (1.93 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> desc customer;
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| id         | int(11)     | YES  |     | NULL    |       |
| first_name | varchar(10) | YES  |     | NULL    |       |
| last_name  | varchar(10) | YES  |     | NULL    |       |
| city       | varchar(10) | YES  |     | NULL    |       |
| country    | varchar(15) | YES  |     | NULL    |       |
| phone      | varchar(15) | YES  |     | NULL    |       |
| dob        | year(4)     | YES  |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+
7 rows in set (0.00 sec)
```

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)
);
```

```
mysql> create table person(id int not null, first_name varchar(25) not null, last_name varchar(25) not null,age int, unique(id));
Query OK, 0 rows affected (1.29 sec)

mysql> desc person;
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| id         | int(11)     | NO   | PRI | NULL    |       |
| first_name | varchar(25) | NO   |     | NULL    |       |
| last_name  | varchar(25) | NO   |     | NULL    |       |
| age        | int(11)     | YES  |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+
4 rows in set (0.03 sec)
```

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);
```

```
mysql> alter table person add unique(first_name);
Query OK, 0 rows affected (0.76 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> desc person;
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| id         | int(11)     | NO   | PRI | NULL    |       |
| first_name | varchar(25) | NO   | UNI | NULL    |       |
| last_name  | varchar(25) | NO   |     | NULL    |       |
| age        | int(11)     | YES  |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+
4 rows in set (0.00 sec)
```

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);
```

```
mysql> ALTER TABLE person
    -> ADD CONSTRAINT UC_Persons UNIQUE (age,last_name);
Query OK, 0 rows affected (1.65 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> desc person;
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| id         | int(11)     | NO   | PRI | NULL    |       |
| first_name | varchar(25) | NO   | UNI | NULL    |       |
| last_name  | varchar(25) | NO   |     | NULL    |       |
| age        | int(11)     | YES  | MUL | NULL    |       |
+------------+-------------+------+-----+---------+-------+
4 rows in set (0.00 sec)
```

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;
```



```
mysql> ALTER TABLE person
    -> drop index UC_Persons;
Query OK, 0 rows affected (0.38 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> desc person;
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| id         | int(11)     | NO   | PRI | NULL    |       |
| first_name | varchar(25) | NO   | UNI | NULL    |       |
| last_name  | varchar(25) | NO   |     | NULL    |       |
| age        | int(11)     | YES  |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+
4 rows in set (0.07 sec)
```

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));

```
mysql> CREATE TABLE person(ID int NOT NULL,
    ->     last_name varchar(255) NOT NULL,
    ->     first_name varchar(255),
    ->     age int,
    ->     PRIMARY KEY (ID)
    -> );
Query OK, 0 rows affected (0.61 sec)

mysql> desc person;
+------------+--------------+------+-----+---------+-------+
| Field      | Type         | Null | Key | Default | Extra |
+------------+--------------+------+-----+---------+-------+
| ID         | int(11)      | NO   | PRI | NULL    |       |
| last_name  | varchar(255) | NO   |     | NULL    |       |
| first_name | varchar(255) | YES  |     | NULL    |       |
| age        | int(11)      | YES  |     | NULL    |       |
+------------+--------------+------+-----+---------+-------+
4 rows in set (0.00 sec)
```

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)
);
```

```
mysql> CREATE TABLE Person1 (
    ->     id int NOT NULL,
    ->     last_name varchar(25) NOT NULL,
    ->     first_name varchar(25),
    ->     age int,
    ->     CONSTRAINT PK_Person PRIMARY KEY (id,last_name)
    -> );
Query OK, 0 rows affected (0.94 sec)

mysql> desc Person1
    -> ;
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| id         | int(11)     | NO   | PRI | NULL    |       |
| last_name  | varchar(25) | NO   | PRI | NULL    |       |
| first_name | varchar(25) | YES  |     | NULL    |       |
| age        | int(11)     | YES  |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+
4 rows in set (0.00 sec)
```

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);
```

```
mysql> alter table student add primary key(id);
Query OK, 0 rows affected (1.71 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> desc student;
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| id         | int(11)     | NO   | PRI | NULL    |       |
| first_name | varchar(25) | NO   |     | NULL    |       |
| last_name  | varchar(25) | NO   |     | NULL    |       |
| age        | int(11)     | NO   |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+
4 rows in set (0.00 sec)
```

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);
```

```
mysql> desc student;
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| id         | int(11)     | NO   |     | NULL    |       |
| first_name | varchar(25) | NO   |     | NULL    |       |
| last_name  | varchar(25) | NO   |     | NULL    |       |
| age        | int(11)     | NO   |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+
4 rows in set (0.00 sec)

mysql> alter table student
    -> ADD CONSTRAINT PK_student PRIMARY KEY (id,first_name);
Query OK, 0 rows affected (1.38 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

- **DROP PRIMARY KEY CONSTRAINTS**

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

```
ALTER TABLE student
DROP PRIMARY KEY;
```

```
mysql> alter table student
    -> drop primary key;
Query OK, 0 rows affected (2.44 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> desc student;
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| id         | int(11)     | NO   |     | NULL    |       |
| first_name | varchar(25) | NO   |     | NULL    |       |
| last_name  | varchar(25) | NO   |     | NULL    |       |
| age        | int(11)     | NO   |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+
4 rows in set (0.05 sec)
```

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)
);
```

```
mysql> 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)
    -> );
Query OK, 0 rows affected (1.05 sec)

mysql>
mysql> desc customer;
+---------+---------------+------+-----+---------+-------+
| Field   | Type          | Null | Key | Default | Extra |
+---------+---------------+------+-----+---------+-------+
| Id      | int(11)       | NO   | PRI | NULL    |       |
| Name    | varchar(20)   | NO   |     | NULL    |       |
| Age     | int(11)       | NO   |     | NULL    |       |
| Address | varchar(25)   | YES  |     | NULL    |       |
| Salary  | decimal(18,2) | YES  |     | NULL    |       |
+---------+---------------+------+-----+---------+-------+
5 rows in set (0.08 sec)
```

