

# MySQL Tutorial

## MySQL Tutorial

[MySQL Tutorial](#)

[MySQL Features](#)

[MySQL Versions](#)

[MySQL Data Types](#)

[MySQL Variables](#)

[Install MySQL](#)

[MySQL Connection](#)

## MySQL Workbench

[MySQL Workbench](#)

## User Management

[MySQL Create User](#)

[MySQL Drop User](#)

[MySQL Show Users](#)

[Change User Password](#)

## MySQL Database

[MySQL Create Database](#)

[MySQL Select Database](#)

[MySQL Show Databases](#)

[MySQL Drop Database](#)

[MySQL Copy Database](#)

## Table & Views

[MySQL CREATE Table](#)

[MySQL ALTER Table](#)

[MySQL Show Tables](#)

[MySQL Rename Table](#)

[MySQL TRUNCATE Table](#)

[MySQL Describe Table](#)

[MySQL DROP Table](#)

[MySQL Temporary Table](#)

[MySQL Copy Table](#)

[MySQL Repair Table](#)

[MySQL Add/Delete Column](#)

[MySQL Show Columns](#)

[MySQL Rename Column](#)

[MySQL Views](#)

[MySQL Table Locking](#)

[MySQL Account Lock](#)

[MySQL Account Unlock](#)

## MySQL Queries

[MySQL Queries](#)

[MySQL Constraints](#)

[MySQL INSERT Record](#)

[MySQL UPDATE Record](#)

[MySQL DELETE Record](#)

[MySQL SELECT Record](#)

[MySQL Replace](#)

[Insert On Duplicate Key Update](#)

[MySQL INSERT IGNORE](#)

[Insert Into Select](#)

## MySQL Indexes

[MySQL Create Index](#)

[MySQL Drop Index](#)

[MySQL Show Indexes](#)

[MySQL Unique Index](#)

[MySQL Clustered Index](#)

## **MySQL Clustered vs Non-Clustered Index**

### **MySQL Clauses**

[MySQL WHERE](#)  
[MySQL DISTINCT](#)  
[MySQL FROM](#)  
[MySQL ORDER BY](#)  
[MySQL GROUP BY](#)  
[MySQL HAVING](#)

### **MySQL Privileges**

[MySQL Grant Privilege](#)  
[MySQL Revoke Privilege](#)

### **Control Flow Function**

[MySQL IF\(\)](#)  
[MySQL IFNULL\(\)](#)  
[MySQL NULLIF\(\)](#)  
[MySQL CASE](#)  
[MySQL IF Statement](#)

### **MySQL Conditions**

[MySQL AND](#)  
[MySQL OR](#)  
[MySQL AND OR](#)  
[MySQL Boolean](#)  
[MySQL LIKE](#)  
[MySQL IN](#)  
[MySQL ANY](#)  
[MySQL Exists](#)  
[MySQL NOT](#)  
[MySQL Not Equal](#)  
[MySQL IS NULL](#)  
[MySQL IS NOT NULL](#)  
[MySQL BETWEEN](#)

### **MySQL Join**

[MySQL JOIN](#)  
[MySQL Inner Join](#)  
[MySQL Left Join](#)  
[MySQL Right Join](#)  
[MySQL CROSS JOIN](#)  
[MySQL SELF JOIN](#)  
[MySQL DELETE JOIN](#)  
[MySQL Update Join](#)  
[MySQL EquiJoin](#)  
[MySQL Natural Join](#)  
[Left Join vs Right Join](#)  
[MySQL Union vs Join](#)

### **MySQL Key**

[MySQL Unique Key](#)  
[MySQL Primary Key](#)  
[MySQL Foreign Key](#)  
[MySQL Composite Key](#)

### **MySQL Triggers**

[MySQL Trigger](#)  
[MySQL Create Trigger](#)  
[MySQL Show Trigger](#)  
[MySQL DROP Trigger](#)  
[Before Insert Trigger](#)  
[After Insert Trigger](#)  
[MySQL BEFORE UPDATE Trigger](#)

[MySQL AFTER UPDATE Trigger](#)

[MySQL BEFORE DELETE Trigger](#)

[MySQL AFTER DELETE Trigger](#)

## Aggregate Functions

[MySQL Aggregate Functions](#)

[MySQL count\(\)](#)

[MySQL sum\(\)](#)

[MySQL avg\(\)](#)

[MySQL min\(\)](#)

[MySQL max\(\)](#)

[MySQL GROUP\\_CONCAT\(\)](#)

[MySQL first\(\)](#)

[MySQL last\(\)](#)

# MySQL Tutorial



MySQL tutorial provides basic and advanced concepts of MySQL. Our MySQL tutorial is designed for beginners and professionals.

MySQL is a relational database management system based on the Structured Query Language, which is the popular language for accessing and managing the records in the database. MySQL is open-source and free software under the GNU license. It is supported by **Oracle Company**.

Our MySQL tutorial includes all topics of MySQL database that provides for how to manage database and to manipulate data with the help of various SQL queries. These queries are: insert records, update records, delete records, select records, create tables, drop tables, etc. There are also given MySQL interview questions to help you better understand the MySQL database.

## What is Database?

It is very important to understand the database before learning MySQL. A database is an application that stores the organized collection of records. It can be accessed and managed by the user very easily. It allows us to organize data into tables, rows, columns, and indexes to find the relevant information very quickly. Each database contains distinct [API](#)

for performing database operations such as creating, managing, accessing, and searching the data it stores. Today, many databases available like MySQL, Sybase, [Oracle](#)

, [MongoDB](#)

, [PostgreSQL](#)

, [SQL Server](#)

, etc. In this section, we are going to focus on MySQL mainly.

Nested Structure in C

Keep Watching

## What is MySQL?

MySQL is currently the most popular database management system software used for managing the relational database. It is open-source database software, which is supported by Oracle Company. It is fast, scalable, and easy to use database management system in comparison with Microsoft SQL Server and Oracle Database. It is commonly used in conjunction with [PHP](#)

scripts for creating powerful and dynamic server-side or web-based enterprise applications.

It is developed, marketed, and supported by [MySQL AB, a Swedish company](#), and written in [C programming language](#)

and [C++ programming language](#)

. The official pronunciation of MySQL is not the My Sequel; it is **My Ess Que Ell**. However, you can pronounce it in your way. Many small and big companies use MySQL. MySQL supports many Operating Systems like [Windows](#)

, [Linux](#)

, [MacOS](#), etc. with C, C++, and [Java languages](#)

.

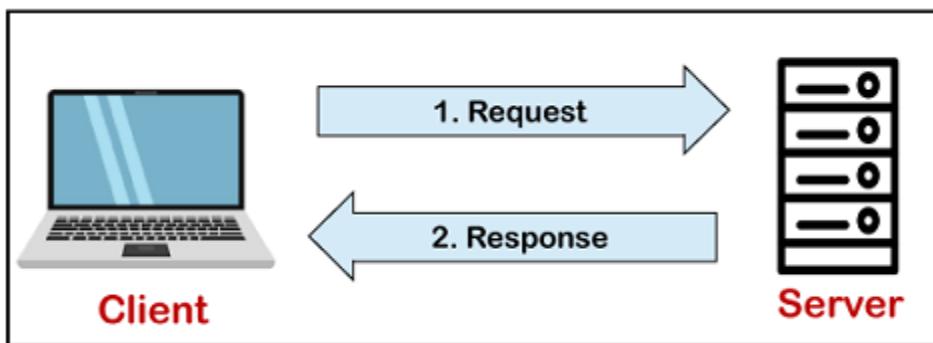
MySQL is a [Relational Database Management System](#)

(RDBMS) software that provides many things, which are as follows:

- It allows us to implement database operations on tables, rows, columns, and indexes.
- It defines the database relationship in the form of tables (collection of rows and columns), also known as relations.
- It provides the Referential Integrity between rows or columns of various tables.
- It allows us to update the table indexes automatically.
- It uses many SQL queries and combines useful information from multiple tables for the end-users.

## How MySQL Works?

MySQL follows the working of Client-Server Architecture. This model is designed for the end-users called clients to access the resources from a central computer known as a server using network services. Here, the clients make requests through a graphical user interface (GUI), and the server will give the desired output as soon as the instructions are matched. The process of MySQL environment is the same as the client-server model.



The core of the MySQL database is the MySQL Server. This server is available as a separate program and responsible for handling all the database instructions, statements, or commands. The working of MySQL database with MySQL Server are as follows:

1. MySQL creates a database that allows you to build many tables to store and manipulate data and defining the relationship between each table.
2. Clients make requests through the GUI screen or command prompt by using specific SQL expressions on MySQL.
3. Finally, the server application will respond with the requested expressions and produce the desired result on the client-side.

A client can use any MySQL [GUI](#)

. But, it is making sure that your GUI should be lighter and user-friendly to make your data management activities faster and easier. Some of the most widely used MySQL GUIs are MySQL Workbench, SequelPro, DBVisualizer, and the Navicat DB Admin Tool. Some GUIs are commercial, while some are free with limited functionality, and some are only compatible with MacOs. Thus, you can choose the GUI according to your needs.

## Reasons for popularity

MySQL is becoming so popular because of these following reasons:

- MySQL is an open-source database, so you don't have to pay a single penny to use it.
- MySQL is a very powerful program that can handle a large set of functionality of the most expensive and powerful database packages.
- MySQL is customizable because it is an open-source database, and the open-source GPL license facilitates programmers to modify the SQL software according to their own specific environment.
- MySQL is quicker than other databases, so it can work well even with the large data set.
- MySQL supports many operating systems with many languages like PHP, PERL, C, C++, JAVA, etc.
- MySQL uses a standard form of the well-known SQL data language.
- MySQL is very friendly with PHP, the most popular language for web development.
- MySQL supports large databases, up to 50 million rows or more in a table. The default file size limit for a table is 4GB, but you can increase this (if your operating system can handle it) to a theoretical limit of 8 million terabytes (TB).

## History of MySQL

The project of MySQL was started in 1979 when MySQL's inventor **Michael Widenius** developed an in-house database tool called **UNIREG** for managing databases. After that, UNIREG has been rewritten in several different languages and extended to handle big databases. After some time, Michael Widenius contacted **David Hughes**, the author of mSQL, to see if Hughes would be interested in connecting mSQL to UNIREG's B+ ISAM handler to provide indexing to mSQL. That's the way MySQL came into existence.

MySQL is named after the daughter of co-founder Michael Widenius whose name is "My".

### History by Year:

Year	Happenings
------	------------

1995	MySQL AB, founded by Michael Widenius (Monty), David Axmark, and Allan Larsson in Sweden.
2000	MySQL goes open-source and releases software under the terms of the GPL. Revenues dropped 80% as a result, and it took a year to make up for it.
2001	Marten Mickos elected CEO at age 38. Marten was the CEO of several nordic companies before joining MySQL and comes with a sales and marketing background. 2 million active installations. Raised series with an undisclosed amount from Scandinavian venture capitalists. It was estimated to be around \$1 to \$2 million.
2002	MySQL launched its headquarters in addition to Swedish headquarters. At that time, 3 million active users. MySQL was ended this year with \$6.5 million in revenue with 1,000 paying customers.
2003	This year raised a \$19.5 million series b from benchmark capital and index ventures. At this time, 4 million active installations and over 30,000 downloads per day. It ended the year with \$12 million in revenue.
2004	With the main revenue coming from the OEM dual-licensing model, MySQL decides to move more into the enterprise market and to focus more on recurring revenue from end-users rather than one-time licensing fees from their OEM partners. It ended the year with \$20 million in revenue.
2005	MySQL launched the MySQL network model after the Redhat network. The MySQL network is a subscription service targeted at end-users that provide updates, alerts, notifications, and product-level support designed to make it easier for companies to manage hundreds of MySQL servers. MySQL 5 ships and includes many new features to go after enterprise users (e.g., stored procedures, triggers, views, cursors, distributed transactions, federated storage engines, etc.) Oracle buys innobase, the 4-person, and a Finland's company behind MySQL's InnoDB storage backend, ended the year with \$34 million in revenue based on 3400 customers.
2006	Marten Mickos confirms that Oracle tried to buy MySQL. Oracle' CEO Larry Ellison commented: "we've spoken to them, in fact, we've spoken to almost everyone. Are we interested? It's a tiny company. I think the revenues from MySQL are between \$30 million and \$40 million. Oracle's revenue next year is \$15 billion." Oracle buys sleepycat, the company that provides MySQL with the Berkeley db transactional storage engine. Marten Mickos announces that they are making MySQL ready for an IPO in 2008 on a projected \$100 million in revenues. 8 million active installations. MySQL has 320 employees in 25 countries, 70 percent of whom work from home, raised an \$18 million series c based on a rumored valuation north of \$300 million. MySQL is estimated to have a 33% market share measured in install base and 0.2% market share measured in revenue (the database market was a \$15 billion market in 2006). It ended the year with \$50 million in revenue.
2007	It ended the year with \$75 million in revenue.
2008	Sun Microsystems acquired MySQL AB for approximately \$1 billion. Michael Widenius (Monty) and David Axmark, two of MySQL AB's co-founders, begin to criticize Sun publicly and leave Sun shortly after.
2009	Marten Mickos leaves Sun and becomes entrepreneur-in-residence at Benchmark Capital. Sun has now lost the business and spiritual leaders that turned MySQL into a success. Sun Microsystems and Oracle announced that they have entered into a definitive agreement under which Oracle will acquire Sun common stock for \$9.50 per share in cash. The transaction is valued at approximately \$7.4 billion.

# MySQL Features

MySQL is a relational database management system (RDBMS) based on the SQL (Structured Query Language) queries. It is one of the most popular languages for accessing and managing the records in the table. MySQL is open-source and free software under the GNU license. Oracle Company supports it.

The following are the most important features of MySQL:

## **Relational Database Management System (RDBMS)**

MySQL is a relational database management system. This database language is based on the SQL queries to access and manage the records of the table.

### **Easy to use**

MySQL is easy to use. We have to get only the basic knowledge of SQL. We can build and interact with MySQL by using only a few simple SQL statements.

### **It is secure**

MySQL consists of a solid data security layer that protects sensitive data from intruders. Also, passwords are encrypted in MySQL.

### **Client/ Server Architecture**

MySQL follows the working of a client/server architecture. There is a database server (MySQL) and arbitrarily many clients (application programs), which communicate with the server; that is, they can query data, save changes, etc.

### **Free to download**

MySQL is free to use so that we can download it from MySQL official website without any cost.

### **It is scalable**

MySQL supports multi-threading that makes it easily scalable. It can handle almost any amount of data, up to as much as 50 million rows or more. The default file size limit is about 4 GB. However, we can increase this number to a theoretical limit of 8 TB of data.

### **Speed**

MySQL is considered one of the very fast database languages, backed by a large number of the benchmark test.

### **High Flexibility**

MySQL supports a large number of embedded applications, which makes MySQL very flexible.

### **Compatible on many operating systems**

MySQL is compatible to run on many operating systems, like Novell NetWare, Windows\*, Linux\*, many varieties of UNIX\* (such as Sun\* Solaris\*, AIX, and DEC\* UNIX), OS/2, FreeBSD\*, and others. MySQL also provides a facility that the clients can run on the same computer as the server or on another computer (communication via a local network or the Internet).

### **Allows roll-back**

MySQL allows transactions to be rolled back, commit, and crash recovery.

### **Memory efficiency**

Its efficiency is high because it has a very low memory leakage problem.

### **High Performance**

MySQL is faster, more reliable, and cheaper because of its unique storage engine architecture. It provides very high-performance results in comparison to other databases without losing an essential functionality of the software. It has fast loading utilities because of the different cache memory.

### **High Productivity**

MySQL uses Triggers, Stored procedures, and views that allow the developer to give higher productivity.

## **Platform Independent**

It can download, install, and execute on most of the available operating systems.

## **Partitioning**

This feature improves the performance and provides fast management of the large database.

## **GUI Support**

MySQL provides a unified visual database graphical user interface tool named "**MySQL Workbench**" to work with database architects, developers, and Database Administrators. **MySQL Workbench** provides SQL development, data modeling, data migration, and comprehensive administration tools for server configuration, user administration, backup, and many more. MySQL has a fully GUI supports from MySQL Server version 5.6 and higher.

## **Dual Password Support**

MySQL version 8.0 provides support for dual passwords: one is the current password, and another is a secondary password, which allows us to transition to the new password.

## **Disadvantages/Drawback of MySQL**

Following are the few disadvantages of MySQL:

- MySQL version less than 5.0 doesn't support ROLE, COMMIT, and stored procedure.
- MySQL does not support a very large database size as efficiently.
- MySQL doesn't handle transactions very efficiently, and it is prone to data corruption.
- MySQL is accused that it doesn't have a good developing and debugging tool compared to paid databases.
- MySQL doesn't support SQL check constraints.

## **MySQL Versions**

Versioning is a process of categorizing either unique version **names or numbers** to the unique set of software program as it is developed and released. The commonly used version name for denoting the initial release of a software or program is version 1.0. There is no industry standard rule available that decide how version number should be formatted. Thus, each company has its own methods for assigning the version name to the software. When the new features in software and programs have introduced, fixes bugs, patched security holes, then version number increased that indicates those improvements.

The latest support for working with MySQL is version number **v5.8**. It contains many essential changes, including new features added and removed, fixed bugs and security issues, etc. This version contains the release history from MySQL 8.0 to MySQL 8.0.21. It is available from **April 2018** and ends the support in **April 2026**.

When you are going to install MySQL in your system, you must have to choose the version and distribution format to use. You can install MySQL in two ways, where first is a **development release**, and the second is **General Availability (GA)** release. The development release provides the newest feature and is not recommended to use in production. The General Availability (GA) release, also known as production or stable release, is mainly used for production. Therefore, you must have to decide the most recent General Availability release.

Let us see what is new in MySQL 8.0 version.

## **Features Added in MySQL 8.0**

The following features are added in MySQL 8.0 version:

**Data Dictionary:** It incorporates the transactional data dictionary to stores information about the database objects. Previous versions stored data in metadata files and non-transactional tables.

**Atomic DDL Statement:** It is an Atomic Data Definition Language statement that combines storage engine operations, data dictionary updates, and binary log associated with a DDL operation into a single atomic transaction.

**Upgrade Procedures:** Previously, the installation of the new MySQL version automatically upgrade the data dictionary table at the next startup, and then DBA is expected to invoke mysql\_upgrade command manually for completing the upgrading

process. After MySQL 8.0.16, it is not dependent on the DBA to invoke mysql\_upgrade command for completing the upgradation process.

**Security and account management:** There is some enhancement added to improve the security and provide it to enable greater DBA flexibility in account management.

**Resource Management:** Now, MySQL allows you to create and support resource groups, assign threads to a particular group so that it can execute according to the resource available for the group. Group attributes can control its resource consumption by threads in the group.

**Table Encryption Management:** Now, table encryption is managed globally by defining and enforcing encryption defaults. The default\_table\_encryption variable or DEFAULT ENCRYPTION clause defines encryption default when creating a schema and general tablespace.

**InnoDB enhancements:** The InnoDB enhancement were added in auto-increment counter, index tree corruption, memcached plugin, InnoDB\_deadlock\_detect, tablespace encryption feature, storage engine, InnoDB\_dedicated\_server, zlib library, and many more.

**Character Set Support:** The default character set now changed from latin1 to utf8mb4. The new character set has many new collations, including utf8mb\_ja\_0900\_as\_cs.

**JSON Enhancements:** The following enhancements or additions are introduced in the MySQL's json functionality: Inline path (->>) operator, json aggregate functions JSON\_ARRAYAGG() and JSON\_OBJECTAGG(), utility function JSON\_PRETTY(), JSON\_STORAGE\_SIZE(), JSON\_STORAGE\_FREE(). In sorting json values, now each value is represented by variable-length part of sort key instead of a fixed 1K size. It also added merge function JSON.Merge\_PATCH to add 2 json object and JSON\_TABLE() function.

**Data Type Support:** In data type specifications, it can support the use of expressions as default values.

**Optimizer Enhancement:** This version added optimizer enhancement such as invisible indexes, descending indexes, support the creation of a functional index. It can use constant folding for comparison between a column and a constant value.

**Window Function:** This version supports many new window functions such as RANK(), LAG(), and NTILE().

Some other important features are:

- It enhances Regular expression support.
- Error Logging re-written to use MySQL component architecture.
- A new backup lock introduced that permits DML while preventing an operation, which can result in an inconsistent state.
- It enhances connection management. Now, TCP/IP port can be configured specifically for administrative connections. It gives more control in compression to minimize the bytes sent over the connection to the server.
- In previous versions, the plugins were written in C or C++. Now, it must be written in only the C++ language. MySQL 8.0.17 version provides clone plugins, which permit InnoDB data locally or from a remote server. The clone plugin also supports replication.
- In this version, the time zone support for TIMESTAMP and DATETIME values.
- This version also added the SQL standard table value constructor and explicit table clause.

## Features Deprecated in MySQL 8.0

The MySQL 8.0 version has deprecated many features and can be removed in the future series. Some of the features are explained below:

- The character set utf8mb3 is deprecated.
- The sha256\_password is deprecated and removed in future versions. Now, the default authentication will be caching\_sha2\_password.
- The validate\_password plugin will be deprecated soon and can be removed in future versions.
- The ENGINE clause will be deprecated for the ALTER TABLESPACE and DROP TABLESPACE.
- AUTO\_INCREMENT and UNSIGNED attribute are deprecated for FLOAT and DOUBLE column type.
- Now, it uses JSON\_MERGE\_PRESERVE() function instead of JSON\_MERGE().
- The SQL\_CALC\_FOUND\_ROWS modifier, FOUND\_ROWS() function, --no--dd--upgrade server option, mysql\_upgrade client, and mysql\_upgrade\_info are also deprecated.
- The use of MYSQL\_PWD environment variable, which specifies the MySQL password, is deprecated now.

## Features Removed in MySQL 8.0

available, your application needs to be updated.

- The `InnoDB_locks_unsafe_for_binlog` system variable was removed, `information_schema_stats` variable is replaced by `information_schema_stats_expiry`.
- Some features related to the account management were removed, which are: `GRANT` statement for creating user, `PASSWORD()` function, `old_passwords` system variable, etc.
- The code related to the InnoDB system table is obsolete and was removed from the MySQL 8.0 version. The `INFORMATION_SCHEMA` view which is based on InnoDB system table now replaced by internal system view and renamed as:

Old Name	New Name
INNODB_SYS_COLUMNS	INNODB_COLUMNS
INNODB_SYS_DATAFILES	INNODB_DATAFILES
INNODB_SYS_FIELDS	INNODB_FIELDS
INNODB_SYS_FOREIGN	INNODB_FOREIGN
INNODB_SYS_FOREIGN_COLS	INNODB_FOREIGN_COLS
INNODB_SYS_INDEXES	INNODB_INDEXES
INNODB_SYS_TABLES	INNODB_TABLES
INNODB_SYS_TABLESPACES	INNODB_TABLESPACES
INNODB_SYS_TABLESTATS	INNODB_TABLESTATS
INNODB_SYS_VIRTUAL	INNODB_VIRTUAL

- This version also removed some query catch, which is `FLUSH QUERY CACHE`, `RESET QUERY CACHE` statement, `SQL_CACHE` `SELECT` modifier, etc.
- The `sync_frm` system variable is removed because of the .frm files become obsolete.
- The `multi_range_count`, `log_warning`, and global scope for the `sql_log_bin` system variable have been removed.
- Some of the encryption-related items such as `ENCODE()`, `DECODE()`, `ENCRYPT()`, etc. have also removed.
- It removed `mysql_install_db` program is removed, and instead if this it uses `--initialize` or `--initialize_insecure` option.

Let us understand the release history of previous versions of MySQL through the following table:

Version Name	Released Date	End of Support	Description
MySQL 5.1	14-11-2008	December 2013	This version contains the releases of MySQL 5.0 to MySQL 5.1.73 versions. To read about the first version of MySQL, click <a href="#">here</a> .
MySQL 5.5	03-12-2010	December 2018	This version contains the releases of MySQL 5.5 to MySQL 5.5.62 versions.
MySQL 5.6	05-02-2013	February 2021	This version contains the releases of MySQL 5.6 to MySQL 5.5.45 versions.
MySQL 5.7	21-10-2015	October 2023	This version contains the releases of MySQL 5.7 to MySQL 5.6.27 versions.

MySQL 8.0	19-04-2018	April 2026	This version contains the releases of MySQL 8.0 to MySQL 8.0.21 versions.
-----------	------------	------------	---

**Note:** Version 6 has stopped working after the Sun Microsystems acquisition, and now it uses MySQL Cluster product versions 7 and 8.

## MySQL Data Types

A Data Type specifies a particular type of data, like integer, floating points, Boolean, etc. It also identifies the possible values for that type, the operations that can be performed on that type, and the way the values of that type are stored. In MySQL, each database table has many columns and contains specific data types for each column.

We can determine the data type in MySQL with the following characteristics:

- The type of values (fixed or variable) it represents.
- The storage space it takes is based on whether the values are a fixed-length or variable length.
- Its values can be indexed or not.
- How MySQL performs a comparison of values of a particular data type.

MySQL supports a lot number of [SQL](#) standard data types in various categories. It uses many different data types that can be broken into the following categories: numeric, date and time, string types, spatial types, and [JSON](#) data types.

### Numeric Data Type

MySQL has all essential SQL numeric data types. These data types can include the exact numeric data types (For example, integer, decimal, numeric, etc.), as well as the approximate numeric data types (For example, float, real, and double precision). It also supports BIT datatype to store bit values. In MySQL, numeric data types are categories into two types, either signed or unsigned except for bit data type.

The following table contains all numeric data types that support in [MySQL](#):

Data Type Syntax	Description
TINYINT	It is a very small integer that can be signed or unsigned. If signed, the allowable range is from -128 to 127. If unsigned, the allowable range is from 0 to 255. We can specify a width of up to 4 digits. It takes 1 byte for storage.
SMALLINT	It is a small integer that can be signed or unsigned. If signed, the allowable range is from -32768 to 32767. If unsigned, the allowable range is from 0 to 65535. We can specify a width of up to 5 digits. It requires 2 bytes for storage.
MEDIUMINT	It is a medium-sized integer that can be signed or unsigned. If signed, the allowable range is from -8388608 to 8388607. If unsigned, the allowable range is from 0 to 16777215. We can specify a width of up to 9 digits. It requires 3 bytes for storage.
INT	It is a normal-sized integer that can be signed or unsigned. If signed, the allowable range is from -2147483648 to 2147483647. If unsigned, the allowable range is from 0 to 4294967295. We can specify a width of up to 11 digits. It requires 4 bytes for storage.
BIGINT	It is a large integer that can be signed or unsigned. If signed, the allowable range is from -9223372036854775808 to 9223372036854775807. If unsigned, the allowable range is from 0 to 18446744073709551615. We can specify a width of up to 20 digits. It requires 8 bytes for storage.
FLOAT(m,d)	It is a floating-point number that cannot be unsigned. You can define the display length (m) and the number of decimals (d). This is not required and will default to 10,2, where 2 is the

	number of decimals, and 10 is the total number of digits (including decimals). Decimal precision can go to 24 places for a float type. It requires 2 bytes for storage.
DOUBLE(m,d)	It is a double-precision floating-point number that cannot be unsigned. You can define the display length (m) and the number of decimals (d). This is not required and will default to 16,4, where 4 is the number of decimals. Decimal precision can go to 53 places for a double. Real is a synonym for double. It requires 8 bytes for storage.
DECIMAL(m,d)	An unpacked floating-point number that cannot be unsigned. In unpacked decimals, each decimal corresponds to one byte. Defining the display length (m) and the number of decimals (d) is required. Numeric is a synonym for decimal.
BIT(m)	It is used for storing bit values into the table column. Here, M determines the number of bit per value that has a range of 1 to 64.
BOOL	It is used only for the true and false condition. It considered numeric value 1 as true and 0 as false.
BOOLEAN	It is Similar to the BOOL.

## Date and Time Data Type:

This data type is used to represent temporal values such as date, time, datetime, timestamp, and year. Each temporal type contains values, including zero. When we insert the invalid value, MySQL cannot represent it, and then zero value is used.

The following table illustrates all date and time data types that support in MySQL:

Data Type Syntax	Maximum Size	Explanation
YEAR[(2 4)]	Year value as 2 digits or 4 digits.	The default is 4 digits. It takes 1 byte for storage.
DATE	Values range from '1000-01-01' to '9999-12-31'.	Displayed as 'yyyy-mm-dd'. It takes 3 bytes for storage.
TIME	Values range from '-838:59:59' to '838:59:59'.	Displayed as 'HH:MM:SS'. It takes 3 bytes plus fractional seconds for storage.
DATETIME	Values range from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'.	Displayed as 'yyyy-mm-dd hh:mm:ss'. It takes 5 bytes plus fractional seconds for storage.
TIMESTAMP(m)	Values range from '1970-01-01 00:00:01' UTC to '2038-01-19 03:14:07' TC.	Displayed as 'YYYY-MM-DD HH:MM:SS'. It takes 4 bytes plus fractional seconds for storage.

## String Data Types:

The string data type is used to hold plain text and binary data, for example, files, images, etc. MySQL can perform searching and comparison of string value based on the pattern matching such as LIKE operator, Regular Expressions, etc.

The following table illustrates all string data types that support in MySQL:

Data Type Syntax	Maximum Size	Explanation
CHAR(size)	It can have a maximum size of 255 characters.	Here size is the number of characters to store. Fixed-length strings. Space padded

		on the right to equal size characters.
VARCHAR(size)	It can have a maximum size of 255 characters.	Here size is the number of characters to store. Variable-length string.
TINYTEXT(size)	It can have a maximum size of 255 characters.	Here size is the number of characters to store.
TEXT(size)	Maximum size of 65,535 characters.	Here size is the number of characters to store.
MEDIUMTEXT(size)	It can have a maximum size of 16,777,215 characters.	Here size is the number of characters to store.
LONGTEXT(size)	It can have a maximum size of 4GB or 4,294,967,295 characters.	Here size is the number of characters to store.
BINARY(size)	It can have a maximum size of 255 characters.	Here size is the number of binary characters to store. Fixed-length strings. Space padded on the right to equal size characters. (introduced in MySQL 4.1.2)
VARBINARY(size)	It can have a maximum size of 255 characters.	Here size is the number of characters to store. Variable-length string. (introduced in MySQL 4.1.2)
ENUM	It takes 1 or 2 bytes that depend on the number of enumeration values. An ENUM can have a maximum of 65,535 values.	It is short for enumeration, which means that each column may have one of the specified possible values. It uses numeric indexes (1, 2, 3...) to represent string values.
SET	It takes 1, 2, 3, 4, or 8 bytes that depends on the number of set members. It can store a maximum of 64 members.	It can hold zero or more, or any number of string values. They must be chosen from a predefined list of values specified during table creation.

## Binary Large Object Data Types (BLOB):

BLOB in MySQL is a data type that can hold a variable amount of data. They are categorized into four different types based on the maximum length of values it can hold.

The following table shows all Binary Large Object data types that support in MySQL:

Data Type Syntax	Maximum Size
TINYBLOB	It can hold a maximum size of 255 bytes.
BLOB(size)	It can hold the maximum size of 65,535 bytes.
MEDIUMBLOB	It can hold the maximum size of 16,777,215 bytes.
LONGBLOB	It can hold the maximum size of 4gb or 4,294,967,295 bytes.

## Spatial Data Types

It is a special kind of data type which is used to hold various geometrical and geographical values. It corresponds to OpenGIS classes. The following table shows all spatial types that support in MySQL:

Data Types	Descriptions
GEOMETRY	It is a point or aggregate of points that can hold spatial values of any type that has a location.
POINT	A point in geometry represents a single location. It stores the values of X, Y coordinates.
POLYGON	It is a planar surface that represents multisided geometry. It can be defined by zero or more interior boundary and only one exterior boundary.
LINESTRING	It is a curve that has one or more point values. If it contains only two points, it always represents Line.
GEOMETRYCOLLECTION	It is a kind of geometry that has a collection of zero or more geometry values.
MULTILINESTRING	It is a multi-curve geometry that has a collection of linestring values.
MULTIPOINT	It is a collection of multiple point elements. Here, the point cannot be connected or ordered in any way.
MULTIPOLYGON	It is a multisurface object that represents a collection of multiple polygon elements. It is a type of two-dimensional geometry.

## JSON Data Type

MySQL provides support for native JSON data type from the version v5.7.8. This data type allows us to store and access the JSON document quickly and efficiently.

The JSON data type has the following advantages over storing JSON-format strings in a string column:

1. It provides automatic validation of JSON documents. If we stored invalid documents in JSON columns, it would produce an error.
2. It provides an optimal storage format.

## MySQL Variables

Variables are used for storing data or information during the execution of a program. It is a way of labeling data with an appropriate name that helps to understand the program more clearly by the reader. The main purpose of the variable is to store data in memory and can be used throughout the program.

MySQL can use variables in **three** different ways, which are given below:

1. User-Defined Variable
2. Local Variable
3. System Variable

### User-Defined Variable

Sometimes, we want to pass values from one statement to another statement. The user-defined variable enables us to store a value in one statement and later can refer it to another statement. MySQL provides a **SET** and **SELECT** statement to declare and initialize a variable. The user-defined variable name starts with **@ symbol**.

The user-defined variables are not case-sensitive such as @name and @NAME; both are the same. A user-defined variable declares by one person cannot visible to another person. We can assign the user-defined variable into limited data types like integer, float, decimal, string, or NULL. The user-defined variable can be a maximum of **64 characters** in length.

## Syntax

The following syntax is used to declare a user-defined variable.

1. By using the **SET** statement

1. **SET** @var\_name = value;

**NOTE:** We can use either '=' or ':=' assignment operator with the SET statement.

2. By using the **SELECT** statement

1. **SELECT** @var\_name := value;

## Example1

Here, we are going to assign a value to a variable by using the SET statement.

1. mysql> **SET** @name='peter';

Then, we can display the above value by using the SELECT statement.

1. mysql> **SELECT** @name;

## Output

```
mysql> SET @name = 'peter';
Query OK, 0 rows affected (0.01 sec)

mysql> SELECT @name;
+-----+
| @name |
+-----+
| peter |
+-----+
1 row in set (0.00 sec)
```

## Example 2

Let us create table **students** in the MySQL database, as shown below:

studentid	firstname	lastname	class	age
1	Rinky	Ponting	12	20
2	Mark	Boucher	11	22
3	Sachin	Tendulkar	10	18
4	Peter	Fleming	10	22
5	Virat	Kohli	12	23
NULL	NULL	NULL	NULL	NULL

Run the following statement to get the maximum age of the student in the 'students' table and assign the age to the user-defined variable **@maxage**.

1. mysql> **SELECT** @maxage:= **MAX**(age) **FROM** students;

It will give the following output.

	@maxage:= MAX(age)
▶	23

Now, run the SELECT statement that uses the @maxage variable to return the maximum age of the student.

1. mysql> **SELECT** firstname, lastname, age **FROM** students **WHERE** age = @maxage;

After successful execution of the above statement, we will get the following result:

	firstname	lastname	age
▶	Virat	Kohli	23

## Example3

If we access the **undeclared** variable, it will give the **NULL** output.

1. Mysql> **SELECT** @var1;

## Output

```
mysql> SELECT @var1;
+-----+
| @var1 |
+-----+
| 0x    |
+-----+
1 row in set (0.00 sec)
```

## Local Variable

It is a type of variable that is not prefixed by **@ symbol**. The local variable is a strongly typed variable. The scope of the local variable is in a stored program block in which it is declared. MySQL uses the **DECLARE** keyword to specify the local variable. The **DECLARE** statement also combines a **DEFAULT** clause to provide a default value to a variable. If you do not provide the **DEFAULT** clause, it will give the initial value **NULL**. It is mainly used in the stored procedure program.

### Syntax

We can use the **DECLARE** statement with the following syntax:

1. **DECLARE** variable\_name datatype(**size**) [**DEFAULT** default\_value];

Let us see the following example to use the local variable.

### Example

1. mysql> **DECLARE** total\_price Oct(8,2) **DEFAULT** 0.0;

We can also define two or more variables with the same data type by using a single **DECLARE** statement.

1. mysql> **DECLARE** a,b,c **INT DEFAULT** 0;

The below example explains how we can use the **DECLARE** statement in a stored procedure.

1. **DELIMITER //**
2. **Create Procedure** Test()
3. **BEGIN**
4.   **DECLARE** A **INT DEFAULT** 100;
5.   **DECLARE** B **INT**;
6.   **DECLARE** C **INT**;
7.   **DECLARE** D **INT**;
8.   **SET** B = 90;
9.   **SET** C = 45;
10.   **SET** D = A + B - C;
11.   **SELECT** A, B, C, D;
12. **END //**
13. **DELIMITER ;**

After successful execution of the above function, call the stored procedure function as below:

1. mysql> **CALL** Test();

It will give the following output:

	A	B	C	D
▶	100	90	45	145

## System Variable

System variables are a special class to all program units, which contains **predefined** variables. MySQL contains various system variables that configure its operation, and each system variable contains a default value. We can change some system variables dynamically by using the **SET** statement at runtime. It enables us to modify the server operation without stop and restart it. The system variable can also be used in the expressions.

MySQL server gives a bunch of system variables such as GLOBAL, SESSION, or MIX types. We can see the GLOBAL variable throughout the lifecycle of the server, whereas the SESSION variable remains active for a particular session only.

We can see the names and values of the system variable by using the following ways:

1. To see the current values used by the running server, execute the following command.

1. mysql> SHOW VARIABLES;
- 2.
3. OR,
- 4.
5. Mysql> **SELECT** @@var\_name;

2. When we want to see the values based on its compiled-in defaults, use the following command.

1. mysql> mysqld --verbose --help

### Example1

1. mysql> SHOW VARIABLES LIKE '%wait\_timeout%';

#### Output

```
mysql> SHOW VARIABLES LIKE '%wait_timeout%';
+-----+-----+
| Variable_name      | Value   |
+-----+-----+
| innodb_lock_wait_timeout | 50      |
| lock_wait_timeout     | 31536000 |
| mysqlx_wait_timeout  | 28800   |
| wait_timeout         | 28800   |
+-----+-----+
4 rows in set (0.01 sec)
```

### Example 2

1. mysql> **SELECT** @@key\_buffer\_size;

#### Output

```
mysql> SELECT @@key_buffer_size;
+-----+
| @@key_buffer_size |
+-----+
|      8388608 |
+-----+
1 row in set (0.00 sec)
```

## How to install MySQL

MySQL is one of the most popular relational database management software that is widely used in today's industry. It provides multi-user access support with various storage engines. It is backed by Oracle Company. In this section, we are going to learn how we can download and install MySQL for beginners.

### Prerequisites

The following requirements should be available in your system to work with [MySQL](#)

:

- o **MySQL Setup Software**
- o Microsoft .NET Framework 4.5.2
- o Microsoft Visual C++ Redistributable for Visual Studio 2019
- o RAM 4 GB (6 GB recommended)

### Download MySQL

Follow these steps:

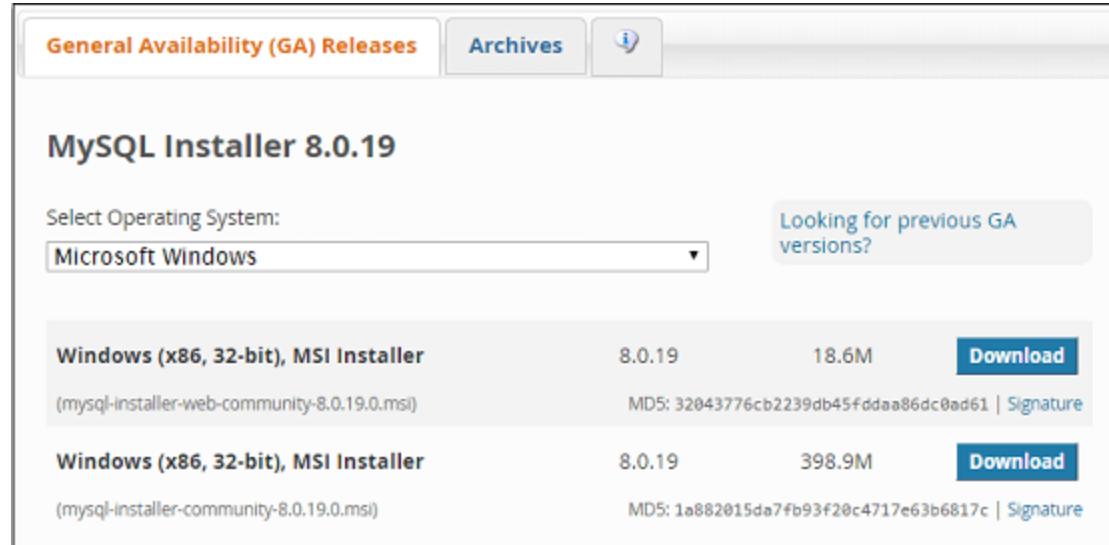
Competitive questions on Structures in Hindi

Keep Watching

## Step 1: Go to the [official website](#)

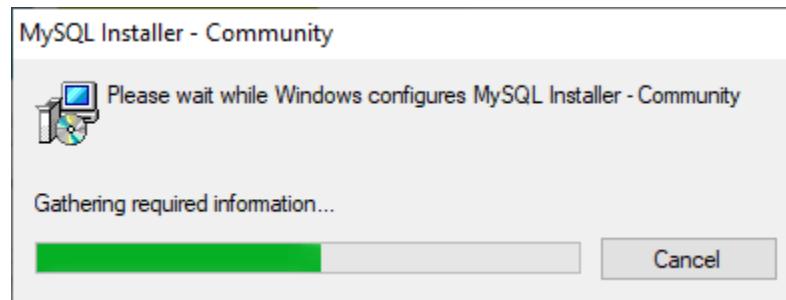
of MySQL and download the community server edition software. Here, you will see the option to choose the Operating System, such as Windows.

**Step 2:** Next, there are two options available to download the setup. Choose the version number for the MySQL community server, which you want. If you have good internet connectivity, then choose the mysql-installer-web-community. Otherwise, choose the other one.



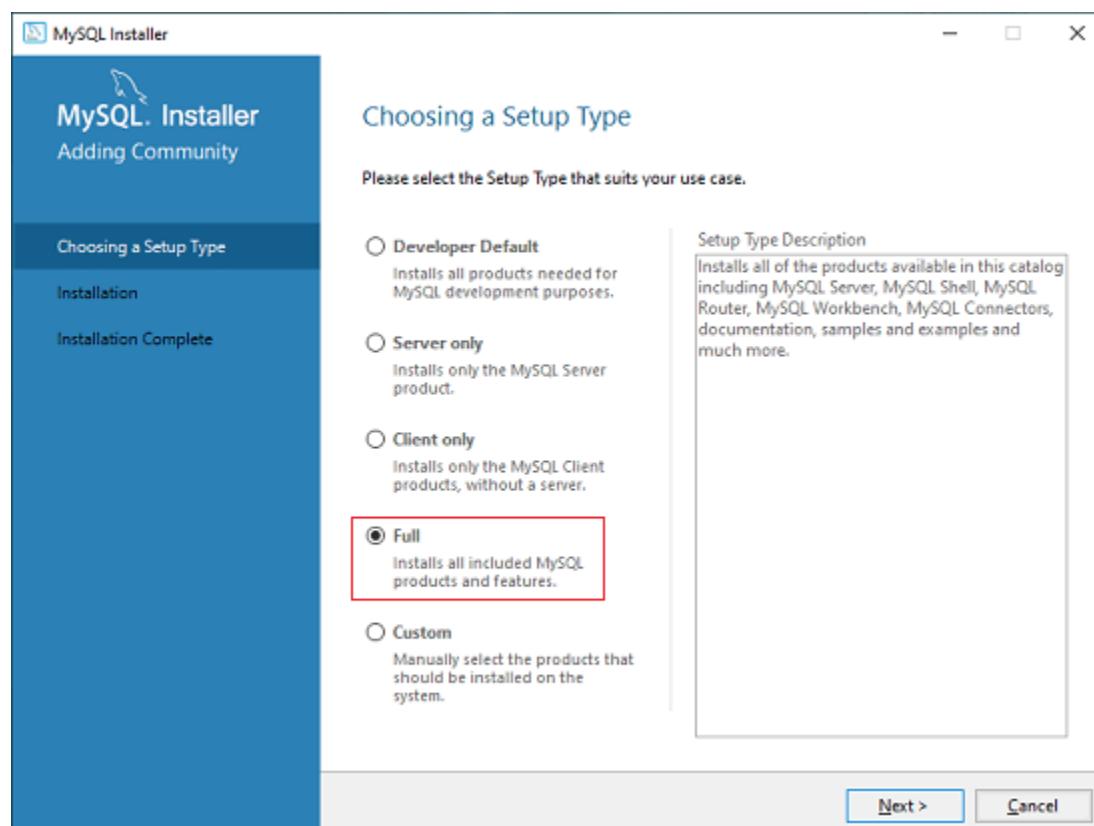
## Installing MySQL on Windows

**Step 1:** After downloading the setup, unzip it anywhere and double click the MSI **installer .exe file**. It will give the following screen:



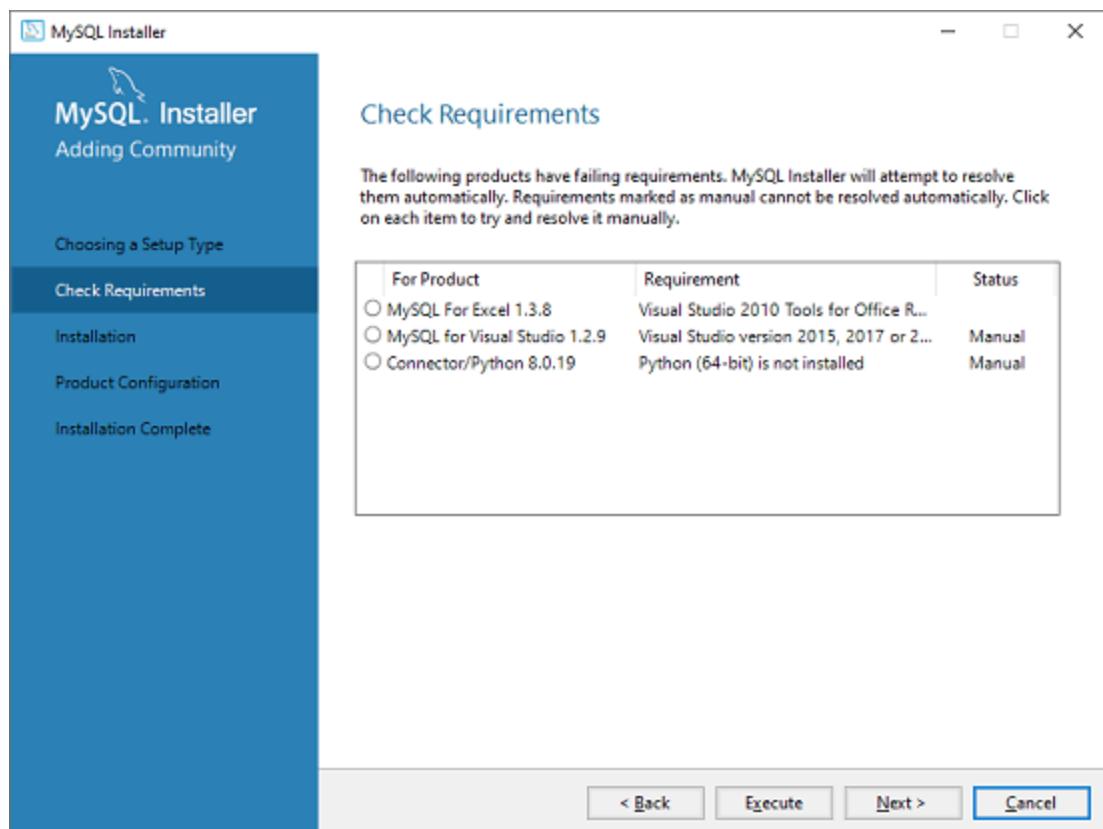
**Step 2:** In the next wizard, choose the **Setup Type**. There are several types available, and you need to choose the appropriate option to install MySQL product and [features](#)

. Here, we are going to select the **Full** option and click on the Next button.

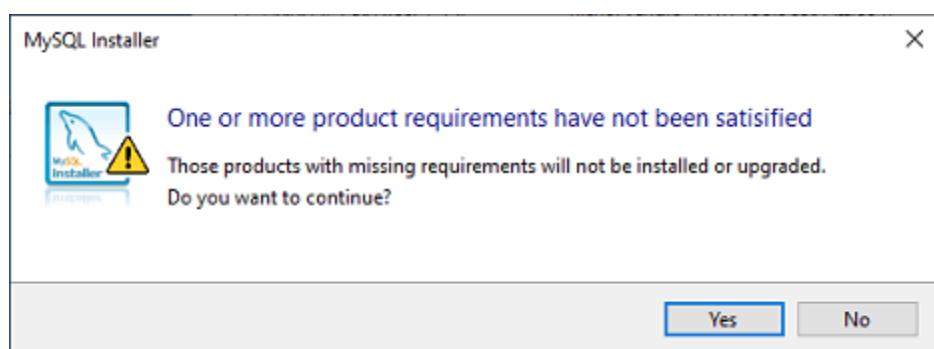


This option will install the following things: MySQL Server, MySQL Shell, MySQL Router, [MySQL Workbench](#), MySQL Connectors, documentation, samples and examples, and many more.

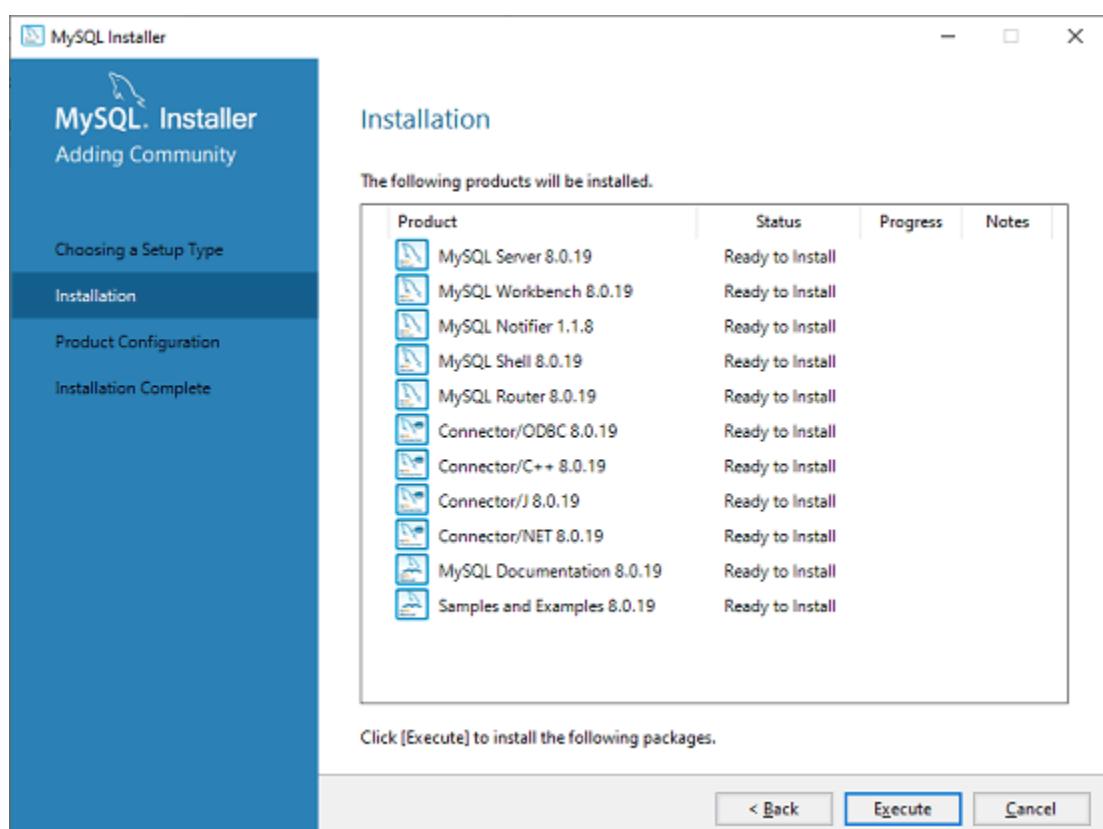
**Step 3:** Once we click on the Next button, it may give information about some features that may fail to install on your system due to a lack of requirements. We can resolve them by clicking on the **Execute** button that will install all requirements automatically or can skip them. Now, click on the Next button.



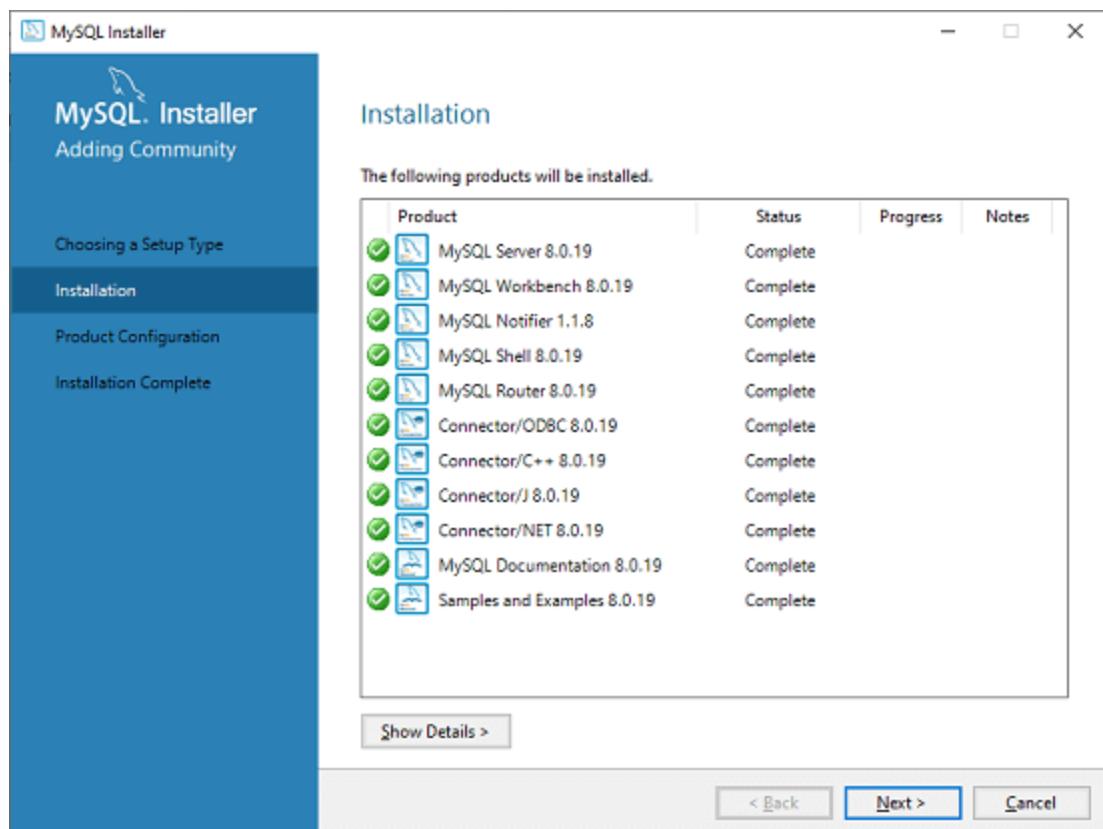
**Step 4:** In the next wizard, we will see a dialog box that asks for our confirmation of a few products not getting installed. Here, we have to click on the **Yes** button.



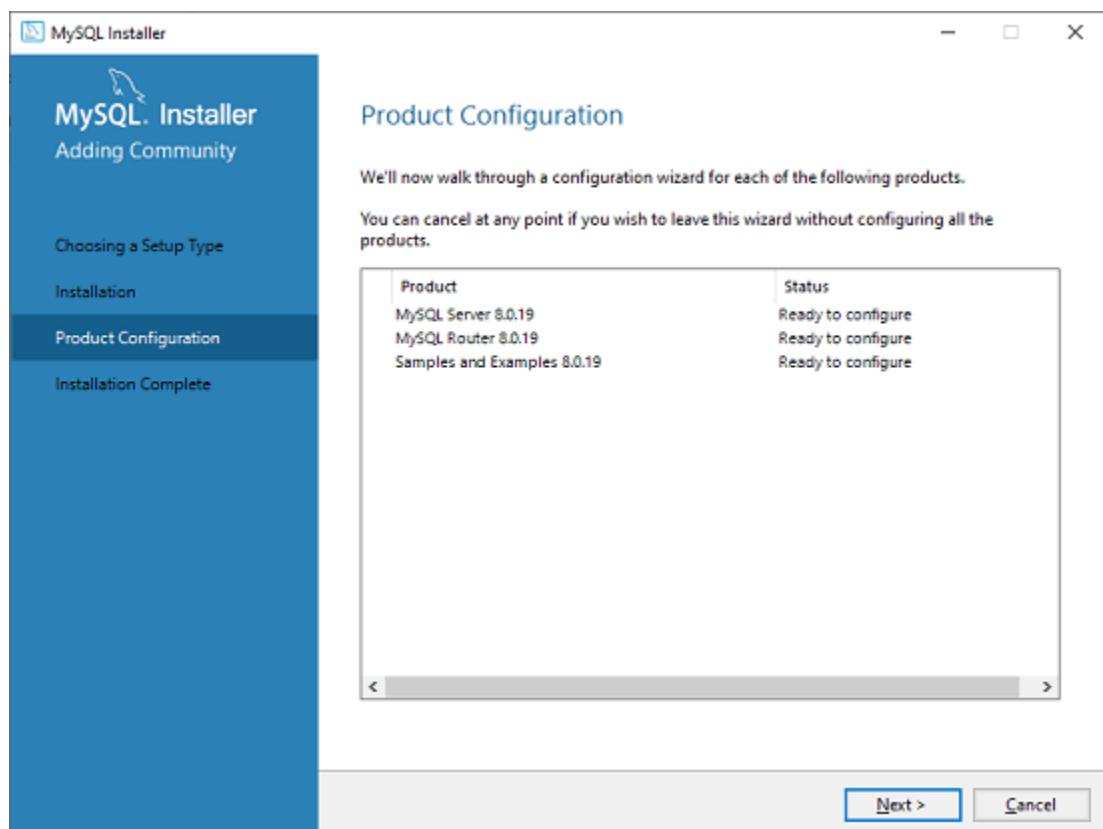
After clicking on the Yes button, we will see the list of the products which are going to be installed. So, if we need all products, click on the Execute button.



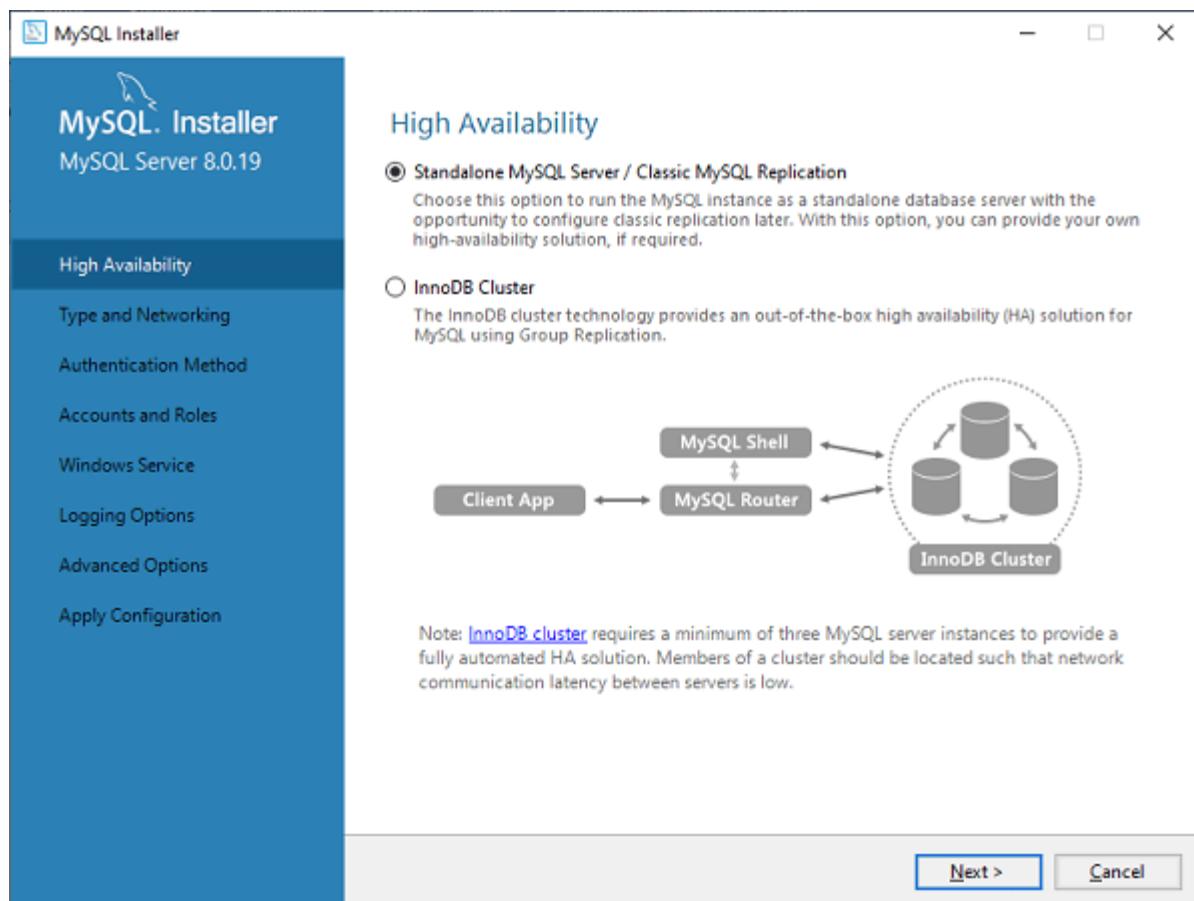
**Step 5:** Once we click on the Execute button, it will download and install all the products. After completing the installation, click on the Next button.



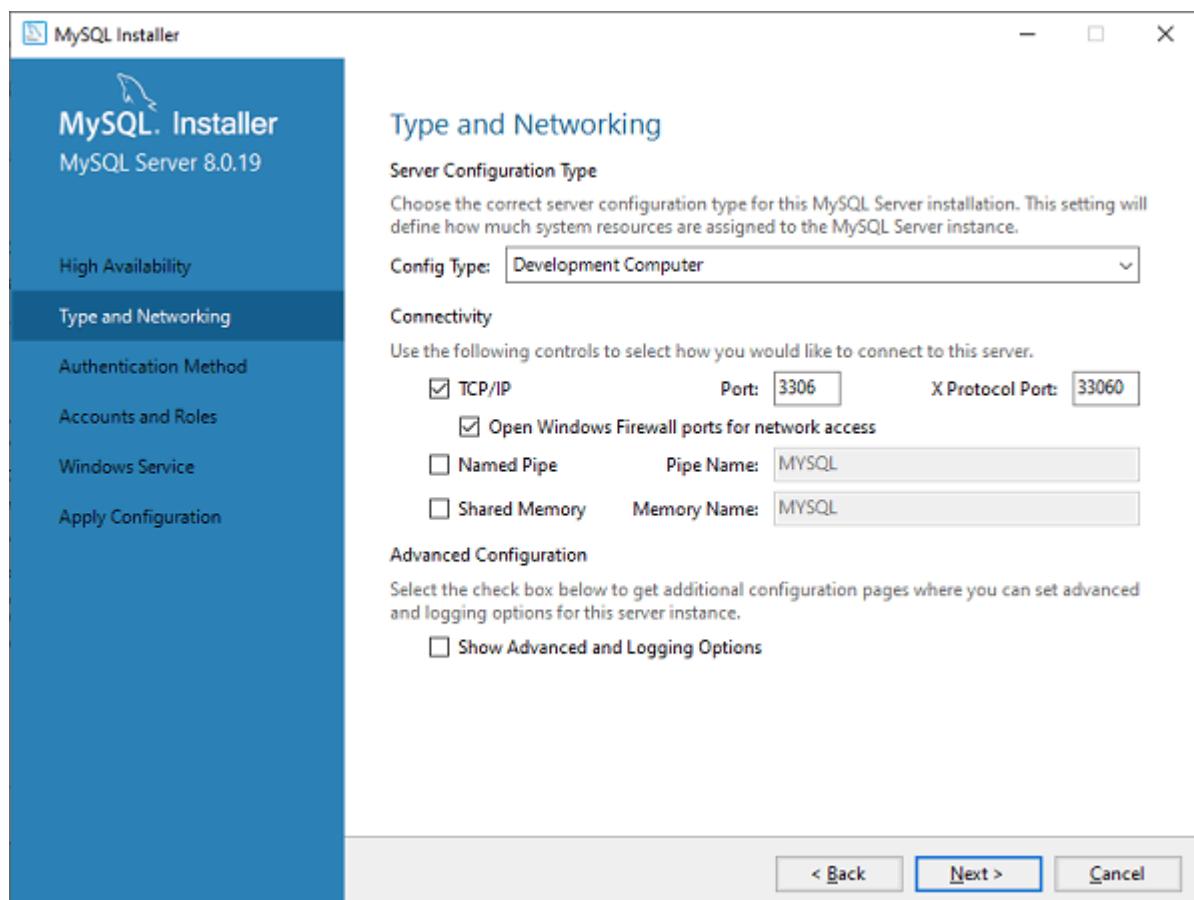
**Step 6:** In the next wizard, we need to configure the MySQL Server and Router. Here, I am not going to configure the Router because there is no need to use it with MySQL. We are going to show you how to configure the server only. Now, click on the Next button.



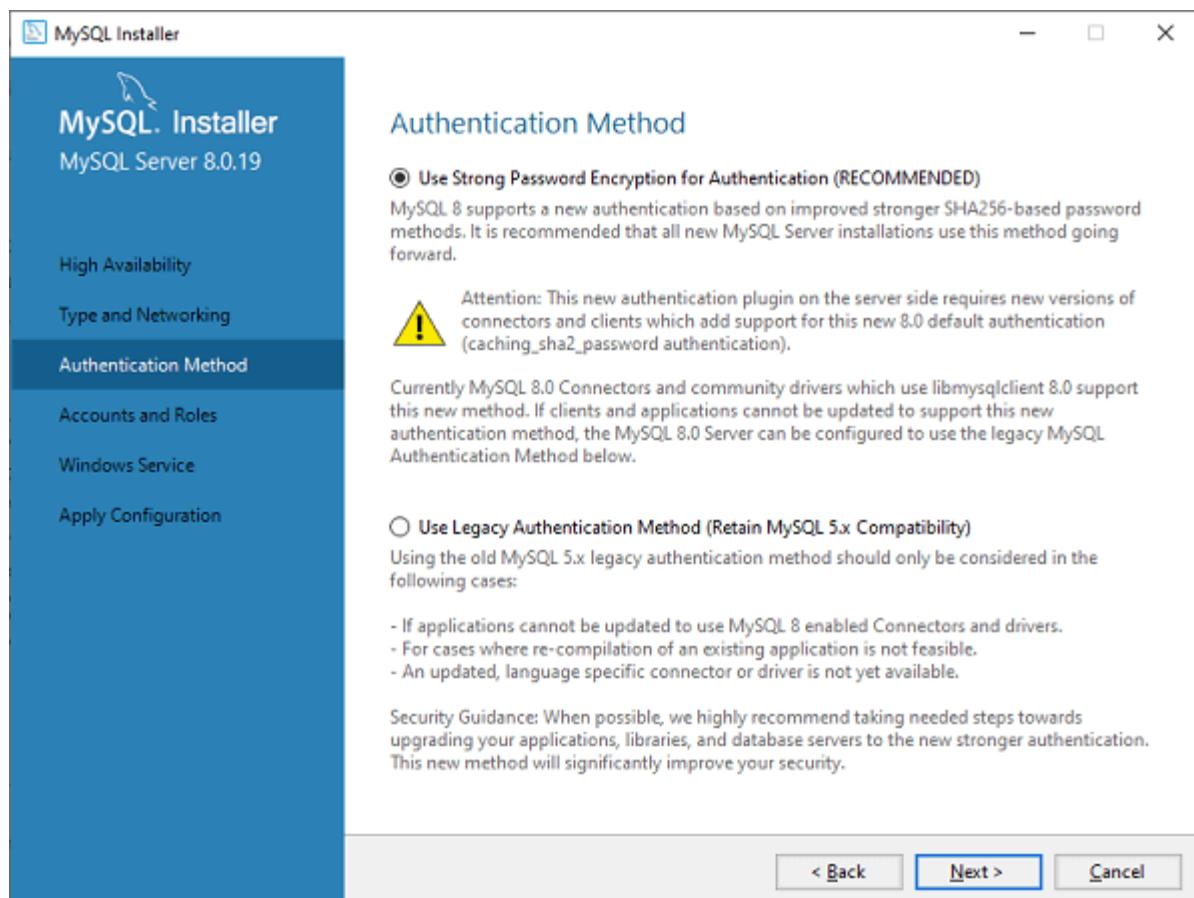
**Step 7:** As soon as you will click on the Next button, you can see the screen below. Here, we have to configure the MySQL Server. Now, choose the Standalone MySQL Server/Classic MySQL Replication option and click on Next. Here, you can also choose the InnoDB Cluster based on your needs.



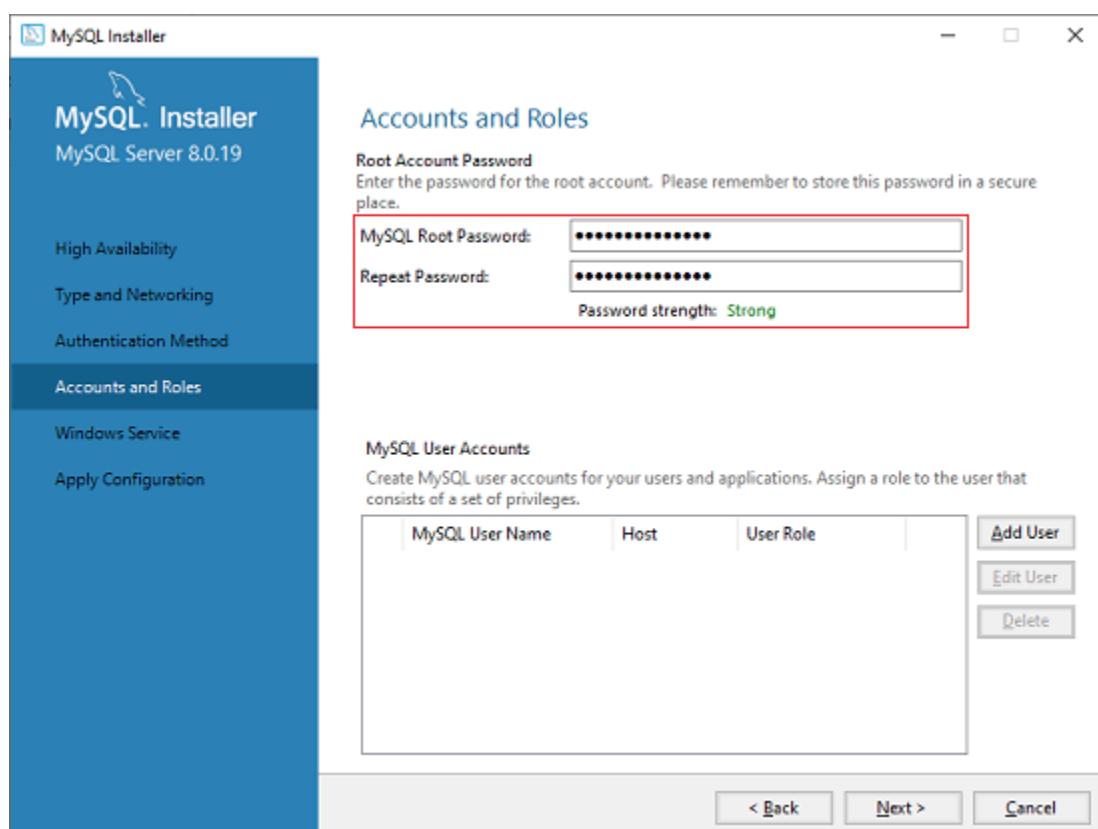
**Step 8:** In the next screen, the system will ask you to choose the Config Type and other connectivity options. Here, we are going to select the **Config Type** as 'Development Machine' and Connectivity as **TCP/IP**, and **Port Number** is 3306, then click on Next.



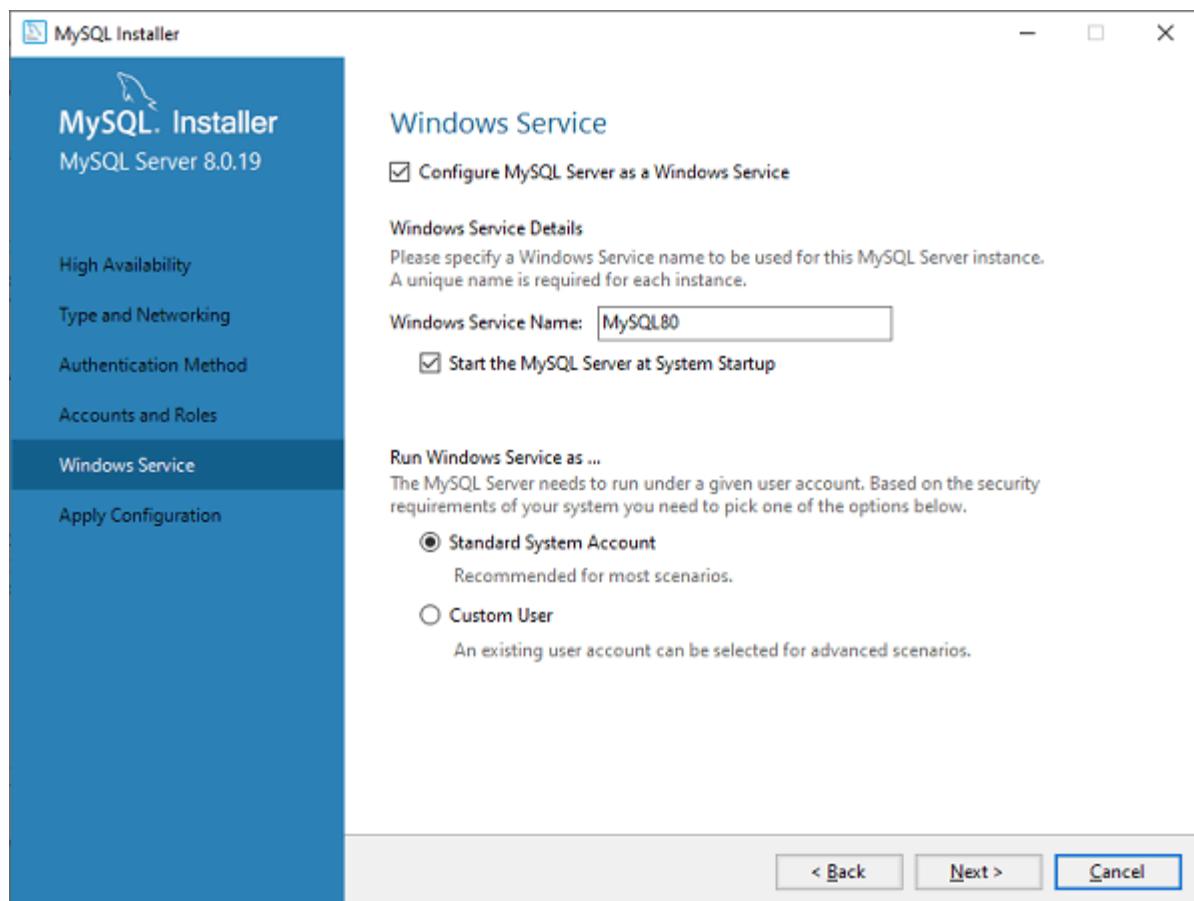
**Step 9:** Now, select the Authentication Method and click on Next. Here, I am going to select the first option.



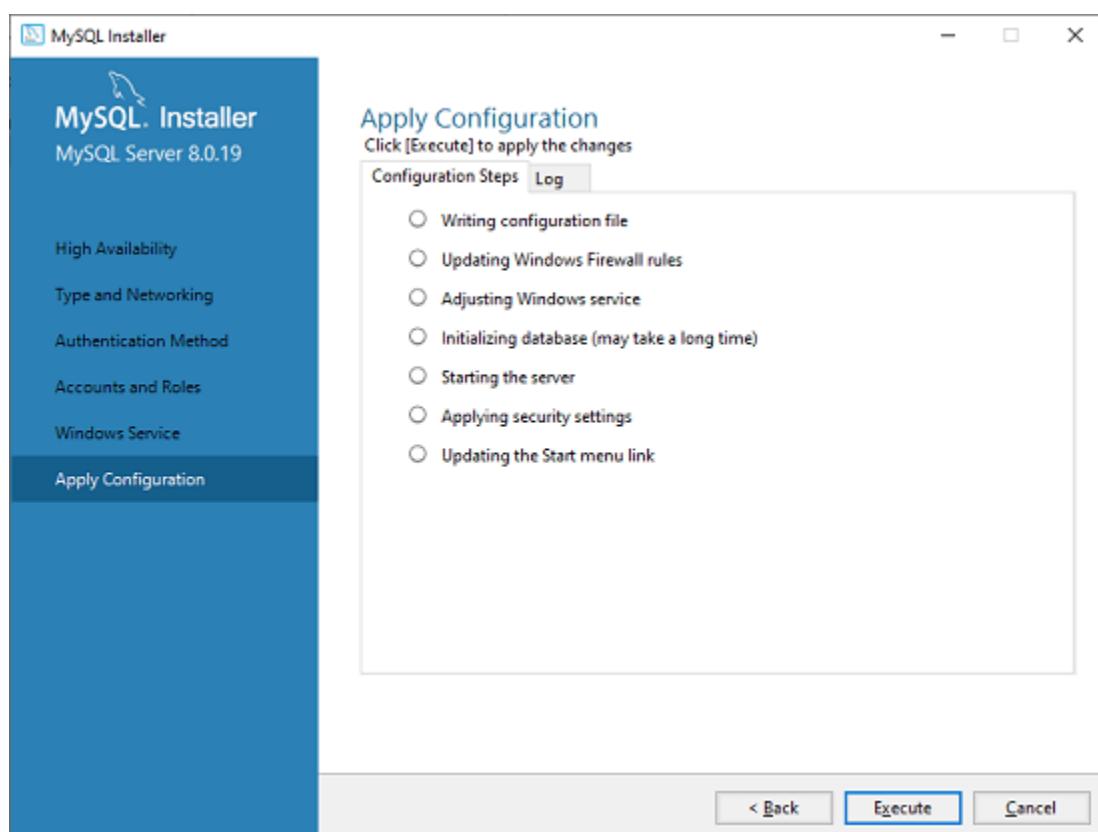
**Step 10:** The next screen will ask you to mention the MySQL Root Password. After filling the password details, click on the Next button.



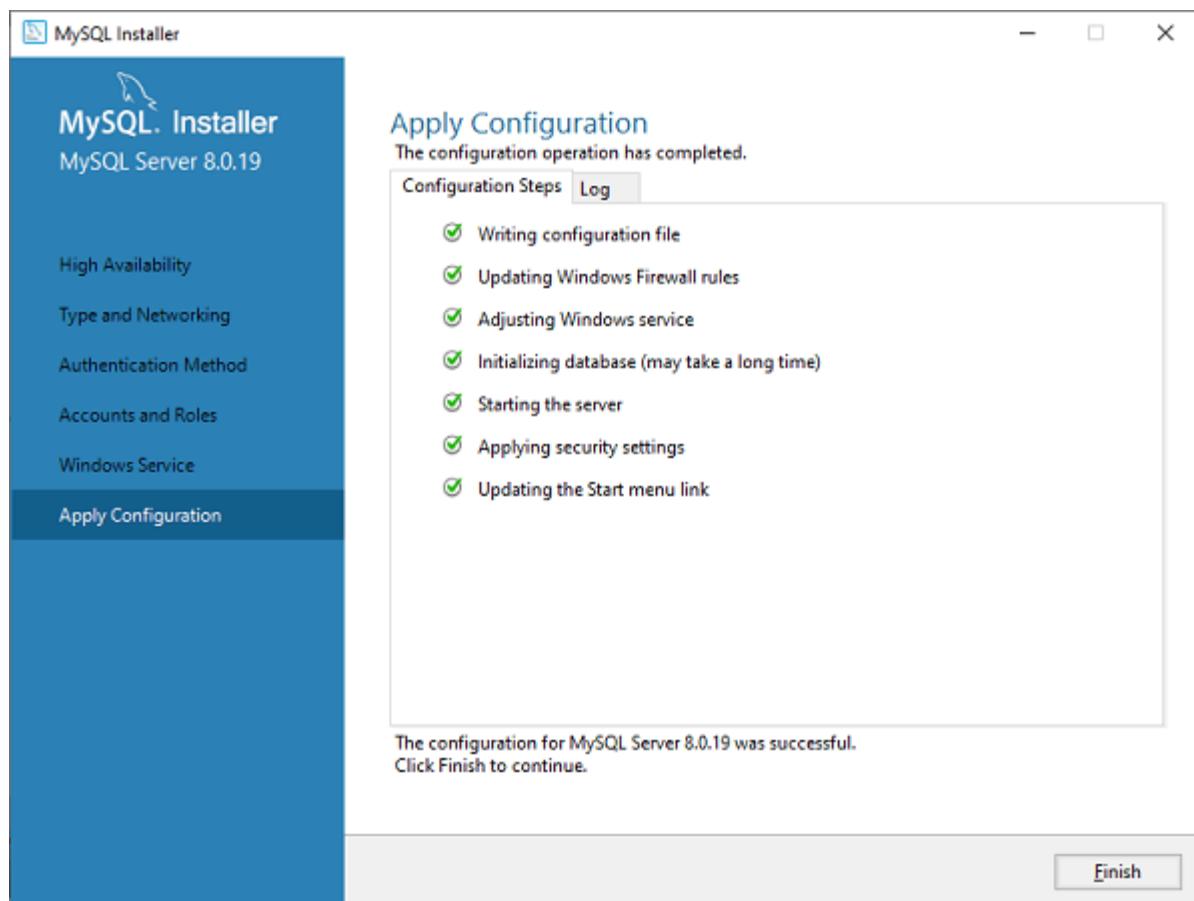
**Step 11:** The next screen will ask you to configure the Windows Service to start the server. Keep the default setup and click on the Next button.



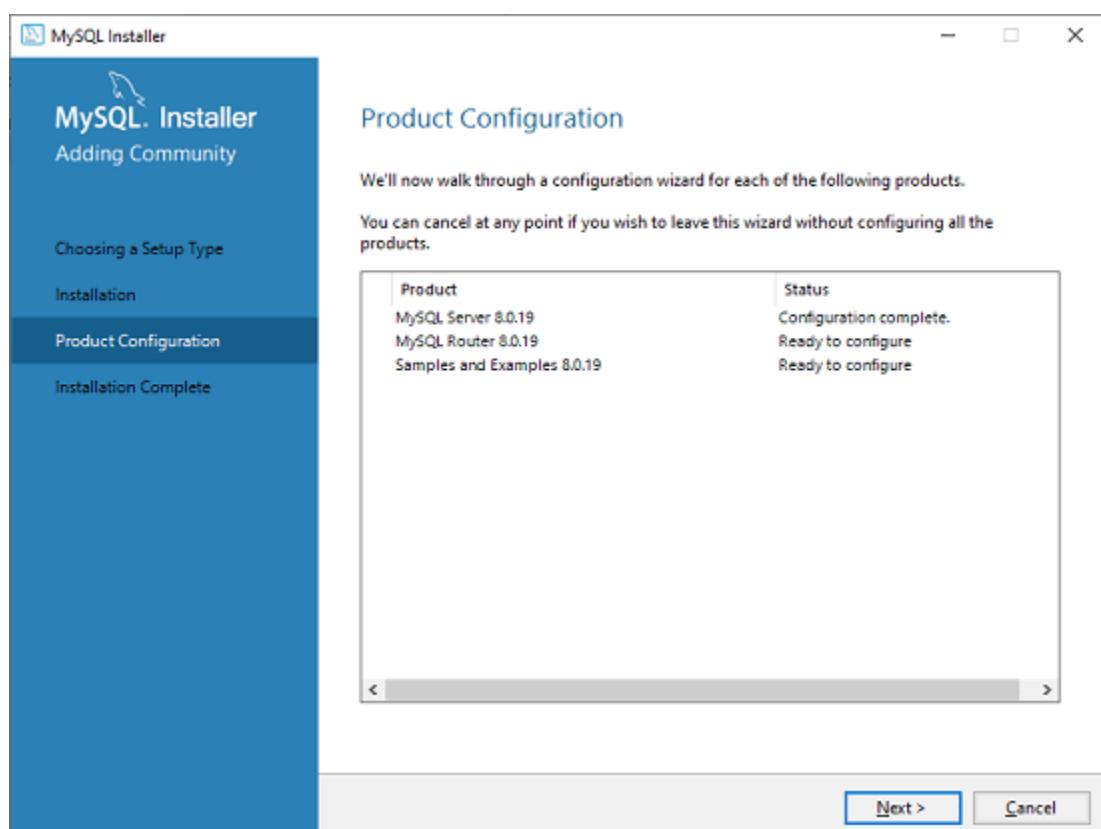
**Step 12:** In the next wizard, the system will ask you to apply the Server Configuration. If you agree with this configuration, click on the Execute button.



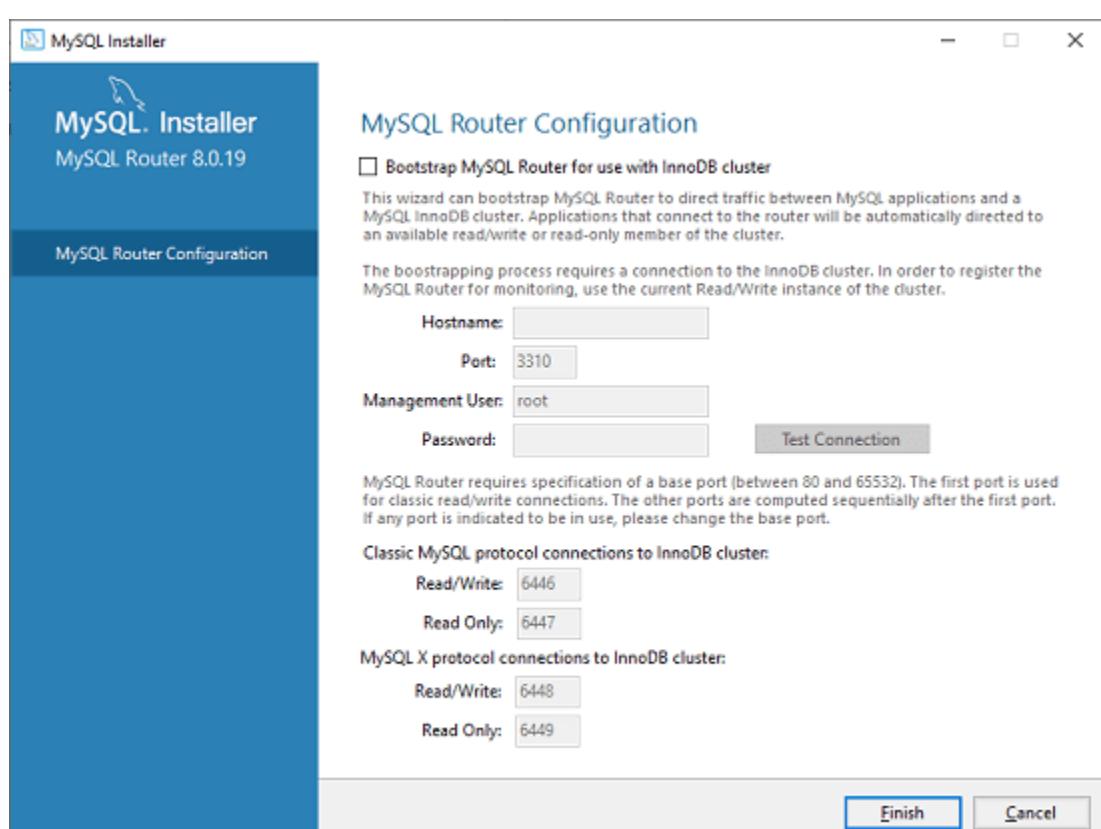
**Step 13:** Once the configuration has completed, you will get the screen below. Now, click on the **Finish** button to continue.



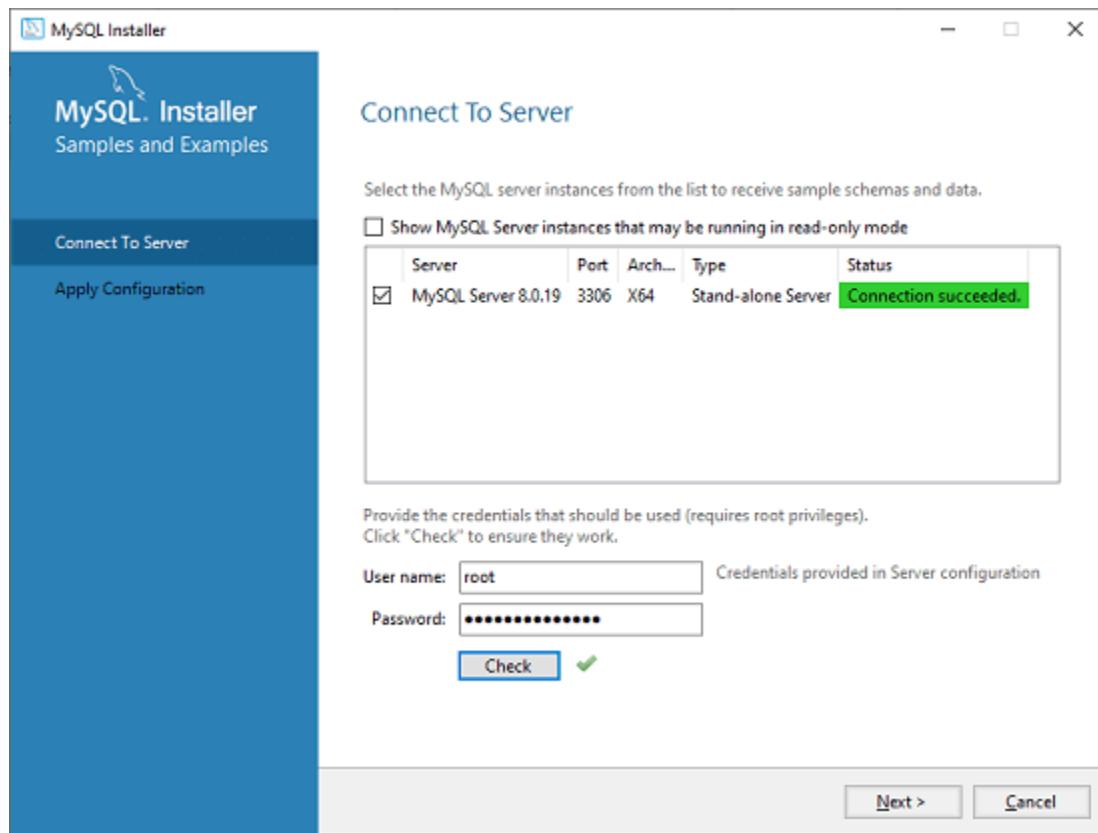
**Step 14:** In the next screen, you can see that the Product Configuration is completed. Keep the default setting and click on the Next-> Finish button to complete the MySQL package installation.



**Step 15:** In the next wizard, we can choose to configure the Router. So click on Next->Finish and then click the Next button.

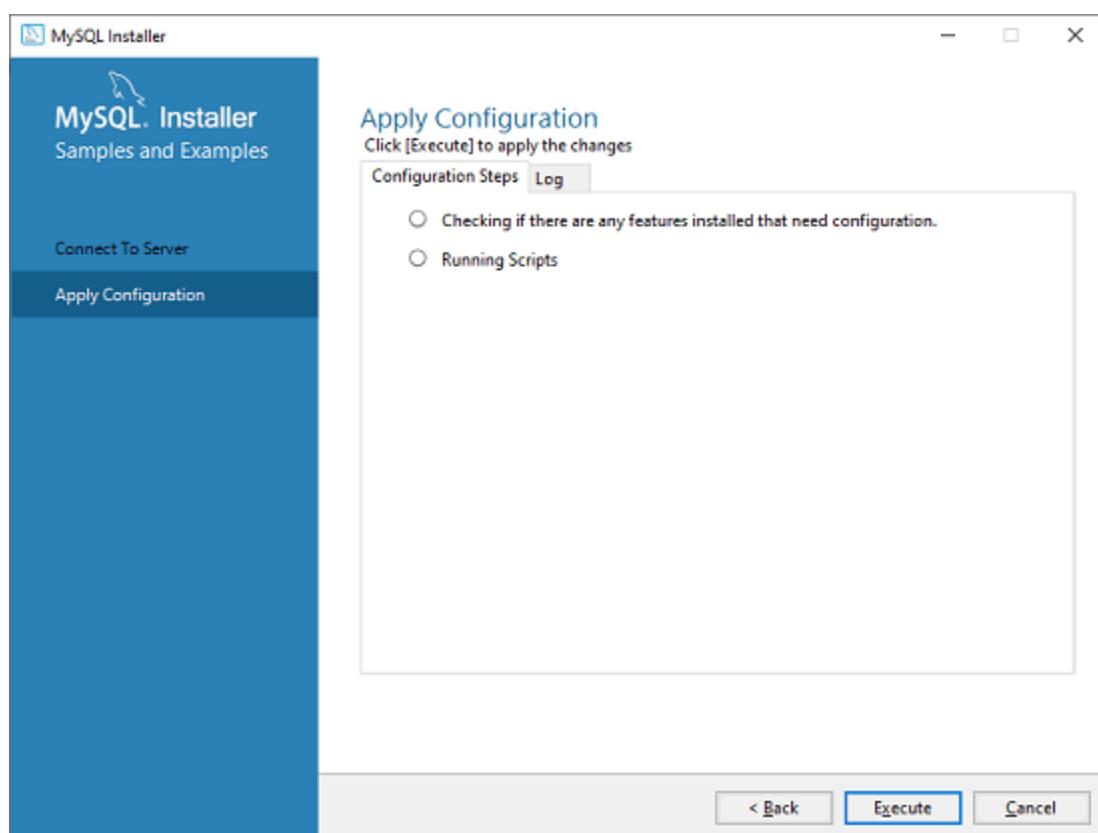


**Step 16:** In the next wizard, we will see the Connect to Server option. Here, we have to mention the root password, which we had set in the previous steps.