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));

```
mysql> CREATE TABLE Orders (
    ->     OrderID int NOT NULL,
    ->     OrderNumber int NOT NULL,
    ->     Id int,
    ->     PRIMARY KEY (OrderID),
    ->     CONSTRAINT FK_customerOrder FOREIGN KEY (Id)
    ->     REFERENCES customer(Id)
    -> );
Query OK, 0 rows affected (1.08 sec)

mysql> desc orders;
+-------------+---------+------+-----+---------+-------+
| Field       | Type    | Null | Key | Default | Extra |
+-------------+---------+------+-----+---------+-------+
| OrderID     | int(11) | NO   | PRI | NULL    |       |
| OrderNumber | int(11) | NO   |     | NULL    |       |
| Id          | int(11) | YES  | MUL | NULL    |       |
+-------------+---------+------+-----+---------+-------+
3 rows in set (0.00 sec)
```

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);
```

```
mysql> ALTER TABLE Orders
    -> ADD FOREIGN KEY (ID) REFERENCES customer(id);
Query OK, 0 rows affected (2.38 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql>
mysql> desc orders;
+-------------+---------+------+-----+---------+-------+
| Field       | Type    | Null | Key | Default | Extra |
+-------------+---------+------+-----+---------+-------+
| OrderID     | int(11) | NO   | PRI | NULL    |       |
| OrderNumber | int(11) | NO   |     | NULL    |       |
| Id          | int(11) | YES  | MUL | NULL    |       |
+-------------+---------+------+-----+---------+-------+
3 rows in set (0.03 sec)
```


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;
```

```
mysql> ALTER TABLE Orders
    -> DROP FOREIGN KEY FK_customerOrder;
Query OK, 0 rows affected (0.19 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

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)
);
```

```
mysql> CREATE TABLE Persons (
    ->       ID int NOT NULL,
    ->     LastName varchar(255) NOT NULL,
    ->     FirstName varchar(255),
    ->     Age int,
    ->     CHECK (Age>=18)
    -> );
Query OK, 0 rows affected (1.19 sec)

mysql> insert into Persons values(1, 'abc', 'aaa', 17);
ERROR 3819 (HY000): Check constraint 'persons_chk_1' is violated.
```

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);

```
mysql> insert into Persons values(1, 'abc', 'aaa', 17);
ERROR 3819 (HY000): Check constraint 'persons_chk_1' is violated.
```

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

```
mysql> CREATE TABLE Persons (
    ->     ID int NOT NULL,
    ->     LastName varchar(255) NOT NULL,
    ->     FirstName varchar(255),
    ->     Age int,
    ->     City varchar(255),
    ->     CONSTRAINT CHK_Person CHECK (Age>=18 AND City='Sandnes')
    -> );
Query OK, 0 rows affected (1.20 sec)
```

CHECK on ALTER TABLE

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

ALTER TABLE Persons

ADD CHECK (Age >= 18)

```
mysql> ALTER TABLE Persons
    -> ADD CHECK (Age>=18)
    -> ;
Query OK, 0 rows affected (2.58 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

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');

```
mysql> ALTER TABLE Persons
    -> ADD CONSTRAINT CHK_PersonAge CHECK (Age>=18 AND City='Sandnes');
Query OK, 0 rows affected (2.31 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

DROP A CHECK CONSTRAINT

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

```
ALTER TABLE Persons
DROP CHECK CHK_PersonAge;
```

```
mysql> ALTER TABLE Persons
    -> DROP CHECK CHK_PersonAge;
Query OK, 0 rows affected (0.38 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

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'
);
```

```
mysql> CREATE TABLE Persons (
    ->     ID int NOT NULL,
    ->     LastName varchar(255) NOT NULL,
    ->     FirstName varchar(255),
    ->     Age int,
    ->     City varchar(255) DEFAULT 'Sandnes'
    -> );
Query OK, 0 rows affected (1.06 sec)

mysql> desc persons
    -> ;
+-----------+--------------+------+-----+---------+-------+
| Field     | Type         | Null | Key | Default | Extra |
+-----------+--------------+------+-----+---------+-------+
| ID        | int(11)      | NO   |     | NULL    |       |
| LastName  | varchar(255) | NO   |     | NULL    |       |
| FirstName | varchar(255) | YES  |     | NULL    |       |
| Age       | int(11)      | YES  |     | NULL    |       |
| City      | varchar(255) | YES  |     | Sandnes |       |
+-----------+--------------+------+-----+---------+-------+
5 rows in set (0.06 sec)
```

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;

```
mysql> ALTER TABLE Persons
    -> ALTER Age SET DEFAULT 20;
Query OK, 0 rows affected (0.44 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> desc persons;
+-----------+--------------+------+-----+---------+-------+
| Field     | Type         | Null | Key | Default | Extra |
+-----------+--------------+------+-----+---------+-------+
| ID        | int(11)      | NO   |     | NULL    |       |
| LastName  | varchar(255) | NO   |     | NULL    |       |
| FirstName | varchar(255) | YES  |     | NULL    |       |
| Age       | int(11)      | YES  |     | 20      |       |
| City      | varchar(255) | YES  |     | Sandnes |       |
+-----------+--------------+------+-----+---------+-------+
5 rows in set (0.04 sec)
```

DROP A DEFAULT CONSTRAINT

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

```
ALTER TABLE Persons
ALTER City DROP DEFAULT;
```

```
mysql> ALTER TABLE Persons
    -> ALTER City DROP DEFAULT;
Query OK, 0 rows affected (0.18 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> desc persons;
+-----------+--------------+------+-----+---------+-------+
| Field     | Type         | Null | Key | Default | Extra |
+-----------+--------------+------+-----+---------+-------+
| ID        | int(11)      | NO   |     | NULL    |       |
| LastName  | varchar(255) | NO   |     | NULL    |       |
| FirstName | varchar(255) | YES  |     | NULL    |       |
| Age       | int(11)      | YES  |     | 20      |       |
| City      | varchar(255) | YES  |     | NULL    |       |
+-----------+--------------+------+-----+---------+-------+
5 rows in set (0.00 sec)
```