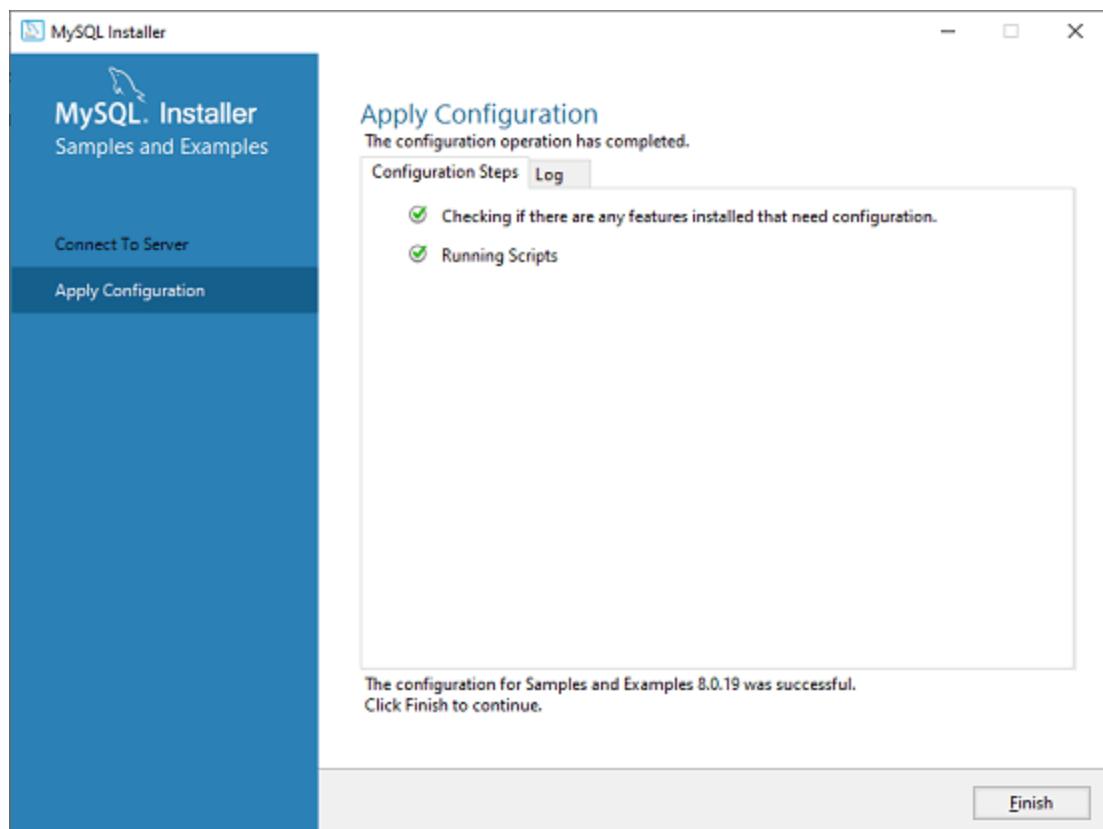


In this screen, it is also required to check about the connection is successful or not by clicking on the Check button. If the connection is successful, click on the Execute button. Now, the configuration is complete, click on Next.

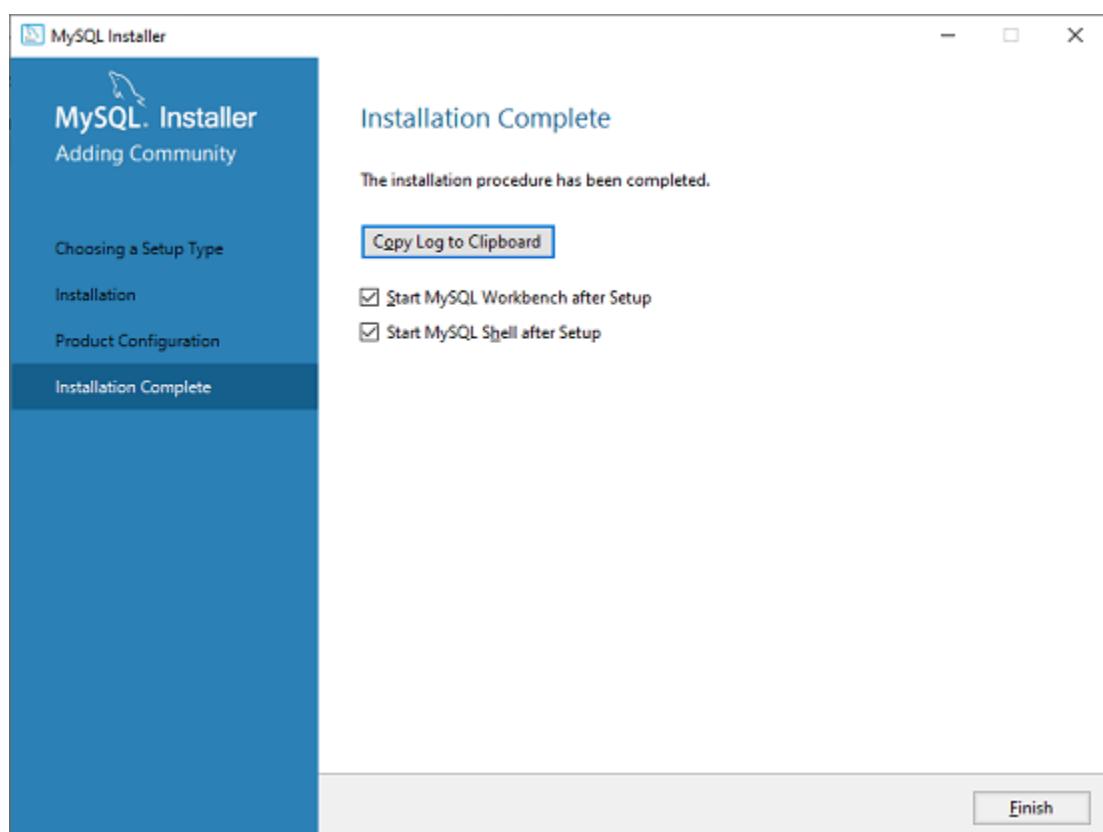
**Step 17:** In the next wizard, select the applied configurations and click on the Execute button.



**Step 18:** After completing the above step, we will get the following screen. Here, click on the Finish button.



**Step 19:** Now, the MySQL installation is complete. Click on the Finish button.

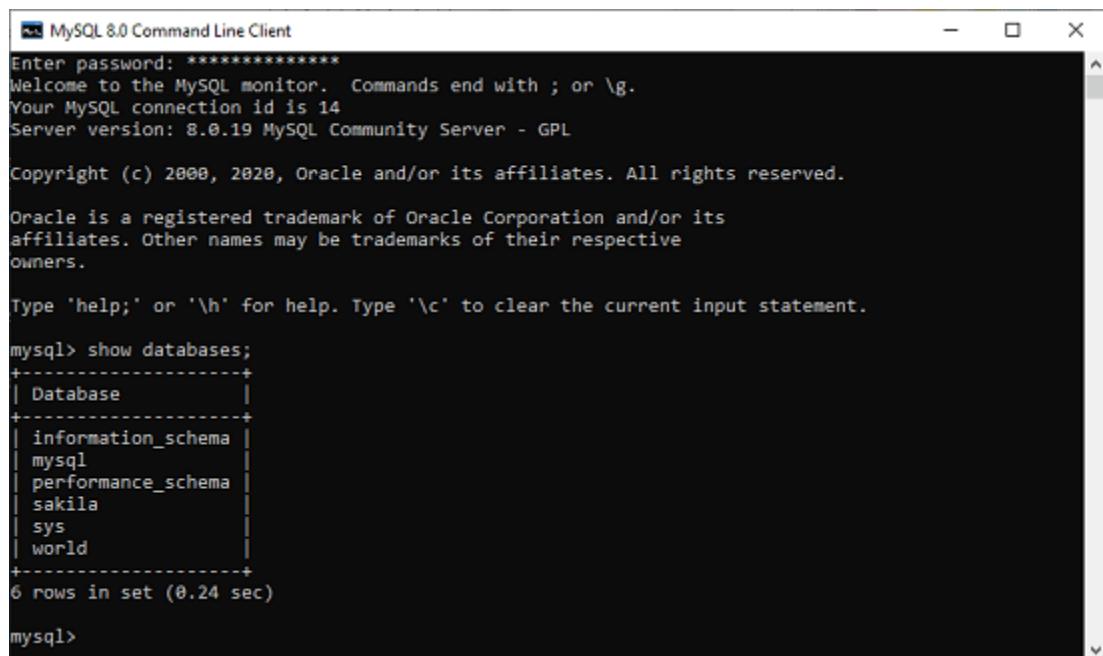


## Verify MySQL installation

Once MySQL has been successfully installed, the base tables have been initialized, and the server has been started, you can verify its working via some simple tests.

Open your MySQL **Command Line Client**; it should have appeared with a **mysql> prompt**. If you have set any password, write your password here. Now, you are connected to the MySQL server, and you can execute all the SQL command at mysql> prompt as follows:

**For example:** Check the already created databases with show databases command:



```
MySQL 8.0 Command Line Client
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 14
Server version: 8.0.19 MySQL Community Server - GPL

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+--------------------+
| Database          |
+--------------------+
| information_schema|
| mysql              |
| performance_schema|
| sakila             |
| sys                |
| world              |
+--------------------+
6 rows in set (0.24 sec)

mysql>
```

## What is the Primary key?

A primary key is a single field or combination of fields that contain a unique record. It must be filled. None of the fields of the primary key can contain a null value. A table can have only one primary key.

**NOTE:** In Oracle, the total number of columns cannot be more than 32.

## MySQL Connection

A connection is a computer science facility that allows the user to connect with the database server software. **A user can connect with the database server, whether on the same machine or remote locations.** Therefore, if we want to work with the database server to send commands and receive answers in the form of a result set, we need connections. In this article, we are going to learn how we can connect to MySQL Server in various ways.

## MySQL Connection Types

MySQL provides various ways to connect with the database server. **Once we have installed the MySQL server, we can connect it using any of the client programs that are listed below:**

1. Command-line client
2. MySQL Workbench
3. PHP Script.

### MySQL Server Connection Using command-line client

MySQL command-line client program provides interaction with the database server in an interactive and non-interactive mode. We can see this program in the **bin directory of the MySQL's installation folder**. We can open the MySQL command prompt by navigating to the bin directory of the MySQL's installation folder and type:

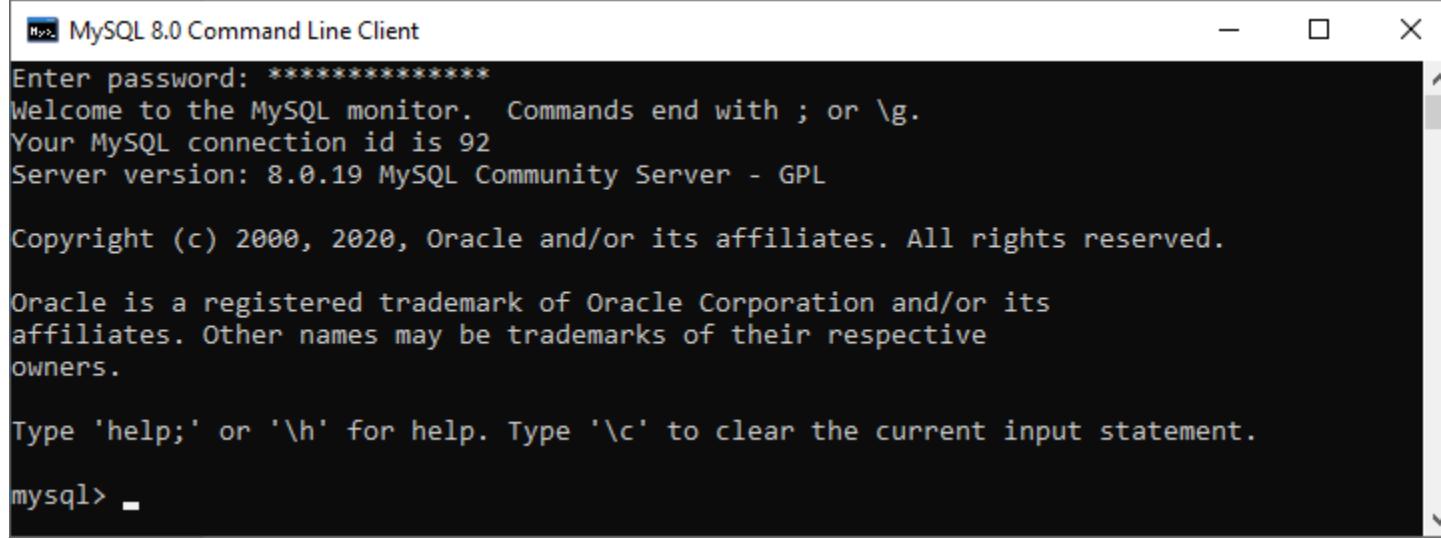
1. MySQL

If we find the MySQL program in the **PATH**, we can use the below command to connect to the MySQL Server:

1. mysql -u root -p

In the syntax, the **-u root indicates** that we will connect to the MySQL server using the root user account and **-p** instructs MySQL to ask for a password.

Next, we need to type the password for the root user account and press **Enter**. If everything is correct, it should give the screen as follows:



MySQL 8.0 Command Line Client

```
Enter password: *****
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 92
Server version: 8.0.19 MySQL Community Server - GPL

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

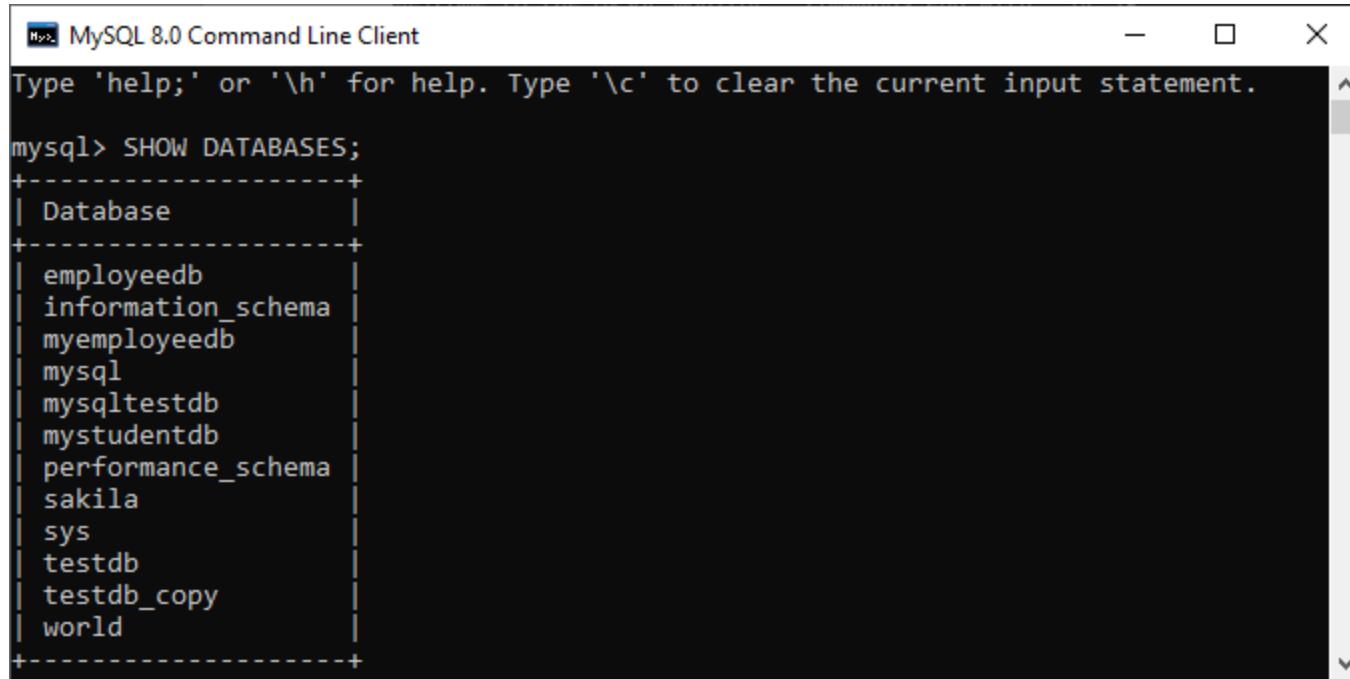
mysql> -
```

This screen indicates that we have successfully connected with the MySQL database server, where we can send commands and receive answers in the form of a result set.

Suppose we want to display all databases available in the current server; we can use the command as follows:

1. mysql> SHOW DATABASES;

It will give the below output:



```
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> SHOW DATABASES;
+-----+
| Database |
+-----+
| employeedb
| information_schema
| myemployeedb
| mysql
| mysqltestdb
| mystudentdb
| performance_schema
| sakila
| sys
| testdb
| testdb_copy
| world
+-----+
```

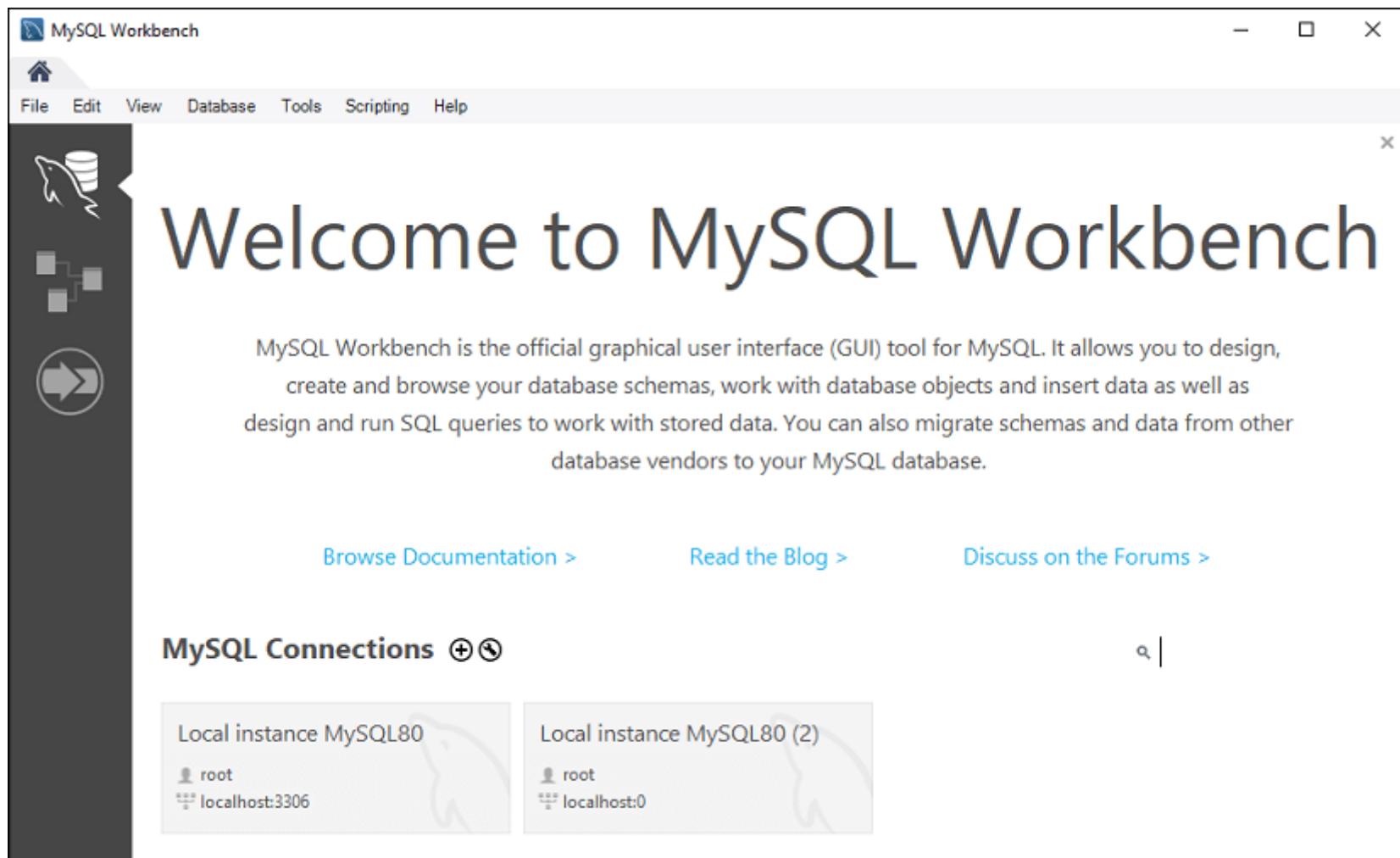
If you want **to disconnect the opened MySQL database server**, you need to use the exit command.

1. mysql> EXIT;

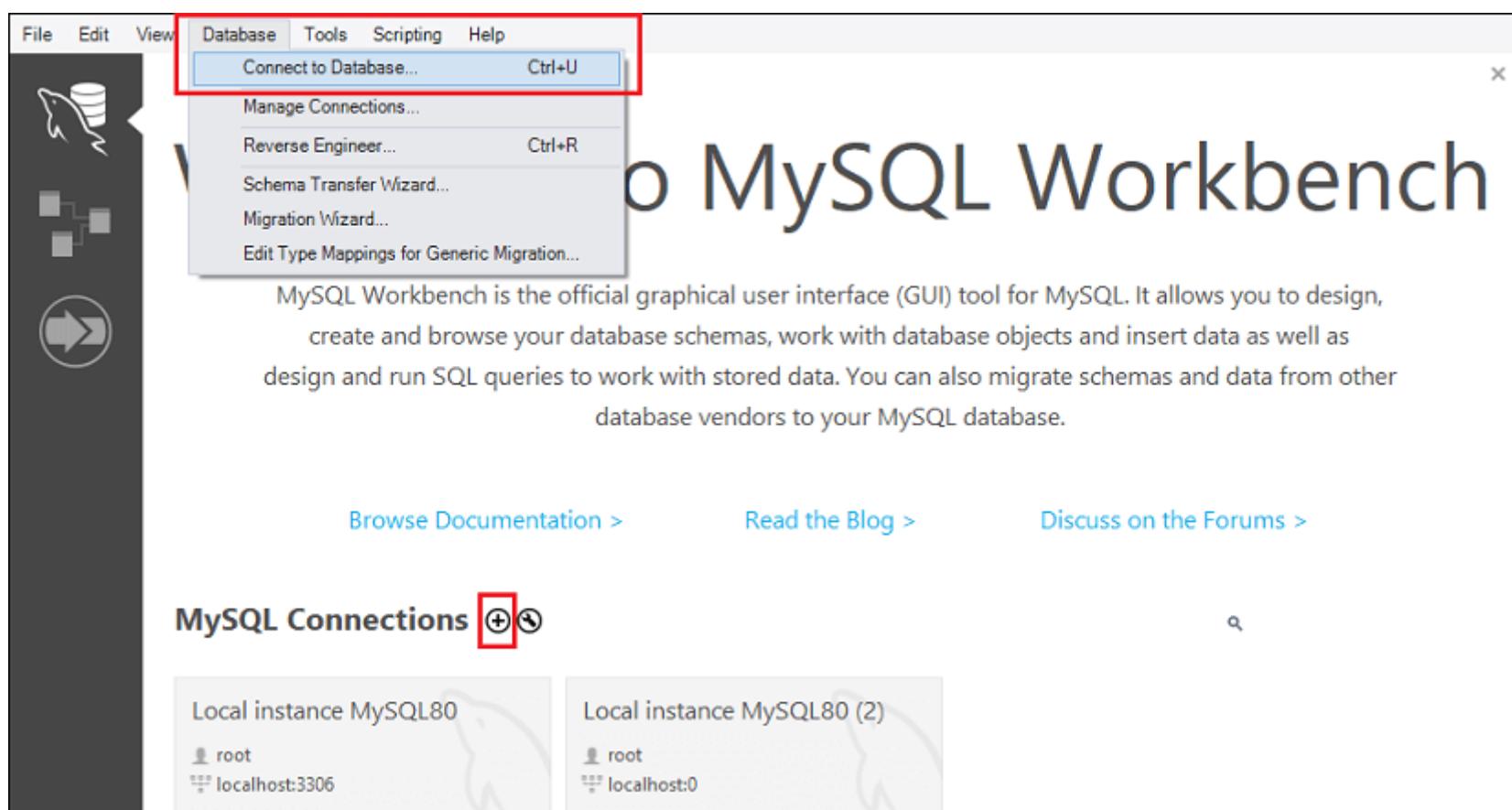
## Connect to Database Server Using MySQL Workbench

We can connect to the MySQL database server in workbench by using the following steps:

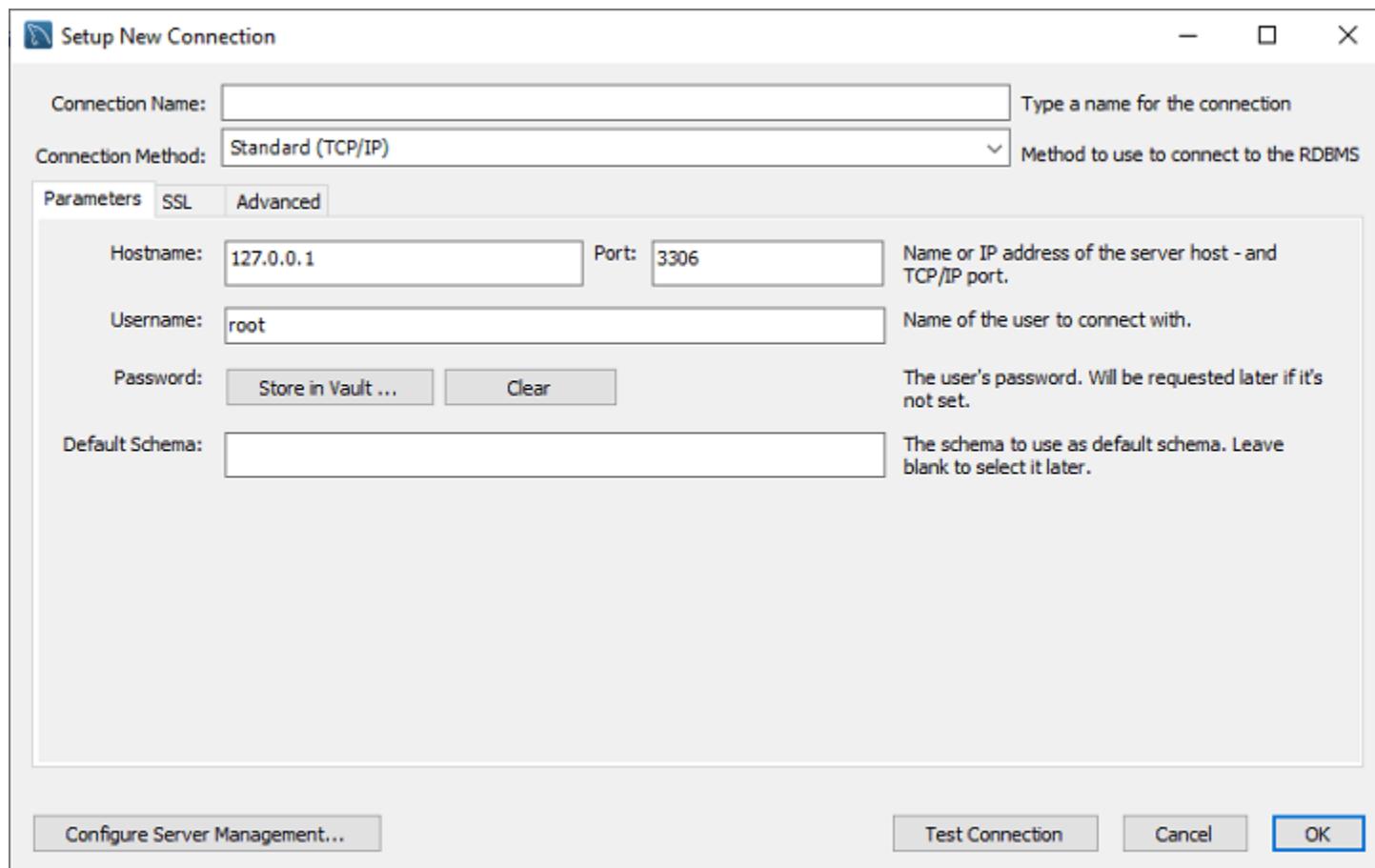
**Step 1:** Launch the MySQL Workbench. We should get the following screen:



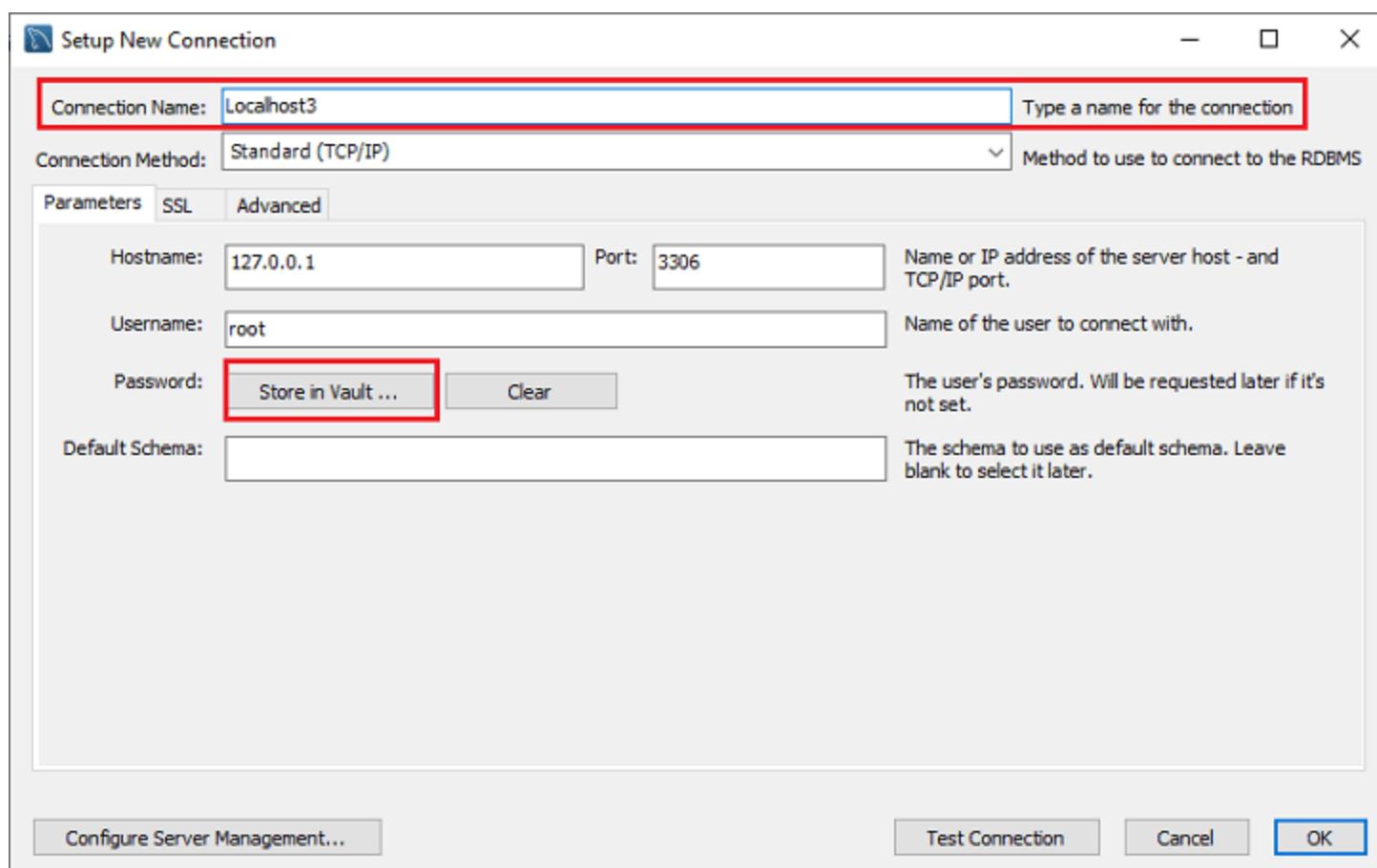
**Step 2:** Navigate to the menu bar, click on the '**Database**' and choose **Connect to Database** option or press the **CTRL+U** command. We can also connect with the database server by just clicking the **plus (+) button** located next to the MySQL Connections. See the below image:



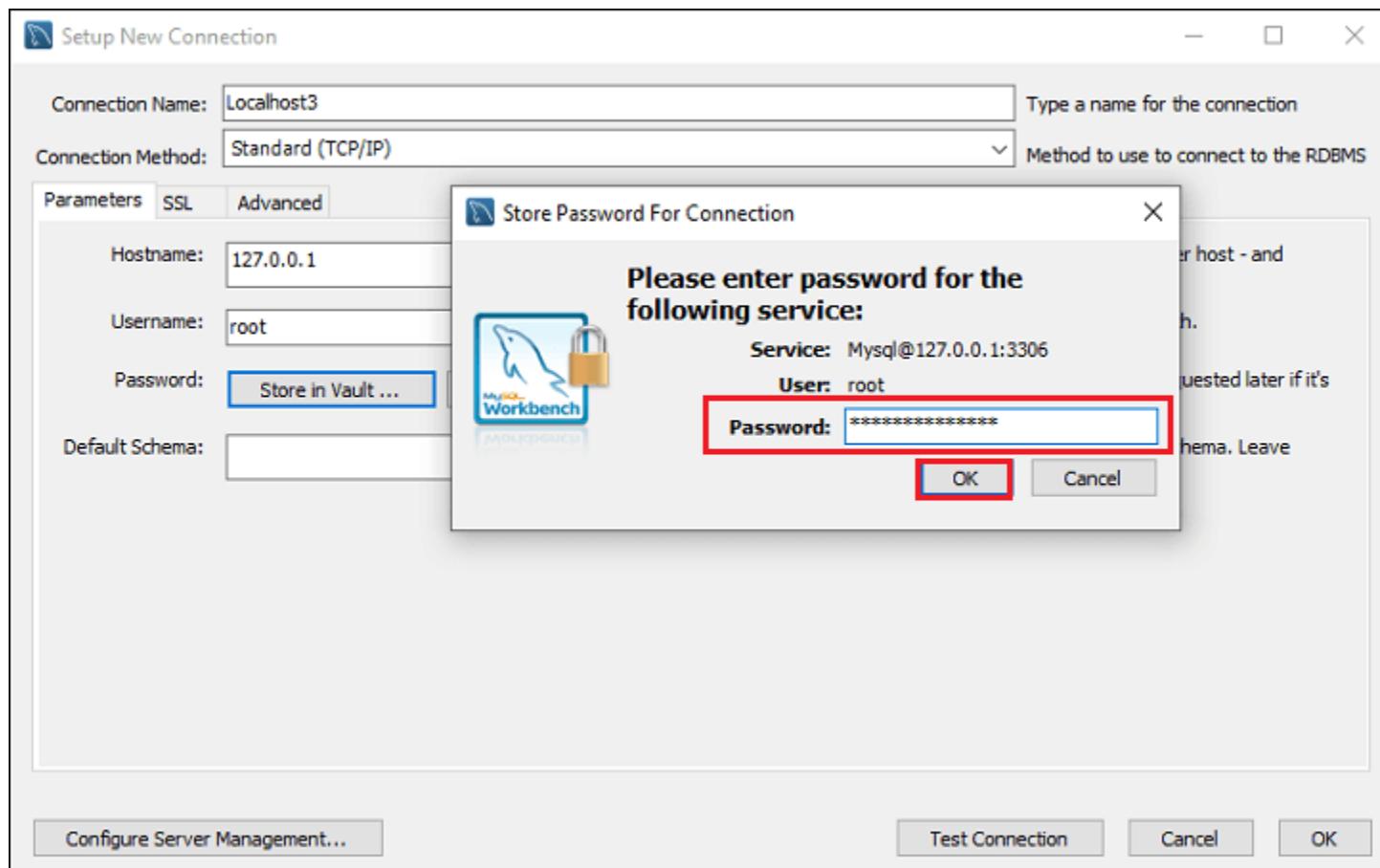
**Step 3:** After choosing any of the options, we will get the below screen:



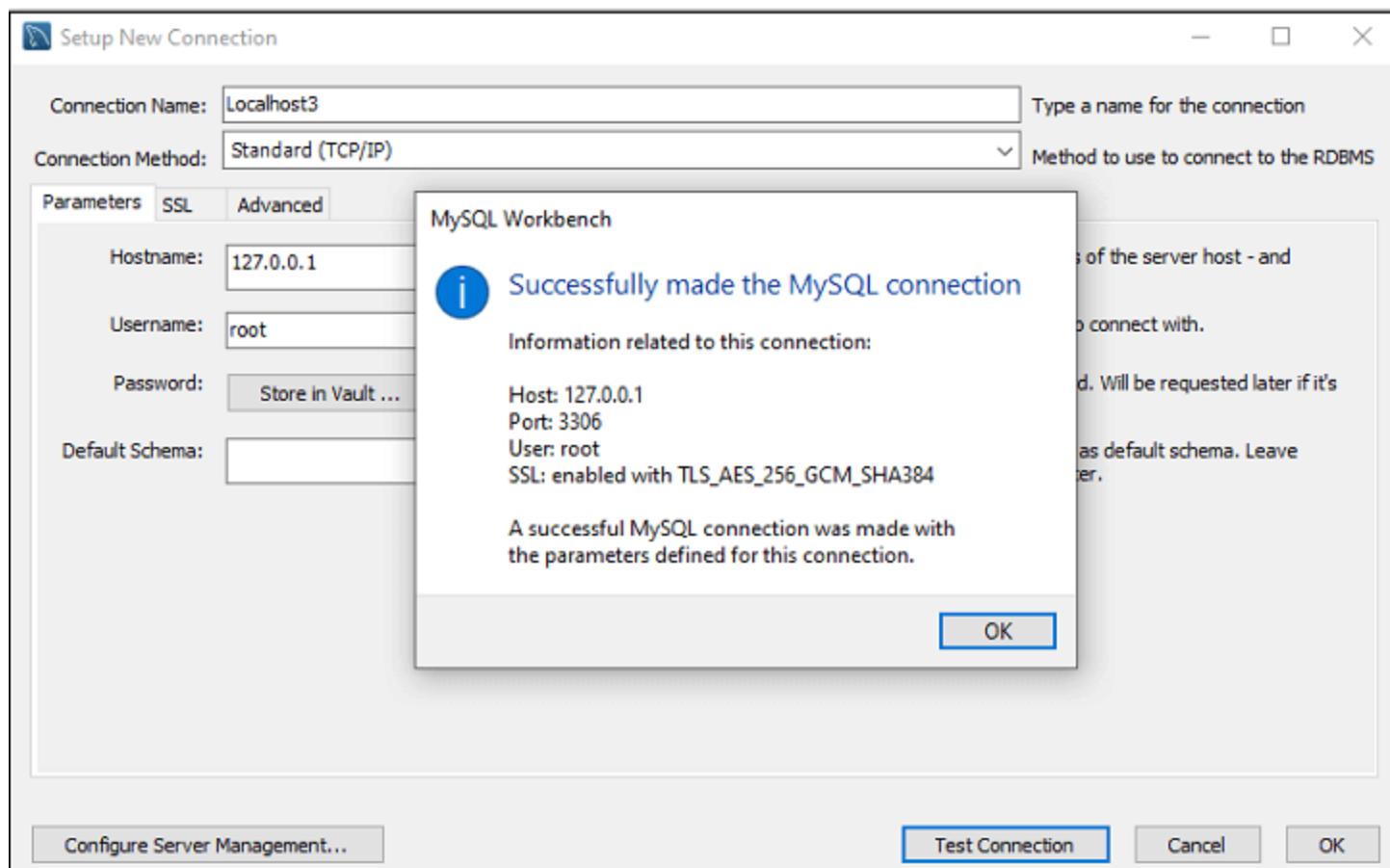
**Step 4:** Fill the box to create a connection, such as **connection name** and **username**, whatever you want. By default, the username is the **root**, but we can also change it with a different username in the Username textbox. After filling all boxes, click the **Store in Vault ... button** to write the password for the given user account.



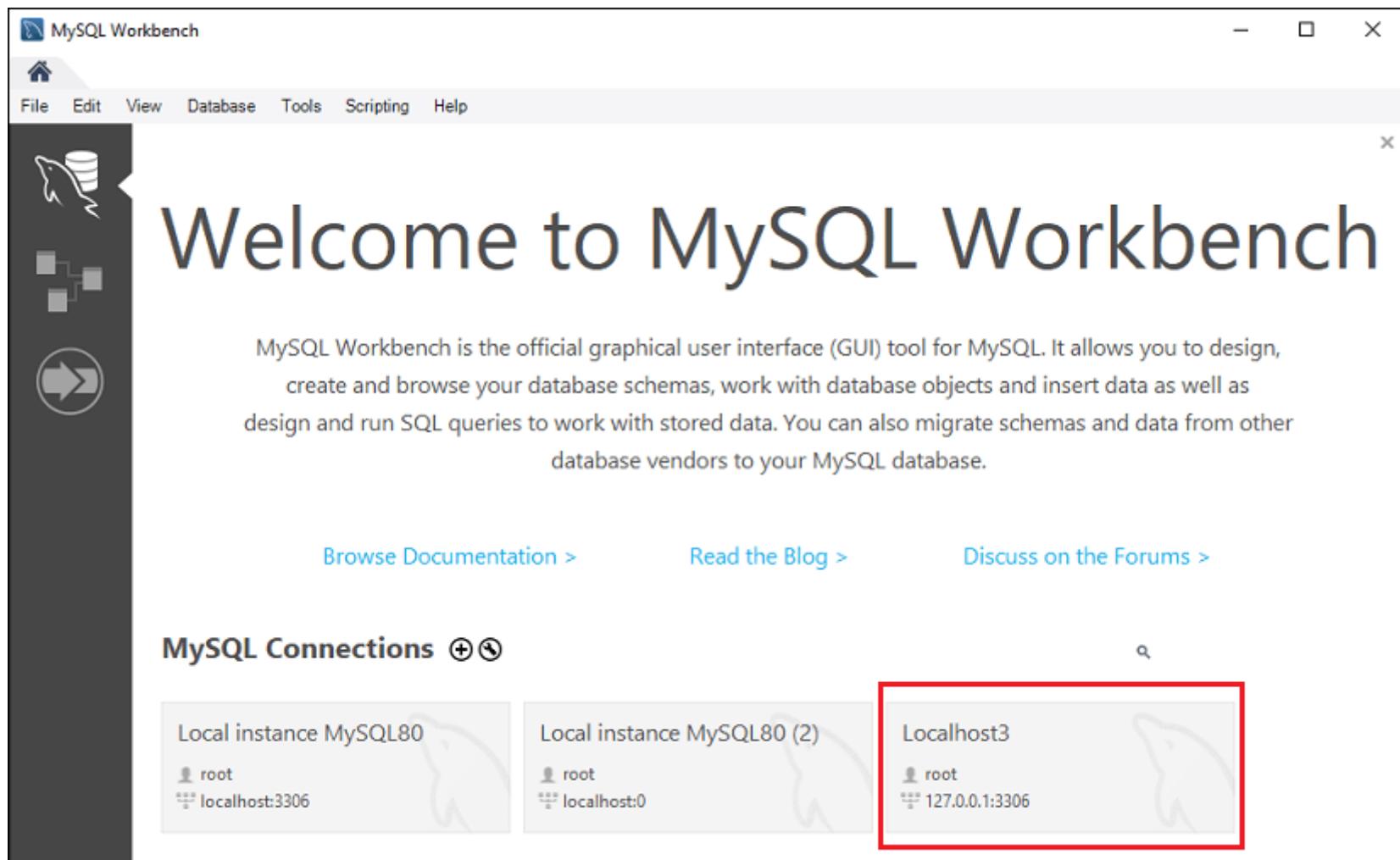
**Step 5:** We will get a new window to write the password and click the **OK** button.



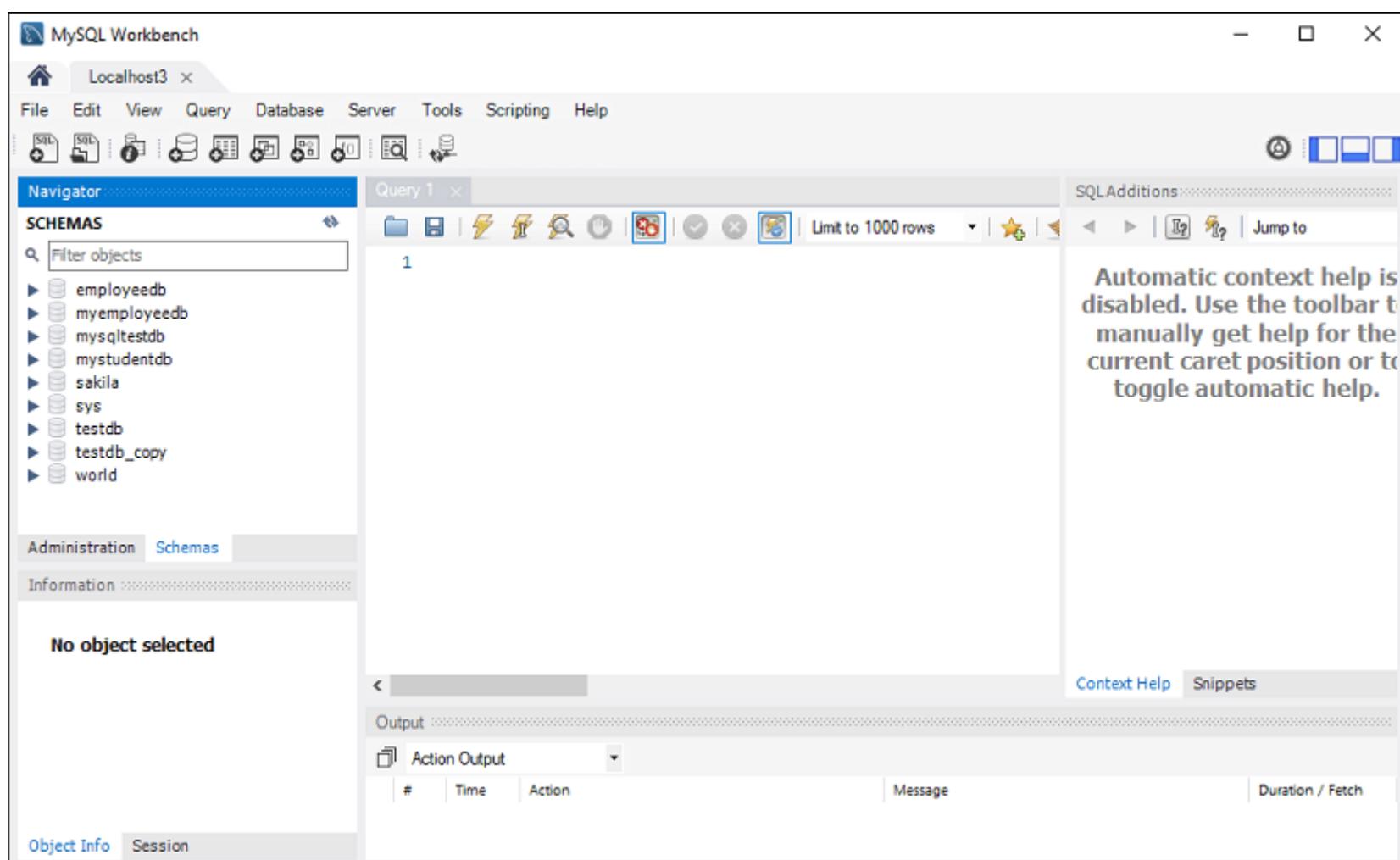
**Step 6:** After entering all the details, click on the **Test Connection** to test the database connectivity is successful or not. If the connection is successful, click on the **OK** button.



**Step 7:** Again, click on the **OK** button for saving connection setup. After finishing all the setup, we can see this connection under **MySQL Connections** for connecting to the MySQL database server. See the below output where we have **localhost3** connection name:



**Step 8:** Now, we can click this newly created connection that displays the current schemas and a pane for entering queries:



## Connect to MySQL Server Using PHP Script

The simplest way to connect with the MySQL database server using the PHP script is to use the **mysql\_connect() function**. This function needs **five parameters** and returns the MySQL link identifier when the connection becomes successful. If the connection is failed, it returns **FALSE**.

### Syntax

The following is the syntax for MySQL connection using PHP Script:

1. **connection** mysql\_connect(server, **user**, **passwordd**, new\_link, client\_flag);

### Let us explain the mysql\_connect() function parameters:

**Server:** It is the name of a host that runs the database server. By default, its value will be Icalhost:3306.

**User:** It is the name of a user who accesses the database. If we will not specify this field, it assumes the default value that will be the name of a user that owns the server process.

**Password:** It is the password of a user whose database you are going to access. If we will not specify this field, it assumes the default value that will be an empty password.

**New\_link:** If we make a second call with the same arguments in the mysql\_connect() function, MySQL does not establish a new connection. Instead, we will get the identifier of the already opened database connection.

**Client\_flags:** This parameter contains a combination of the below constants:

- MYSQL\_CLIENT\_SSL: It uses SSL encryption.
- MYSQL\_CLIENT\_COMPRESS: It uses a compression protocol.
- MYSQL\_CLIENT\_IGNORE\_SPACE: It provides space after function names.
- MYSQL\_CLIENT\_INTERACTIVE: It provides a timeout before closing the connection.

If we want to disconnect from the MySQL database server, we can use another PHP function named **mysql\_close()**. It accepts only a single parameter that will be a connection returned by the mysql\_connect() function. Its syntax is given below:

1. bool mysql\_close ( resource \$link\_identifier );

If we do not specify any resource, MySQL will close the last opened database. This function returns true when the connection is closed successfully. Otherwise returns a FALSE value.

### Example

The following example explain how to connect to a MySQL server using PHP Script:

```
1. <html>
2.   <head>
3.     <title>MySQL Server Connection</title>
4.   </head>
5.   <body>
6.     <?php
7.       $servername = 'localhost:3306';
8.       $username = 'javatpoint';
9.       $dbpass = 'jtp123';
10.      $conn = mysql_connect($servername, $username, $password);
11.      if(! $conn ) {
12.        die('Connection failed: ' . mysql_error());
13.      }
14.      echo 'Connection is successful';
15.      mysql_close($conn);
16.    ?>
17.   </body>
18. </html>
```

## MySQL Workbench

# MySQL Workbench (Download and Installation)

MySQL Workbench is a unified visual database designing or graphical user interface tool used for working with database architects, developers, and Database Administrators. It is developed and maintained by Oracle. It provides SQL development, data modeling, data migration, and comprehensive administration tools for server configuration, user administration, backup, and many more. We can use this Server Administration for creating new physical data models, E-R diagrams, and for SQL development (run queries, etc.). It is available for all major operating systems like Mac OS, Windows, and Linux. MySQL Workbench fully supports MySQL Server version v5.6 and higher.

MySQL Workbench covers **five main functionalities**, which are given below:

**SQL Development:** This functionality provides the capability that enables you to execute SQL queries, create and manage connections to the database Servers with the help of built-in SQL editor.

**Data Modelling (Design):** This functionality provides the capability that enables you to create models of the database Schema graphically, performs reverse and forward engineering between a Schema and a live database, and edit all aspects of the database using the comprehensive Table editor. The Table editor gives the facilities for editing tables, columns, indexes, views, triggers, partitioning, etc.

**Server Administration:** This functionality enables you to administer MySQL Server instances by administering users, inspecting audit data, viewing database health, performing backup and recovery, and monitoring the performance of MySQL Server.

**Data Migration:** This functionality allows you to migrate from Microsoft SQL Server, SQLite, Microsoft Access, PostgreSQL, Sybase ASE, SQL Anywhere, and other RDBMS tables, objects, and data to MySQL. It also supports migrating from the previous versions of MySQL to the latest releases.

**MySQL Enterprise Supports:** This functionality gives the support for Enterprise products such as MySQL firewall, MySQL Enterprise Backup, and MySQL Audit.

## MySQL Workbench Editions

MySQL Workbench is mainly available in three editions, which are given below:

1. Community Edition (Open Source, GPL)
2. Standard Edition (Commercial)
3. Enterprise Edition (Commercial)

### Community Edition

The Community Edition is an open-source and freely downloadable version of the most popular database system. It came under the GPL license and is supported by a huge community of developers.

### Standard Edition

It is the commercial edition that provides the capability to deliver high-performance and scalable Online Transaction Processing (OLTP) applications. It has made MySQL famous along with industrial-strength, performance, and reliability.

### Enterprise Edition

It is the commercial edition that includes a set of advanced features, management tools, and technical support to achieve the highest scalability, security, reliability, and uptime. This edition also reduces the risk, cost, complexity in the development, deployment, and managing MySQL applications.

Let us understand it with the following comparison chart.

Functionality	Community Edition	Standard Edition	Enterprise Edition
Visual SQL Development	Yes	Yes	Yes
Visual Database Administration	Yes	Yes	Yes
Performance Tuning	Yes	Yes	Yes
User and Session Management	Yes	Yes	Yes
Connection Management	Yes	Yes	Yes
Object Management	Yes	Yes	Yes
Data Management	Yes	Yes	Yes
Visual Data Modelling	Yes	Yes	Yes
Reverse Engineering	Yes	Yes	Yes

Forward Engineering	Yes	Yes	Yes
Schema Synchronization	Yes	Yes	Yes
Schema & Model Validation	No	Yes	Yes
DBDoc	No	Yes	Yes
GUI for MySQL Enterprise Backup	No	No	Yes
GUI for MySQL Enterprise Audit	No	No	Yes
GUI for MySQL Enterprise Firewall	No	Yes	Yes
Scripting & Plugins	Yes	Yes	Yes
Database Migration	Yes	Yes	Yes

## MySQL Workbench Environment Setup

Here, we are going to learn how we can download and install MySQL Workbench.

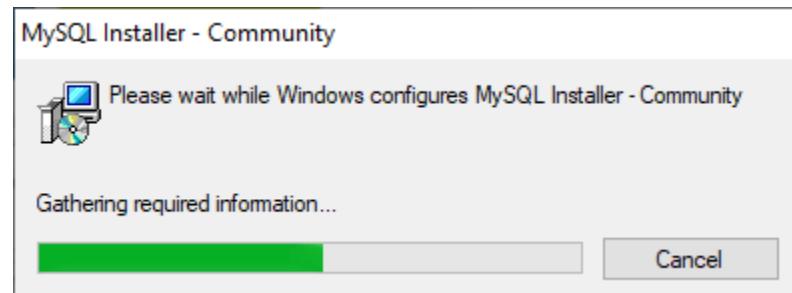
### Prerequisites

The following requirements should be available in your system to work with MySQL Workbench:

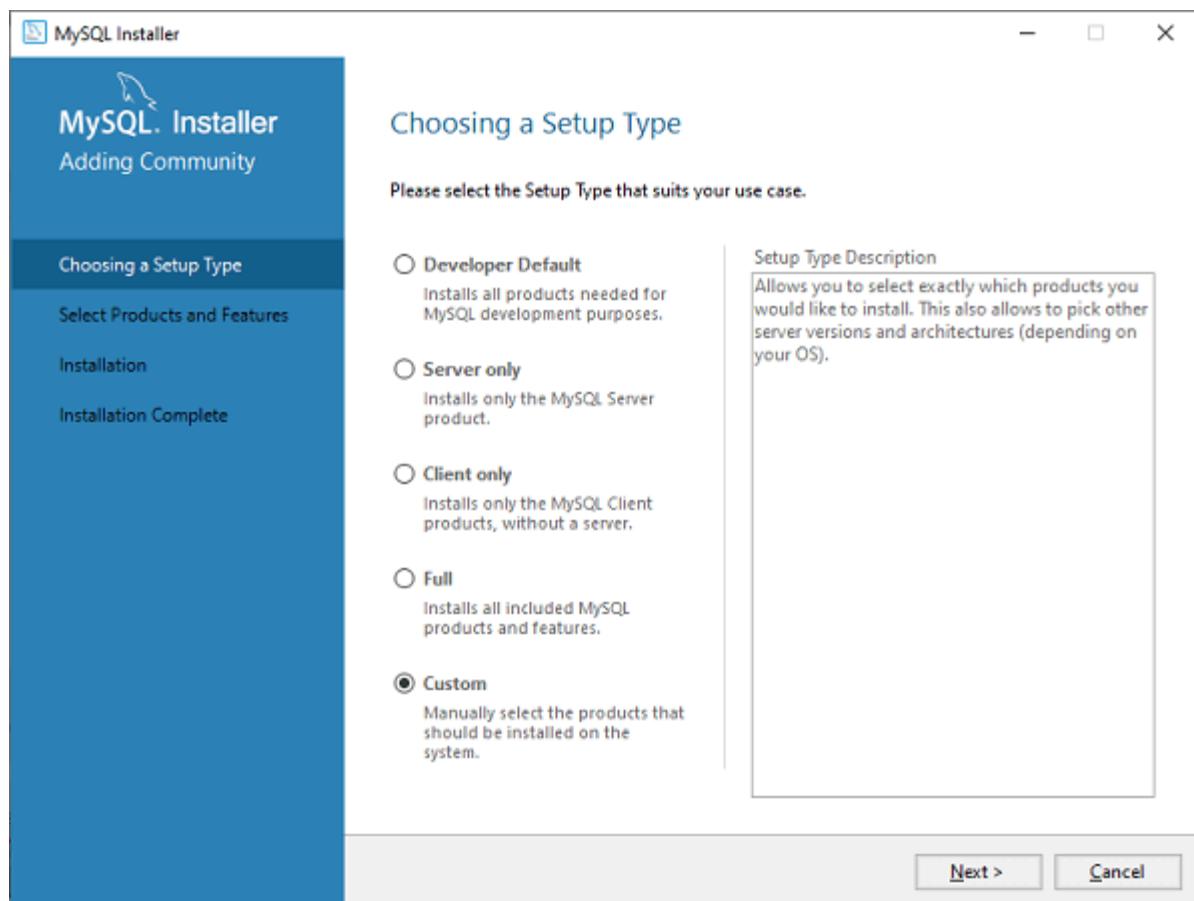
- **MySQL Server:** You can download it from [here](#).
- **MySQL Workbench:** You can download it from [here](#).
- Microsoft .NET Framework 4.5.2
- Microsoft Visual C++ Redistributable for Visual Studio 2019
- RAM 4 GB (6 GB recommended)

### Installation

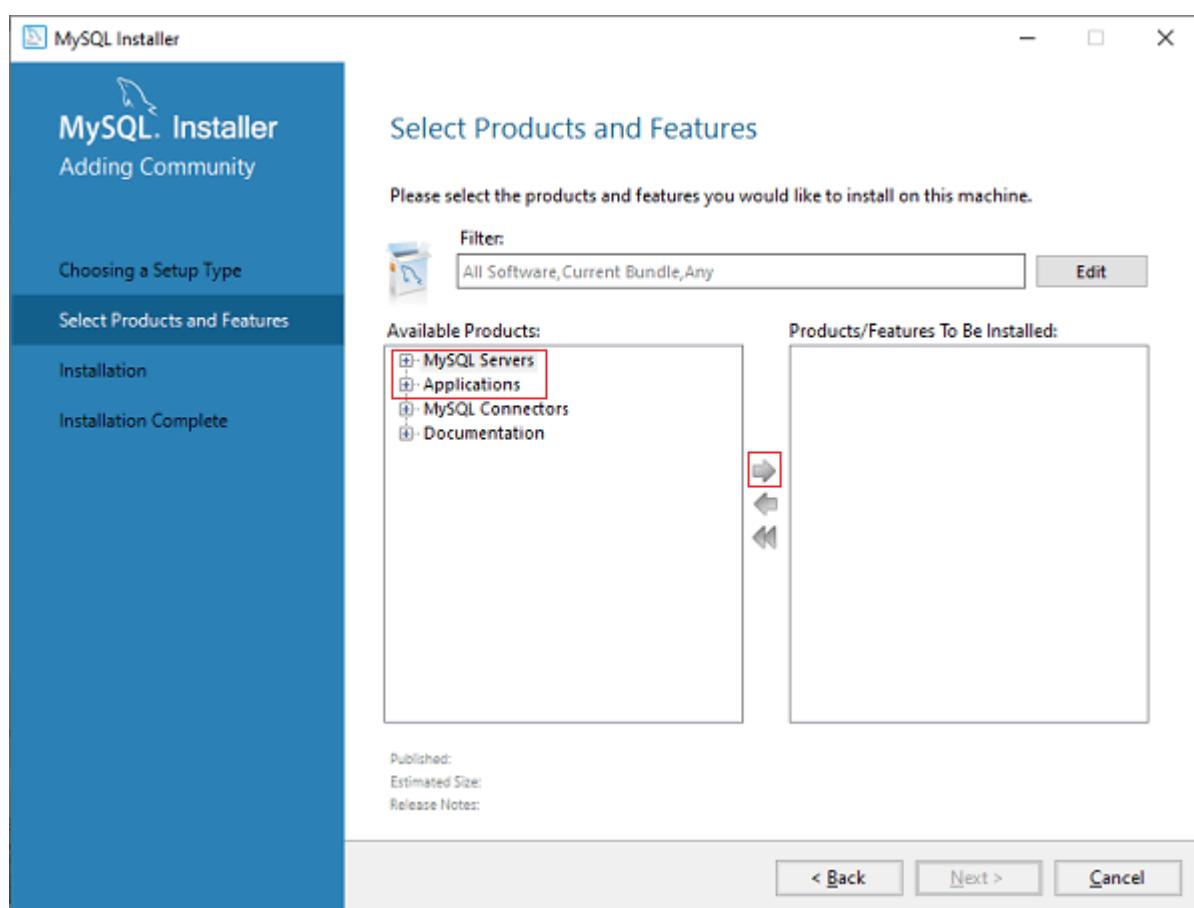
**Step 1:** Install the MySQL Community Server. To install MySQL Server, double click the MySQL **installer .exe file**. After clicking the .exe file, you can see the following screen:



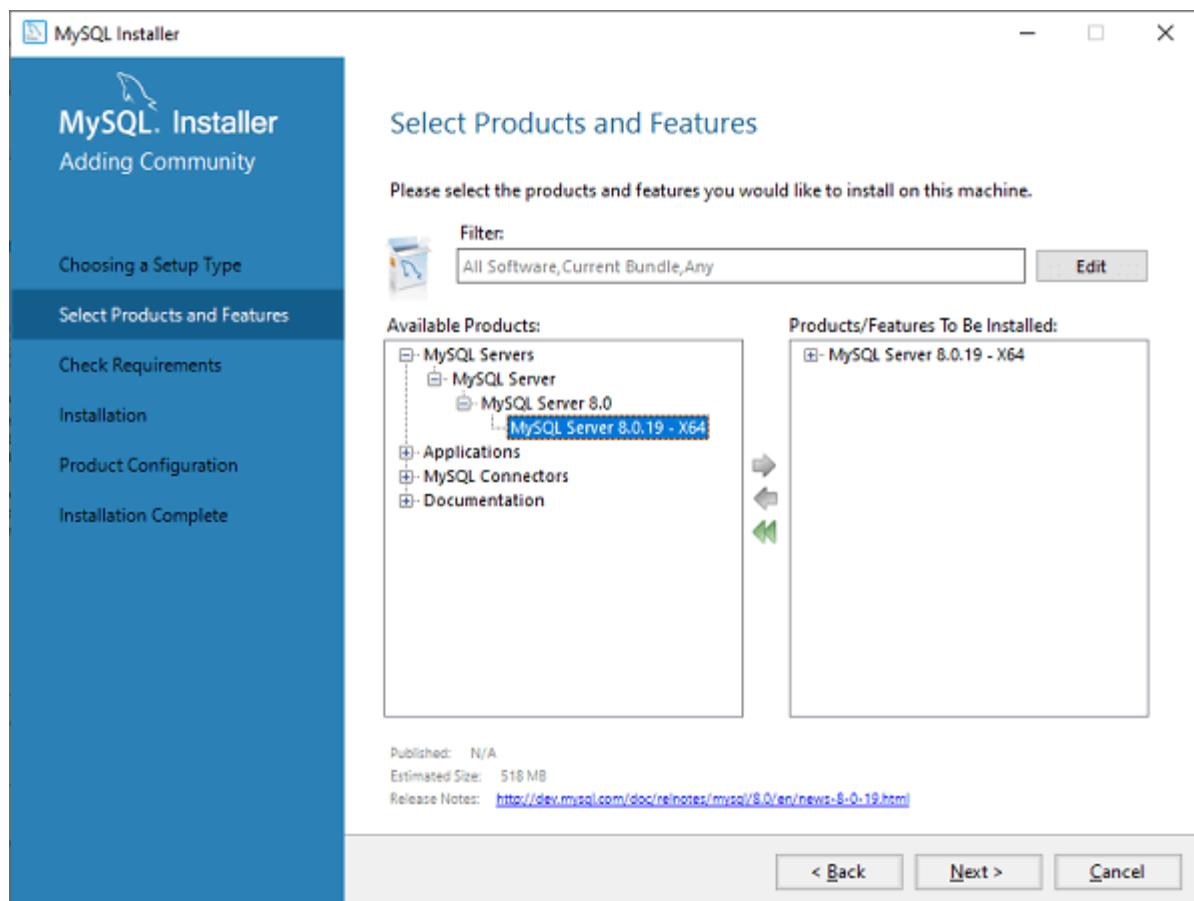
**Step 2:** Choose the **Setup Type** and click on the **Next** button. There are several types available, and you need to choose the appropriate option to install MySQL product and features. Here, we are going to select a Custom option because there is a need for only MySQL Server and Workbench. If you need more features, you can choose the Full option.



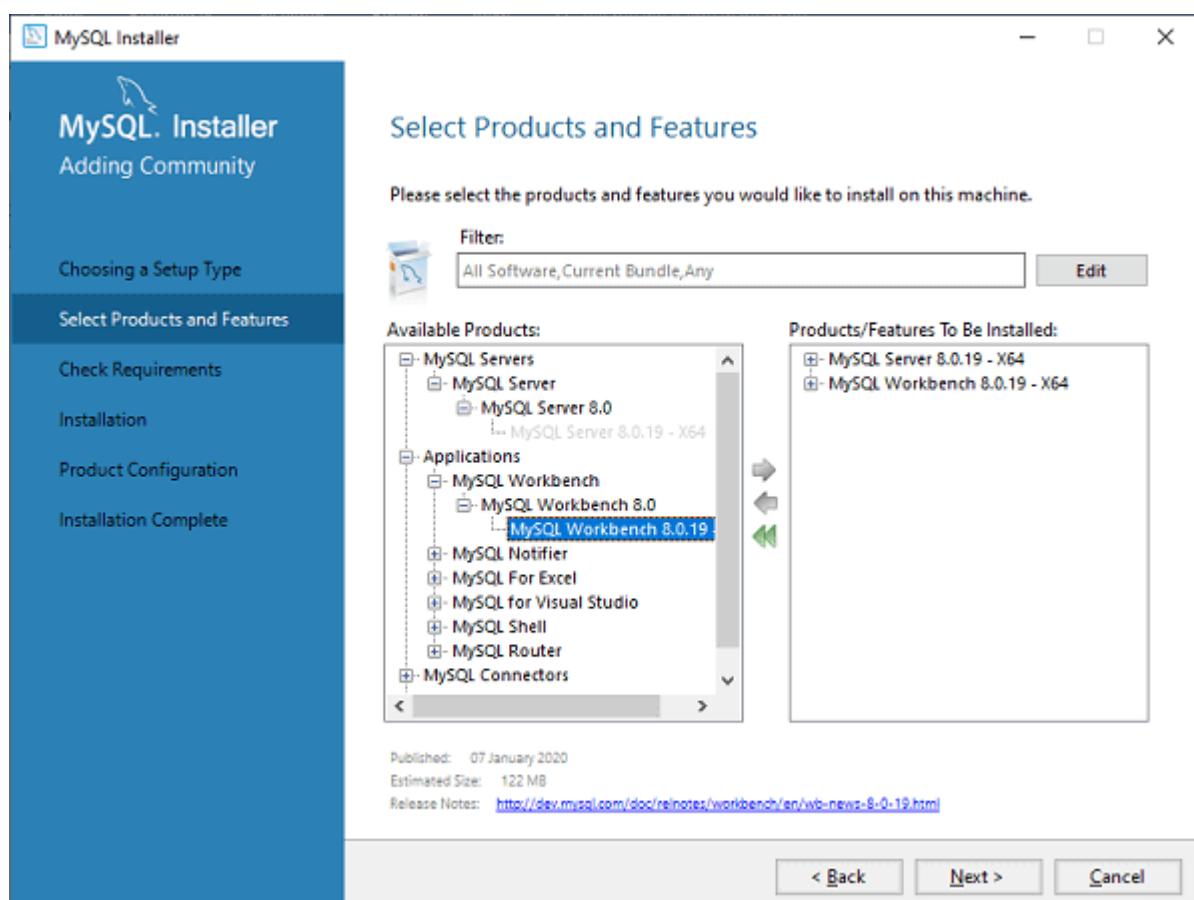
**Step 3:** When you click on the Next button, it will give the following screen.



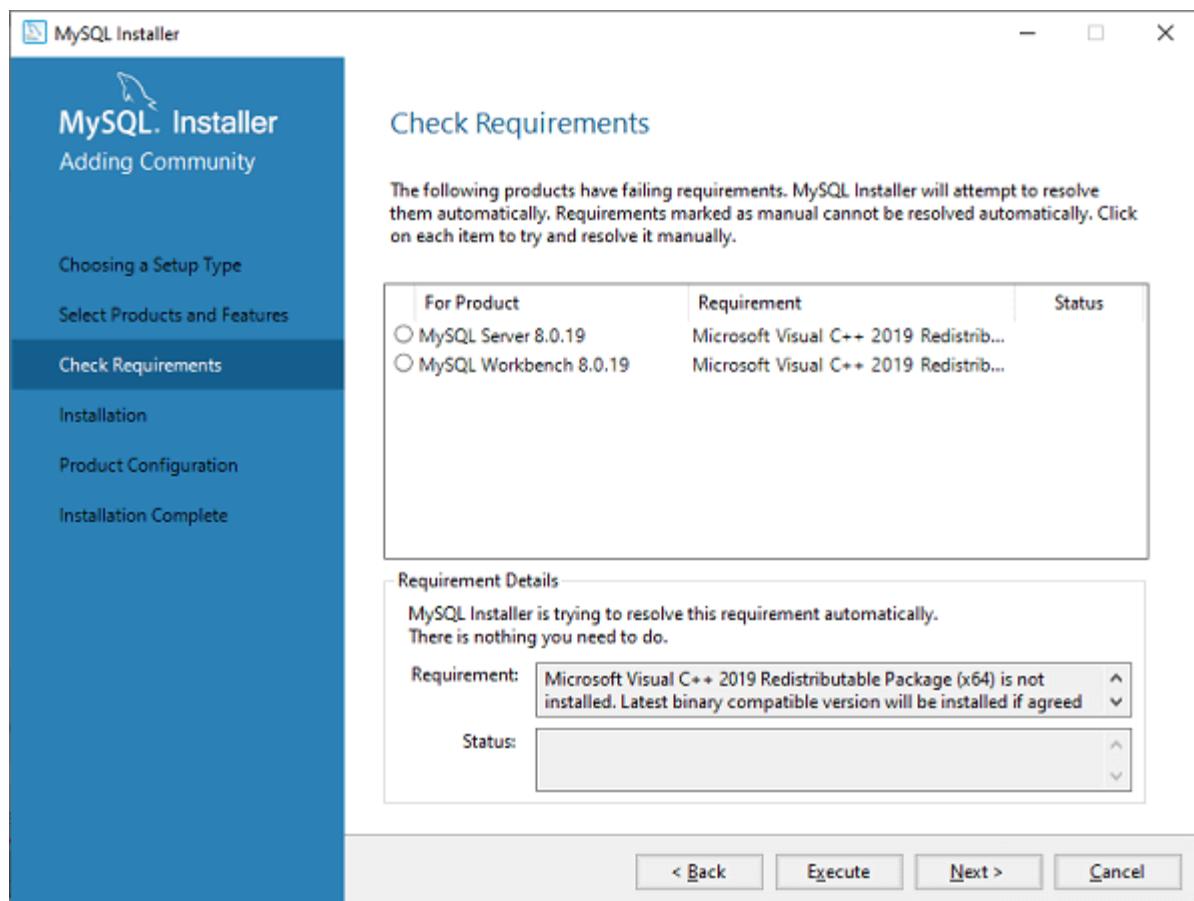
In this screen, go to the **MySQL Server** section, click the plus (+) icon. Here, you need to choose the MySQL Server and add it to the right side box by clicking on the right arrow symbol.



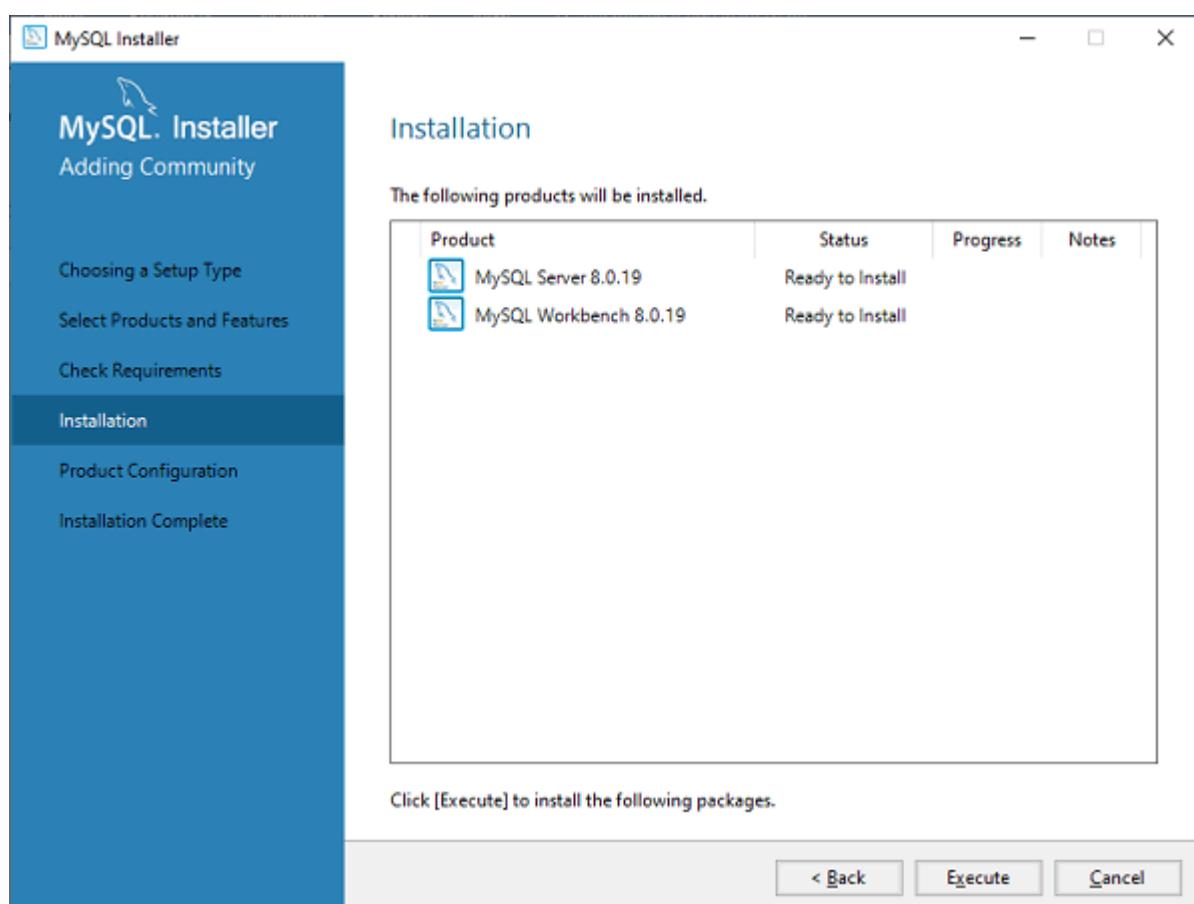
Now, in the **Application section**, you need to do the same thing that you had to perform with MySQL Server and click on the Next button. The following screen explains it more clearly.



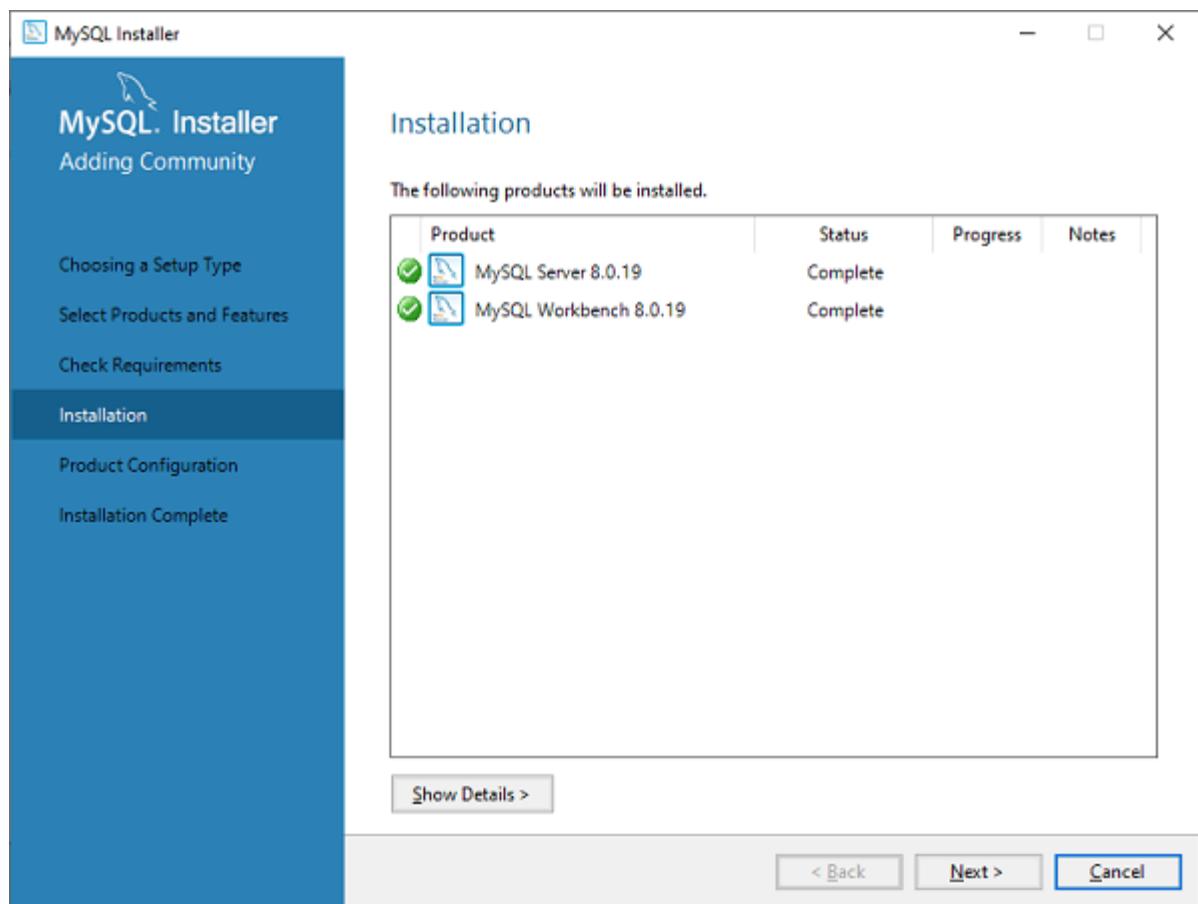
**Step 4:** When you click on Next, it will give the following screen. This screen checks all the requirements for installing MySQL Server and Workbench. As soon as you click on the **Execute** button, it will install all requirements automatically. Now, click on the Next button.



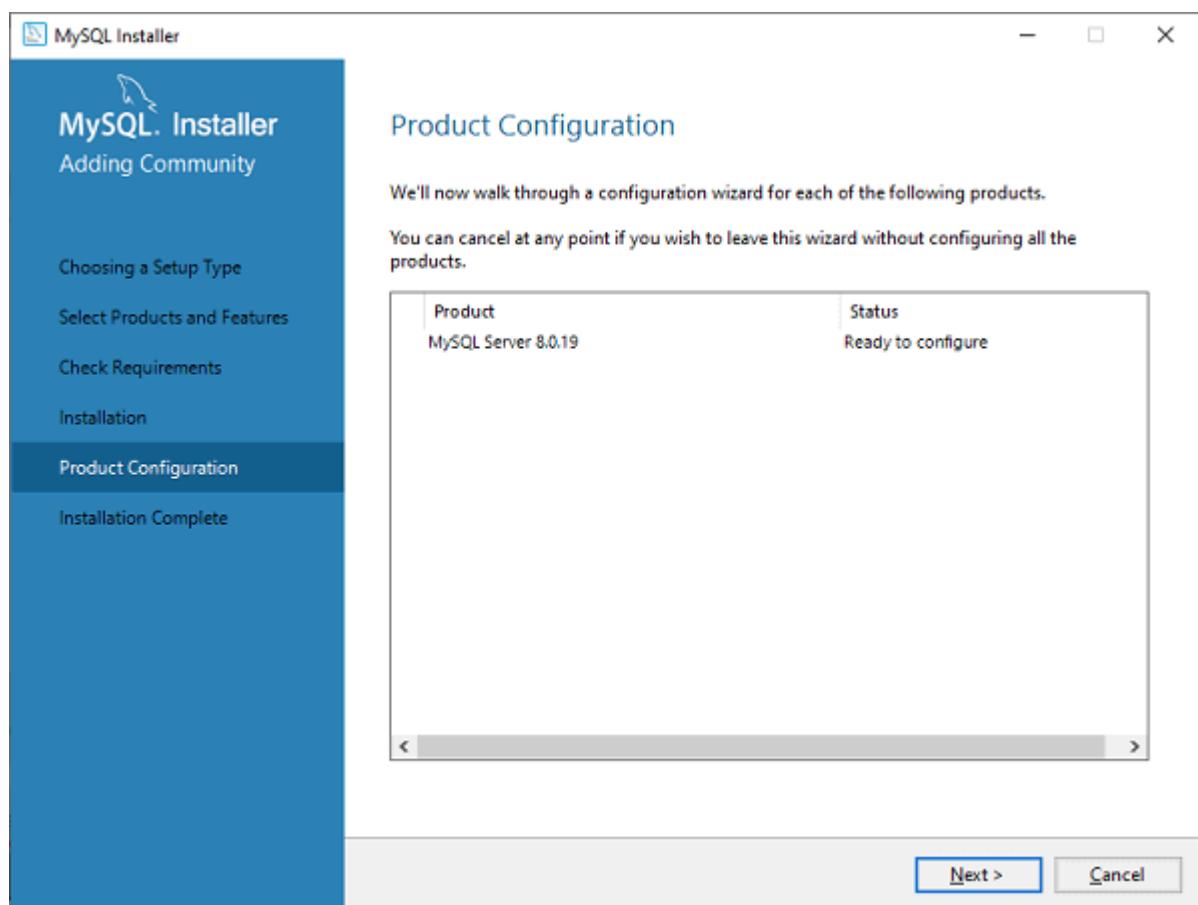
**Step 5:** In this screen, click on the Execute button to download and install the MySQL Server and Workbench.



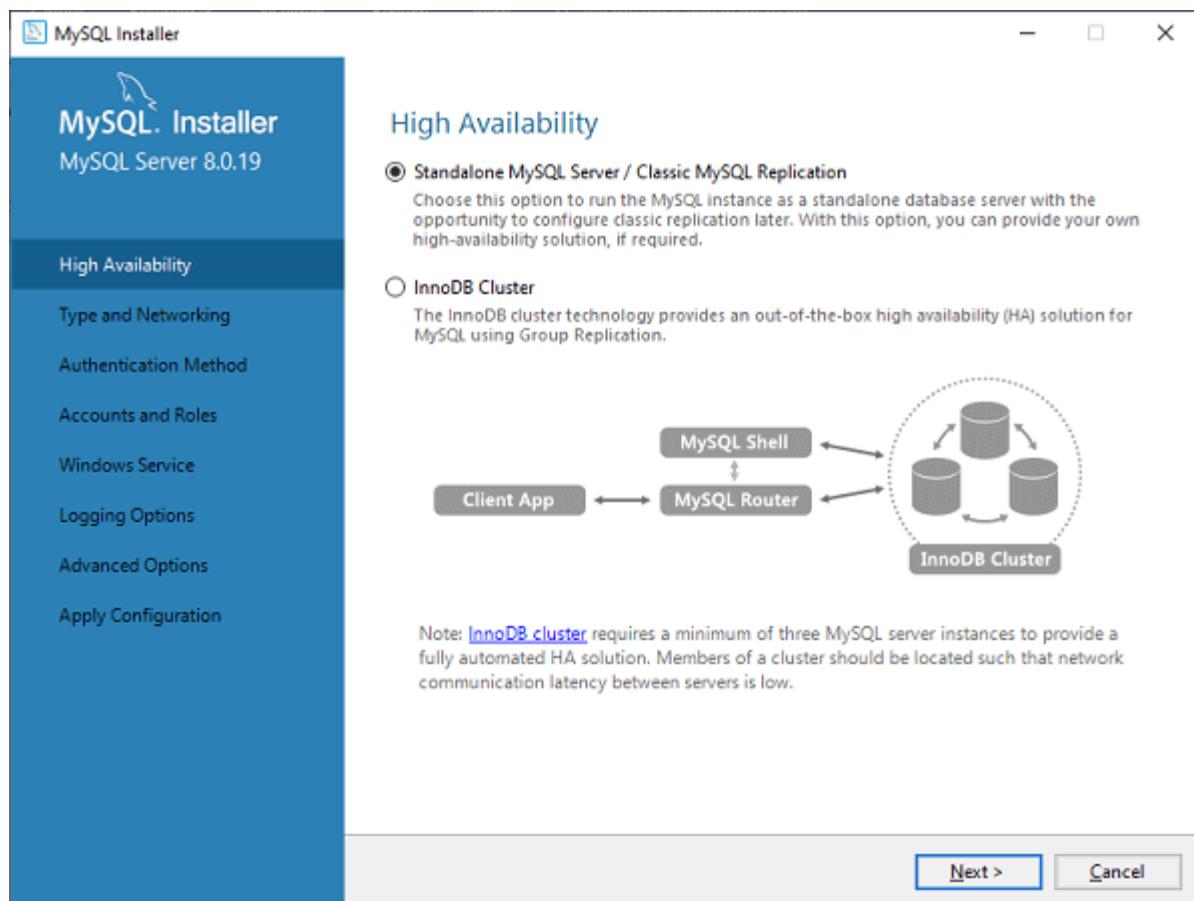
When the downloading and installation is complete, click on Next button.



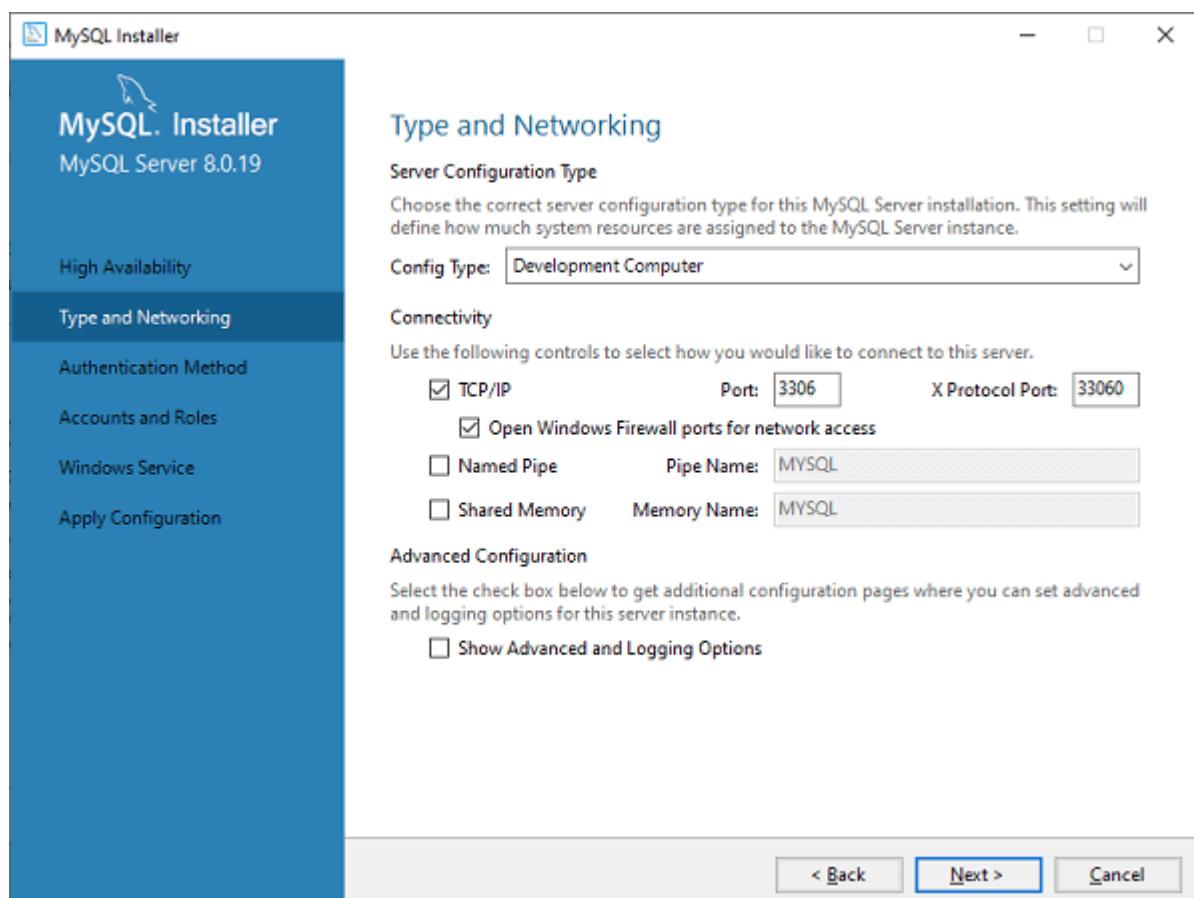
**Step 6:** In the next screen, we need to configure the MySQL Server and click on Next button.



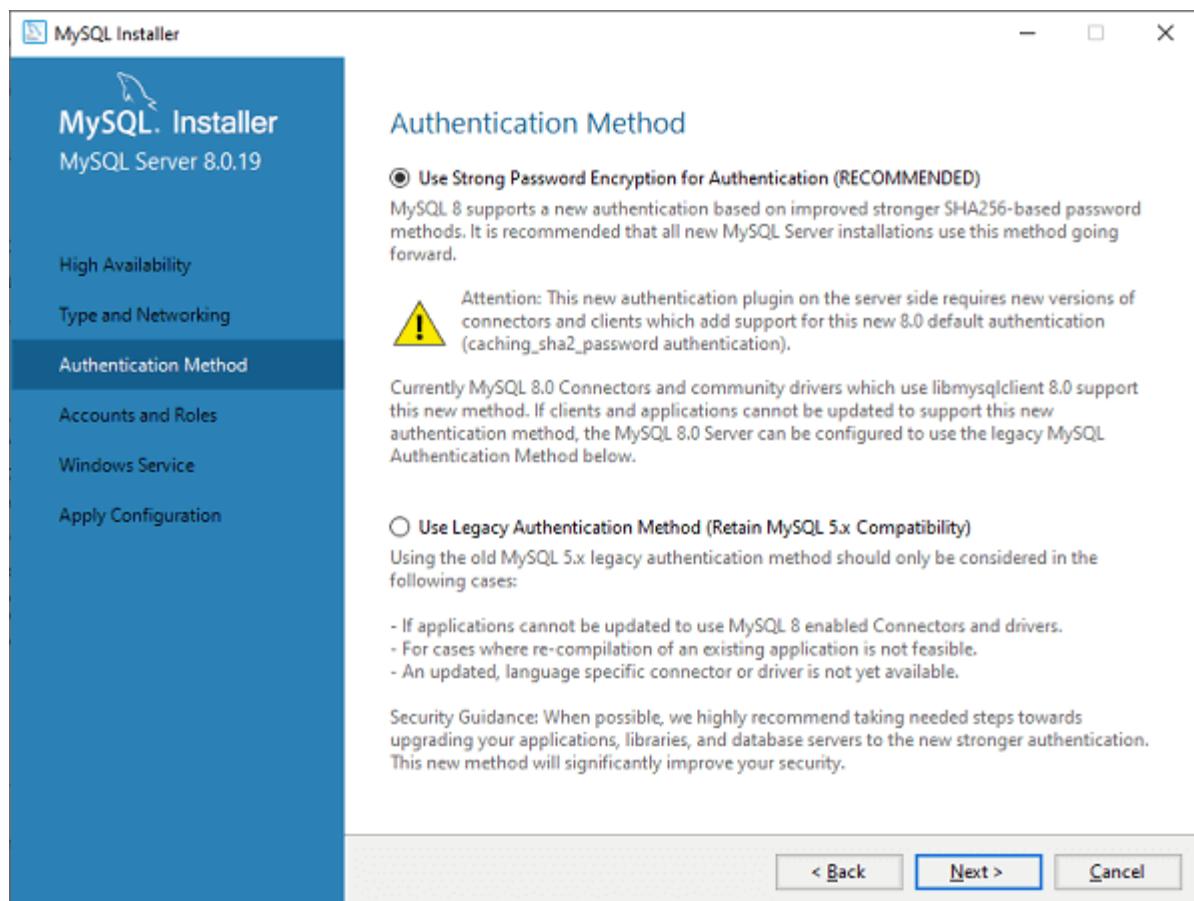
**Step 7:** As soon as you will click on the Next button, you can see the screen below. Here, we have to configure the MySQL Server. Now, choose the Standalone MySQL Server/Classic MySQL Replication option and click on Next.



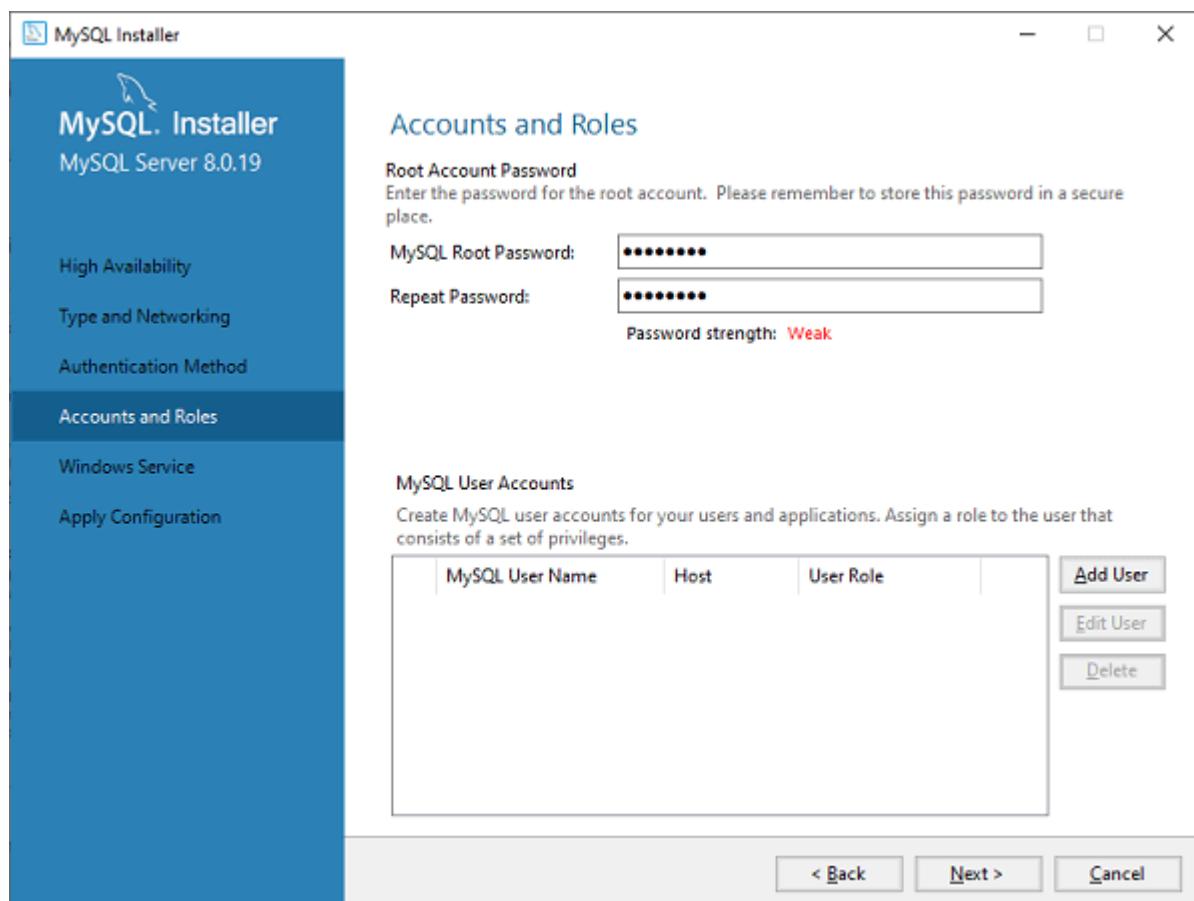
**Step 8:** In the next screen, the system will ask you to choose the Config Type and other connectivity options. Here, we are going to select the Config Type as '**Development Machine**' and Connectivity as **TCP/IP**, and **Port Number** is 3306, then click on Next.