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

7. 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)
```

```
mysql> CREATE INDEX idex_lastname
    -> ON Persons (LastName);
Query OK, 0 rows affected (0.86 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> desc persons;
+-----------+--------------+------+-----+---------+-------+
| Field     | Type         | Null | Key | Default | Extra |
+-----------+--------------+------+-----+---------+-------+
| ID        | int(11)      | NO   |     | NULL    |       |
| LastName  | varchar(255) | NO   | MUL | NULL    |       |
| FirstName | varchar(255) | YES  |     | NULL    |       |
| Age       | int(11)      | YES  |     | 20      |       |
| City      | varchar(255) | YES  |     | NULL    |       |
+-----------+--------------+------+-----+---------+-------+
5 rows in set (0.00 sec)
```

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;
```

8. 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

```
mysql> select * from customers;
+------+--------+-----------+--------+------+
| id   | name   | address   | salary | age  |
+------+--------+-----------+--------+------+
|    1 | ramesh | ahamdabad |  35000 |   25 |
|    2 | khilan | dubai     |  45000 |   35 |
|    3 | delhi  | ram       |  44500 |   32 |
|    4 | patna  | komal     |  50500 |   27 |
+------+--------+-----------+--------+------+
4 rows in set (0.00 sec)
```

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;
```

```
mysql> CREATE VIEW CUSTOMERS_VIEW AS
    -> SELECT name, age
    -> FROM customers;
Query OK, 0 rows affected (0.22 sec)

mysql> select * from CUSTOMERS_VIEW;
+--------+------+
| name   | age  |
+--------+------+
| ramesh |   25 |
| khilan |   35 |
| delhi  |   32 |
| patna  |   27 |
+--------+------+
4 rows in set (0.13 sec)
```

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;
```

```
mysql> CREATE VIEW CUSTOMER_VIEW AS
    -> SELECT name, age
    -> FROM  customers
    -> WHERE age IS NOT NULL
    -> WITH CHECK OPTION;
Query OK, 0 rows affected (0.17 sec)

mysql> select * from CUSTOMER_VIEW;
+--------+------+
| name   | age  |
+--------+------+
| ramesh |   25 |
| khilan |   35 |
| delhi  |   32 |
| patna  |   27 |
+--------+------+
4 rows in set (0.00 sec)
```

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;
```

```
mysql> DELETE FROM CUSTOMER_VIEW
    ->    WHERE age = 25;
Query OK, 1 row affected (0.16 sec)

mysql> select * from CUSTOMER_VIEW;
+--------+------+
| name   | age  |
+--------+------+
| khilan |   35 |
| delhi  |   32 |
| patna  |   27 |
+--------+------+
3 rows in set (0.00 sec)
```

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;

```
mysql> drop view customer_view;
Query OK, 0 rows affected (0.19 sec)

mysql> select * from CUSTOMER_VIEW;
ERROR 1146 (42S02): Table 'testdb.customer_view' doesn't exist
```

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.

9. 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.

9.1. 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.

9.1.1. 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.

```
1   use sakila;    #using the sakila database
2
3   DELIMITER $$
4   CREATE PROCEDURE Customer()
5   BEGIN
6       SELECT
7           first_name,
8           email,
9           address_id
10      FROM
11          customer
12      ORDER BY first_name;
13  END$$
14  DELIMITER ;
15
```

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.

9.1.2. 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.

```
1 •  CALL Customers();
```

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

first_name	email	address_id
AARON	AARON.SELBY@sakilacustomer.org	380
ADAM	ADAM.GOOCH@sakilacustomer.org	372
ADRIAN	ADRIAN.CLARY@sakilacustomer.org	531
AGNES	AGNES.BISHOP@sakilacustomer.org	221
ALAN	ALAN.KAHN@sakilacustomer.org	394

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.

9.1.3. 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

```
1 •  drop procedure Customer;
```

3 16:23:59 drop procedure Customer 0 row(s) affected 0.313 sec

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;

```
1  drop procedure [if exist] Customer;
```

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

9.1.4. 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.**

```
1     DELIMITER //
2
3     CREATE PROCEDURE Getactive(
4         IN Active VARCHAR(25)
5     )
6     BEGIN
7         SELECT *
8          FROM customer
9         WHERE active = Active;
10    END //
11
12    DELIMITER ;
13
14    call Getactive(1);
```

Output:

customer_id	store_id	first_name	last_name	email	address_id	active	create_date	last_update
1	1	MARY	SMITH	MARY.SMITH@sakilacustomer.org	5	1	2006-02-14 22:04:36	2006-02-15 04:57:20
2	1	PATRICIA	JOHNSON	PATRICIA.JOHNSON@sakilacustomer.org	6	1	2006-02-14 22:04:36	2006-02-15 04:57:20
3	1	LINDA	WILLIAMS	LINDA.WILLIAMS@sakilacustomer.org	7	1	2006-02-14 22:04:36	2006-02-15 04:57:20
4	2	BARBARA	JONES	BARBARA.JONES@sakilacustomer.org	8	1	2006-02-14 22:04:36	2006-02-15 04:57:20
5	1	ELIZABETH	BROWN	ELIZABETH.BROWN@sakilacustomer.org	9	1	2006-02-14 22:04:36	2006-02-15 04:57:20
6	2	JENNIFER	DAVIS	JENNIFER.DAVIS@sakilacustomer.org	10	1	2006-02-14 22:04:36	2006-02-15 04:57:20
7	1	MARIA	MILLER	MARIA.MILLER@sakilacustomer.org	11	1	2006-02-14 22:04:36	2006-02-15 04:57:20
8	2	SUSAN	WILSON	SUSAN.WILSON@sakilacustomer.org	12	1	2006-02-14 22:04:36	2006-02-15 04:57:20
9	2	MARGARET	MOORE	MARGARET.MOORE@sakilacustomer.org	13	1	2006-02-14 22:04:36	2006-02-15 04:57:20

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.