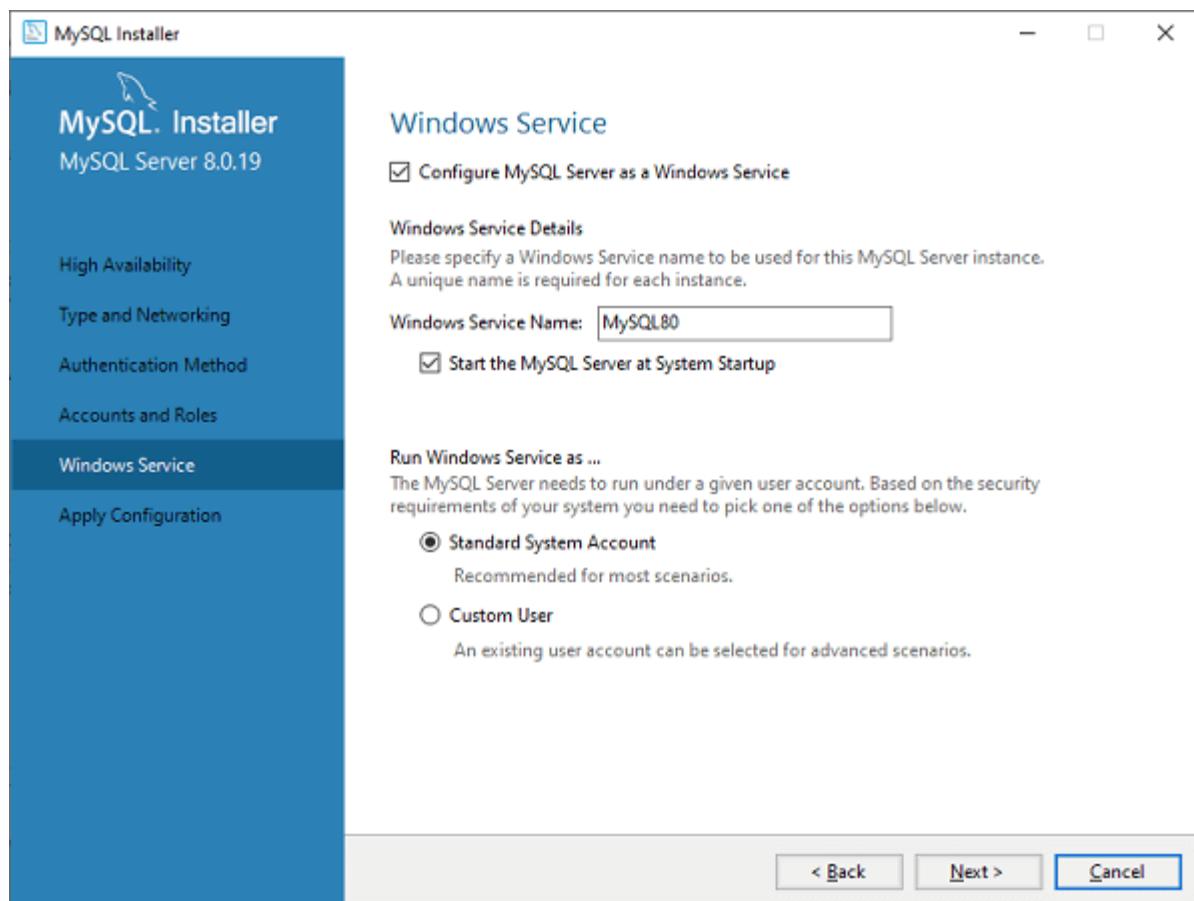
**Step 9:** Now, select the Authentication Method and click on Next.



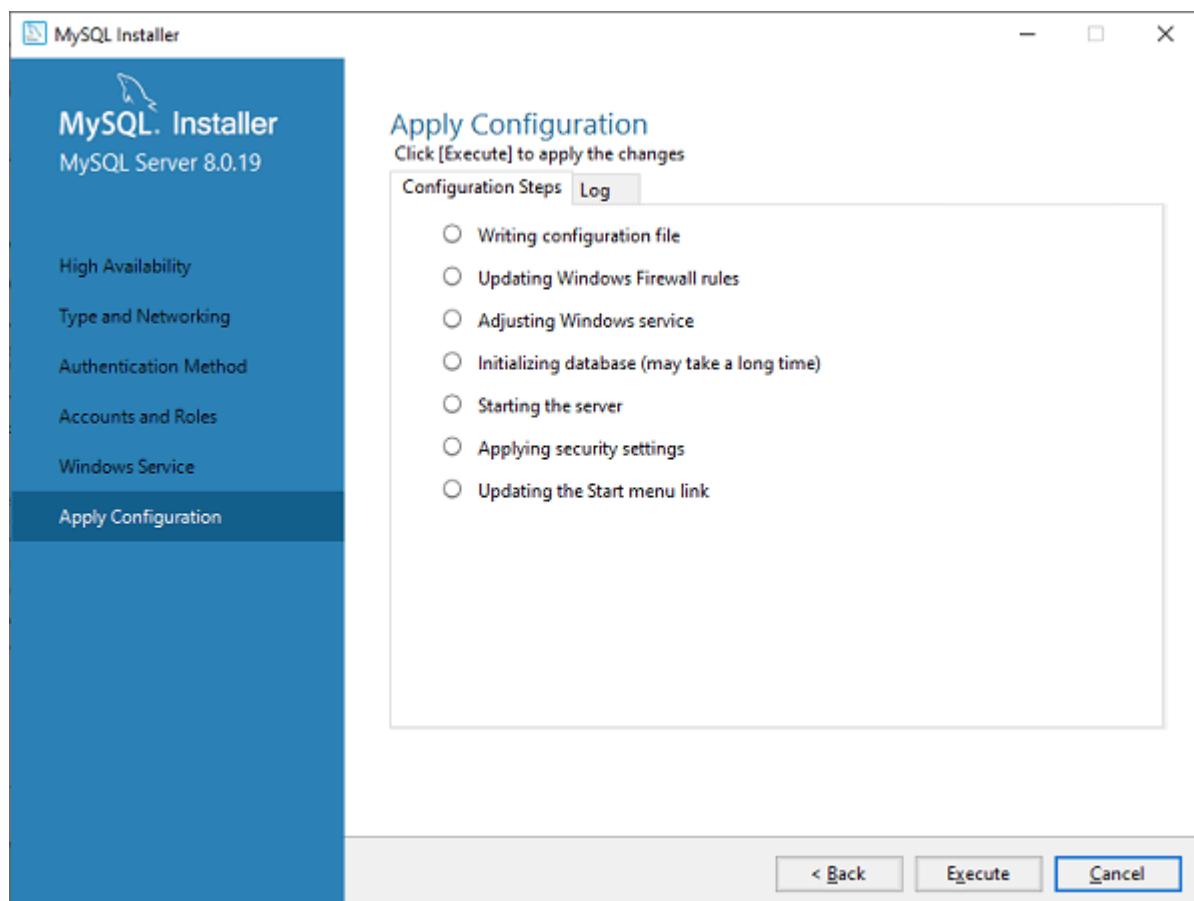
**Step 10:** The next screen will ask you to choose the account, username, and password. After filling all the details, click on the Next button.



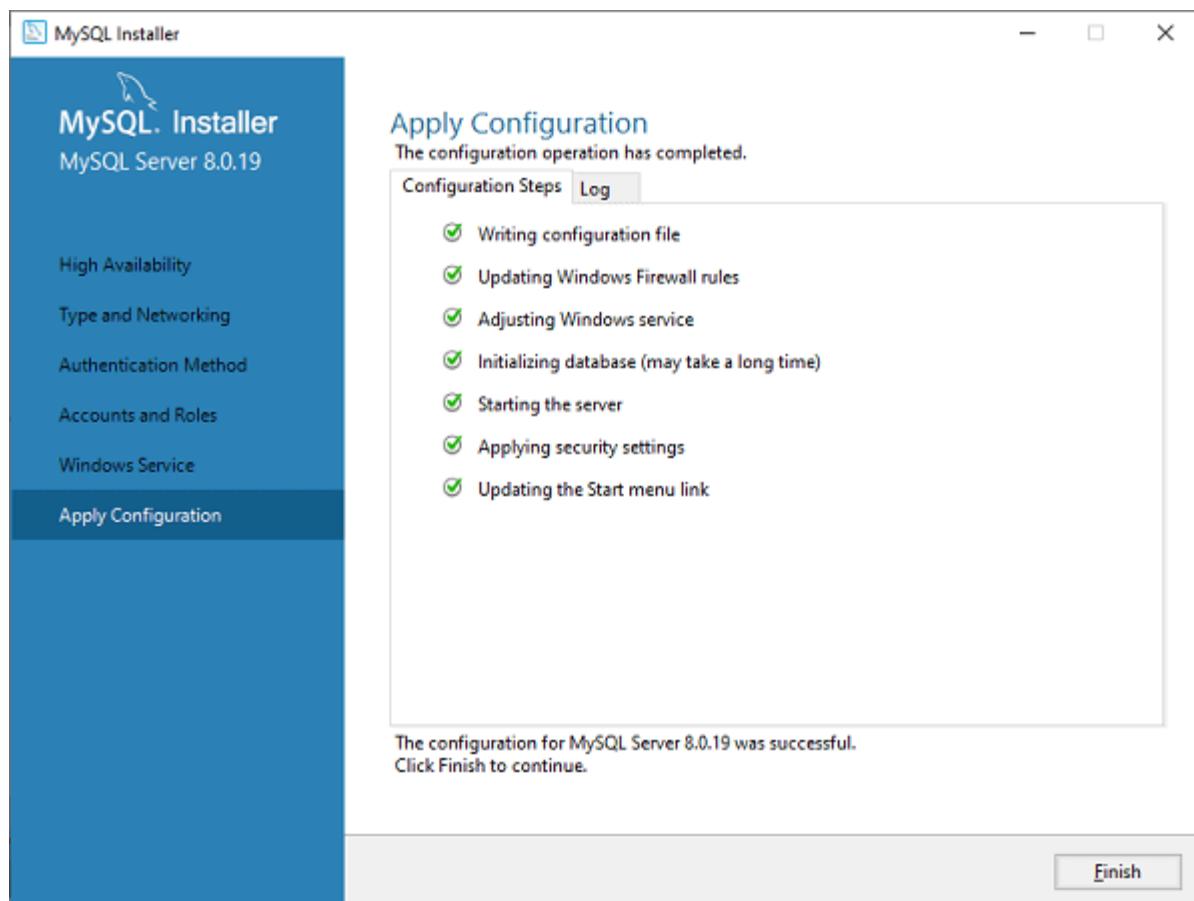
**Step 11:** The next screen will ask you to configure the Windows Service. Keep the default setup and click on Next.



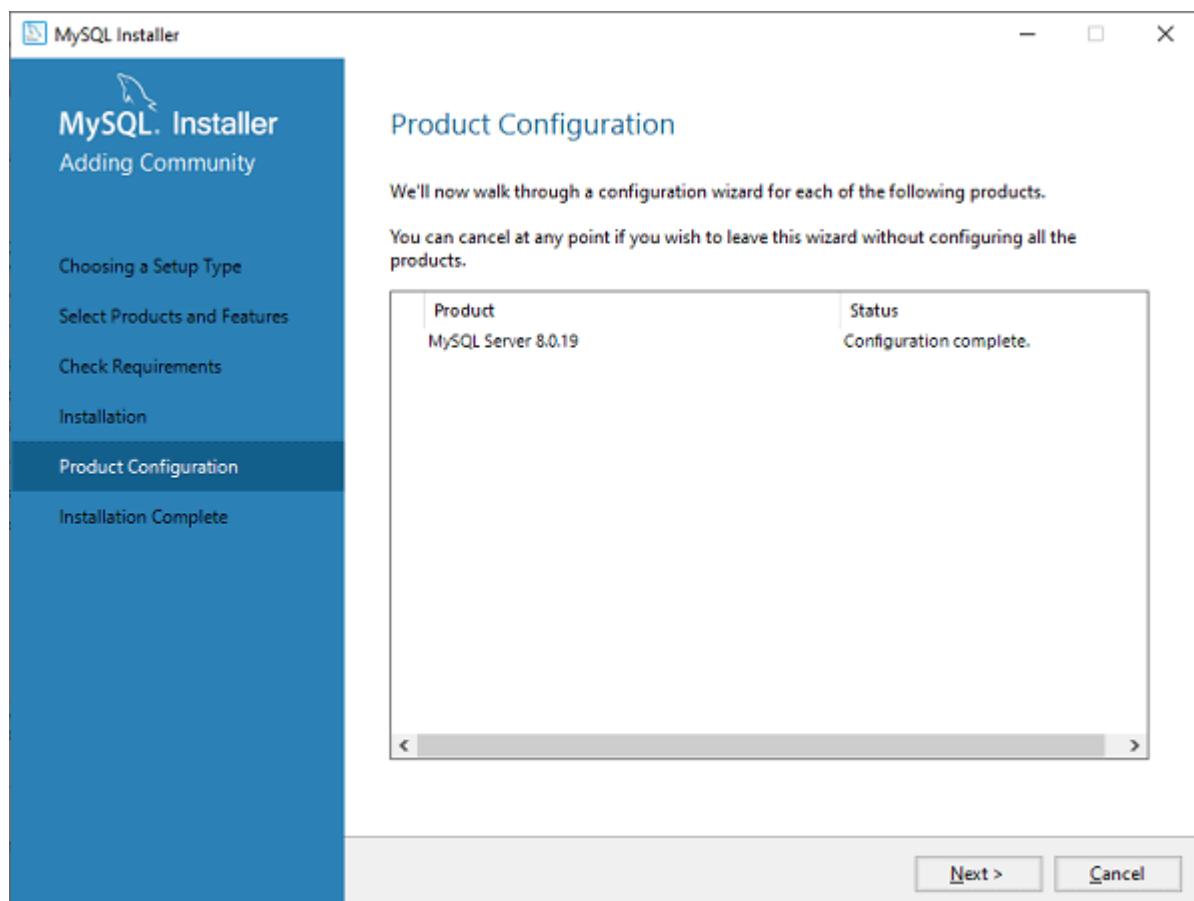
**Step 12:** In the next screen, the system will ask you to apply the Server Configuration. For this configuration, click on the Execute button.

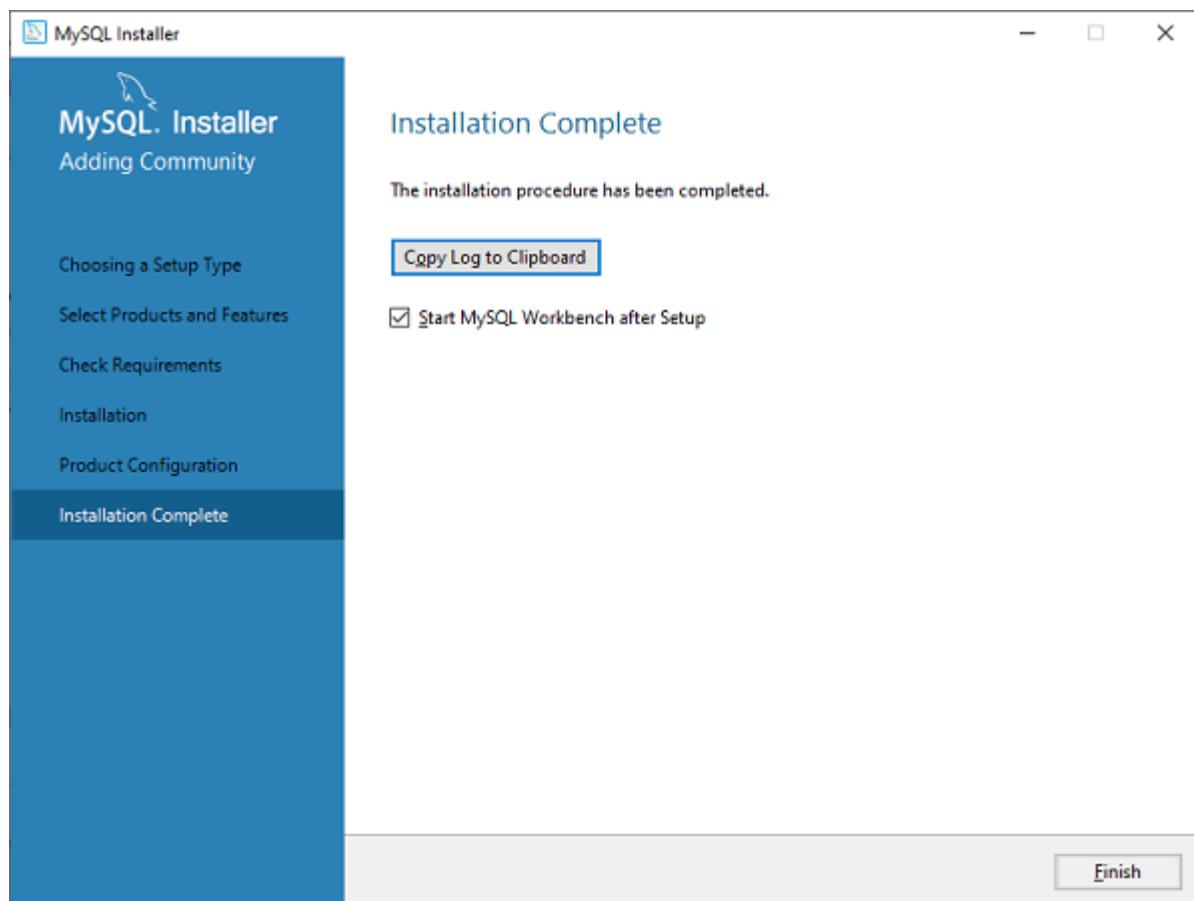


**Step 13:** Once the configuration has completed, you will get the screen below. Now, click on the **Finish** button to continue.

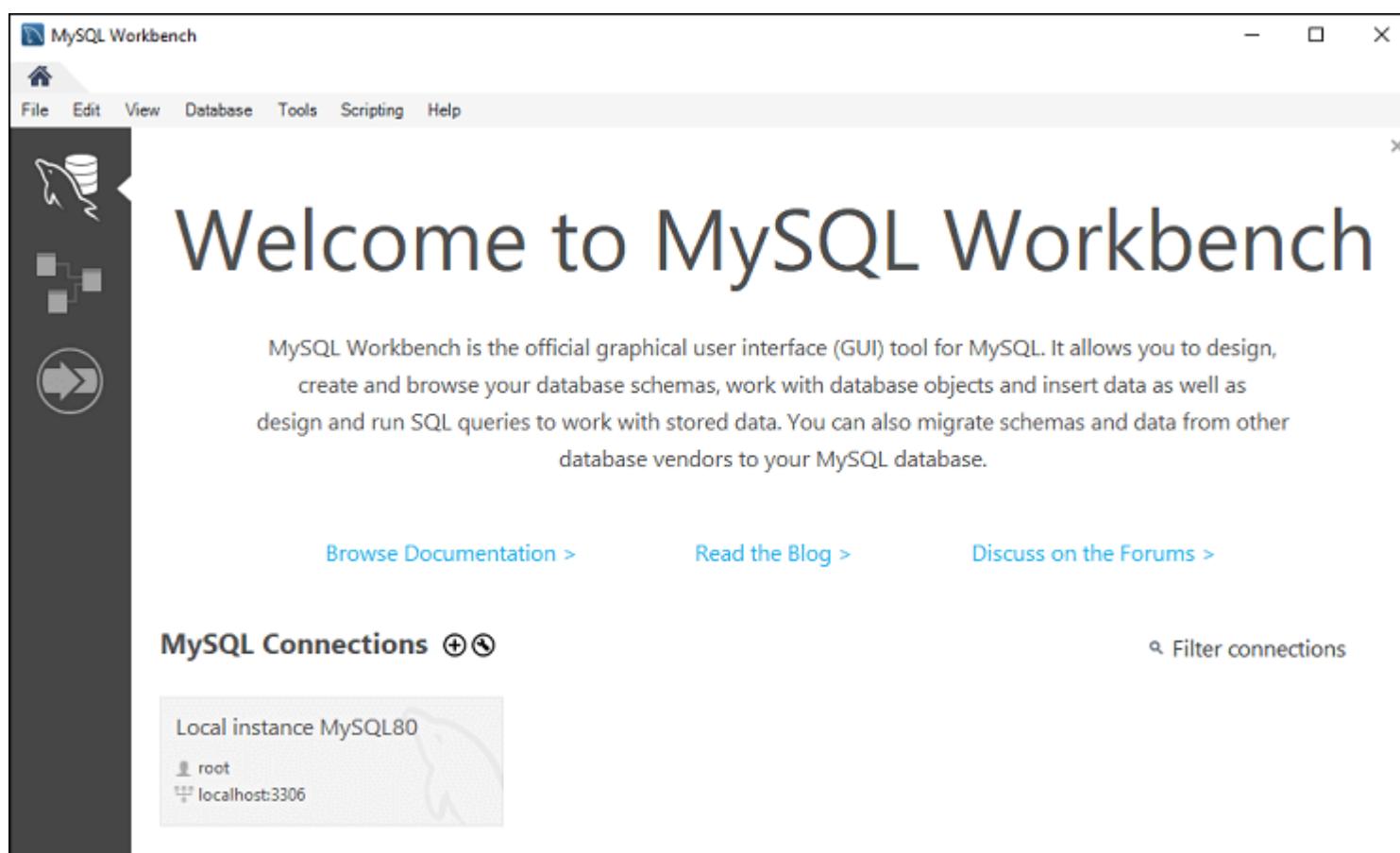


**Step 14:** In the next screen, you can see that the Product Configuration is completed. Keep the default setting and click on the Next-> Finish button to complete the MySQL package installation.

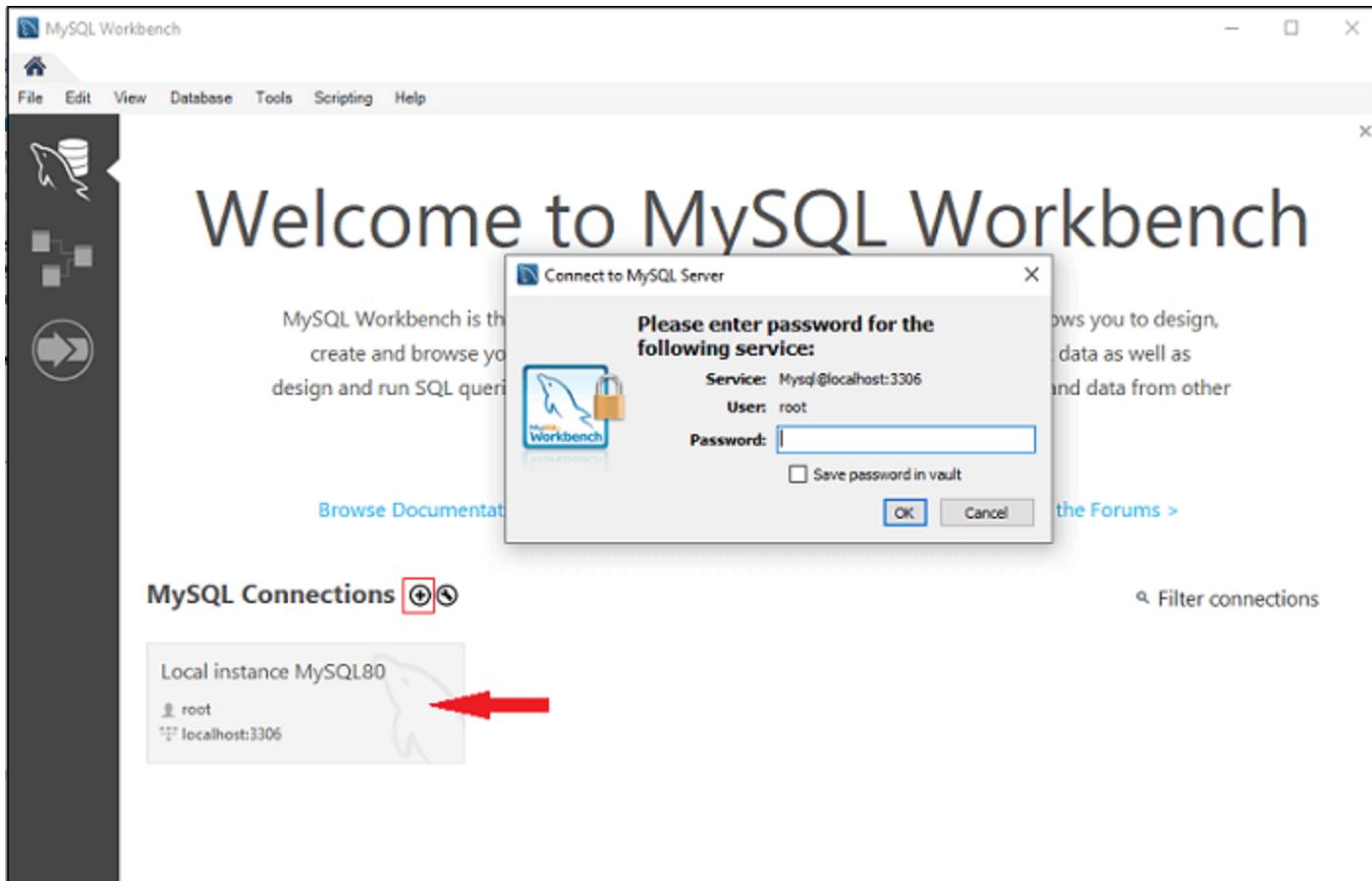




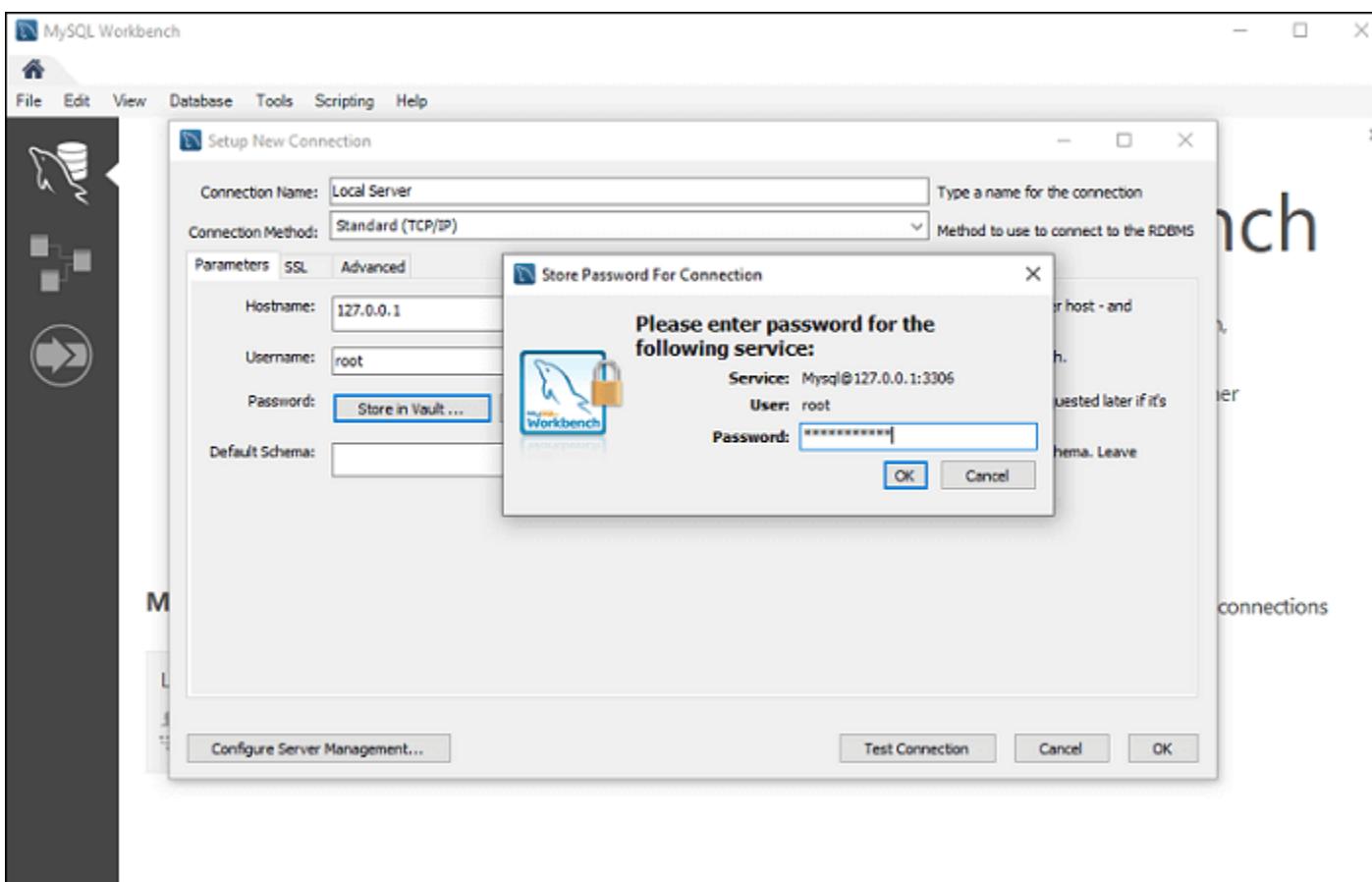
**Step 15:** Once you click the Finish button, the MySQL Workbench should be open on your system, as shown in the screen below.



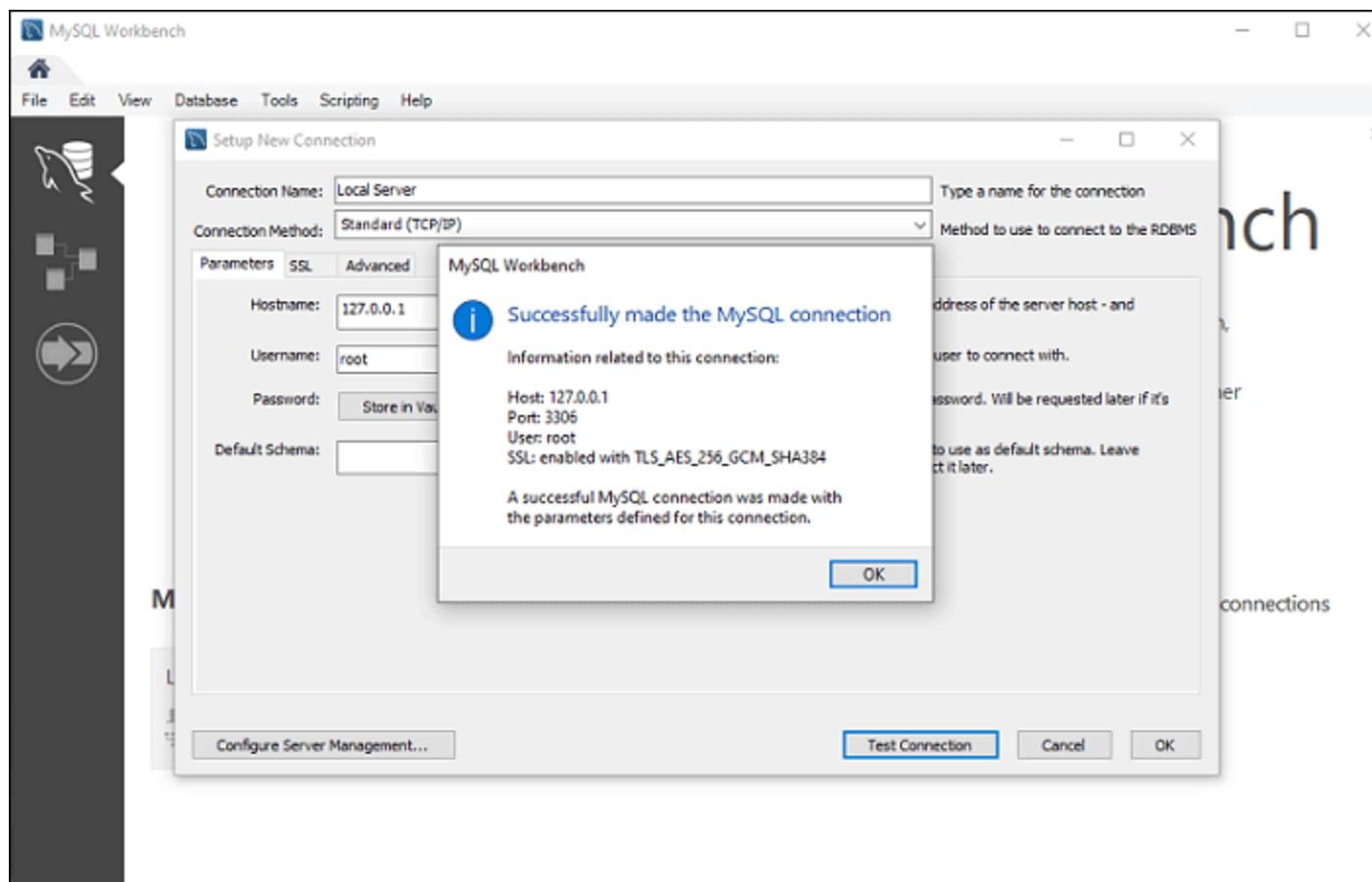
**Step 16:** In the above screen, you need to make a connection. To do this, double click the box designated by the **red arrow**. Here, you will get the popup screen that asks to enter the password created earlier during the installation. After entering the password, you are able to connect with the Server.



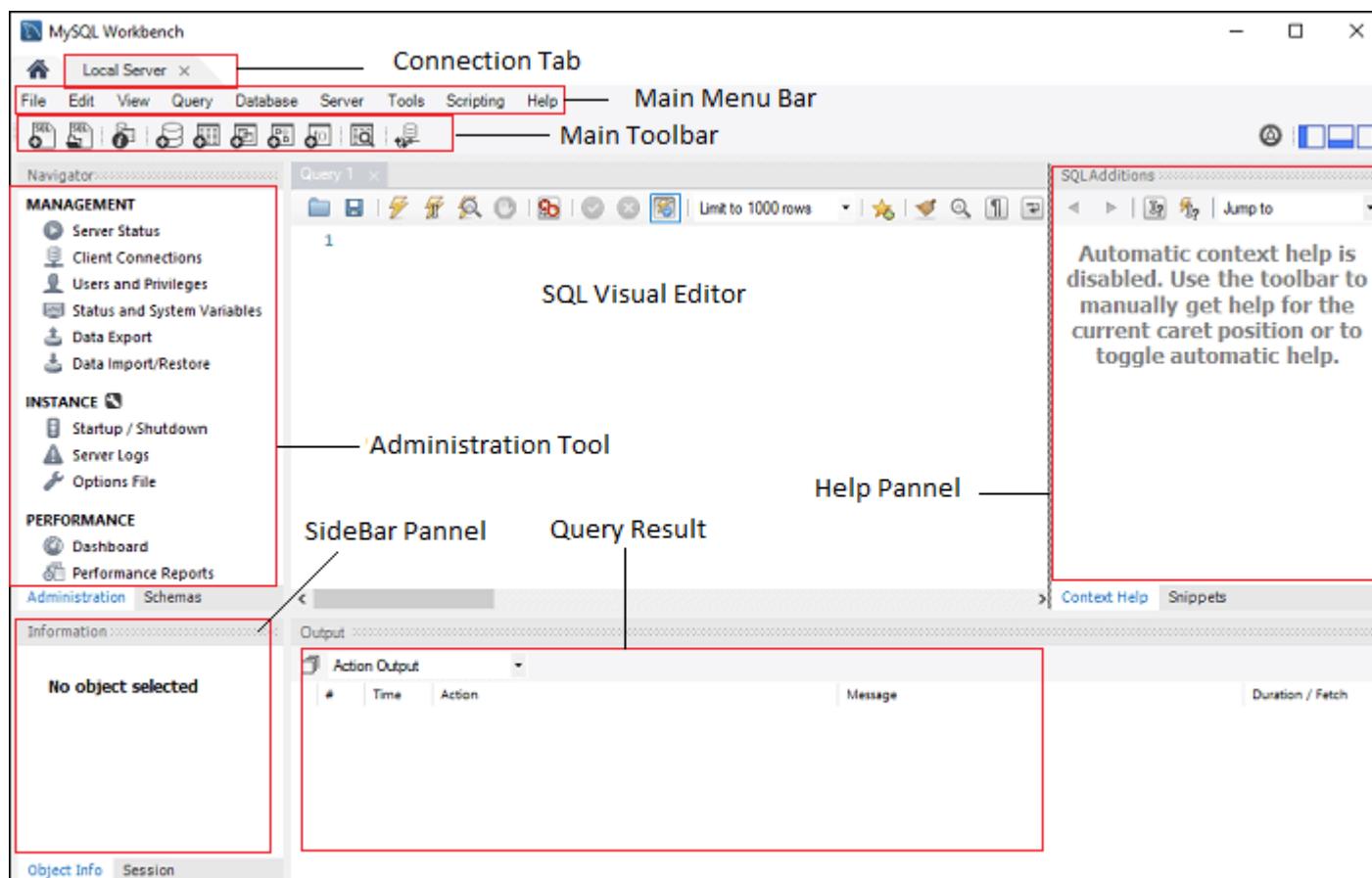
**Step 17:** If you do not have a connection, you can create a new connection. To make a connection, click the plus (+) icon or go to the menu bar -> Database -> Connect to Database, the following screen appears. Now, you need to fill all the details. Here, you have to make sure that the entered password should be the same as you have created earlier.



**Step 18:** After entering all the details, click on the **Test Connection** to test the database connectivity. If the connection is successful, you will get the following screen. Now, click on OK->OK button to finish the setup.



**Step 19:** Once you have finished all the setup, it will open the MySQL Workbench screen. Now, double click on the newly created connection, you will get the following screen where the SQL command can be executed.



## MySQL Workbench Administration Tool

The Administration Tool plays an important role in securing the data of the company. Here, we are going to discuss the user's management, Server configuration, Database backup and restorations, Server logs, and many more.

### User Administration

It is a visual utility that allows for managing the user that relate to an active MySQL Server instance. Here, you can add and manage user accounts, grant and drop privileges, view user-profiles, and expire passwords.

### Server Configuration

It allows for advanced configuration of the Server. It provides detailed information about the Server and status variable, a number of threads, buffer allocation size, fine-tuning for optimal performance, and many more.

### Database backup and restorations

It is a visual tool, which is used for importing/exporting MySQL dump files. The dump files contain SQL scripts for creating databases, tables, views, and stored procedures.

### Server Logs

It displays log information for the MySQL Server by each connection tab. For each connection tab, it includes an additional tab for the general error logs.

## Performance Dashboard

This tab provides the statistical view of the Server performance. You can open it by navigating to the Navigation tab, and under the Performance section, choose Dashboard.

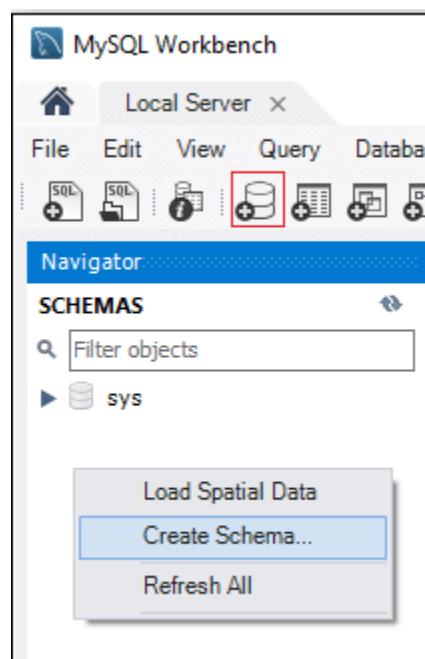
## MySQL Workbench Create, Alter, Drop Database

In this section, we are going to see how a database is created, altered, and drop by using the MySQL Workbench. Let us see in detail one by one.

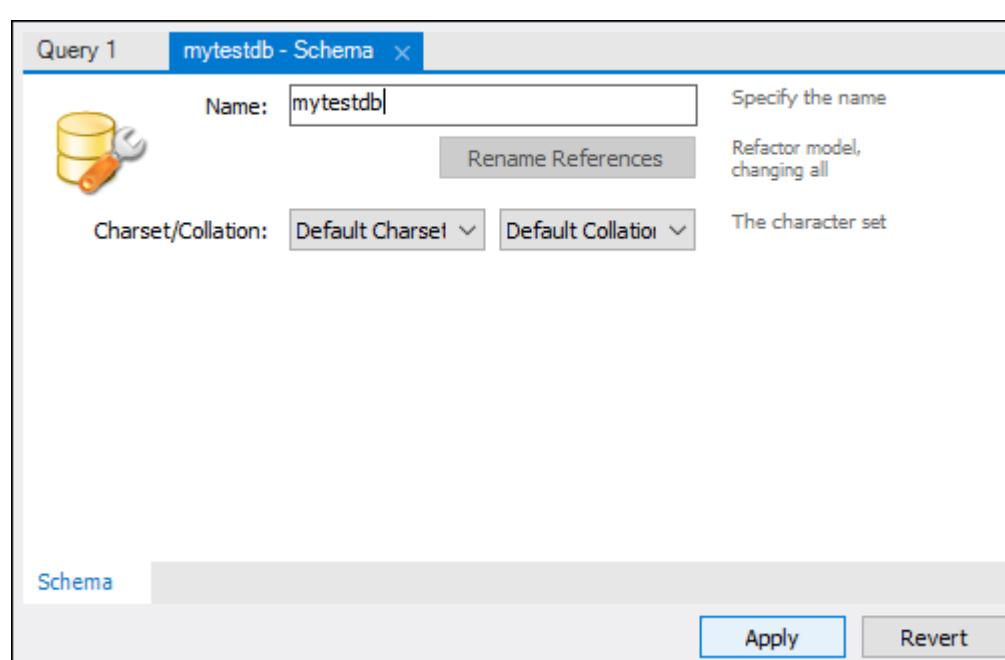
### Create Database

To create a database, do the following steps:

1. Open the MySQL Workbench and logged in using username and password. Then, go to the Navigation tab and click on the **Schema menu**. Here, you can see all the previously created databases.
2. If you want to create a new database, right-click under the Schema menu and select **Create Schema** or click the database icon (red rectangle), as shown in the following screen.



3. The new Schema window screen open. Enter the new database name (for example, mytestdb) and use default **Collation**. Collation is used to store specific data characters, mainly useful for storing foreign languages. Now, click on the Apply button as shown in the screen below:

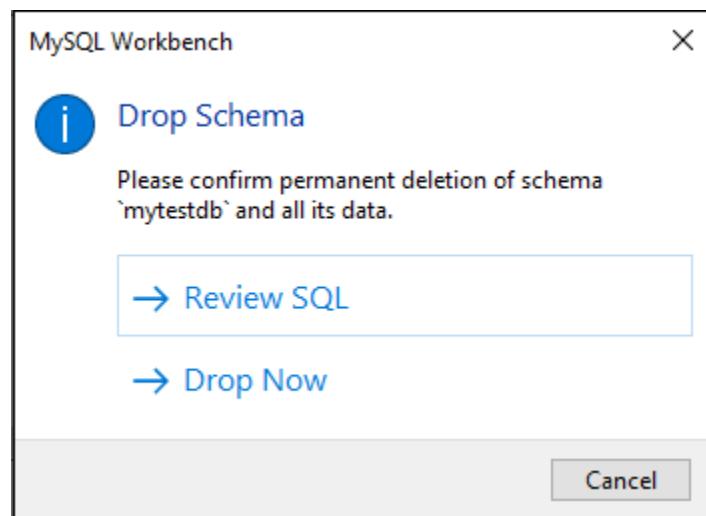


4. A new popup window appears, click Apply->Finish button to create a new database.
5. After the successful creation of the database, you can see this new database in the Schema menu. If you do not see this, click on the refresh icon into the Schema menu.
6. If you want to see more information about the database, select mytestdb database, and click on the 'i' icon. The information window displays several options, like Table, Column, Functions, Users, and many more.

7. MySQL Workbench does not provide an option to rename the database name, but we can create, update, and delete the table and data rows from the database.

### Drop Database

1. To delete a database, you need to choose the database, right-click on it, and select the **Drop Schema** option. The following screen appears:



2. Select **Drop Now** option in the popup window and the database including table, data rows will be deleted from the database Server.

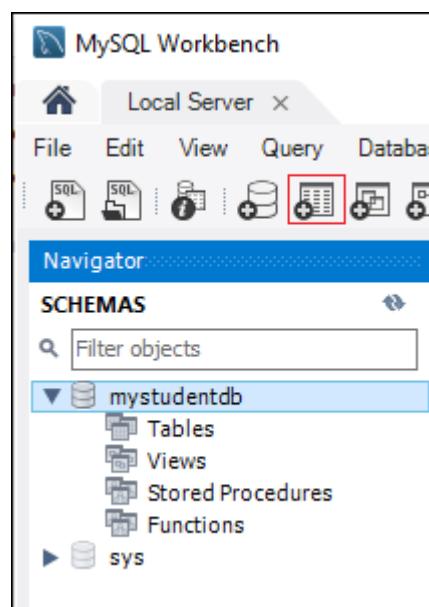
## MySQL Workbench Create, Alter, Drop Table

In this section, we are going to see how a table is created, altered, and drop by using the MySQL Workbench. Let us see in detail one by one.

### Create Table

To create a table, do the following steps:

1. Open the MySQL Workbench and logged in using username and password. Then, go to the Navigation tab and click on the Schema menu. Here, you can see all the previously created databases. You can also create a new database.
2. Select the newly created database, double click on it, and you will get the sub-menu under the database. The sub-menu under the database are Tables, Views, Functions, and Stored Procedures, as shown in the below screen.



3. Select Tables sub-menu, right-click on it and select **Create Table** option. You can also click on create a new table icon (shown in red rectangle) to create a table.
4. On the new table screen, you need to fill all the details to create a table. Here, we are going to enter the table name (for example, student) and use default collation and engine.
5. Click inside the middle window and fill the column details. Here, the column name contains many attributes such as Primary Key(PK), Not Null (NN), Unique Index (UI), Binary(B), Unsigned Data type(UN), Auto Incremental (AI), etc. The following screen explains it more clearly. After filling all the details, click on the **Apply** button.

Table Name: student Schema: mystudentdb

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
studentid	INT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
firstname	VARCHAR(30)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
lastname	VARCHAR(30)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
class	VARCHAR(10)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
age	INT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

Column Name: age Data Type: INT Default:

Charset/Collation: Default Charset Default Collation

Comments:

Storage:  Virtual  Stored  
 Primary Key  Not Null  Unique  
 Binary  Unsigned  Zero Fill  
 Auto Increment  Generated

Columns Indexes Foreign Keys Triggers Partitioning Options Apply Revert

6. As soon as you click on the Apply button, it will open the SQL statement window. Again, click on the Apply button to execute the statement and Finish button to save the changes.

7. Now, go to the Schema menu and select the database which contains the newly created table, as shown in the screen below.

Navigator

SCHEMAS

mystudentdb

Tables

student

Columns

studentid, firstname, lastname, class, age

Indexes, Foreign Keys, Triggers

Views, Stored Procedures, Functions

sys

Administration Schemas

Information

Table: student

Columns:

- studentid int UN AI PK
- firstname varchar(30)
- lastname varchar(30)
- class varchar(10)
- age int UN

## Alter Table

To alter a table, do the following steps:

1. Select the table you want to modify, click on the 'i' icon, and you will get the following screen.

Navigator

SCHEMAS

mystudentdb

Tables

student

i

Table Details

Engine: InnoDB

Row format: Dynamic

Column count: 5

Table rows: 0

AVG row length: 0

Data length: 16.0 KiB

Index length: 16.0 KiB

Max data length: 0.0 bytes

Data free: 0.0 bytes

Table size (estimate): 32.0 KiB

File format:

Data path: C:\ProgramData\MySQL\MySQL Server 8.0\Data\mystudentdb\student.ibd

Table: student

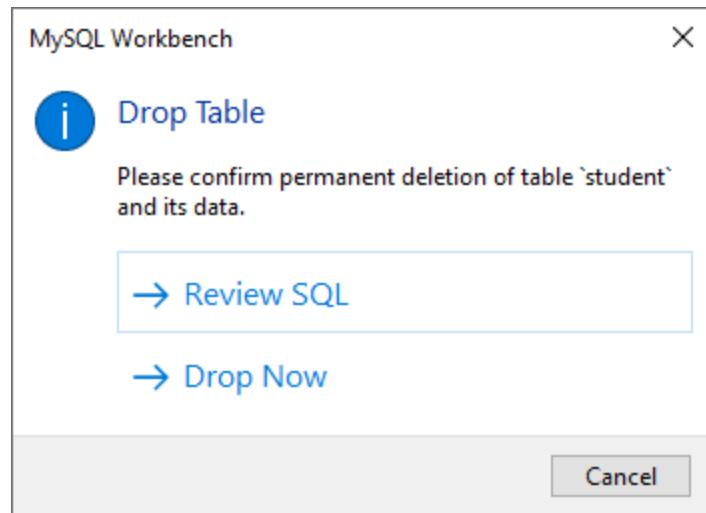
Columns:

Information on this page may be outdated. Click Analyze Table to update it.

2. In the above screen, you can modify the column name, data type, and other table settings.

### Drop a Table

1. To delete a table, you need to choose the table, right-click on it, and select the Drop Table option. The following screen appears:



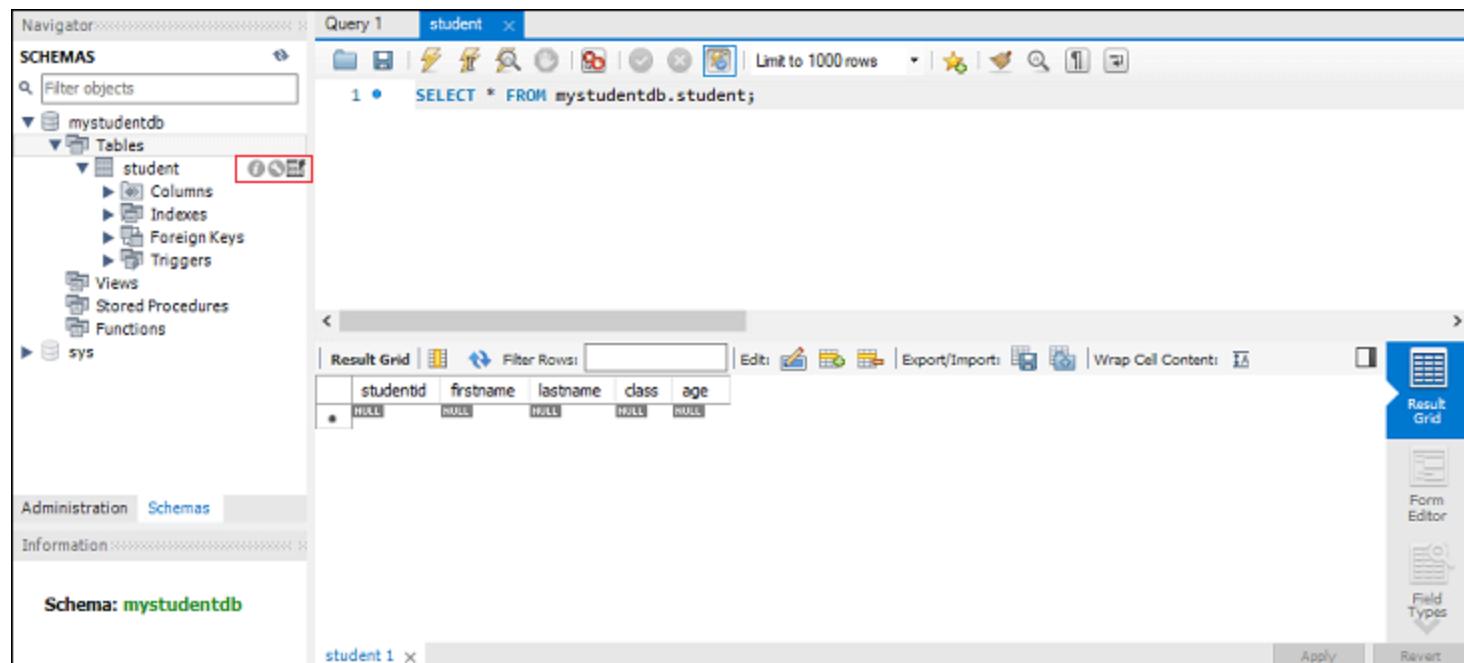
2. Select **Drop Now** option in the popup window to delete the table from the database instantly.

### MySQL Workbench Insert, Read, Update, Delete Data Rows

In this section, we are going to see how we can insert, read, update, and delete data rows by using the MySQL Workbench. Let us see in detail one by one.

1. Open the MySQL Workbench and logged in using username and password. Then, go to the Navigation tab and click on the Schema menu. Here, we have successfully created a database (mystudentdb) and student table using MySQL Workbench.

2. Select the table, and when we hover a mouse pointer over the student table, you can see the table icons appears here. Click the table, which will open a new window where the upper section shows the MySQL statement, and the lower section shows the data rows.



3. To enter a data row, select the respected column, and insert the data value. Inserting data value in rows is similar to the Microsoft Excel Worksheet.

4. After entering the data rows, click on the Apply->Apply>Finish button to save the data rows.

5. Similarly, we can edit or modify the previously saved data rows. After modification, save new value, click on the Apply button to save changes. It will generate an SQL update statement save the changes to the database.

### Delete Row

1. To delete an individual row from the table, you need to select a data row, right-click on the right icon in front of the row and select Delete Row(s) option.

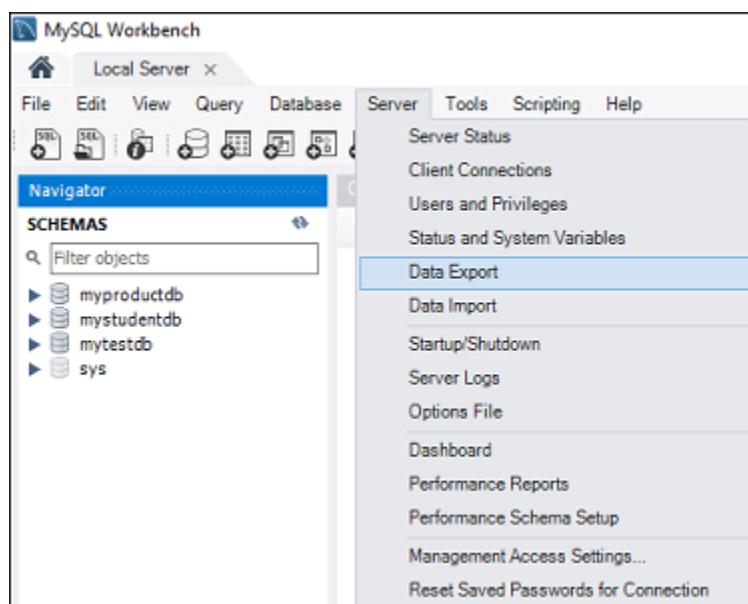
2. Now, click Apply->Apply->Finish button to save changes to the database.

### MySQL Workbench Export and Import Database(Table)

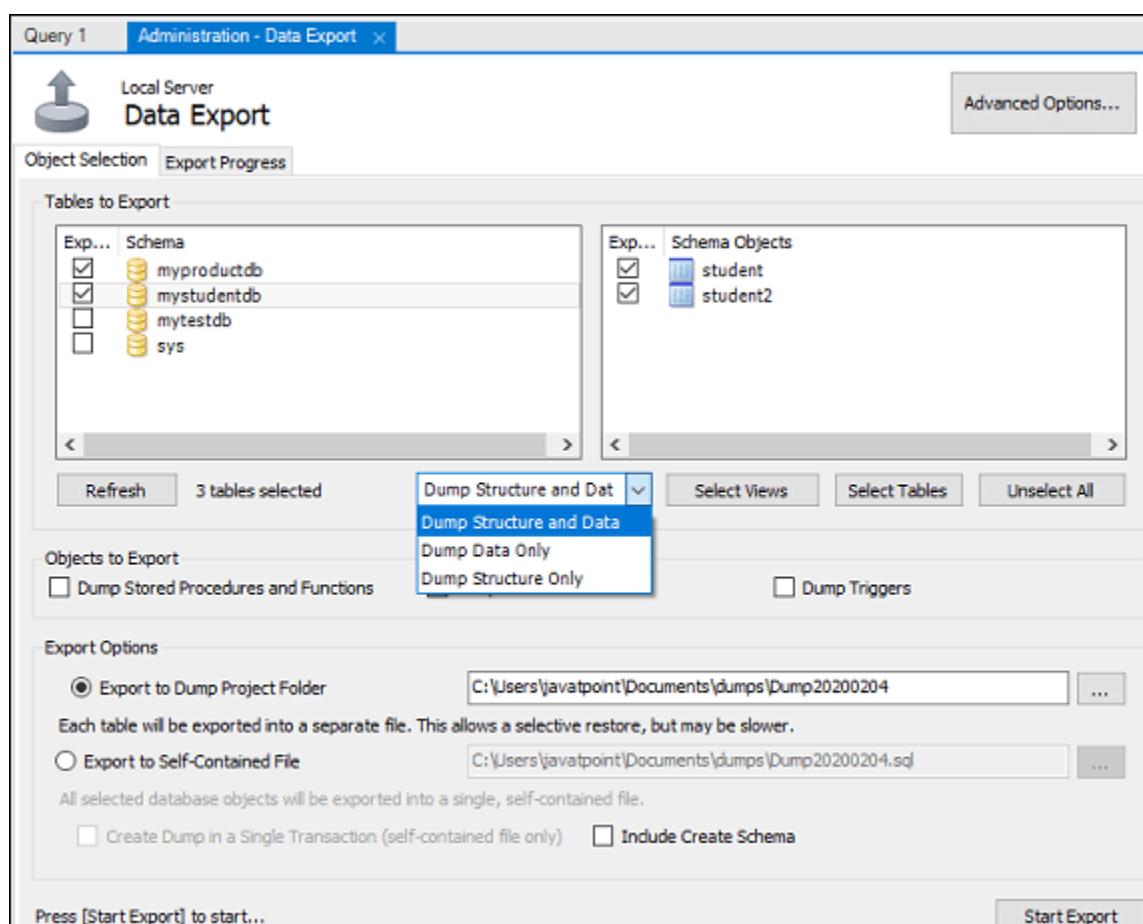
In this section, we are going to learn how we can export and import the database or table by using the MySQL Workbench.

## Export Databases(Tables)

1. To export databases or tables, go to the Menu bar, click on Server, and select the Data Export option, as shown in the following screen. It will open a new window of **data export** settings and options.



2. Select any database, and it will display all the corresponding tables under the selected database. Here, we can also select one or multiple database checkboxes to include the database in the Export file. Similarly, we can select one or multiple tables from the left section of the window.



3. Let us select two databases, namely (myproductdb and mystudentdb), including all tables under this database. Now, go to the drop-down setting, we can select 'Dump Structure and Data', 'Dump Data Only', and 'Dump Structure Only' option.

- **Dump Data and Structure:** It will save both table structure and data rows.
- **Dump Data Only:** It will save only the inserted rows in the tables.
- **Dump Structure Only:** It will save only the table structure, which are database columns and data types defined by us.

4. In the Export option, you can select the export path of your choice. Here, I will keep the default setting. Also, there are two radio buttons that are explained below.

- **Export to Dump Project Folder:** It will save all the tables as separate SQL files under one folder. It will be useful when you import or restore the export file one by one table.
- **Export to Self-Contained File:** It will store all the databases and tables in a single SQL file. It is a good option when you want to import all the databases, tables, and data rows using a single SQL file.

5. Click the Start Export button, which displays the progress bar and log. Now, open the Document folder in your system to locate the export files.

## Import Databases (Tables)

1. To import databases or tables, go to the Menu bar, click on Server, and select the **Data Import** option. It will open a new window of data import settings and options.

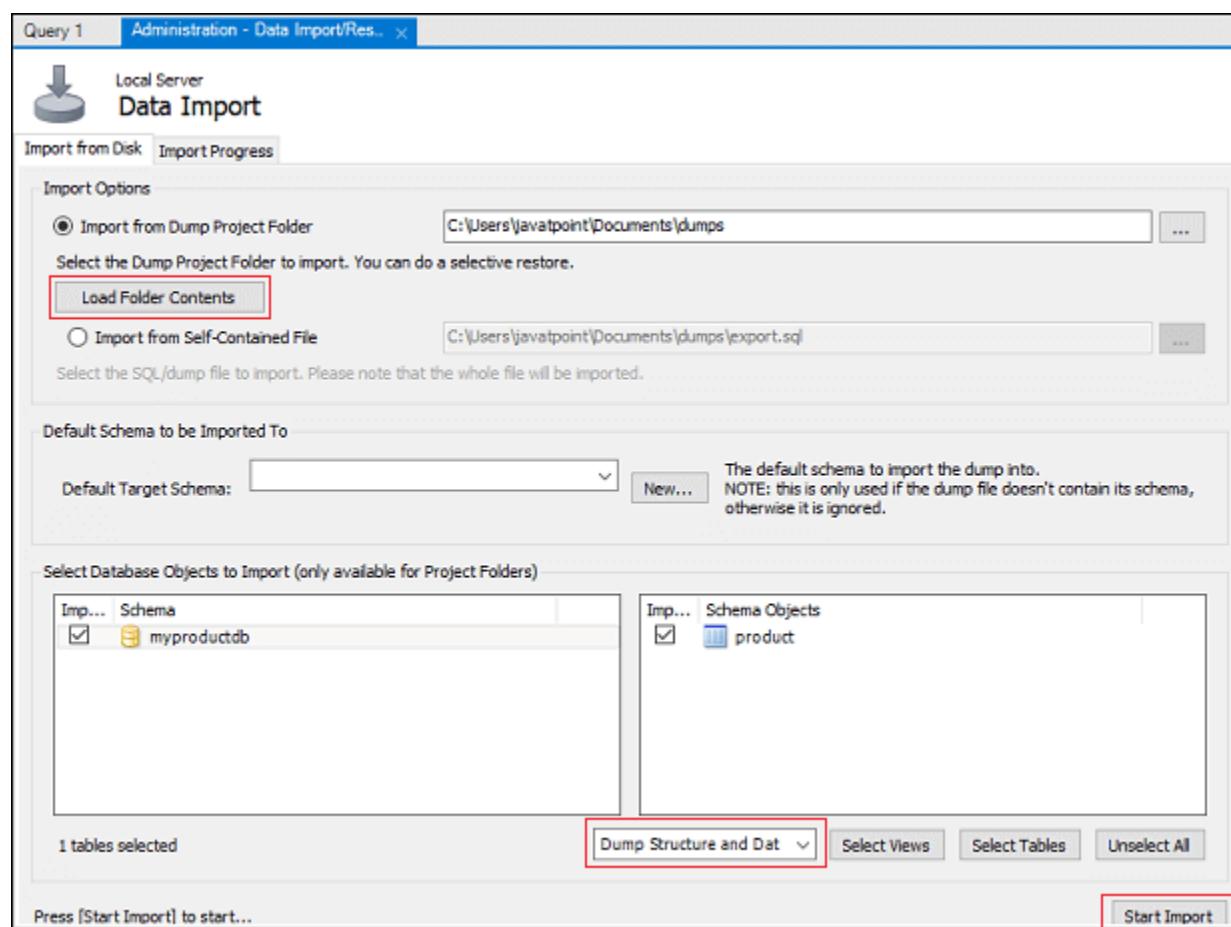
2. Here, you can see the two radio options to import databases and tables, which are:

- Import from Dump Project Folder
- Import by using Self-Contained File

3. We are going to select 'Import from Dump Project Folder' and click on 'Load Folder Content' to display all the available databases in the project folder.

4. Select **myproductdb** database from the Data Import option and also select the corresponding product table.

5. Choose the 'Dump Structure and Data' option and click the **Start Import** button to import the databases and tables from the backup file.



6. Now, go to the Schema->myproductdb->table and refresh it to see the currently imported database or table.

## User Management

# MySQL Create User

The MySQL user is a record in the **USER** table of the MySQL server that contains the login information, account privileges, and the host information for MySQL account. It is essential to create a user in MySQL for accessing and managing the databases.

The MySQL Create User statement allows us to create a new user account in the database server. It provides authentication, SSL/TLS, resource-limit, role, and password management properties for the new accounts. It also enables us to control the accounts that should be initially locked or unlocked.

If you want to use the Create User, it is required to have a **global** privilege of Create User statement or the **INSERT** privilege for the MySQL system schema. When you create a user that already exists, it gives an error. But if you use, **IF NOT EXISTS** clause, the statement gives a warning for each named user that already exists instead of an error message.

## Why Did Users require in MySQL server?

When the MySQL server installation completes, it has a **ROOT** user account only to access and manage the databases. But, sometimes, you want to give the database access to others without granting them full control. In that case, you will create a non-root user and grant them specific privileges to access and modify the database.

## Syntax

The following syntax is used to create a user in the database server.

1. `CREATE USER [IF NOT EXISTS] account_name IDENTIFIED BY 'password';`

In the above syntax, the **account\_name** has two parts one is the **username**, and another is the **hostname**, which is separated by @ symbol. Here, the username is the name of the user, and the hostname is the name of the host from which the user can connect with the database server.

1. `username@hostname`

The hostname is optional. If you have not given the hostname, the user can connect from any host on the server. The user account name without hostname can be written as:

1. `username@%`

**Note:** The Create User creates a new user with full access. So, if you want to give privileges to the user, it is required to use the GRANT statement.

## MySQL CREATE USER Example

The following are the step required to create a new user in the MySQL server database.

**Step 1:** Open the MySQL server by using the **mysql client tool**.

**Step 2:** Enter the password for the account and press Enter.

1. Enter **Password:** \*\*\*\*\*

**Step 3:** Execute the following command to show all users in the current MySQL server.

1. `mysql> select user from mysql.user;`

We will get the output as below:

user
mysql.infoschema
mysql.session
mysql.sys
root

4 rows in set (0.00 sec)

**Step 4:** Create a new user with the following command.

1. `mysql> create user peter@localhost identified by 'jtp12345';`

Now, run the command to show all users again.

user
mysql.infoschema
mysql.session
mysql.sys
peter
root

5 rows in set (0.00 sec)

In the above output, we can see that the user **peter** has been created successfully.

**Step 5:** Now, we will use the IF NOT EXISTS clause with the CREATE USER statement.

1. `mysql> CREATE USER IF NOT EXISTS adam@localhost IDENTIFIED BY 'jtp123456';`

## Grant Privileges to the MySQL New User

MySQL server provides multiple types of privileges to a new user account. Some of the most commonly used privileges are given below:

1. **ALL PRIVILEGES:** It permits all privileges to a new user account.
2. **CREATE:** It enables the user account to create databases and tables.

3. **DROP:** It enables the user account to drop databases and tables.
4. **DELETE:** It enables the user account to delete rows from a specific table.
5. **INSERT:** It enables the user account to insert rows into a specific table.
6. **SELECT:** It enables the user account to read a database.
7. **UPDATE:** It enables the user account to update table rows.

If you want to give all privileges to a newly created user, execute the following command.

1. mysql> **GRANT ALL PRIVILEGES ON \*.\* TO peter@localhost;**

If you want to give specific privileges to a newly created user, execute the following command.

1. mysql> **GRANT CREATE, SELECT, INSERT ON \*.\* TO peter@localhost;**

Sometimes, you want to **flush** all the privileges of a user account for changes occurs immediately, type the following command.

1. **FLUSH PRIVILEGES;**

If you want to see the existing privileges for the user, execute the following command.

1. mysql> **SHOW GRANTS FOR username;**

## MySQL Drop User

The MySQL Drop User statement allows us to **remove** one or more user accounts and their **privileges** from the database server. If the account does not exist in the database server, it gives an error.

If you want to use the Drop User statement, it is required to have a **global** privilege of Create User statement or the **DELETE** privilege for the MySQL system schema.

### Syntax

The following syntax is used to delete the user accounts from the database server completely.

1. **DROP USER 'account\_name';**

The **account\_name** can be identified with the following syntax:

1. **username@hostname**

Here, the **username** is the name of the account, which you want to delete from the database server and the **hostname** is the server name of the user account.

## MySQL Drop USER Example

The following are the step required to delete an existing user from the MySQL server database.

**Step 1:** Open the MySQL server by using the **mysql client tool**.

**Step 2:** Enter the password for the account and press Enter.

1. Enter **Password:** \*\*\*\*\*

**Step 3:** Execute the following command to show all users in the current MySQL server.

1. mysql> **select user from mysql.user;**

We will get the output as below:

user
adam
john
martin
mysql.infoschema
mysql.session
mysql.sys
peter
root

8 rows in set (0.00 sec)

**Step 4:** To drop a user account, you need to execute the following statement.

1. **DROP USER** martin@localhost;

Here, we are going to remove the username '**martin**' from the MySQL server. After the successful execution of the above command, you need to execute the show user statement again. You will get the following output where username martin is not present.

user
adam
john
mysql.infoschema
mysql.session
mysql.sys
peter
root

7 rows in set (0.00 sec)

**Step 5:** The **DROP USER** statement can also be used to remove more than one user accounts at once. We can drop multiple user accounts by separating account\_name with **comma** operator. To delete multiple user accounts, execute the following command.

1. **DROP USER** john@localhost, peter@localhost;

Here, we are going to remove **john** and **peter** accounts from the above image. After the successful execution of the above command, you need to execute the show user statement again. You will get the following output where username john and peter is not present.

user
adam
mysql.infoschema
mysql.session
mysql.sys
root

5 rows in set (0.00 sec)

**NOTE:** This statement cannot close any open user sessions automatically. In the case when the **DROP USER** statement executed and the session of this account is active, this statement does not take effect until its session is closed. The user account is dropped only when the session is closed, and that the user's next attempt will not be able to log in again.

## MySQL Show Users/List All Users

Sometimes you want to manage a database in MySQL. In that case, we need to see the list of all user's accounts in a database. Most times, we assume that there is a **SHOW USERS** command similar to SHOW DATABASES, SHOW TABLES, etc. for displaying the list of all users available in the database server. Unfortunately, MySQL database does not have a SHOW USERS command to display the list of all users in the MySQL server. We can use the following query to see the list of all user in the database server:

1. mysql> **Select user from** mysql.user;

After the successful execution of the above statement, we will get the user data from the user table of the MySQL database server.

Let us see how we can use this query. First, we have to open the MySQL server by using the **mysql client tool** and log in as an administrator into the server database. Execute the following query:

1. > mysql -u root -p
2. Enter **password**: \*\*\*\*\*
3. mysql> use mysql;
4. **Database** changed
5. mysql> **SELECT user FROM user;**

We will get the following output where we can see the **five** users in our local database:

```
mysql> select user from user;
+-----+
| user |
+-----+
| adam |
| mysql.infoschema |
| mysql.session |
| mysql.sys |
| root |
+-----+
5 rows in set (0.00 sec)
```

If we want to see more information on the user table, execute the command below:

1. mysql> **DESC user;**

It will give the following output that lists all the available columns of the **mysql.user** database:

```
MySQL 8.0 Command Line Client
mysql> DESC user;
+-----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| Host | char(255) | NO | PRI | PRI |       |
| User | char(32) | NO | PRI | PRI |       |
| Select_priv | enum('N','Y') | NO |       | N |       |
| Insert_priv | enum('N','Y') | NO |       | N |       |
| Update_priv | enum('N','Y') | NO |       | N |       |
| Delete_priv | enum('N','Y') | NO |       | N |       |
| Create_priv | enum('N','Y') | NO |       | N |       |
| Drop_priv | enum('N','Y') | NO |       | N |       |
| Reload_priv | enum('N','Y') | NO |       | N |       |
| Shutdown_priv | enum('N','Y') | NO |       | N |       |
| Process_priv | enum('N','Y') | NO |       | N |       |
| File_priv | enum('N','Y') | NO |       | N |       |
| Grant_priv | enum('N','Y') | NO |       | N |       |
| References_priv | enum('N','Y') | NO |       | N |       |
| Index_priv | enum('N','Y') | NO |       | N |       |
| Alter_priv | enum('N','Y') | NO |       | N |       |
| Show_db_priv | enum('N','Y') | NO |       | N |       |
| Super_priv | enum('N','Y') | NO |       | N |       |
| Create_tmp_table_priv | enum('N','Y') | NO |       | N |       |
| Lock_tables_priv | enum('N','Y') | NO |       | N |       |
| Execute_priv | enum('N','Y') | NO |       | N |       |
| Repl_slave_priv | enum('N','Y') | NO |       | N |       |
| Repl_client_priv | enum('N','Y') | NO |       | N |       |
| Create_view_priv | enum('N','Y') | NO |       | N |       |
| Show_view_priv | enum('N','Y') | NO |       | N |       |
+-----+-----+-----+-----+-----+-----+
```

To get the selected information like as hostname, password expiration status, and account locking, execute the query as below:

1. mysql> **SELECT user, host, account\_locked, password\_expired FROM user;**

After the successful execution, it will give the following output:

```
mysql> SELECT user, host, account_locked, password_expired FROM user;
+-----+-----+-----+-----+
| user | host | account_locked | password_expired |
+-----+-----+-----+-----+
| adam | localhost | N | N |
| mysql.infoschema | localhost | Y | N |
| mysql.session | localhost | Y | N |
| mysql.sys | localhost | Y | N |
| root | localhost | N | N |
+-----+-----+-----+-----+
5 rows in set (0.00 sec)
```

## Show Current User

We can get information of the current user by using the **user()** or **current\_user()** function, as shown below:

1. mysql> **Select user();**
2. or,
3. mysql> **Select current\_user();**

After executing the above command, we will get the following output:

```

mysql> Select user();
+-----+
| user() |
+-----+
| root@localhost |
+-----+
1 row in set (0.00 sec)

mysql> Select current_user();
+-----+
| current_user() |
+-----+
| root@localhost |
+-----+
1 row in set (0.00 sec)

```

## Show Current Logged User

We can see the currently logged user in the database server by using the following query in the MySQL server:

1. mysql> **SELECT** user, host, db, command **FROM** information\_schema.processlist;

The above command gives the output, as shown below:

```

mysql> SELECT user, host, db, command FROM information_schema.processlist;
+-----+-----+-----+-----+
| user | host | db | command |
+-----+-----+-----+-----+
| root | localhost:64982 | mystudentdb | Sleep |
| root | localhost:64983 | mystudentdb | Sleep |
| event_scheduler | localhost | NULL | Daemon |
| root | localhost:49742 | mysql | Query |
+-----+-----+-----+-----+
4 rows in set (0.00 sec)

```

In this output, we can see that there are currently **four** users logged in the database, where one is executing a **Query**, and others show in **Sleep or Daemon** status.

## Change MySQL User Password

MySQL user is a record that contains the login information, account privileges, and the host information for MySQL account to access and manage the database. The login information includes the user name and password. In some cases, there is a need to change the user password in the MySQL database.

To change the password of any user account, you must have to keep this information in your mind:

- o The details of the user account that you want to change.
- o An application used by the user whose password you want to change. If you reset the user account password without changing an application connection string, then the application cannot connect with the database server.

MySQL allows us to change the user account password in three different ways, which are given below:

1. UPDATE Statement
2. SET PASSWORD Statement
3. ALTER USER Statement

Let us see how we can change the user account password in MySQL by using the above statement in detail:

### Change user account password using the UPDATE statement

This statement is the first way to change the user password for updating the user table of the MySQL database. Here, you have to use the **FLUSH PRIVILEGE** statement after executing an UPDATE statement for reloading privileges from the grant table of the MySQL database.

Suppose, you want to change or update the password for a user **peter** that connects from the localhost with the password **jtp12345**, execute the SQL statements as below:

1. mysql> USE mysql;
- 2.
3. mysql> **UPDATE** user **SET** password = **PASSWORD**('jtp12345') **WHERE** user = 'peter' AND host = 'localhost';
- 4.
5. mysql> **FLUSH** **PRIVILEGES**;

If you are using the MySQL version 5.7.6 or higher, the above statement will not work. It is because the MySQL user table contains the **authentication\_string** column that stores the password only. Now, the higher versions contain the authentication\_string column in the UPDATE statement, like the following statement.

1. mysql> USE mysql;
- 2.
3. mysql> **UPDATE user SET** authentication\_string = **PASSWORD('jtp12345')** **WHERE** user = 'peter' AND host = 'localhost';
- 4.
5. mysql> FLUSH **PRIVILEGES**;

#### Change user account password using SET PASSWORD statement

The SET PASSWORD statement is the second way to change the user password in the MySQL database. If you want to change the other account password, you must have the UPDATE privilege. The SET PASSWORD statement uses the user account in the **username@localhost** format.

There is no need to use the FLUSH PRIVILEGES statement for reloading privileges from the grant tables of the MySQL database. We can use the following statement to change the password of user account peter by using the SET PASSWORD statement:

1. mysql> **SET PASSWORD FOR 'peter'@'localhost' = PASSWORD('jtp12345');**

If you are using the MySQL version 5.7.6 or higher, the above statement deprecated and will not work in future releases. Instead, we need to use the following statement:

1. mysql> **SET PASSWORD FOR 'peter'@'localhost' = jtp12345;**

#### Change user account password using ALTER USER statement

The ALTER USER statement is the third way to change the user password in the MySQL database. MySQL uses ALTER USER statement with the IDENTIFIED BY clause for changing the password of a user account. We need to use the following syntax to change the password of a user **peter** with **jtp123**.

1. mysql> **ALTER USER peter@localhost IDENTIFIED BY 'jtp123';**

Sometimes, you need to reset the MySQL **root** account password. In that case, you can force to stop and restart the MySQL database server without using the grant table validation.

## MySQL Database

# MySQL Create Database

A database is used to store the collection of records in an organized form. It allows us to hold the data into tables, rows, columns, and indexes to find the relevant information frequently. We can access and manage the records through the database very easily.

[MySQL](#) implements a database as a directory that stores all files in the form of a table. It allows us to create a database mainly in **two ways**:

1. MySQL Command Line Client
2. MySQL Workbench

## MySQL Command Line Client

We can create a new database in MySQL by using the **CREATE DATABASE** statement with the below syntax:

1. **CREATE DATABASE [IF NOT EXISTS] database\_name**
2. **[CHARACTER SET charset\_name]**
3. **[COLLATE collation\_name];**

#### Parameter Explanation

The parameter descriptions of the above syntax are as follows:

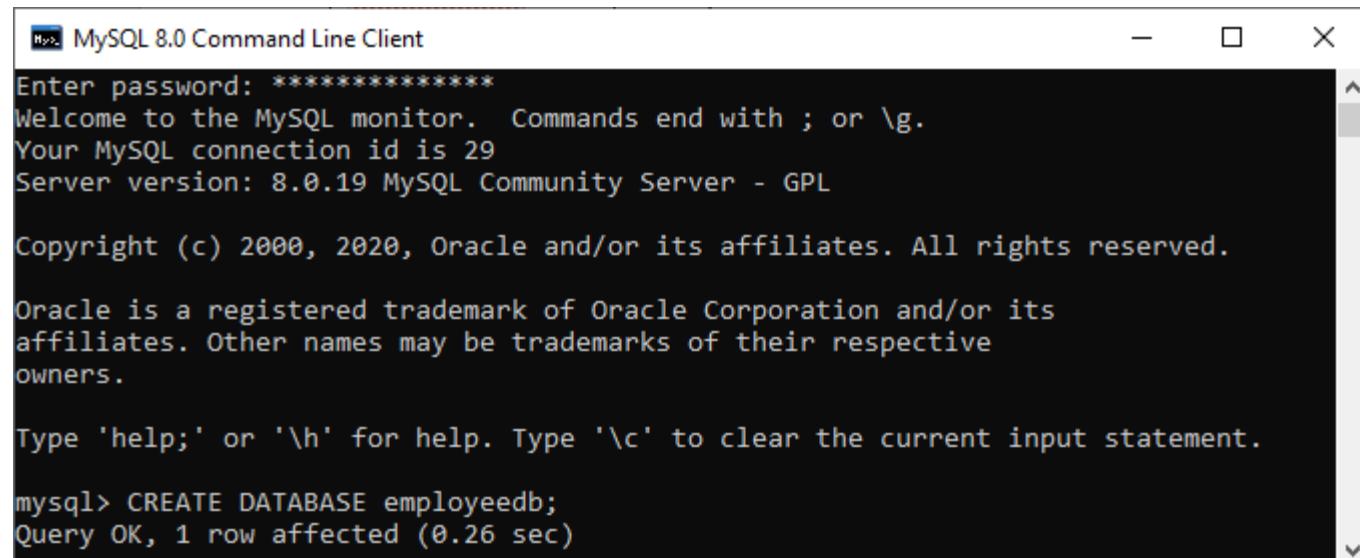
Parameter	Description
database_name	It is the name of a new database that should be unique in the MySQL server instance. The <b>IF NOT EXIST</b> clause avoids an error when we create a database that already exists.
charset_name	It is optional. It is the name of the character set to store every character in a string. MySQL database server supports many character sets. If we do not provide this in the statement, MySQL takes the default character set.
collation_name	It is optional that compares characters in a particular character set.

### Example

Let us understand how to create a database in MySQL with the help of an example. Open the MySQL console and write down the password, if we have set during installation. Now we are ready to create a database. Here, we are going to create a database name "**employeedb**" using the following statement:

1. mysql> **CREATE DATABASE** employeedb;

It will look like the below output:



```

MySQL 8.0 Command Line Client
Enter password: *****
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 29
Server version: 8.0.19 MySQL Community Server - GPL

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

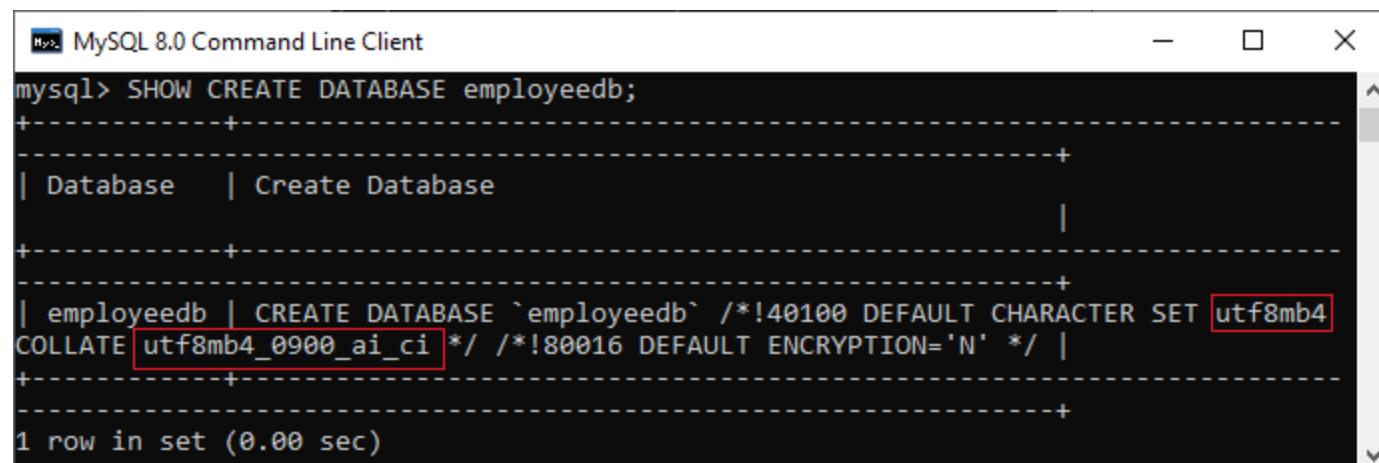
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> CREATE DATABASE employeedb;
Query OK, 1 row affected (0.26 sec)

```

We can review the newly created database using the below query that returns the database name, character set, and collation of the database:

1. mysql> **SHOW CREATE DATABASE** employeedb;



```

MySQL 8.0 Command Line Client
mysql> SHOW CREATE DATABASE employeedb;
+-----+
| Database | Create Database
+-----+
| employeedb | CREATE DATABASE `employeedb` /*!40100 DEFAULT CHARACTER SET utf8mb4
COLLATE utf8mb4_0900_ai_ci */ /*!80016 DEFAULT ENCRYPTION='N' */
+-----+
1 row in set (0.00 sec)

```

We can check the created database using the following query:

1. mysql> **SHOW DATABASES;**

After executing the above query, we can see all the created databases in the server.

```

MySQL 8.0 Command Line Client
mysql> SHOW DATABASES;
+-----+
| Database |
+-----+
| employeedb |
| information_schema |
| myemployeedb |
| mysql |
| mysqltestdb |
| mystudentdb |
| mytestdb_copy |
| performance_schema |
| sakila |
| sys |
| testdb |
| world |
+-----+

```

Finally, we can use the below command to access the database that enables us to create a table and other database objects.

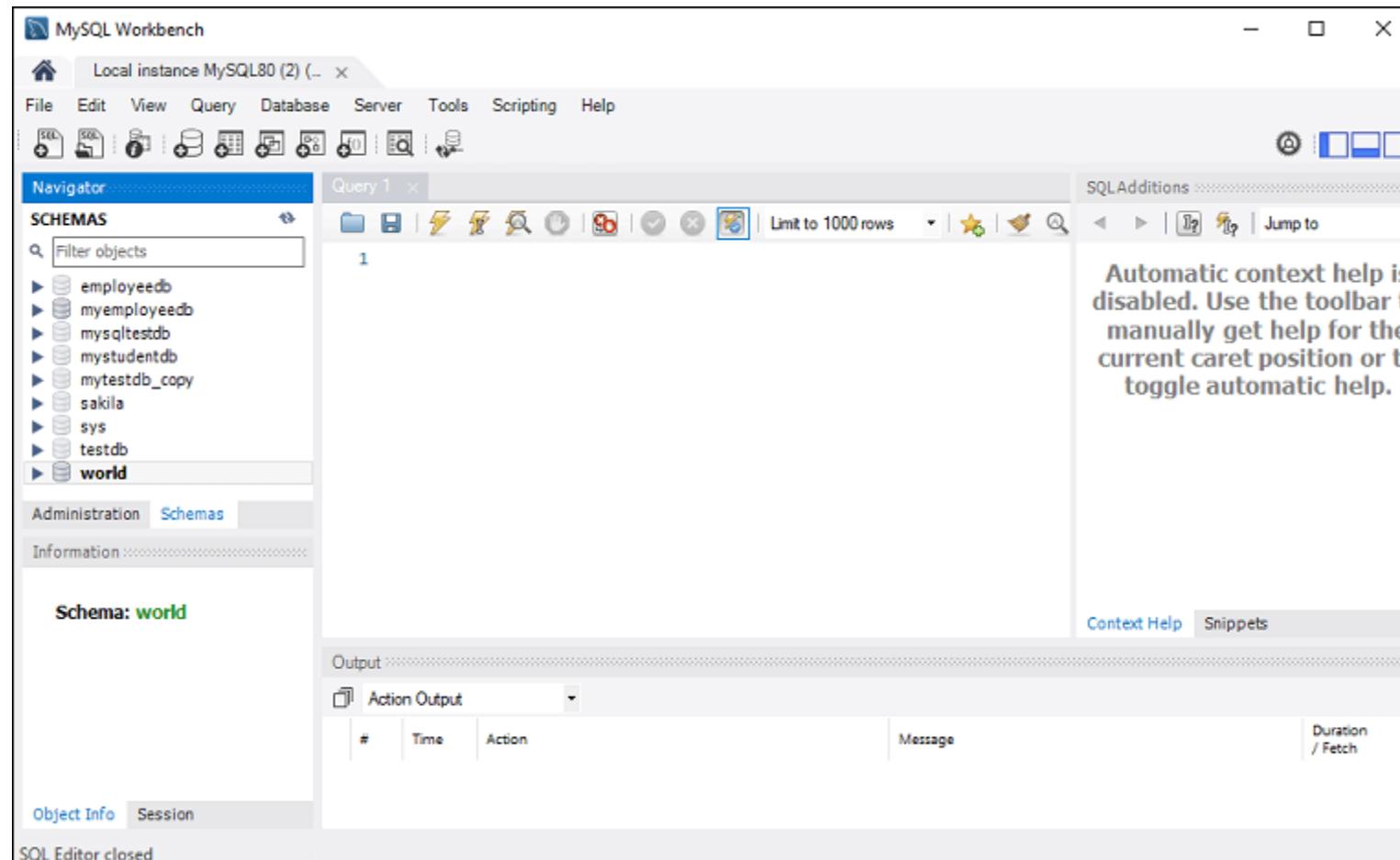
1. mysql> USE employeedb;

**NOTE:** All the database names, table names, and table field names are case sensitive. We must have to use proper names while giving any SQL command.

## MySQL Workbench

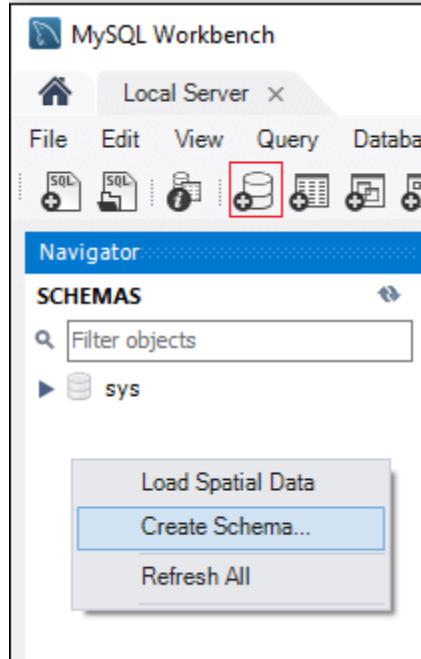
It is a visual database designing or GUI tool used to work with database architects, developers, and Database Administrators. This visual tool supports [SQL](#) development, data modeling, data migration, and comprehensive administration tools for server configuration, user administration, backup, and many more. It allows us to create new physical data models, E-R diagrams, and SQL development (run queries, etc.).

To create a new database using this tool, we first need to launch the [MySQL Workbench](#) and log in using the username and password that you want. It will show the following screen:

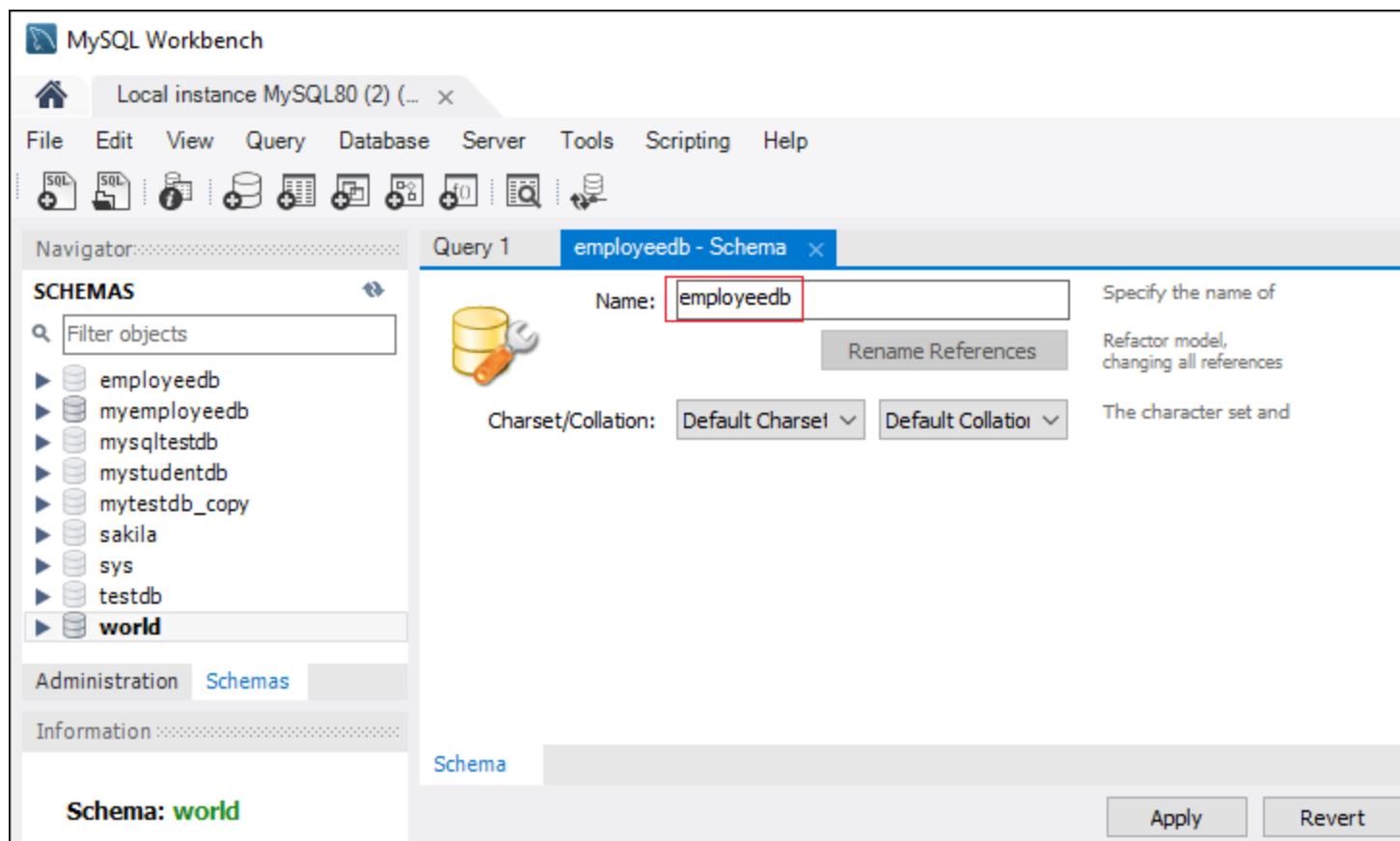


Now do the following steps for database creation:

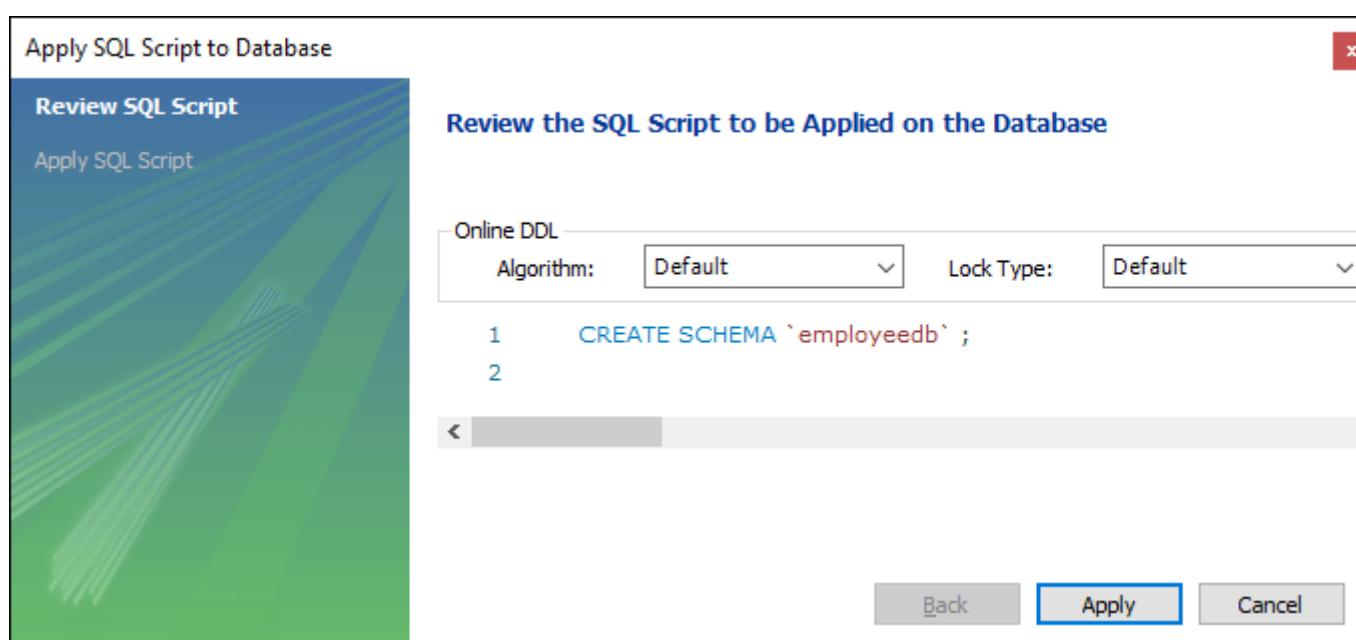
1. Go to the Navigation tab and click on the **Schema menu**. Here, we can see all the previously created databases. If we want to create a new database, right-click under the Schema menu and select Create Schema or click the database icon (red rectangle), as shown in the following screen.