```
1   DELIMITER $$
2
3   CREATE PROCEDURE my_sqrt(input_number INT, OUT out_number FLOAT)
4   BEGIN
5       SET out_number=SQRT(input_number);
6   END$$
7   DELIMITER ;
8
9   #print the output
10  call my_sqrt(4, @out_number);
11  select @out_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.

	@out_number
▶	2

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.

```
1     DELIMITER $$
2     CREATE PROCEDURE SetCounter(
3         INOUT counter INT,
4         IN inc INT
5     )
6     BEGIN
7         SET counter = counter + inc;
8     END$$
9     DELIMITER ;
10
11    #print the output
12    set @counter = 1;
13    call SetCounter(@counter,1);
14    call SetCounter(@counter,3);
15    select @counter
```

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.

	@counter
▶	5

9.1.5. 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.

```
DECLARE person_age INT DEFAULT 0;
```

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

```
DECLARE X,Y INT DEFAULT 0;
```

ASSIGNING VARIABLE

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

```
DECLARE Total INT DEFAULT 0;
SET Total = 10
```

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

```
1   DELIMITER //
2   CREATE PROCEDURE User_Variables()
3   BEGIN
4   SET @x = 15;
5   SET @y = 10;
6   SELECT @x, @y, @x-@y;
7   END//
8
9   CALL User_Variables();
```

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:

@x	@y	@x-@y
15	10	5

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.

9.2. 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.

9.2.1. 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** and if the condition is TRUE, otherwise it will go to the next END IF.

```
1    Delimiter //
2    create procedure student_ifthen(IN s_subject varchar(15), OUT S_course varchar(15))
3    BEGIN
4    declare sub varchar(10);
5    select Subject from student where s_subject = subject;
6    if sub = 'computer' Then
7    set S_course = 'B.Tech';
8    END if;
9    END //
```

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’.**

```
1     call student_ifthen('Computer', @s_course);
2 •   select @s_course
```

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

	@s_course
▶	B.Tech

9.2.2. 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”.

```
Delimiter //
create procedure student_ifthenelse(IN s_subject varchar(20), OUT S_course varchar(50))
BEGIN
declare sub varchar(10);
select Subject INTO SUB
from student where s_subject = subject;
if sub = 'computer' Then
set S_course = 'B.Tech';
else
set S_course = 'subject is not available';
END if;
END //
```

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

```
call student_ifthenelse('History', @S_course);
select @S_course
```

@S_course
subject is not available

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'.

9.2.3. 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

EISEIF-Statement;

ELSE

EISE-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.

```
1   Delimiter //
2   create procedure student_ifthenelseif(IN s_subject varchar(20), OUT S_course varchar(50))
3   BEGIN
4   declare sub varchar(10);
5   select Subject INTO SUB
6   from student where s_subject = subject;
7   if sub = 'computer' Then
8   set S_course = 'B.Tech';
9   elseif sub = 'History' then
10  set S_course = 'BA';
11  else
12  set S_course = 'subject is not available';
13  END if;
14  END //
```

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

```
1   call student_ifthenelseif('history', @S_course);
2   select @S_course
```

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

	@S_course
	BA

let’s call the else part,

```
1   call student_ifthenelseif('maths', @S_course);
2   select @S_course
```

	@S_course
▶	subject is not available

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

9.3. 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.

9.3.1. 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.

```
Delimiter //
create procedure student_case(IN s_subject varchar(20), OUT S_course varchar(50)
)
BEGIN
declare sub varchar(10);
select Subject INTO SUB
from student where s_subject = subject;

case sub
 when 'computer' Then
    set S_course = 'B.Tech';
 when 'History' then
    set S_course = 'BA';
 else
set S_course = 'subject is not available';
END case;
END //
```

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:

```
call student_case('computer', @S_course);
select @S_course
```

	@S_course
▶	B.Tech

And also,

```
call student_case('History', @S_course);
select @S_course
```

	@S_course
▶	BA

Now, Let's see the else block, if the case_values are not satisfied then else block will executed.

```
call student_case('maths', @S_course);
select @S_course
```

	@S_course
▶	subject is not available

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

9.3.2. 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.

```
1   Delimiter //
2   create procedure student_searchedcase(IN s_subject varchar(20), OUT S_course varchar(50)
3   )
4   BEGIN
5   declare sub varchar(10);
6   select Subject INTO SUB
7   from student where s_subject = subject;
8
9   case
10   when sub = 'computer' Then
11      set S_course = 'B.Tech';
12   when sub = 'History' then
13      set S_course = 'BA';
14   else
15      set S_course = 'subject is not available';
16  END case;
17  END //
```

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.

```
1 •   call student_searchedcase('computer', @S_course);
2 •   select @S_course
```

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

	@S_course
▶	B.Tech

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

```
1 •   call student_searchedcase('maths', @S_course);
2 •   select @S_course
```

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

	@S_course
▶	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.

9.4. 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.

```
1   DELIMITER //
2   CREATE PROCEDURE Loop_example()
3   BEGIN
4       DECLARE x  INT;
5       DECLARE str  VARCHAR(255);
6
7       SET x = 1;
8       SET str =  '';
9           #if x is greater than 10, exit the loop
10      loop_label:  LOOP
11          IF  x > 10 THEN
12              LEAVE  loop_label;
13          END  IF;
14
15      #increase the x by 1
16          SET  x = x + 1;
17          IF  (x mod 2) THEN
18              ITERATE  loop_label;
19          ELSE
20              SET  str = CONCAT(str, x, ',');
21          END  IF;
22      END LOOP;
23      SELECT str;
24  END //
25  DELIMITER ;
```

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

```
1 •   call Loop_example()
```

	str
▶	2,4,6,8,10,

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

9.5. 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.

```
1   DELIMITER //
2   CREATE PROCEDURE WHILE_LOOP()
3   BEGIN
4   DECLARE X INT;
5   DECLARE string_val varchar(100);
6
7   set X = 1;
8   set string_val = '';
9
10  WHILE X < 10 DO
11      SET string_val = concat(string_val, x , ',');
12      SET X = X + 1;
13
14      END WHILE;
15  Select string_val;
16  END //
```

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

```
1   call WHILE_LOOP()
```

string_val
1,2,3,4,5,6,7,8,9,

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.

9.6. 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.

```
1   DELIMITER //
2
3   CREATE PROCEDURE Repeat_loop()
4   BEGIN
5       DECLARE count INT DEFAULT 1;
6       #creating output as empty string by default
7       DECLARE output VARCHAR(100) DEFAULT '';
8
9       REPEAT
10          SET output = CONCAT(output,count,',');
11          SET count = count + 1;
12      UNTIL count >= 20
13      END REPEAT;
14
15      SELECT output;
16  END//
17  DELIMITER ;
```

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

```
1   call Repeat_loop()
```

output
1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,

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

9.7. 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.

2. OPEN Cursor_name.

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

3. 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.

4. 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.

5. 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.

```
1   delimiter //
2   CREATE procedure customerdetails()
3   begin
4   DECLARE finished int default 0;
5   DECLARE email_list varchar(500) default ' ';
6   DECLARE customer_email varchar(100) default '';
7
8   DECLARE user_data CURSOR FOR SELECT email FROM customer limit 5;
9   DECLARE CONTINUE HANDLER FOR NOT FOUND SET finished = 1;
10
11   OPEN user_data; #open the user_data where the emails are saved
12    get_emails: LOOP
13      FETCH user_data INTO customer_email;
14      IF finished = 1 THEN
15         LEAVE get_emails;
16      END IF;
17      #concating the emails
18      SET email_list =  CONCAT(email_list," , ",customer_email);
19     END LOOP get_emails;
20   CLOSE user_data;  #closing the user_data
21       SELECT email_list;
22  END
```

Let's check the output by calling the stored procedure

```
1     call customerdetails()
```

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.

email_list
, MARY.SMITH@sakilacustomer.org , PATRICIA.JOHNSON@sakilacustomer.org , LINDA.WILLIAMS@sakilacustomer.org , BARBARA.JONES@sakilacustomer.org , ELIZABETH.BROWN@sakilacustomer.org

9.8. 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.

```
delimiter //
DECLARE EXIT HANDLER FOR SQLEXCEPTION
BEGIN
    ROLLBACK;
    SELECT 'An error has occurred, operation roll_backed
                and the stored procedure was terminated';
END;
```


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).

```
CREATE TABLE EmployeeDetails
(
     EmpID INTEGER
    ,EmpName VARCHAR(50)
    ,EmailAddress VARCHAR(50)
    ,CONSTRAINT pk_EmployeeDetails_EmpID PRIMARY KEY (EmpID)
);
```

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.

```
DELIMITER //
CREATE PROCEDURE InsertEmployeeDetails
(
     InputEmpID INTEGER ,InputEmpName VARCHAR(50) ,InputEmailAddress VARCHAR(50)
)
BEGIN
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SELECT 'Error occured';
 INSERT INTO EmployeeDetails
 (
  EmpID ,EmpName ,EmailAddress
 )
 VALUES
 (
  InputEmpID ,InputEmpName  ,InputEmailAddress
 );
 SELECT *FROM EmployeeDetails;
END
// DELIMITER ;
```

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.

```
CALL Employee.usp_InsertEmployeeDetails (1,'abc','abc@gmail.com');
CALL Employee.usp_InsertEmployeeDetails (1,'def','def@gmail.com');
CALL Employee.usp_InsertEmployeeDetails (2,'Roy','Roy@gmail.com');
```

Error occured
Error occured

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.

EmpID	EmpName	EmailAddress
1	abc	abc@gmail.com
2	Roy	Roy@gmail.com

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.

```
1   DELIMITER //
2   CREATE PROCEDURE InsertEmployeeDetailsexit
3   (
4        InputEmpID INTEGER ,InputEmpName VARCHAR(50)
5       ,InputEmailAddress VARCHAR(50)
6   )
7
8   BEGIN
9       DECLARE EXIT HANDLER FOR SQLEXCEPTION SELECT 'Error occured';
10   INSERT INTO EmployeeDetails
11   (
12    EmpID ,EmpName  ,EmailAddress
13   )
14   VALUES
15   (
16    InputEmpID ,InputEmpName ,InputEmailAddress
17   );
18   SELECT *FROM EmployeeDetails;
19  END
20  // DELIMITER ;
```

Here we are inserting the two rows as follow:

```
1   CALL InsertEmployeeDetailsexit (1,'abc','abc@gmail.com');
2   CALL InsertEmployeeDetailsexit (1,'def','def@gmail.com');
```

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.

	Error occured
▶	Error occured

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

9.8.1. 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
```

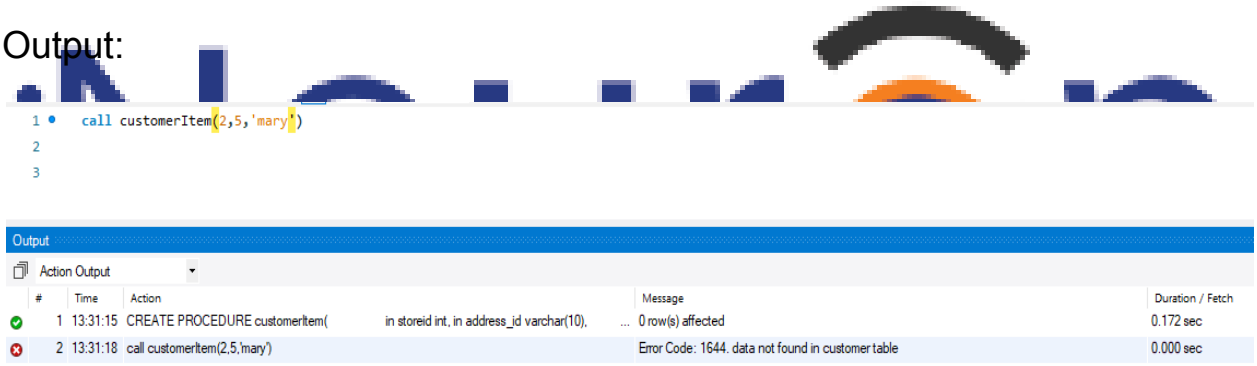
SIGNAL keyword is a SQLSTATE values or condition name declare by using DECLARE CONDITION. The condition_information_item_name can beMESSAGE_TEXT, MYSQL_ERROR, CURSOR_NAME etc.

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.