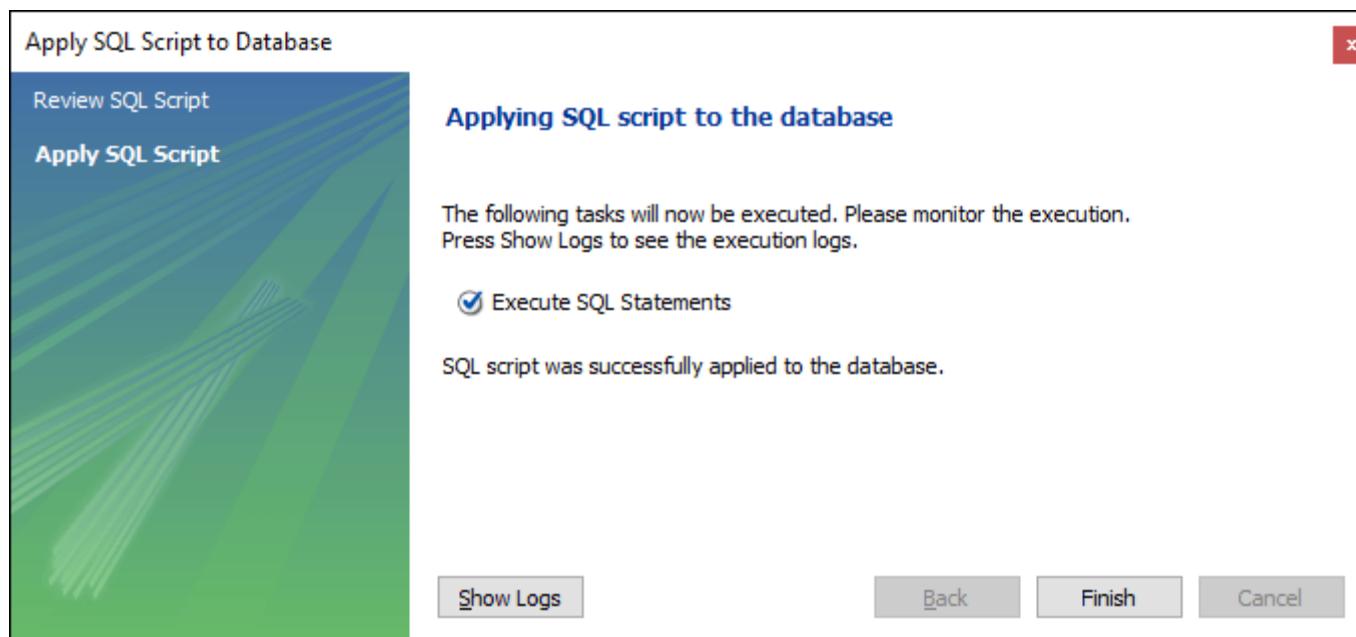
2. The new Schema window screen open. Enter the new database name (for example, **employeedb**) and use default character set and collation. Now, click on the Apply button as shown in the screen below:



3. A new popup window appears. Click on the **Apply** button.

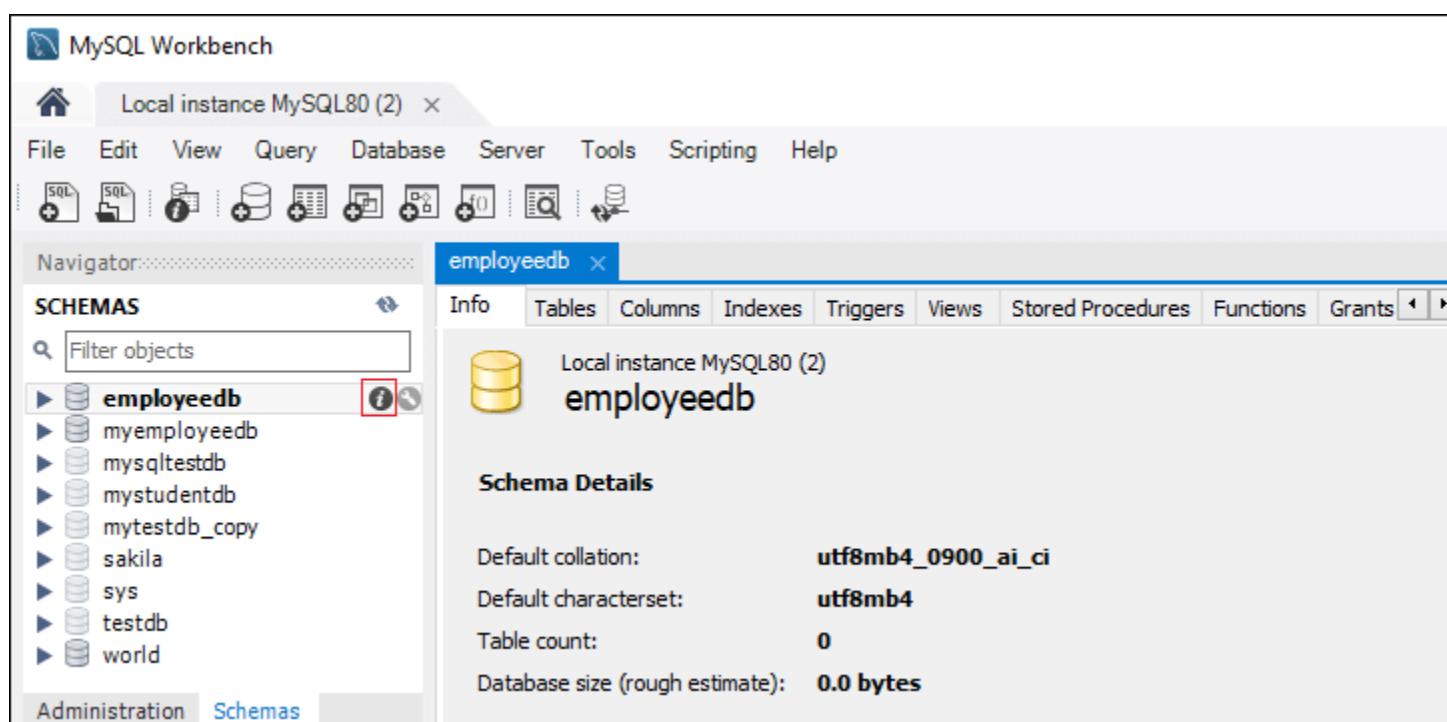


4. A new popup screen appears. Click on the **Finish** button to complete the database creation.



5. After successful database creation, we can see new databases in the Schema menu. If we do not see this, click on the **refresh icon** into the Schema menu.

6. We can see more information about the database by selecting the database and click on the '**i**' icon. The information window displays several options, like Table, Triggers, Indexes, Users, and many more.



7. MySQL Workbench does not provide an option to rename the database name, but we can create, update, and delete the table and data rows from the database.

## MySQL SELECT Database

SELECT Database is used in MySQL to select a particular database to work with. This query is used when multiple databases are available with MySQL Server.

You can use SQL command **USE** to select a particular database.

### Syntax:

1. USE database\_name;

### Example:

Let's take an example to use a database name "customers".

1. USE customers;

It will look like this:



The screenshot shows the MySQL 5.5 Command Line Client window. The title bar says "MySQL 5.5 Command Line Client". The main area displays the following SQL session:

```
+-----+  
| Database |  
+-----+  
| information_schema |  
| customers |  
| mysql |  
| performance_schema |  
| sssit |  
| test |  
+-----+  
6 rows in set <0.01 sec>  
  
mysql> USE customers;  
Database changed  
mysql>
```

**Note:** All the database names, table names and table fields name are case sensitive. You must have to use proper names while giving any SQL command.

## MySQL Show/List Databases

When we work with the MySQL server, it is a common task to show or list the databases, displaying the table from a particular database, and information of user accounts and their privileges that reside on the server. In this article, we are going to focus on how to list databases in the MySQL server.

We can list all the databases available on the MySQL server host using the following command, as shown below:

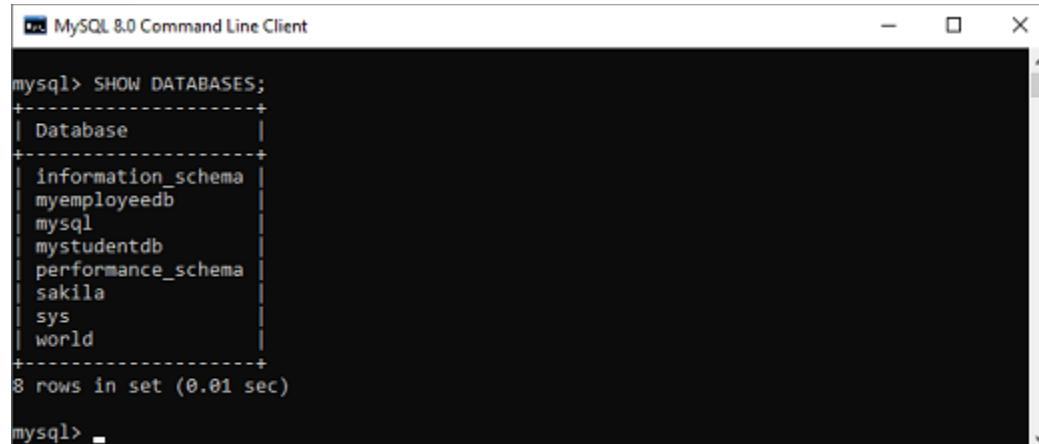
1. mysql> SHOW DATABASES;

Open the MySQL Command Line Client that appeared with a **mysql> prompt**. Next, **log in** to the MySQL database server using the **password** that you have created during the installation of MySQL. Now, you are connected to the MySQL server host, where you can execute all the SQL statements. Finally, run the SHOW Databases command to list/show databases.

We can see the following output that explains it more clearly:

Competitive questions on Structures in Hindi

Keep Watching



The screenshot shows the MySQL 8.0 Command Line Client window. The title bar says "MySQL 8.0 Command Line Client". The main area displays the following SQL session:

```
+-----+  
| Database |  
+-----+  
| information_schema |  
| myemployeedb |  
| mysql |  
| mystudentdb |  
| performance_schema |  
| sakila |  
| sys |  
| world |  
+-----+  
8 rows in set (0.01 sec)  
  
mysql>
```

MySQL also allows us another command to list the databases, which is a **SHOW SCHEMAS** statement. This command is the synonyms of the SHOW DATABASES and gives the same result. We can understand it with the following output:



The screenshot shows the MySQL 8.0 Command Line Client window. The title bar says "MySQL 8.0 Command Line Client". The main area displays the following SQL session:

```
+-----+  
| Database |  
+-----+  
| information_schema |  
| myemployeedb |  
| mysql |  
| mystudentdb |  
| performance_schema |  
| sakila |  
| sys |  
| world |  
+-----+  
8 rows in set (0.00 sec)  
  
mysql>
```

## List Databases Using Pattern Matching

Show Databases command in MySQL also provides an option that allows us to **filter** the returned database using different pattern matching with **LIKE** and **WHERE** clause. The LIKE clause list the database name that matches the specified pattern. The WHERE clause provides more flexibility to list the database that matches the given condition in the SQL statement.

## Syntax

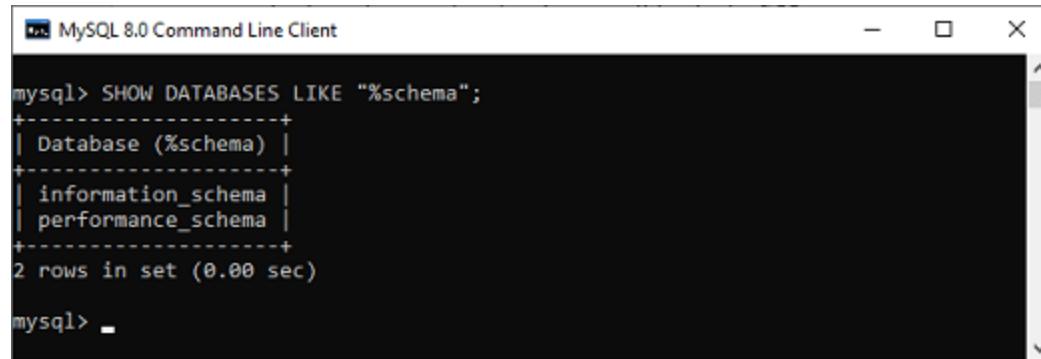
The following are the syntax to use pattern matching with Show Databases command:

1. mysql> SHOW DATABASES **LIKE** pattern;
2. OR,
3. mysql> SHOW DATABASES **WHERE** expression;

We can understand it with the example given below where **percent (%) sign** assumes zero, one, or multiple characters:

1. mysql> SHOW DATABASES **LIKE** "%schema";

The above statement will give the following output:



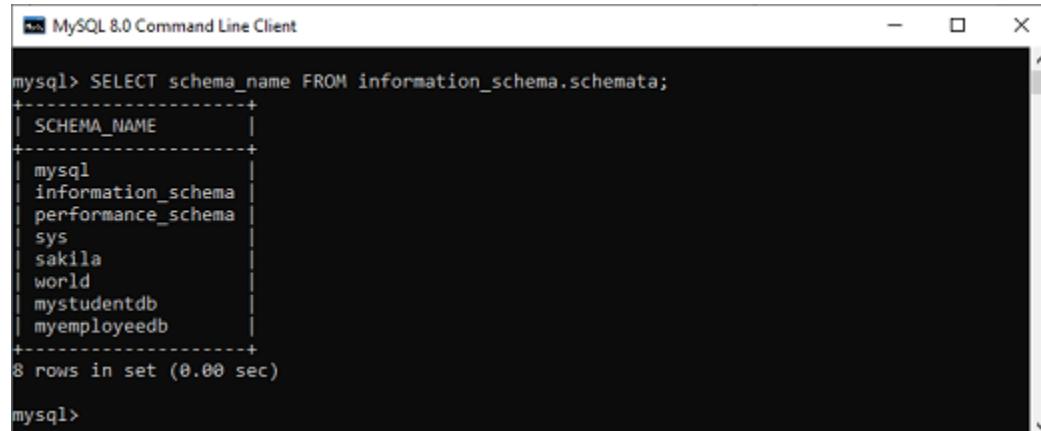
```
MySQL 8.0 Command Line Client
mysql> SHOW DATABASES LIKE "%schema";
+-----+
| Database (%schema) |
+-----+
| information_schema |
| performance_schema |
+-----+
2 rows in set (0.00 sec)

mysql>
```

Sometimes the LIKE clause is not sufficient; then, we can make a more complex search to query the database information from the schemata table in the information schema. The information schema in MySQL is an information database so that we can use it to get the output using the SHOW DATABASES command.

1. mysql> **SELECT** schema\_name **FROM** information\_schema.schemata;

This statement will give the same result as the SHOW DATABASES command:



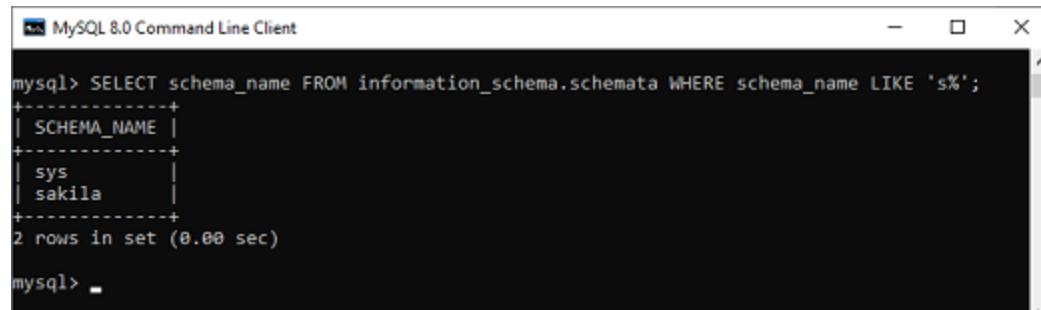
```
MySQL 8.0 Command Line Client
mysql> SELECT schema_name FROM information_schema.schemata;
+-----+
| SCHEMA_NAME |
+-----+
| mysql        |
| information_schema |
| performance_schema |
| sys          |
| sakila       |
| world        |
| mystudentdb  |
| myemployeedb |
+-----+
8 rows in set (0.00 sec)

mysql>
```

Now, we are going to see how we can use the WHERE clause with the SHOW DATABASES command. This statement returns the database whose schema name starts with "s":

1. mysql> **SELECT** schema\_name **FROM** information\_schema.schemata **WHERE** schema\_name **LIKE** 's%';

It will give the following output:



```
MySQL 8.0 Command Line Client
mysql> SELECT schema_name FROM information_schema.schemata WHERE schema_name LIKE 's%';
+-----+
| SCHEMA_NAME |
+-----+
| sys          |
| sakila       |
+-----+
2 rows in set (0.00 sec)

mysql>
```

**NOTE:** It is to be noted that if the MySQL server started with the "--skip-show-database" option, we could not use the SHOW DATABASES command unless we have the SHOW DATABASES privilege.

## MySQL DROP Database

We can drop/delete/remove a MySQL database quickly with the MySQL DROP DATABASE command. It will delete the database along with all the tables, indexes, and constraints permanently. Therefore, we should have to be very careful while removing the database in MySQL because we will lose all the data available in the database. If the database is not available in the MySQL server, the DROP DATABASE statement throws an error.

[MySQL](#) allows us to drop/delete/remove a database mainly in **two ways**:

- MySQL Command Line Client
- MySQL Workbench

## MySQL Command Line Client

We can drop an existing database in MySQL by using the DROP DATABASE statement with the below syntax:

1. **DROP DATABASE** [IF EXISTS] database\_name;

In MySQL, we can also use the below syntax for deleting the database. It is because the **schema** is the synonym for the database, so we can use them interchangeably.

1. **DROP SCHEMA** [IF EXISTS] database\_name;

### Parameter Explanation

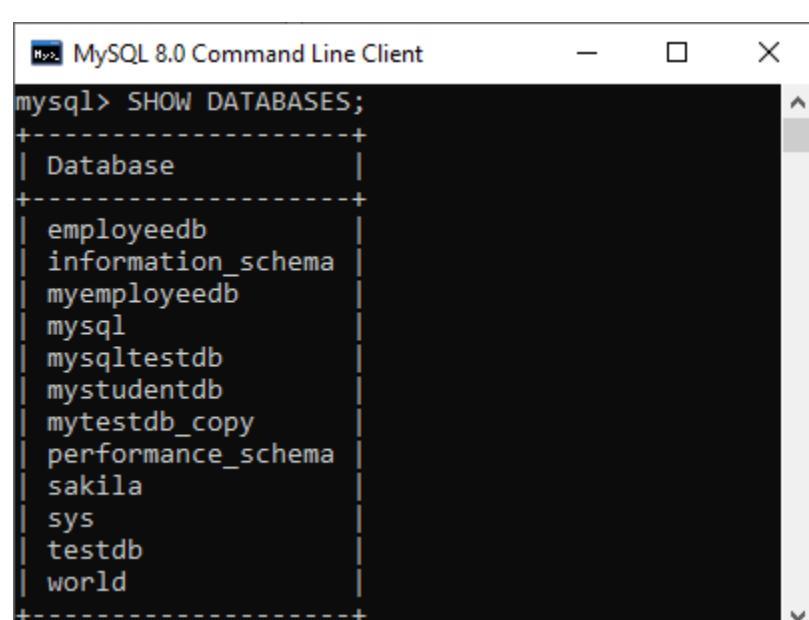
The parameter descriptions of the above syntax are as follows:

Parameter	Description
database_name	It is the name of an existing database that we want to delete from the server. It should be unique in the MySQL server instance.
IF EXISTS	It is optional. It is used to prevent from getting an error while removing a database that does not exist.

### Example

Let us understand how to drop a database in MySQL with the help of an example. Open the MySQL console and write down the password, if we have set during installation. Now we are ready to delete a database.

Next, use the **SHOW DATABASES** statement to see all available database in the server:



```
MySQL 8.0 Command Line Client - 2024-01-15 14:45:22
+-----+
| Database |
+-----+
| employeedb |
| information_schema |
| myemployeedb |
| mysql |
| mysqltestdb |
| mystudentdb |
| mytestdb_copy |
| performance_schema |
| sakila |
| sys |
| testdb |
| world |
+-----+
```

Suppose we want to remove a database named "**mytestdb\_copy**". Execute the below statement:

1. **DROP DATABASE** mytestdb\_copy;

Now we can verify that either our database is removed or not by executing the following query. It will look like this:

```
MySQL 8.0 Command Line Client
mysql> DROP DATABASE mytestdb_copy;
Query OK, 3 rows affected (1.85 sec)

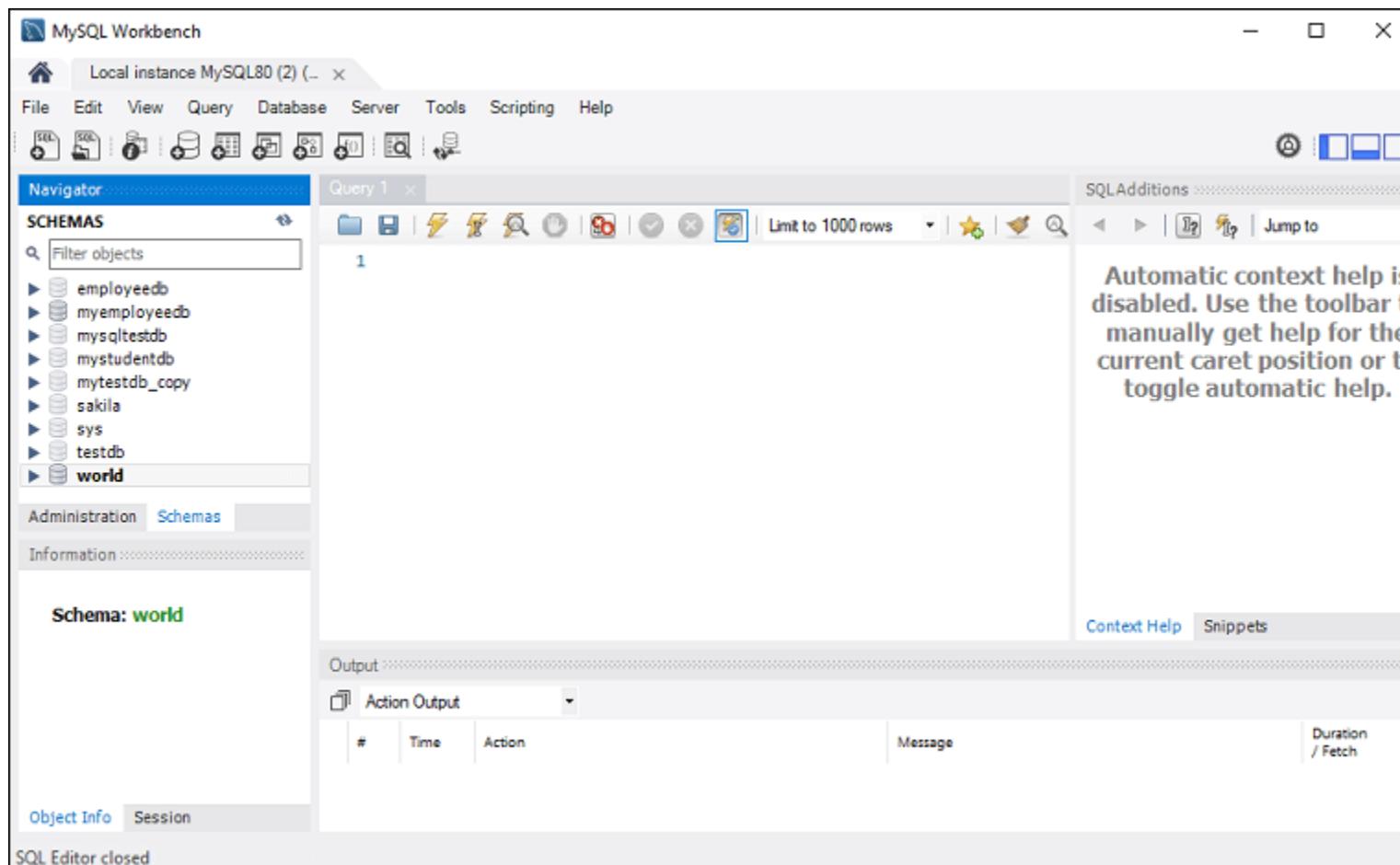
mysql> SHOW DATABASES;
+-----+
| Database |
+-----+
| employeedb |
| information_schema |
| myemployeedb |
| mysql |
| mysqltestdb |
| mystudentdb |
| performance_schema |
| sakila |
| sys |
| testdb |
| world |
+-----+
```

From the above, we can see that the database "mytestdb\_copy" is removed successfully.

**Note:** All the database names, table names, and table field names are case sensitive. We must have to use proper names while giving any SQL command.

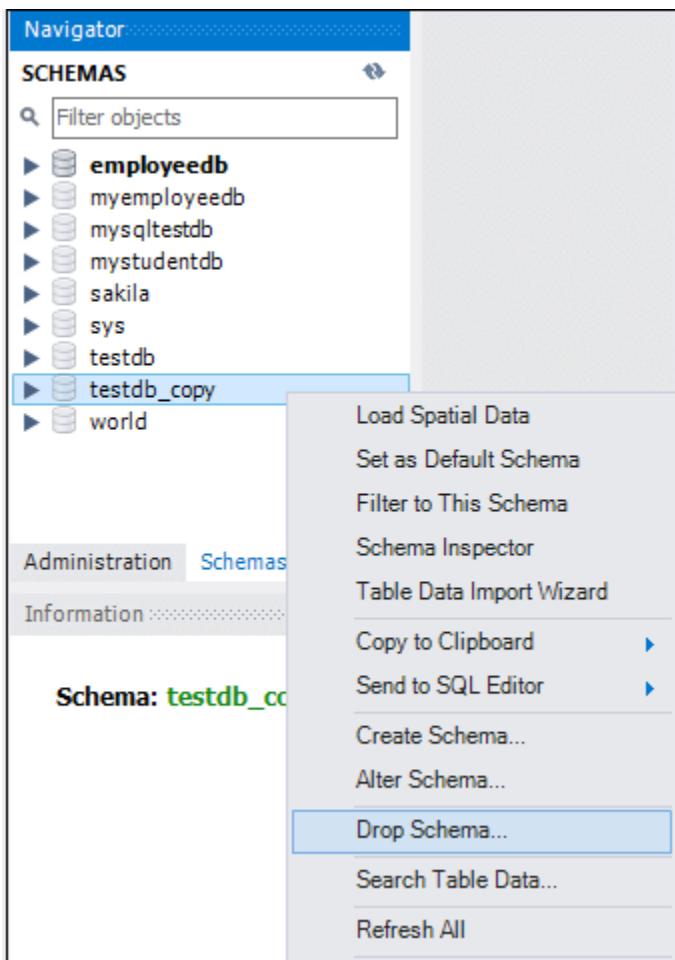
## DROP Database using MySQL Workbench

To drop a database using this tool, we first need to launch the [MySQL Workbench](#) and log in with the **username** and **password** to the MySQL server. It will show the following screen:

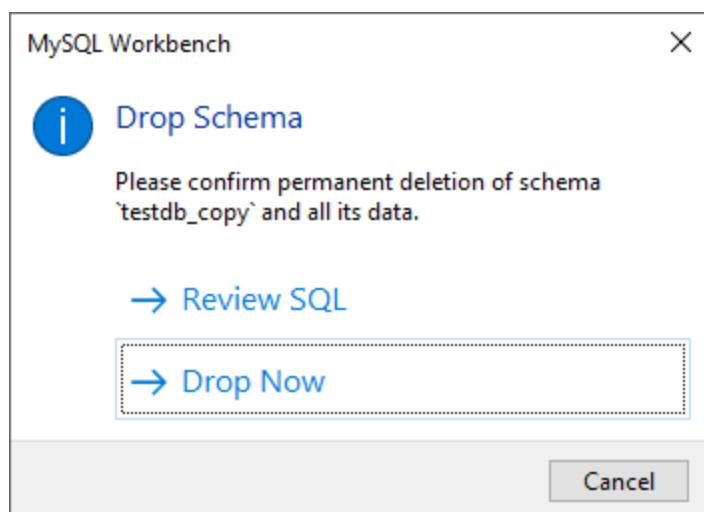


Now do the following steps for database deletion:

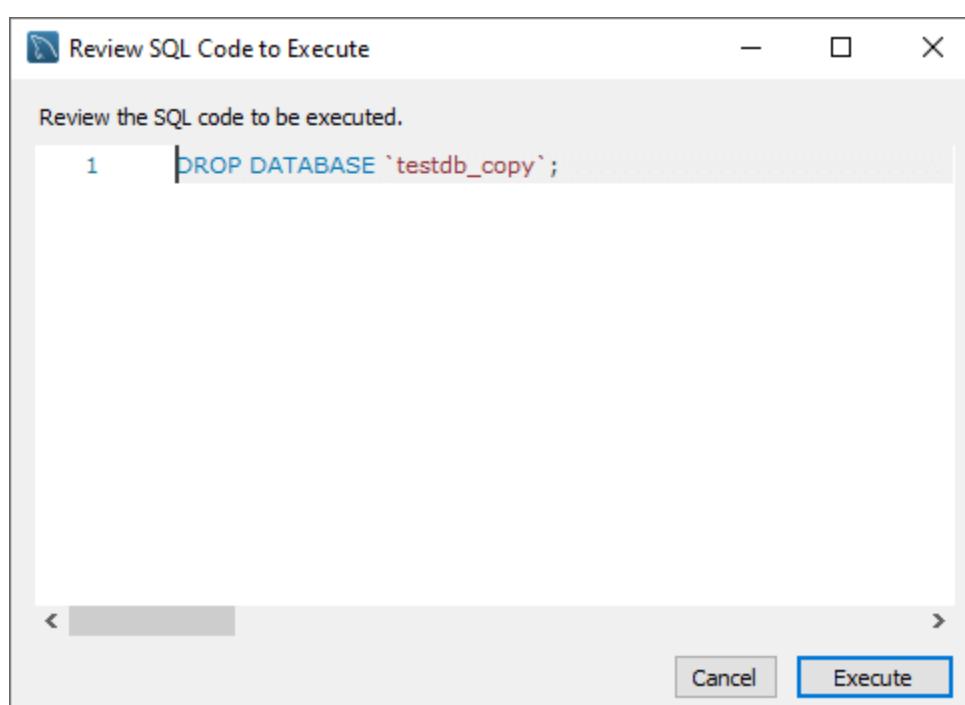
1. Go to the Navigation tab and click on the **Schema menu**. Here, we can see all the previously created databases. If we want to delete a database, right-click the database that you want to remove, for example, **testdb\_copy** under the Schema menu and select **Drop Schema** option, as shown in the following screen.



When we click the Drop Schema option, MySQL Workbench displays a dialog box to confirm the deletion process. If we select **Review SQL**, it will produce the [SQL](#) statement that will be executed. And if we choose **Drop Now** option, the database will be deleted permanently.



If we want the safe deletion of the database, it is required to choose the Review SQL option. Once we are sure, click the Execute button to execute the statement. The below screen explains it more clearly:



Once we click the execute button, MySQL will return the below message indicating that the database is dropped successfully. Since the database testdb\_copy is an empty database, the number of affected rows is zero.

Output ::::::					
Action Output			Message		Duration / Fetch
#	Time	Action			
1	11:08:31	DROP DATABASE `testdb_copy`	0 row(s) affected		0.234 sec

If we verify the schemas tab, we will not find the testdb\_copy database on the list anymore.

## MySQL COPY Database

A database is an application used for storing the organized collection of records that can be accessed and managed by the user. It holds the data into tables, rows, columns, and indexes to quickly find the relevant information.

MySQL copy or clone database is a feature that allows us to create a **duplicate copy of an existing database**, including the table structure, indexes, constraints, default values, etc. Making a duplicate copy of an original database into a new database is very useful when accidentally our database is **lost or failure**. The most common use of making a duplicate copy of the database is for data backups. It is also useful when planning the major changes to the structure of the original database.

In [MySQL](#), making the clone of an original database is a **three-step process**: First, the original database records are dumped (copied) to a temporary file that holds the SQL commands for reinserting the data into the new database. Second, it is required to create a new database. Finally, the [SQL](#) file is processed, and the data will be copied into the new database.

We need to follow these steps to copy a database to another database:

1. First, use the **CREATE DATABASE** statement to create a new database.
2. Second, store the data to an **SQL file**. We can give any name to this file, but it must end with a **.sql** extension.
3. Third, export all the database objects along with its data to copy using the **mysqldump** tool and then import this file into the new database.

For the demonstration, we will copy the **testdb** database to **testdb\_copy** database using the following steps:

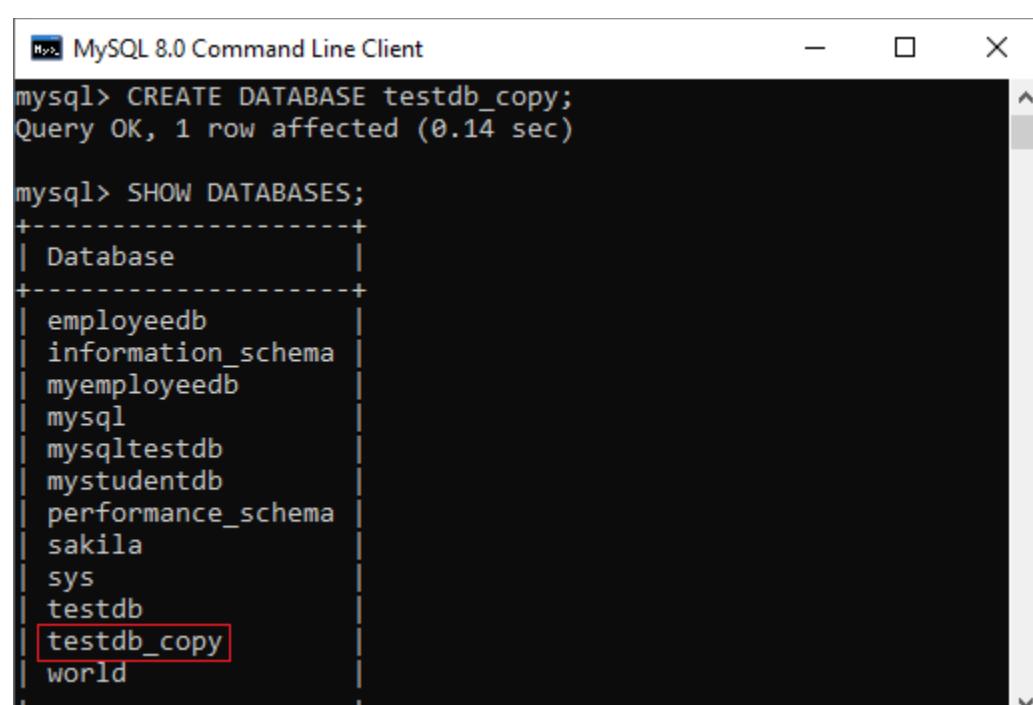
Open the MySQL console and write down the password, if we have set during installation. Now we are ready to create a duplicate database of testdb using the command below:

1. mysql> **CREATE DATABASE** testdb\_copy;

Next, use the SHOW DATABASES statement for verification:

1. mysql> **SHOW DATABASES;**

This command will return all available database in the server where we can see the newly created database in red rectangle box:



```
MySQL 8.0 Command Line Client
mysql> CREATE DATABASE testdb_copy;
Query OK, 1 row affected (0.14 sec)

mysql> SHOW DATABASES;
+-----+
| Database |
+-----+
| employedb
| information_schema
| myemployedb
| mysql
| mysqltestdb
| mystudentdb
| performance_schema
| sakila
| sys
| testdb
| testdb_copy
| world
+-----+
```

The 'testdb\_copy' database is highlighted with a red rectangle.

Now, open a DOS or terminal window to access the MySQL server on the command line. For example, if we have installed the MySQL in the **C folder**, copy the following folder and paste it in our DOS command. Then, press the **Enter** key.

1. C:\Users\javatpoint> CD C:\Program Files\MySQL\MySQL Server 8.0\bin

In the next step, we need to use the mysqldump tool to copy the database objects and data into the SQL file. Suppose we want to dump (copy) the database objects and data of the testdb into an SQL file located at **D:\Database\_backup folder**. To do this, execute the below statement:

1. mysqldump -u root -p testdb > D:\Database\_backup\testdb.sql
2. Enter **password**: \*\*\*\*\*

The above statement instructs mysqldump tool to log in to the MySQL database server using the username and password and then exports the database objects and data of the testdb database to **D:\Database\_backup\testdb.sql**. It is to note that the operator (>) used for exporting the database from one location to another.

In the next step, we need to import the D:\Database\_backup\testdb.sql file into testdb\_copy database. To do this, execute the below statement:

1. mysql -u root -p testdb\_copy < D:\Database\_backup\testdb.sql
2. Enter **password**: \*\*\*\*\*

It is to note that the operator (<) used for importing the database from one location to another.

```
Command Prompt
C:\Users\javatpoint>cd C:\Program Files\MySQL\MySQL Server 8.0\bin
C:\Program Files\MySQL\MySQL Server 8.0\bin>mysqldump -u root -p testdb > D:\Database_backup\testdb.sql
Enter password: *****
C:\Program Files\MySQL\MySQL Server 8.0\bin>mysql -u root -p testdb_copy < D:\Database_backup\testdb.sql
Enter password: *****
```

Finally, we can verify whether the above operation is successful or not by using the **SHOW TABLES** command in the MySQL command-line tool:

1. mysql> SHOW TABLES;

```
MySQL 8.0 Command Line Client
mysql> use testdb_copy;
Database changed
mysql> SHOW TABLES;
+-----+
| Tables_in_testdb_copy |
+-----+
| contact
| courses
| customer
| trainer
+-----+
```

In this output, we can see that all the objects and data from the testdb database to testdb\_copy database have successfully copied.

## Table & Views

# MySQL CREATE TABLE

A table is used to organize data in the form of rows and columns and used for both storing and displaying records in the structure format. It is similar to worksheets in the spreadsheet application. A table creation command requires **three things**:

- o Name of the table
- o Names of fields
- o Definitions for each field

MySQL allows us to create a table into the database mainly in **two ways**:

1. MySQL Command Line Client
2. MySQL Workbench

## MySQL Command Line Client

MySQL allows us to create a table into the database by using the **CREATE TABLE** command. Following is a generic **syntax** for creating a MySQL table in the database.

```
1. CREATE TABLE [IF NOT EXISTS] table_name(  
2.   column_definition1,  
3.   column_definition2,  
4.   .....,  
5.   table_constraints  
6. );
```

#### Parameter Explanation

The parameter descriptions of the above syntax are as follows:

Parameter	Description
database_name	It is the name of a new table. It should be unique in the MySQL database that we have selected. The <b>IF NOT EXIST</b> clause avoids an error when we create a table into the selected database that already exists.
column_definition	It specifies the name of the column along with data types for each column. The columns in table definition are separated by the comma operator. The syntax of column definition is as follows: <b>column_name1 data_type(size) [NULL   NOT NULL]</b>
table_constraints	It specifies the table constraints such as PRIMARY KEY, UNIQUE KEY, FOREIGN KEY, CHECK, etc.

#### Example

Let us understand how to create a table into the database with the help of an example. Open the MySQL console and write down the password, if we have set during installation. Now open the database in which you want to create a table. Here, we are going to create a table name "**employee\_table**" in the database "**employeedb**" using the following statement:

```
1. mysql> CREATE TABLE employee_table(  
2.   id int NOT NULL AUTO_INCREMENT,  
3.   name varchar(45) NOT NULL,  
4.   occupation varchar(35) NOT NULL,  
5.   age int NOT NULL,  
6.   PRIMARY KEY (id)  
7. );
```

#### NOTE:

1. Here, **NOT NULL** is a field attribute, and it is used because we don't want this field to be **NULL**. If we try to create a record with a **NULL** value, then MySQL will raise an error.
2. The field attribute **AUTO\_INCREMENT** specifies MySQL to go ahead and add the next available number to the **id** field. **PRIMARY KEY** is used to define a column's uniqueness. We can use multiple columns separated by a comma to define a primary key.

#### Visual representation of creating a MySQL table:

```
MySQL 8.0 Command Line Client
Database
+-----+
| employeedb
| information_schema
| myemployeedb
| mysql
| mysqltestdb
| mystudentdb
| mytestdb_copy
| performance_schema
| sakila
| sys
| testdb
| world
+-----+
12 rows in set (0.01 sec)

mysql> USE employeedb;
Database changed
mysql> CREATE TABLE employee_table(
    -> id int NOT NULL AUTO_INCREMENT,
    -> name varchar(45) NOT NULL,
    -> occupation varchar(35) NOT NULL,
    -> age int NOT NULL,
    -> PRIMARY KEY (id)
    -> );
Query OK, 0 rows affected (1.47 sec)
```

We need to use the following command to see the newly created table:

1. mysql> SHOW TABLES;

It will look like the below output:

```
MySQL 8.0 Command Line Client
mysql> SHOW TABLES;
+-----+
| Tables_in_employeedb |
+-----+
| employee_table       |
+-----+
1 row in set (0.00 sec)
```

#### See the table structure:

We can use the following command to see the information or structure of the newly created table:

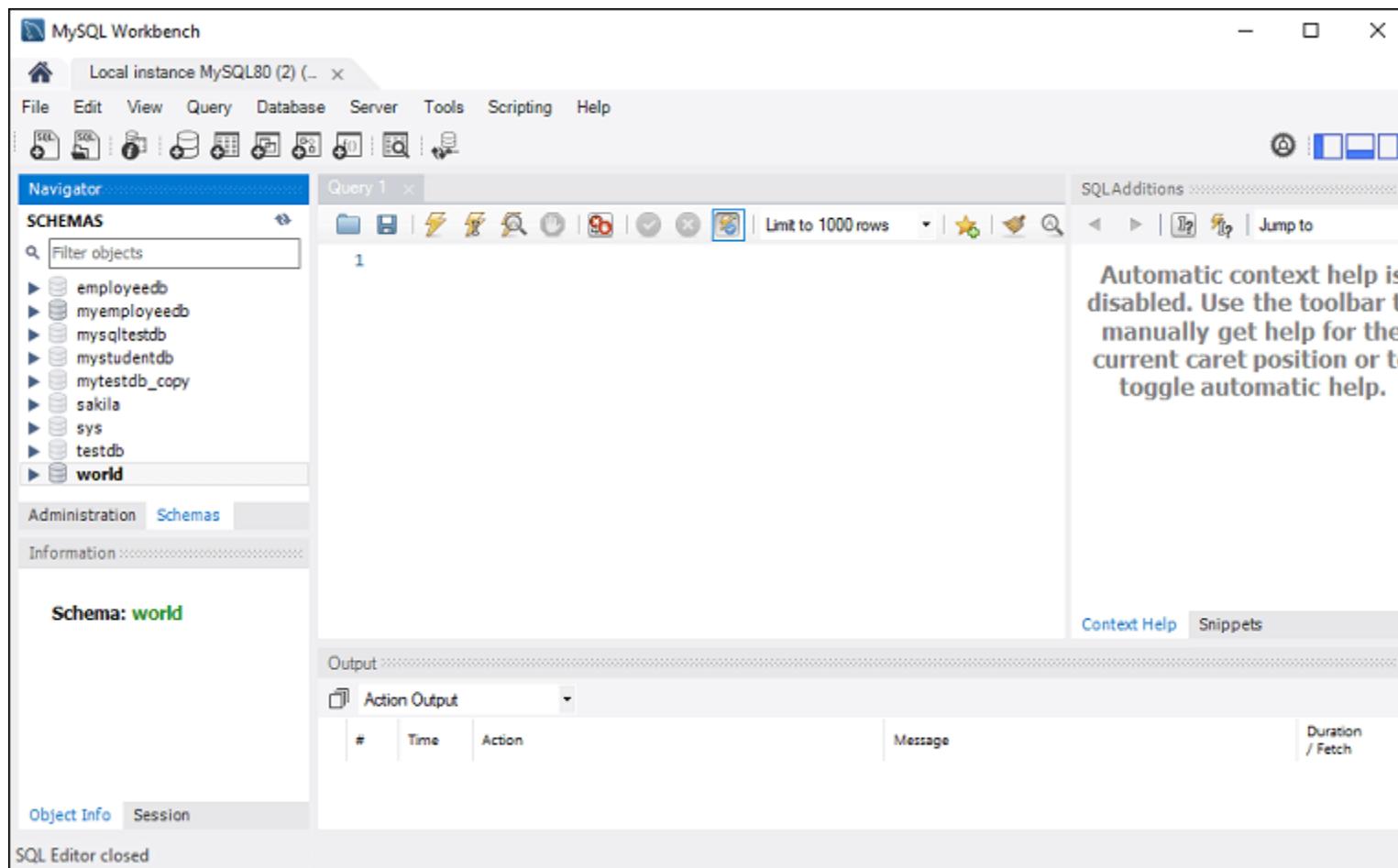
1. mysql> DESCRIBE employee\_table;

It will look like this:

```
MySQL 8.0 Command Line Client
mysql> DESCRIBE employee_table;
+-----+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key  | Default | Extra       |
+-----+-----+-----+-----+-----+-----+
| id    | int    | NO   | PRI  | NULL    | auto_increment |
| name  | varchar(45) | NO  |     | NULL    |              |
| occupation | varchar(35) | NO  |     | NULL    |              |
| age   | int    | NO   |     | NULL    |              |
+-----+-----+-----+-----+-----+-----+
```

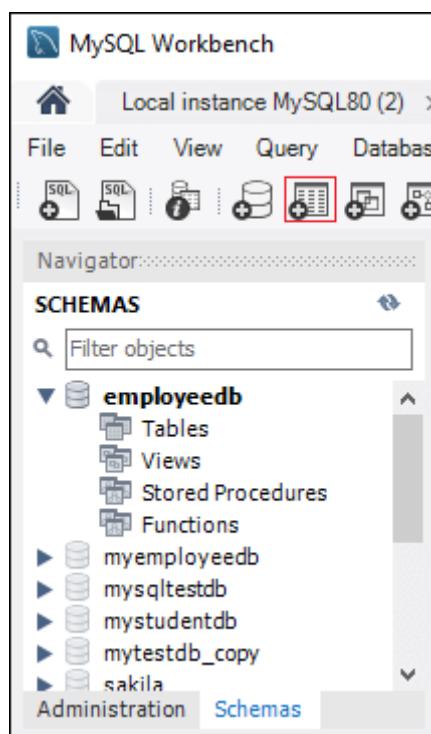
## Create Table Using MySQL Workbench

It is a visual GUI tool used to create databases, tables, indexes, views, and stored procedures quickly and efficiently. To create a new database using this tool, we first need to launch the [MySQL Workbench](#) and log in using the username and password that you want. It will show the following screen:



Now do the following steps for table creation:

1. Go to the Navigation tab and click on the **Schema menu**. Here, we can see all the previously created databases. Now we are ready to select the database in which a table is created.
2. Select the database, double click on it, and we will get the sub-menu under the database. These **sub-menus** are Tables, Views, Functions, and Stored Procedures, as shown in the below screen.



3. Select Tables sub-menu, right-click on it, and select **Create Table** option. We can also click on create a new table icon (shown in red rectangle) to create a table.
4. On the new table screen, we need to fill all the details to create a table. Here, we will enter the table name (**for example**, employee\_table) and use default collation and engine.
5. Click inside the middle window and fill the column details. Here, the column name contains many attributes such as Primary Key(PK), Not Null (NN), Unique Index (UI), Binary(B), Unsigned Data type(UN), Auto Incremental (AI), etc. The following screen explains it more clearly. After filling all the details, click on the **Apply** button.

The screenshot shows the 'employee\_table - Table' configuration window in MySQL Workbench. At the top, the table name is set to 'employee\_table' in schema 'employeedb'. The engine is set to InnoDB. Below this, the table structure is defined with four columns: 'id' (INT, PK, AI), 'name' (VARCHAR(55)), 'occupation' (VARCHAR(35)), and 'age' (INT). The 'age' column is highlighted. On the right, detailed settings for the 'age' column are shown: Data Type (INT), Default (empty), Storage (Virtual), Primary Key (unchecked), Not Null (checked), Unique (unchecked), Binary (unchecked), Unsigned (unchecked), Zero Fill (unchecked), Auto Increment (unchecked), and Generated (unchecked). Below the table structure, tabs for Columns, Indexes, Foreign Keys, Triggers, Partitioning, and Options are visible. At the bottom right are 'Apply' and 'Revert' buttons.