```
1     DELIMITER $$
2     CREATE PROCEDURE customerItem(
3                     in storeid int, in address_id varchar(10),
4                 in first_name varchar(10))
5     BEGIN
6         DECLARE C INT;
7         SELECT COUNT(store_id) INTO C
8         FROM customer
9         WHERE store_id = storeid;
10         -- check if Number exists
11        IF(C != 1) THEN
12            SIGNAL SQLSTATE '45000'
13                SET MESSAGE_TEXT = 'data not found in customer table';
14        END IF;
15    END
```

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”.

```
1   DELIMITER $$
2   CREATE PROCEDURE Divide_resignal(IN numerator INT, IN denominator INT, OUT output double)
3   BEGIN
4       DECLARE division_by_zero CONDITION FOR SQLSTATE '22012';
5
6       DECLARE CONTINUE HANDLER FOR division_by_zero
7       RESIGNAL SET MESSAGE_TEXT = 'Divided by zero / Denominator cannot be zero';
8
9       IF denominator = 0 THEN
10          SIGNAL division_by_zero;
11      ELSE
12          SET output := numerator / denominator;
13      END IF;
14  END
```

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

```
1   call Divide_resignal(5,0, @output)
2
```

Output

Action Output

#	Time	Action	Message	Duration / Fetch
1	13:50:10	call Divide_resignal(5,0, @output)	Error Code: 1644. Divided by zero / Denominator cannot be zero	0.000 sec

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

9.9. 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.

9.9.1. 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.

```
1   DELIMITER $$
2   CREATE FUNCTION studentssubject(
3       sub varchar(10)
4   )
5   RETURNS VARCHAR(20)
6   DETERMINISTIC
7   BEGIN
8       DECLARE course VARCHAR(20);
9
10      IF sub = 'computer' THEN
11          SET course = 'B.Tech';
12      ELSEIF sub = 'History' THEN
13          SET course = 'BA';
14      ELSEIF sub = 'science' THEN
15          SET course = 'B.Sc';
16      ELSEIF sub = 'maths' THEN
17          SET course = 'mathematics';
18      END IF;
19      #return the course
20      RETURN (course);
21  END$$
22  DELIMITER ;
```

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

```
SELECT
    subject,
    studentssubject(subject)
FROM
    student
ORDER BY
    subject;
```

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

subject	studentssubject(subject)
computer	B.Tech
History	BA
maths	mathematics
science	B.Sc

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

9.9.2. 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.

```
DROP FUNCTION studentssubject
```

Output

Action Output

#	Time	Action	Message	Duration / Fetch
1	14:36:47	DROP FUNCTION studentssubject	0 row(s) affected	0.172 sec

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

```
1 •  DROP FUNCTION IF EXISTS studentssubject
2
3
```

Output

Action Output

#	Time	Action	Message	Duration / Fetch
⚠ 1	14:38:03	DROP FUNCTION IF EXISTS studentssubject	0 row(s) affected, 1 warning(s): 1305 FUNCTION sakila.studentssubject does not exist	0.062 sec

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

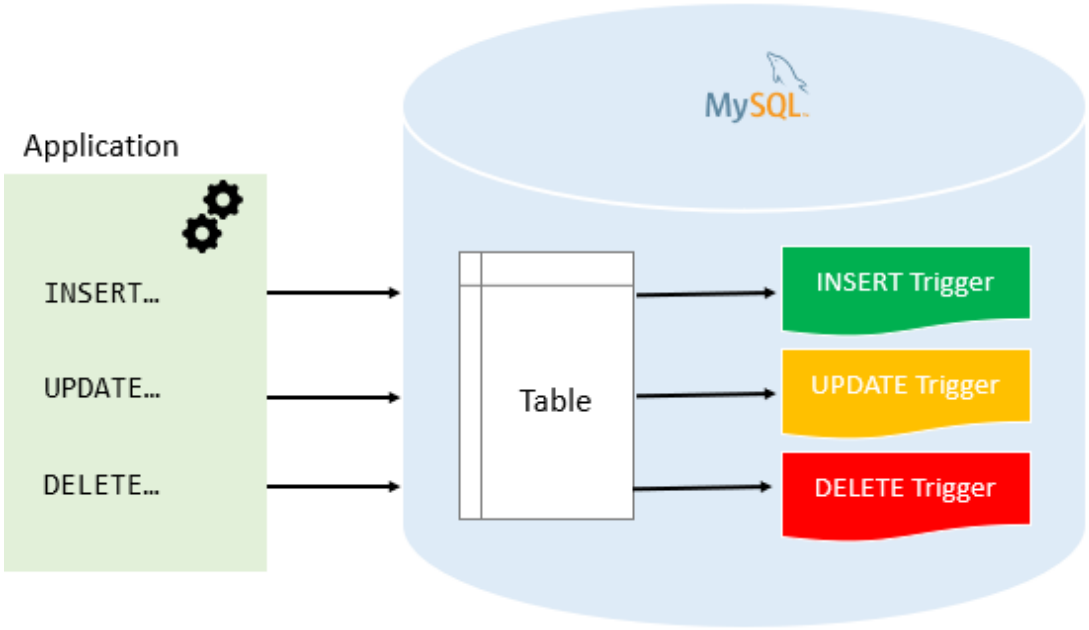
10. TRIGGERS

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.

10.1. 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 TABLE EmployeeDetail (
    id INT AUTO_INCREMENT PRIMARY KEY,
    employee_Number INT NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    change_date DATETIME DEFAULT NULL,
    action VARCHAR(50) DEFAULT NULL
);
```

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".

```
CREATE TABLE employee (
    id INT AUTO_INCREMENT PRIMARY KEY,
    employee_Number INT ,
    last_name VARCHAR(50) ,
    change_date DATETIME
);

insert into employee values(1, 101, 'singh', '2020-02-28')
```

Result Grid | Filter Rows: | Edit: | Export/Import:

id	employee_Number	last_name	change_date
1	101	singh	2020-02-28 00:00:00
NULL	NULL	NULL	NULL

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".

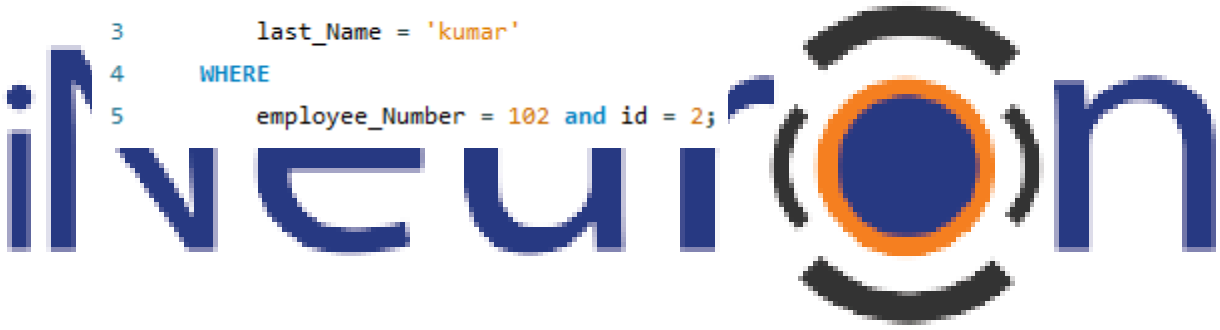
```
1   CREATE TRIGGER before_on_employee_update
2       BEFORE UPDATE ON employee
3       FOR EACH ROW
4    INSERT INTO EmployeeDetail
5    SET action = 'update',
6        employee_Number = OLD.employee_Number,
7        last_name = OLD.last_name,
8        change_date = NOW();
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Trigger	Event	Table	Statement	Timing	Created	sql_mode	Definer
customer_create_date	INSERT	customer	SET NEW.create_date = NOW()	BEFORE	2020-02-03 14:51:01.43	STRICT_TRANS_TABLES,STRICT_ALL_TABLES,...	root@localhost
before_on_employee_update	UPDATE	employee	INSERT INTO EmployeeDetail SET action = 'upd...	BEFORE	2020-02-28 15:51:34.44	STRICT_TRANS_TABLES,NO_ENGINE_SUBSTIT...	root@localhost

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

```
1   UPDATE employee
2   SET
3       last_Name = 'kumar'
4   WHERE
5       employee_Number = 102 and id = 2;
```



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.

```
1 •   select * from EmployeeDetail
```

Result Grid | Filter Rows: | Edit:

id	employee_Number	last_name	change_date	action
1	102	yadav	2020-02-28 16:05:07	update

10.2. 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.

```
1    DROP TRIGGER before_on_employee_update
```

The trigger **before_on_employee_update** has been deleted from the database.

10.3. 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;**

```
CREATE TABLE totalamount (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    amount INT NOT NULL
);
```

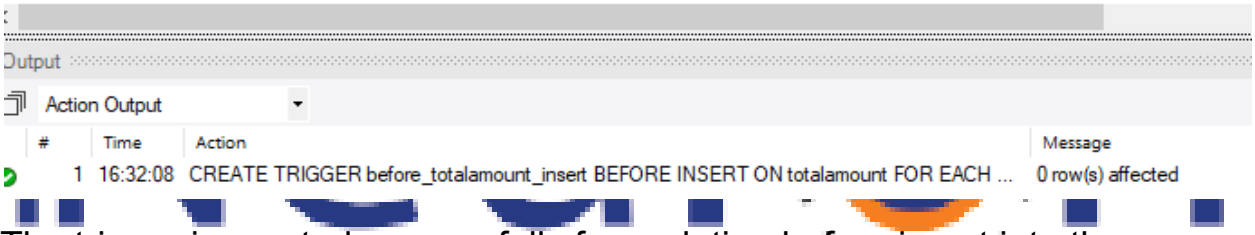
Now, let's create another table as **totalamountstatus** to store the summary of the triggers.

```
CREATE TABLE totalamountstatus(
    totalamount INT NOT NULL
);
```

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.

```
DELIMITER $$
CREATE TRIGGER before_totalamount_insert
BEFORE INSERT
ON totalamount FOR EACH ROW
BEGIN
    DECLARE rowcount INT;
    SELECT COUNT(*)
    INTO rowcount
    FROM totalamountstatus;

    IF rowcount > 0 THEN
        UPDATE totalamountstatus
        SET totalamount = totalamount + new.amount;
    ELSE
        INSERT INTO totalamountstatus(totalamount)
        VALUES(new.amount);
    END IF;
END $$
 DELIMITER ;
```



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

```
INSERT INTO totalamount(name, amount)
VALUES('singh',1000);
```

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.

```
select * from totalamountstatus
```

totalamount
1000

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.

10.4. 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**.

```
1  CREATE TABLE members (
2      id INT AUTO_INCREMENT,
3      name VARCHAR(100) NOT NULL,
4      email_id VARCHAR(255),
5      birth_Date DATE,
6      PRIMARY KEY (id)
7  );
```

Create another table as **remembers.**

```
1  CREATE TABLE reminders (
2      id INT AUTO_INCREMENT,
3      member_Id INT,
4      message VARCHAR(255) NOT NULL,
5      PRIMARY KEY (id , member_Id)
6  );
```

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.

```
DELIMITER $$

CREATE TRIGGER after_member_insert
AFTER INSERT
ON members FOR EACH ROW
BEGIN
    IF NEW.birth_Date IS NULL THEN
        INSERT INTO reminders(member_Id, message)
        VALUES(new.id,CONCAT('Hello ', NEW.name, ', Update your date_of_birth.'));
    END IF;
END$$

DELIMITER ;
```

Let's test the alter insert trigger.

```
INSERT INTO members(name, email_id, birth_Date)
VALUES
    ('hemant', 'hemant@gmail.com', NULL),
    ('vikash', 'vikash@gmail.com','2000-01-01');
```

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

id	name	email_id	birth_Date
1	hemant	hemant@gmail.com	NULL
2	vikash	vikash@gmail.com	2000-01-01

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.

id	member_Id	message
1	1	Hello hemant, Update your date_of_birth.

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.

10.5. 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;

```
1  CREATE TABLE sales (
2      id INT AUTO_INCREMENT,
3      product VARCHAR(100) NOT NULL,
4      quantity INT NOT NULL DEFAULT 0,
5      fiscal_Year SMALLINT NOT NULL,
6      fiscal_Month TINYINT NOT NULL,
7      CHECK(fiscal_Month >= 1 AND fiscal_Month <= 12),
8      CHECK(fiscal_Year BETWEEN 2000 and 2050),
9      CHECK (quantity >=0),
10     UNIQUE(product, fiscal_Year, fiscal_Month),
11     PRIMARY KEY(id)
12 );
```

Insert few rows into the sales table;

id	product	quantity	fiscal_Year	fiscal_Month
1	2003 Harley-Davidson Eagle Drag Bike	120	2020	1
2	1969 Corvair Monza	150	2020	1
3	1970 Plymouth Hemi Cuda	200	2020	1

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

```
DELIMITER $$
CREATE TRIGGER before_update_sales
BEFORE UPDATE
ON sales FOR EACH ROW
BEGIN
    DECLARE errorMessage VARCHAR(255);
    SET errorMessage = CONCAT('The new quantity ',
                        NEW.quantity,
                        ' cannot be 3 times greater than the current quantity ',
                        OLD.quantity);

    IF new.quantity > old.quantity * 3 THEN
        SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = errorMessage;
    END IF;
END $$
DELIMITER ;
```

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

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

```
UPDATE sales
SET quantity = 150
WHERE id = 1;
```

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;

id	product	quantity	fiscal_Year	fiscal_Month
1	2003 Harley-Davidson Eagle Drag Bike	150	2020	1
2	1969 Corvair Monza	150	2020	1
3	1970 Plymouth Hemi Cuda	200	2020	1

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

```
1  UPDATE sales
2  SET quantity = 500
3  WHERE id = 1;
4
```

Output

Action Output

#	Time	Action	Message	Duration / Fetch
1	17:33:06	UPDATE sales SET quantity = 500 WHERE id = 1	Error Code: 1644. The new quantity 500 cannot be 3 times greater than the current quantity 150	0.000 sec

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".

10.6. 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.

```
CREATE TABLE Sales_Changes (
    id INT AUTO_INCREMENT PRIMARY KEY,
    sales_id INT,
    before_Quantity INT,
    after_Quantity INT,
    changed_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```

Now, let's create the AFTER UPDATE TRIGGER;

```
DELIMITER $$
CREATE TRIGGER after_update_sales
AFTER UPDATE
ON sales FOR EACH ROW
BEGIN
    IF OLD.quantity <> new.quantity THEN
        INSERT INTO SalesChanges(sales_id,before_Quantity, after_Quantity)
        VALUES(old.id, old.quantity, new.quantity);
    END IF;
END$$
DELIMITER ;
```

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

```
UPDATE Sales
SET quantity = 350
WHERE id = 1;
```

Let's check the **sales_changes** table;

id	sales_id	before_Quantity	after_Quantity	changed_at
1	1	150	350	2020-02-28 17:51:11

As we can see the value is updated automatically in the sales_changes.

10.7. 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;**

```
CREATE TABLE Salary (
    employee_no INT PRIMARY KEY,
    valid_From DATE NOT NULL,
    amount DEC(12 , 2 ) NOT NULL DEFAULT 0
);
```

Insert few rows into salary table;

employee_no	valid_From	salary
1016	2020-01-01	90000.00
1022	2020-01-01	80000.00
1026	2020-01-01	70000.00

Create another table as deleted_salary to store the deleted salaries;

```sql
CREATE TABLE deleted_Salary (
    id INT PRIMARY KEY AUTO_INCREMENT,
    employee_no INT PRIMARY KEY,
    valid_From DATE NOT NULL,
    salary DEC(12 , 2 ) NOT NULL DEFAULT 0,
    deleted_At TIMESTAMP DEFAULT NOW()
);
```

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.

```sql
CREATE TRIGGER before_deleted_salaries
BEFORE DELETE
ON salary FOR EACH ROW
BEGIN
    INSERT INTO deleted_Salary(employee_no,valid_From,salary)
    VALUES(OLD.employee_no,OLD.valid_From,OLD.salary);
END$$

DELIMITER ;
```

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

```sql
DELETE FROM salary
WHERE employee_no = 1022;
```

Now, check the deleted_Salary table to check whether the data is stored or not.

id	employee_no	valid_From	salary	deleted_At
1	1022	2020-01-01	80000.00	2020-02-28 18:27:25

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

10.8. 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;

```
1  create table salary(employee_no INT PRIMARY KEY,
2      salary DECIMAL(10,2) NOT NULL DEFAULT 0)
```

employee_no	salary
1016	60000.00
1022	50000.00
1026	70000.00

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

```
1  CREATE TABLE deleted_Salary(
2      total DECIMAL(15,2) NOT NULL
3  );
```

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.

```
1 •   INSERT INTO deleted_Salary(total)
2     SELECT SUM(salary)
3     FROM Salary;
```

So the total amount is 180000.

	total
▶	180000.00

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.

```
1 •   CREATE TRIGGER after_salaries_delete
2     AFTER DELETE
3     ON Salary FOR EACH ROW
4     UPDATE deleted_Salary
5     SET total = total - old.salary;
```

Let's delete a row where the employee_no = 1022 inside the salary table;

```
1 •   SET SQL_SAFE_UPDATES=0;
2 •   DELETE FROM Salary
3     WHERE employee_no = 1022;
```

Check the deleted_Salary;

	total
▶	130000.00

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