6. As soon as you click on the Apply button, it will open the SQL statement window. Again, click on the Apply button to execute the statement and **Finish** button to save the changes.

The screenshot shows the 'Review SQL Script to be Applied on the Database' dialog. It displays the SQL code for creating the 'employee\_table':

```

CREATE TABLE `employeedb`.`employee_table` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(55) NOT NULL,
  `occupation` VARCHAR(35) NOT NULL,
  `age` INT NOT NULL,
  PRIMARY KEY (`id`)
)
ENGINE = InnoDB;

```

Below the code, there are 'Back', 'Apply', and 'Cancel' buttons.

7. Now, go to the Schema menu and select the database which contains the newly created table, as shown in the screen below.

The screenshot shows the Navigator pane under the 'SCHEMAS' tab. The 'employeedb' schema is selected, and its 'Tables' section shows the 'employee\_table' table, which is currently selected.

## MySQL ALTER Table

MySQL ALTER statement is used when you want to change the name of your table or any table field. It is also used to add or delete an existing column in a table.

The ALTER statement is always used with "ADD", "DROP" and "MODIFY" commands according to the situation.

## 1) ADD a column in the table

**Syntax:**

1. **ALTER TABLE** table\_name
2. **ADD** new\_column\_name column\_definition
3. [ **FIRST | AFTER** column\_name ];

### Parameters

**table\_name:** It specifies the name of the table that you want to modify.

**new\_column\_name:** It specifies the name of the new column that you want to add to the table.

**column\_definition:** It specifies the data type and definition of the column (NULL or NOT NULL, etc).

**FIRST | AFTER column\_name:** It is optional. It tells MySQL where in the table to create the column. If this parameter is not specified, the new column will be added to the end of the table.

### Example:

In this example, we add a new column "cus\_age" in the existing table "cus\_tbl".

Use the following query to do this:

1. **ALTER TABLE** cus\_tbl
2. **ADD** cus\_age **varchar(40)** NOT NULL;

### Output:

The screenshot shows a Windows command-line interface window titled "MySQL 5.5 Command Line Client". The session starts with a "SELECT \* FROM cus\_tbl;" query, which returns three rows of data:

cus_id	cus_firstname	cus_surname
5	Ajeet	Maurya
6	Deepika	Chopra
7	Uimal	Jaiswal

Following this, an "ALTER TABLE" command is run to add a new column "cus\_age" of type "varchar(40)" with the constraint "NOT NULL". The output shows the query was successful, with 3 rows affected and 0 warnings.

```
mysql> SELECT * FROM cus_tbl;
+-----+-----+-----+
| cus_id | cus_firstname | cus_surname |
+-----+-----+-----+
|      5 | Ajeet        | Maurya      |
|      6 | Deepika      | Chopra      |
|      7 | Uimal        | Jaiswal     |
+-----+-----+-----+
3 rows in set (0.00 sec)

mysql> ALTER TABLE cus_tbl
    -> ADD cus_age varchar(40) NOT NULL;
Query OK, 3 rows affected (0.48 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql>
```

**See the recently added column:**

1. **SELECT\* FROM** cus\_tbl;

### Output:

```
MySQL 5.5 Command Line Client
mysql> ALTER TABLE cus_tbl
    -> ADD cus_age varchar(40) NOT NULL;
Query OK, 3 rows affected (0.48 sec)
Records: 3  Duplicates: 0  Warnings: 0
mysql> SELECT*FROM cus_tbl;
+ cus_id + cus_firstname + cus_surname + cus_age +
+ 5       + Ajeet        + Maurya      +
+ 6       + Deepika      + Chopra      +
+ 7       + Uimal        + Jaiswal     +
3 rows in set (0.00 sec)
mysql>
```

## 2) Add multiple columns in the table

Syntax:

1. **ALTER TABLE** table\_name
2. **ADD** new\_column\_name column\_definition
3. [ **FIRST** | **AFTER** column\_name ],
4. **ADD** new\_column\_name column\_definition
5. [ **FIRST** | **AFTER** column\_name ],
6. ...
7. ;

Example:

In this example, we add two new columns "cus\_address", and cus\_salary in the existing table "cus\_tbl". cus\_address is added after cus\_surname column and cus\_salary is added after cus\_age column.

Use the following query to do this:

1. **ALTER TABLE** cus\_tbl
2. **ADD** cus\_address **varchar**(100) NOT NULL
3. **AFTER** cus\_surname,
4. **ADD** cus\_salary **int**(100) NOT NULL
5. **AFTER** cus\_age ;

```
MySQL 5.5 Command Line Client
mysql> ALTER TABLE cus_tbl
    -> ADD cus_address varchar(100) NOT NULL
    -> AFTER cus_surname,
    -> ADD cus_salary int(100) NOT NULL
    -> AFTER cus_age ;
Query OK, 3 rows affected (0.34 sec)
Records: 3  Duplicates: 0  Warnings: 0
mysql>
```

See the recently added columns:

1. **SELECT\* FROM** cus\_tbl;

The screenshot shows the MySQL 5.5 Command Line Client interface. The command entered is:

```
mysql> ALTER TABLE cus_tbl
-> ADD cus_address varchar(100) NOT NULL
-> AFTER cus_surname,
-> ADD cus_salary int(100) NOT NULL
-> AFTER cus_age ;
Query OK, 3 rows affected <0.34 sec>
Records: 3  Duplicates: 0  Warnings: 0
```

Following this, a SELECT query is run:

```
mysql> SELECT * FROM cus_tbl;
```

cus_id	cus_firstname	cus_surname	cus_address	cus_age	cus_salary
5	Ajeet	Maurya			0
6	Deepika	Chopra			0
7	Vimal	Jaiswal			0

3 rows in set <0.02 sec>

```
mysql> _
```

### 3) MODIFY column in the table

The MODIFY command is used to change the column definition of the table.

**Syntax:**

1. **ALTER TABLE** table\_name
2. **MODIFY** column\_name column\_definition
3. [ **FIRST** | **AFTER** column\_name ];

**Example:**

In this example, we modify the column cus\_surname to be a data type of varchar(50) and force the column to allow NULL values.

**Use the following query to do this:**

1. **ALTER TABLE** cus\_tbl
2. **MODIFY** cus\_surname **varchar**(50) **NULL**;

The screenshot shows the MySQL 5.5 Command Line Client interface. The command entered is:

```
mysql> ALTER TABLE cus_tbl
-> MODIFY cus_surname varchar(50) NULL;
Query OK, 3 rows affected <0.33 sec>
Records: 3  Duplicates: 0  Warnings: 0
```

```
mysql> _
```

**See the table structure:**

```

MySQL 5.5 Command Line Client
Records: 3  Duplicates: 0  Warnings: 0
mysql> SELECT* FROM cus_tbl;
+ cus_id | cus_firstname | cus_surname | cus_address | cus_age | cus_salary |
|      5 | Ajeet         | Maurya     |           |          |       0 |
|      6 | Deepika       | Chopra     |           |          |       0 |
|      7 | Vimal         | Jaiswal    |           |          |       0 |
+-----+
3 rows in set <0.00 sec>

mysql> DESCRIBE cus_tbl;
+ Field   | Type    | Null | Key | Default | Extra |
| cus_id  | int(11) | NO   | PRI | NULL    | auto_increment |
| cus_firstname | varchar(100) | NO | NULL | NULL |
| cus_surname | varchar(50) | YES | NULL | NULL |
| cus_address | varchar(100) | NO | NULL | NULL |
| cus_age | varchar(40) | NO | NULL | NULL |
| cus_salary | int(100) | NO | NULL | NULL |
+-----+
6 rows in set <0.01 sec>

```

## 4) DROP column in table

Syntax:

1. **ALTER TABLE** table\_name
2. **DROP COLUMN** column\_name;

Let's take an example to drop the column name "cus\_address" from the table "cus\_tbl".

Use the following query to do this:

1. **ALTER TABLE** cus\_tbl
2. **DROP COLUMN** cus\_address;

Output:

```

MySQL 5.5 Command Line Client
mysql> SELECT*FROMM cus_tbl;
+ cus_id | cus_firstname | cus_surname | cus_address | cus_age | cus_salary |
|      5 | Ajeet         | Maurya     |           |          |       0 |
|      6 | Deepika       | Chopra     |           |          |       0 |
|      7 | Vimal         | Jaiswal    |           |          |       0 |
+-----+
3 rows in set <0.00 sec>

mysql> ALTER TABLE cus_tbl
    -> DROP COLUMN cus_address;
Query OK, 3 rows affected <0.37 sec>
Records: 3  Duplicates: 0  Warnings: 0
mysql> _

```

See the table structure:

```

MySQL 5.5 Command Line Client
3 rows in set <0.00 sec>

mysql> ALTER TABLE cus_tbl
    -> DROP COLUMN cus_address;
Query OK, 3 rows affected <0.37 sec>
Records: 3  Duplicates: 0  Warnings: 0
mysql> SELECT* FROMM cus_tbl;
+ cus_id | cus_firstname | cus_surname | cus_age | cus_salary |
|      5 | Ajeet         | Maurya     |          |       0 |
|      6 | Deepika       | Chopra     |          |       0 |
|      7 | Vimal         | Jaiswal    |          |       0 |
+-----+
3 rows in set <0.00 sec>
mysql> _

```

## 5) RENAME column in table

Syntax:

1. **ALTER TABLE** table\_name
2. CHANGE **COLUMN** old\_name new\_name
3. column\_definition
4. [ **FIRST** | **AFTER** column\_name ]

Example:

In this example, we will change the column name "cus\_surname" to "cus\_title".

Use the following query to do this:

1. **ALTER TABLE** cus\_tbl
2. CHANGE **COLUMN** cus\_surname cus\_title
3. **varchar(20)** NOT NULL;

Output:

The screenshot shows the MySQL 5.5 Command Line Client window. The command line displays the following SQL queries and their results:

```
mysql> SELECT * FROM cus_tbl;
+-----+-----+-----+-----+
| cus_id | cus_firstname | cus_surname | cus_age | cus_salary |
+-----+-----+-----+-----+
| 5     | Ajeet        | Maurya      |          | 0          |
| 6     | Deepika      | Chopra      |          | 0          |
| 7     | Uimal         | Jaiswal     |          | 0          |
+-----+-----+-----+-----+
3 rows in set (0.00 sec)

mysql> ALTER TABLE cus_tbl
    -> CHANGE COLUMN cus_surname cus_title
    -> varchar(20) NOT NULL;
Query OK, 3 rows affected (0.25 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> _
```

## 6) RENAME table

Syntax:

1. **ALTER TABLE** table\_name
2. **RENAME TO** new\_table\_name;

Example:

In this example, the table name cus\_tbl is renamed as cus\_table.

1. **ALTER TABLE** cus\_tbl
2. **RENAME TO** cus\_table;

Output:

The screenshot shows the MySQL 5.5 Command Line Client window. The command `SELECT * FROM cus_tbl;` is run, displaying three rows of data. Then, the command `ALTER TABLE cus_tbl RENAME TO cus_table;` is run, which is successful as indicated by the message "Query OK, 0 rows affected (0.07 sec)". Finally, the prompt `mysql>` is shown.

```

mysql> SELECT * FROM cus_tbl;
+ cus_id + cus_firstname + cus_title + cus_age + cus_salary +
| 5 | Ajeet | Maurya |          | 0 |
| 6 | Deepika | Chopra |          | 0 |
| 7 | Vimal | Jaiswal |          | 0 |
+-----+
3 rows in set (0.00 sec)

mysql> ALTER TABLE cus_tbl
-> RENAME TO cus_table;
Query OK, 0 rows affected (0.07 sec)

mysql>

```

**See the renamed table:**

The screenshot shows the MySQL 5.5 Command Line Client window. The command `SHOW tables;` is run, displaying the results in a table format. It shows two tables: `Tables_in_customers` and `cus_table`. The prompt `mysql>` is shown at the bottom.

```

mysql> SELECT * FROM cus_tbl;
+-----+
3 rows in set (0.00 sec)

mysql> ALTER TABLE cus_tbl
-> RENAME TO cus_table;
Query OK, 0 rows affected (0.07 sec)

mysql> SHOW tables;
+-----+
| Tables_in_customers |
| cus_table           |
+-----+
1 row in set (0.00 sec)

mysql>

```

## MySQL Show>List Tables

The show or list table is very important when we have many databases that contain various tables. Sometimes the table names are the same in many databases; in that case, this query is very useful. We can get the number of table information of a database using the following statement:

1. `mysql> SHOW TABLES;`

The following steps are necessary to get the list of tables:

**Step 1:** Open the MySQL Command Line Client that appeared with a **mysql> prompt**. Next, **log in** to the MySQL database server using the **password** that you have created during the installation of MySQL. Now, you are connected to the MySQL server, where you can execute all the SQL statements.

**Step 2:** Next, choose the specific database by using the command below:

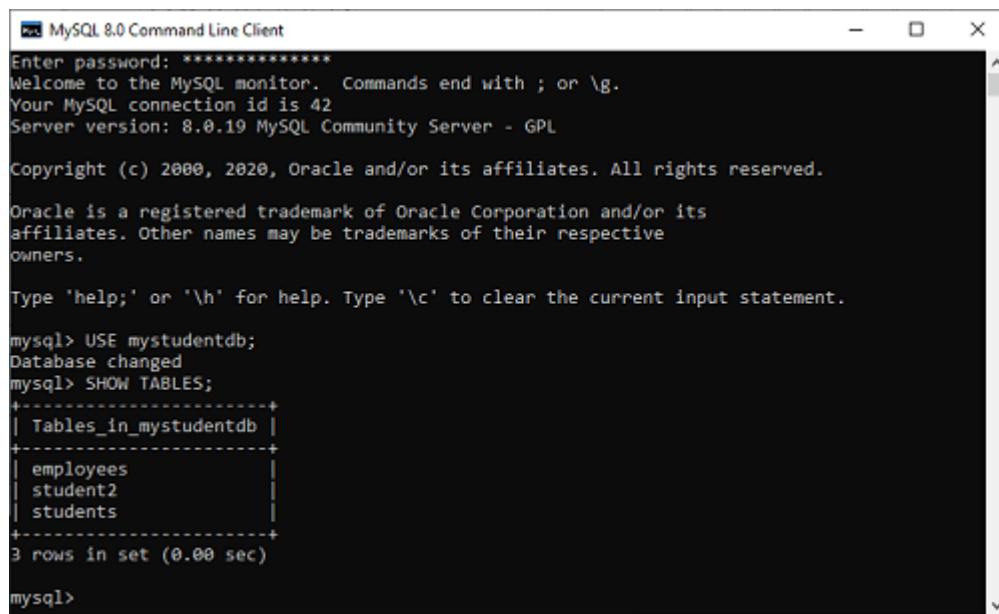
1. `mysql> USE database_name;`

**Step 3:** Finally, execute the `SHOW TABLES` command.

Let us understand it with the example given below. Suppose we have a database name "**mystudentdb**" that contains many tables. Then execute the below statement to list the table it contains:

1. `mysql> USE mystudentdb;`
2. `mysql> SHOW TABLES;`

The following output explains it more clearly:



```
MySQL 8.0 Command Line Client
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 42
Server version: 8.0.19 MySQL Community Server - GPL

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

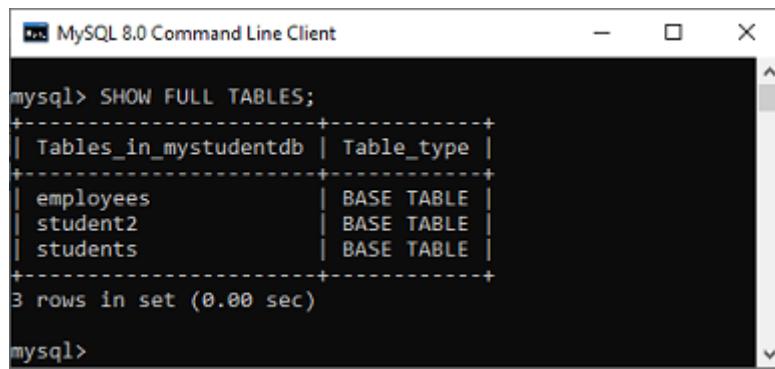
mysql> USE mystudentdb;
Database changed
mysql> SHOW TABLES;
+-----+
| Tables_in_mystudentdb |
+-----+
| employees
| student2
| students
+-----+
3 rows in set (0.00 sec)

mysql>
```

We can also use the **FULL modifier** with the SHOW TABLES query to get the type of table (Base or View) that appears in a second output column.

1. mysql> SHOW **FULL** TABLES;

This statement will give the following output:



```
MySQL 8.0 Command Line Client
- □ X
mysql> SHOW FULL TABLES;
+-----+-----+
| Tables_in_mystudentdb | Table_type |
+-----+-----+
| employees           | BASE TABLE |
| student2            | BASE TABLE |
| students             | BASE TABLE |
+-----+-----+
3 rows in set (0.00 sec)

mysql>
```

If we want to show or list the table name from different databases or database to which you are not connected without switching, MySQL allows us to use the FROM or IN clause followed by the database name. The following statement explains it more clearly:

1. mysql> SHOW TABLES IN database\_name;

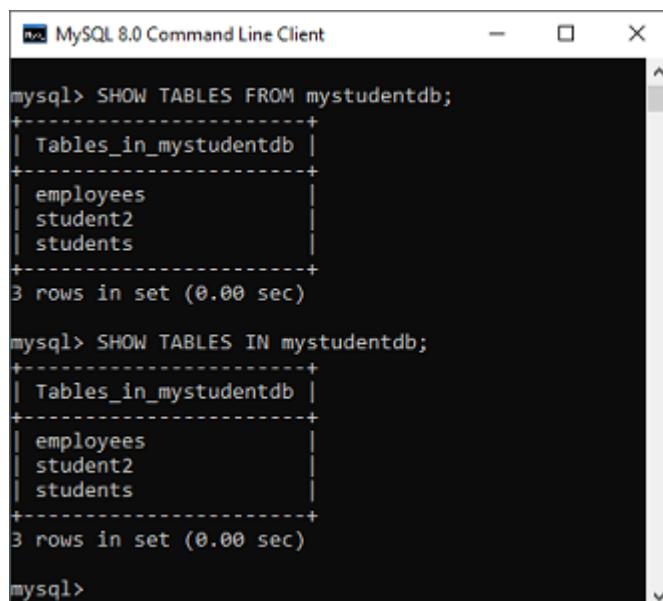
The above statement can also be written as:

1. mysql> SHOW TABLES **FROM** database\_name;

When we execute the below statements, we will get the same result:

1. mysql> SHOW TABLES **FROM** mystudentdb;
2. OR,
3. mysql> SHOW TABLES IN mystudentdb;

#### Output:



```
MySQL 8.0 Command Line Client
- □ X
mysql> SHOW TABLES FROM mystudentdb;
+-----+
| Tables_in_mystudentdb |
+-----+
| employees
| student2
| students
+-----+
3 rows in set (0.00 sec)

mysql> SHOW TABLES IN mystudentdb;
+-----+
| Tables_in_mystudentdb |
+-----+
| employees
| student2
| students
+-----+
3 rows in set (0.00 sec)

mysql>
```

## Show Tables Using Pattern Matching

Show Tables command in MySQL also provides an option that allows us to **filter** the returned table using different pattern matching with LIKE and WHERE clause.

## Syntax

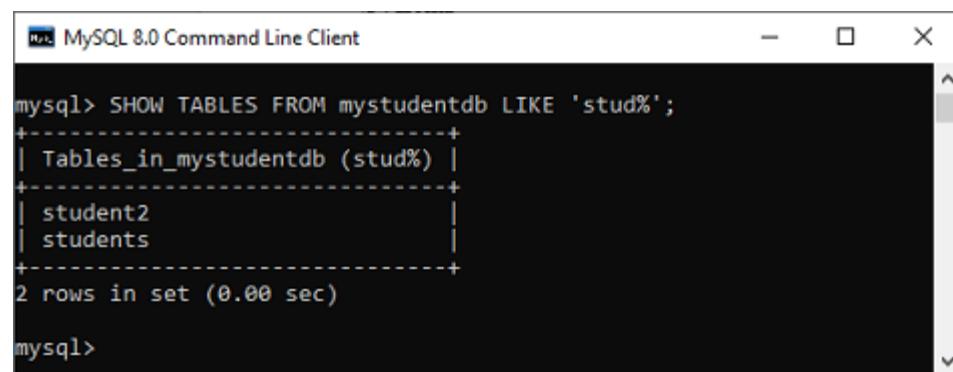
The following are the syntax to use pattern matching with show table command:

1. mysql> SHOW TABLES **LIKE** pattern;
2. OR,
3. mysql> SHOW TABLES **WHERE** expression;

We can understand it with the example given below where percent (%) sign assumes zero, one, or multiple characters:

1. mysql> SHOW TABLES **FROM** mystudentdb **LIKE** "stud%";

The above statement will give the following output:



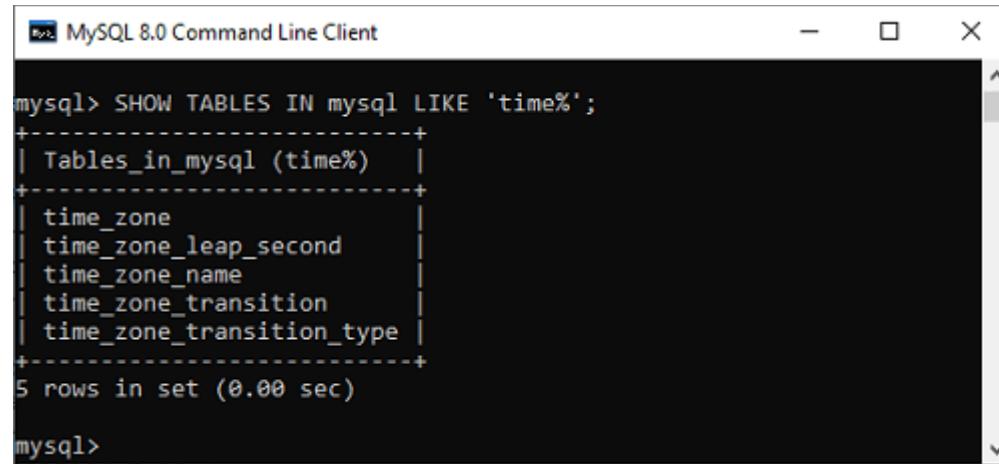
```
MySQL 8.0 Command Line Client
mysql> SHOW TABLES FROM mystudentdb LIKE 'stud%';
+-----+
| Tables_in_mystudentdb (stud%) |
+-----+
| student2
| students
+-----+
2 rows in set (0.00 sec)

mysql>
```

Let us see another statement that returned the table names starting with "**time**":

1. mysql> SHOW TABLES **IN** mysql **LIKE** "time%";

The above query will give the following output:



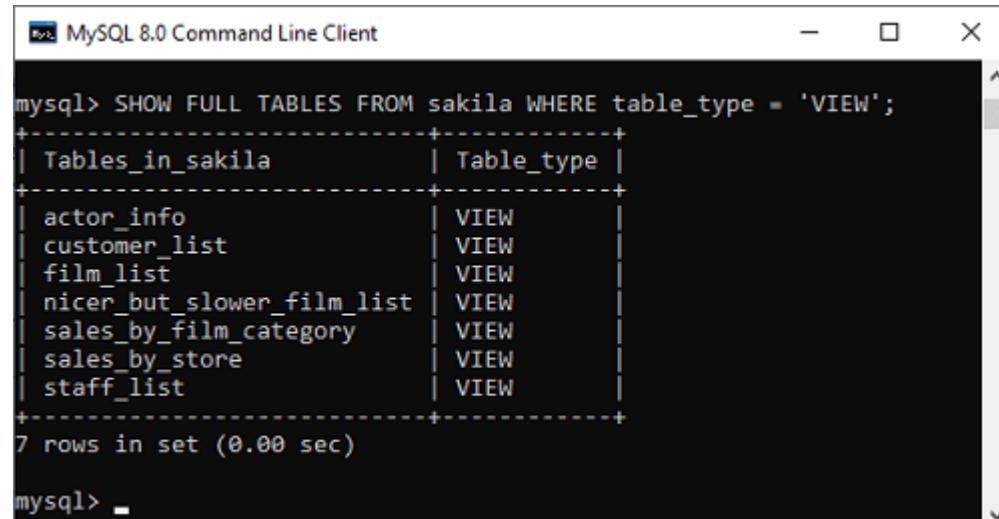
```
MySQL 8.0 Command Line Client
mysql> SHOW TABLES IN mysql LIKE 'time%';
+-----+
| Tables_in_mysql (time%) |
+-----+
| time_zone
| time_zone_leap_second
| time_zone_name
| time_zone_transition
| time_zone_transition_type
+-----+
5 rows in set (0.00 sec)

mysql>
```

Now, we are going to see how we can use the **WHERE** clause with the SHOW TABLES command to list different types of tables (either Base or View type) in the selected database:

1. mysql> SHOW TABLES **FROM** sakila **WHERE** table\_type= "**VIEW**";

This statement gives the below output:



```
MySQL 8.0 Command Line Client
mysql> SHOW FULL TABLES FROM sakila WHERE table_type = 'VIEW';
+-----+-----+
| Tables_in_sakila | Table_type |
+-----+-----+
| actor_info      | VIEW      |
| customer_list   | VIEW      |
| film_list       | VIEW      |
| nicer_but_slower_film_list | VIEW |
| sales_by_film_category | VIEW |
| sales_by_store  | VIEW      |
| staff_list      | VIEW      |
+-----+-----+
7 rows in set (0.00 sec)

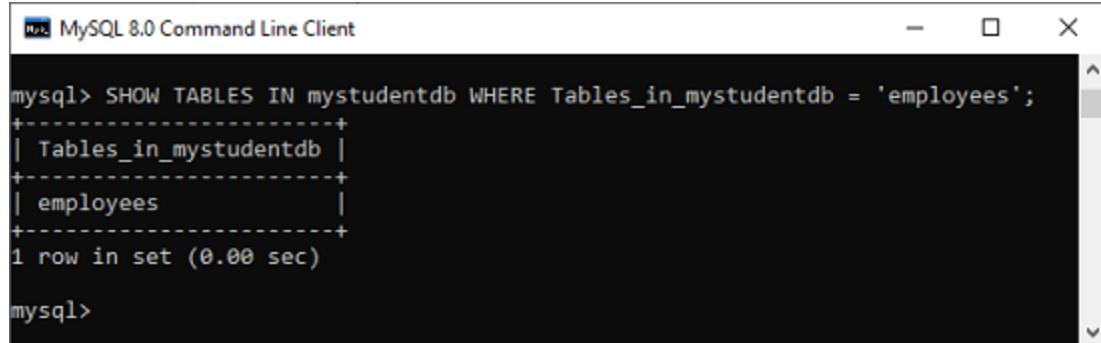
mysql>
```

It is noted that if MySQL does not provide the privileges for accessing a Base table or view, then we cannot get the tables in the result set of the SHOW TABLES command.

Here, we can also see another example of Show Tables statement with the WHERE clause:

1. mysql> SHOW TABLES IN mystudentdb **WHERE** Tables\_in\_mystudentdb= "employees";

It will give the following output:



The screenshot shows the MySQL 8.0 Command Line Client window. The command entered is 'SHOW TABLES IN mystudentdb WHERE Tables\_in\_mystudentdb = 'employees';'. The output shows one row: 'employees'. The message '1 row in set (0.00 sec)' is displayed at the bottom.

## MySQL Rename Table

Sometimes our table name is non-meaningful, so it is required to rename or change the name of the table. MySQL provides a useful syntax that can rename one or more tables in the current database.

### Syntax

The following are the syntax used to change the name of the table:

1. mysql> RENAME old\_table **TO** new\_table;

Here, we have to make sure that **new\_table\_name** must not exist, and **old\_table\_name** should be present in the database. Otherwise, it will throw an error message. It is to ensure that the table is not locked as well as there are no active transactions before executing this statement.

**NOTE:** If we use the **RENAME TABLE** statement, it is required to have **ALTER** and **DROP TABLE** privileges to the existing table. Also, this statement cannot change the name of a temporary table.

We can also use the MySQL **RENAME TABLE** statement to change more than one table name with a single statement, as shown below:

1. RENAME **TABLE** old\_tab1 **TO** new\_tab1,
2.       old\_tab2 **TO** new\_tab2, old\_tab3 **TO** new\_tab3;

From the **MySQL 8.0.13** version, we can change the old table name locked with a LOCK statement and also uses the WRITE LOCK clause. For example, following are the valid statement:

1. mysql> LOCK **TABLE** old\_tab\_name1 **WRITE**;
2. RENAME **TABLE** old\_tab\_name1 **TO** new\_tab\_name1,
3.       new\_tab\_name1 **TO** new\_tab\_name2;

Following statement are not permitted:

1. mysql> LOCK **TABLE** old\_tab\_name1 **READ**;
2. RENAME **TABLE** old\_tab\_name1 **TO** new\_tab\_name1,
3.       new\_tab\_name1 **TO** new\_tab\_name2;

Before MySQL 8.0.13 version, we cannot change the table name that was locked with the LOCK TABLE statement.

MySQL also use the RENAME TABLE statement for moving a table from one database to other database, which is show below:

1. mysql> RENAME **TABLE** current\_db.table1\_name **TO** other\_db.table1\_name;

## MySQL RENAME TABLE Example

Let us understand how the RENAME TABLE statement works in MySQL through the various examples. Suppose we have a table named **EMPLOYEE**, and due to some reason, there is a need to change it into the table named **CUSTOMER**.

**Table Name:** employee

```
MySQL 8.0 Command Line Client

mysql> SELECT * FROM employee;
+----+-----+-----+-----+-----+
| id | name | occupation | working_date | working_hours |
+----+-----+-----+-----+-----+
| 1  | Joseph | Business   | 2020-04-10  | 10          |
| 2  | Stephen | Doctor     | 2020-04-10  | 15          |
| 3  | Mark   | Engineer   | 2020-04-10  | 12          |
| 4  | Peter   | Teacher    | 2020-04-10  | 9           |
| 1  | Joseph | Business   | 2020-04-12  | 10          |
| 2  | Stephen | Doctor     | 2020-04-12  | 15          |
| 4  | Peter   | Teacher    | 2020-04-12  | 9           |
| 3  | Mark   | Engineer   | 2020-04-12  | 12          |
| 1  | Joseph | Business   | 2020-04-14  | 10          |
| 4  | Peter   | Teacher    | 2020-04-14  | 9           |
+----+-----+-----+-----+-----+
10 rows in set (0.00 sec)
```

Next, execute the following syntax to change the table name:

1. mysql> RENAME employee TO customer;

#### Output

We will see that the table named "employee" will be changed into a new table name "customer":

```
MySQL 8.0 Command Line Client

mysql> RENAME TABLE employee TO customer;
Query OK, 0 rows affected (0.79 sec)

mysql> SELECT * FROM employee;
ERROR 1146 (42S02): Table 'myemployeedb.employee' doesn't exist
mysql>
mysql> SELECT * FROM customer;
+----+-----+-----+-----+-----+
| id | name | occupation | working_date | working_hours |
+----+-----+-----+-----+-----+
| 1  | Joseph | Business   | 2020-04-10  | 10          |
| 2  | Stephen | Doctor     | 2020-04-10  | 15          |
| 3  | Mark   | Engineer   | 2020-04-10  | 12          |
| 4  | Peter   | Teacher    | 2020-04-10  | 9           |
| 1  | Joseph | Business   | 2020-04-12  | 10          |
| 2  | Stephen | Doctor     | 2020-04-12  | 15          |
| 4  | Peter   | Teacher    | 2020-04-12  | 9           |
| 3  | Mark   | Engineer   | 2020-04-12  | 12          |
| 1  | Joseph | Business   | 2020-04-14  | 10          |
| 4  | Peter   | Teacher    | 2020-04-14  | 9           |
+----+-----+-----+-----+-----+
10 rows in set (0.13 sec)
```

In the above output, we can see that if we use the table name employee after executing a RENAME TABLE statement, it will throw an error message.

## How to RENAME Multiple Tables

RENAME TABLE statement in MySQL also allows us to change more than one table name within a single statement. See the below statement:

Suppose our database "**myemployeedb**" having the following tables:

```
MySQL 8.0 Command Line Client

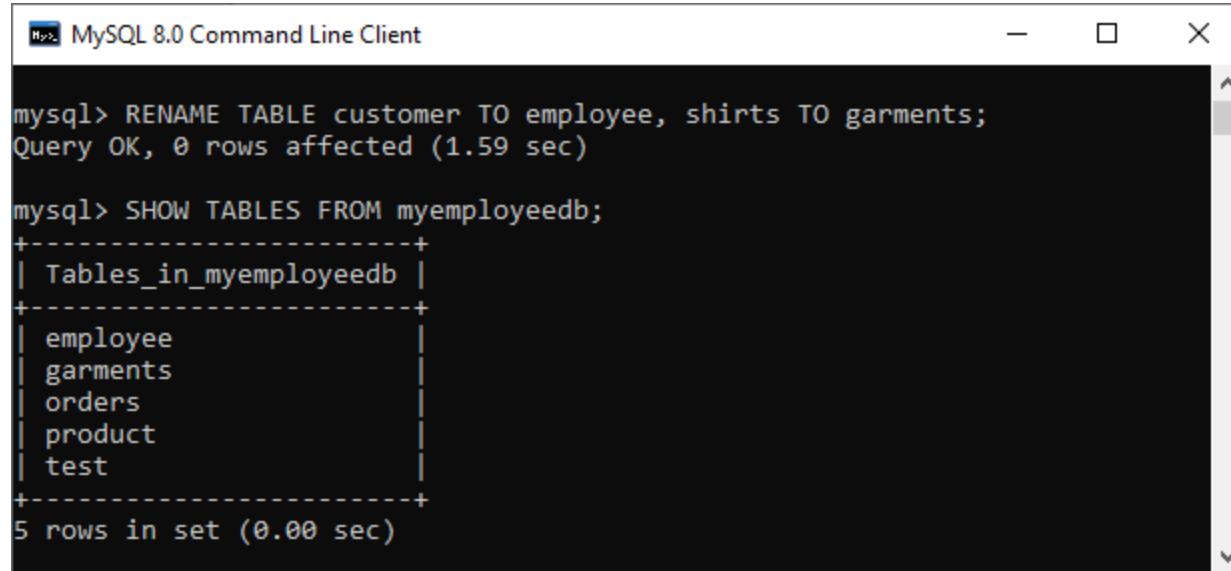
mysql> SHOW TABLES FROM myemployeedb;
+-----+
| Tables_in_myemployeedb |
+-----+
| customer
| orders
| product
| shirts
| test
+-----+
5 rows in set (0.00 sec)
```

If we want to change the table name customer into employee and table name shirts into garments, execute the following statement:

1. mysql> RENAME TABLE customer TO employee, shirts TO garments;

#### Output

We can see that the table name customer into employee and table name shirts into garments have successfully renamed.



The screenshot shows the MySQL 8.0 Command Line Client interface. The command entered was 'mysql> RENAME TABLE customer TO employee, shirts TO garments;'. The output shows 'Query OK, 0 rows affected (1.59 sec)'. Then, 'mysql> SHOW TABLES FROM myemployeedb;' is run, and the results are displayed in a table:

Tables_in_myemployeedb
employee
garments
orders
product
test

5 rows in set (0.00 sec)

## Rename table using ALTER statement

The ALTER TABLE statement can also be used to rename the existing table in the current database. The following are the syntax of the ALTER TABLE statement:

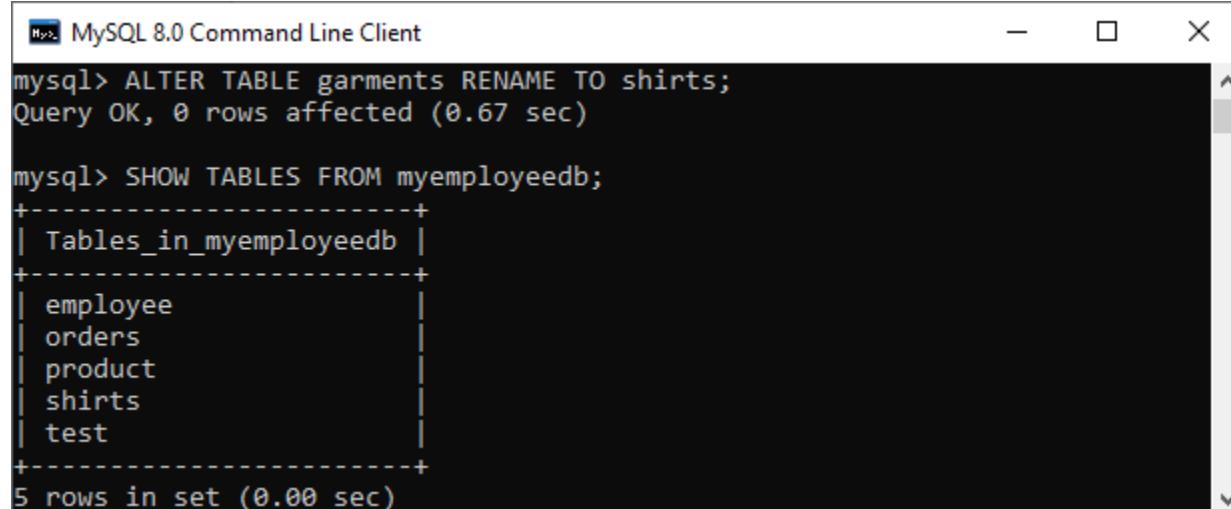
1. **ALTER TABLE** old\_table\_name RENAME TO new\_table\_name;

See the following query that changes the existing table name garments into new table name shirts:

1. mysql> **ALTER TABLE** garments RENAME TO shirts;

#### Output

Here, we can see that the table name garments renamed into table name shirts.



The screenshot shows the MySQL 8.0 Command Line Client interface. The command entered was 'mysql> ALTER TABLE garments RENAME TO shirts;'. The output shows 'Query OK, 0 rows affected (0.67 sec)'. Then, 'mysql> SHOW TABLES FROM myemployeedb;' is run, and the results are displayed in a table:

Tables_in_myemployeedb
employee
orders
product
shirts
test

5 rows in set (0.00 sec)

## How to RENAME Temporary Table

A temporary table allows us to keep temporary data, which is visible and accessible in the current session only. So, first, we need to create a temporary table using the following statement:

1. mysql> **CREATE TEMPORARY TABLE** Students( **name VARCHAR**(40) NOT NULL, total\_marks **DECIMAL**(12,2) NOT NULL **DEFAULT** 0.00, total\_subjects **INT UNSIGNED** NOT NULL **DEFAULT** 0);

Next, insert values into this table:

1. mysql> **INSERT INTO** Students(**name**, total\_marks, total\_subjects) **VALUES** ('Joseph', 150.75, 2), ('Peter', 180.75, 2);

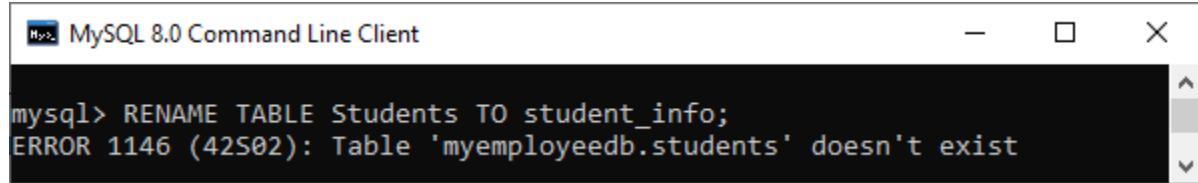
Next, run the show table command to check the temporary table:

1. mysql> **SELECT \* FROM** Students;

Now, run the following command to change the name of the temporary table:

1. mysql> RENAME TABLE Students TO student\_info;

It will throw an error message, as shown below:

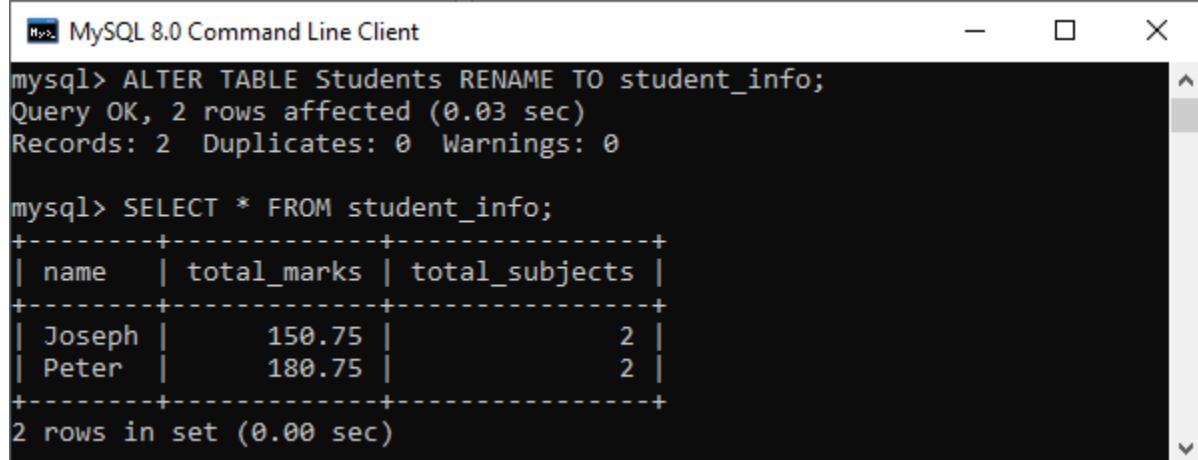


The screenshot shows a terminal window titled "MySQL 8.0 Command Line Client". The command entered is "mysql> RENAME TABLE Students TO student\_info;". The response is "ERROR 1146 (42S02): Table 'myemployeedb.students' doesn't exist".

Thus, MySQL allows ALTER table statement to rename the temporary table:

1. mysql> ALTER TABLE Students RENAME TO student\_info;

#### Output



The screenshot shows a terminal window titled "MySQL 8.0 Command Line Client". The command entered is "mysql> ALTER TABLE Students RENAME TO student\_info;". The response shows "Query OK, 2 rows affected (0.03 sec)". Below this, the command "mysql> SELECT \* FROM student\_info;" is run, displaying the following data:

name	total_marks	total_subjects
Joseph	150.75	2
Peter	180.75	2

There are 2 rows in the set.

## MySQL TRUNCATE Table

The TRUNCATE statement in MySQL removes the complete data without removing its structure. It is a part of **DDL or data definition language command**. Generally, we use this command when we want to delete an entire data from a table without removing the table structure.

The TRUNCATE command works the same as a DELETE command without using a **WHERE clause** that deletes complete rows from a table. However, the TRUNCATE command is more efficient as compared to the **DELETE** command because it removes and recreates the table instead of deleting single records one at a time. Since this command internally drops the table and recreates it, the number of rows affected by the truncate statement is zero, unlike the delete statement that returns the number of deleted rows.

This command does not maintain the transaction log during the execution. It deallocates the data **pages instead of rows** and makes an entry for the deallocating pages instead of rows in transaction logs. This command also locks the pages instead of rows; thus, it requires fewer locks and resources.

The following points must be considered while using the TRUNCATE command:

- We cannot use the **WHERE** clause with this command so that filtering of records is not possible.
- We **cannot rollback the deleted data** after executing this command because the log is not maintained while performing this operation.
- We cannot use the truncate statement when a table is referenced by a **foreign key** or participates in an **indexed view**.
- The TRUNCATE command doesn't fire **DELETE triggers** associated with the table that is being truncated because it does not operate on individual rows.

## Syntax

The following syntax explains the TRUNCATE command to remove data from the table:

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

In this syntax, first, we will specify the **table name** which data we are going to remove. The TABLE keyword in the syntax is not mandatory. But it's a good practice to use it to distinguish between the **TRUNCATE()** function and the **TRUNCATE TABLE statement**.

## MySQL Truncate Table Example

Let us demonstrate how we can truncate the table with the help of an example. First, we are going to create a table named "customer" using the below statement:

```
1. CREATE TABLE customer (
2.   Id int PRIMARY KEY NOT NULL,
3.   Name varchar(45) NOT NULL,
4.   Product varchar(45) DEFAULT NULL,
5.   Country varchar(25) DEFAULT NULL,
6.   Year int NOT NULL
7. );
```

Next, we will add values to this table using the below statement:

```
1. INSERT INTO customer ( Id, Name, Product, Country, Year)
2. VALUES (1, 'Stephen', 'Computer', 'USA', 2015),
3. (2, 'Joseph', 'Laptop', 'India', 2016),
4. (3, 'John', 'TV', 'USA', 2016),
5. (4, 'Donald', 'Laptop', 'England', 2015),
6. (5, 'Joseph', 'Mobile', 'India', 2015),
7. (6, 'Peter', 'Mouse', 'England', 2016);
```

Now, verify the table by executing the **SELECT statement** whether the records inserted or not:

```
1. mysql> SELECT * FROM customer;
```

We will get the output, as shown below:



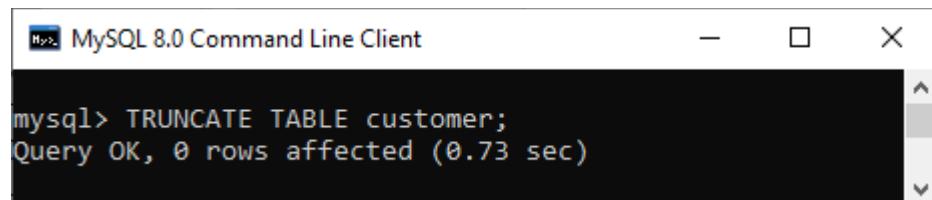
The screenshot shows the MySQL 8.0 Command Line Client window. The command entered is 'SELECT \* FROM customer;'. The result set is displayed as a table with columns: Id, Name, Product, Country, and Year. The data consists of six rows with the following values:

Id	Name	Product	Country	Year
1	Stephen	Computer	USA	2015
2	Joseph	Laptop	India	2016
3	John	TV	USA	2016
4	Donald	Laptop	England	2015
5	Joseph	Mobile	India	2015
6	Peter	Mouse	England	2016

Now, execute the following statement that truncates the table customer using the TRUNCATE syntax discussed above:

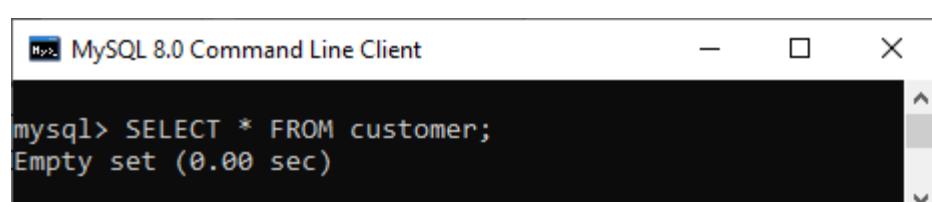
```
1. mysql> TRUNCATE TABLE customer;
```

After the successful execution, we will get the following output:



The screenshot shows the MySQL 8.0 Command Line Client window. The command entered is 'TRUNCATE TABLE customer;'. The response indicates 'Query OK, 0 rows affected (0.73 sec)'.

As we can see, this query returns **0 rows are affected** even if all the table records are deleted. We can verify the deletion of the data by executing the SELECT statement again. This command gives the following output that shows none of the records present in the table:



The screenshot shows the MySQL 8.0 Command Line Client window. The command entered is 'SELECT \* FROM customer;'. The response indicates 'Empty set (0.00 sec)'.

## How to Truncate Table with Foreign key?

If we perform the TRUNCATE operation for the table that uses a foreign key constraint, we will get the following error:

```
1. ERROR 1217 (23000): Cannot delete or update a parent row: a foreign key constraint fails
```

In that case, we need to log into the MySQL server and **disable foreign key** checks before executing the TRUNCATE statement as below:

1. **SET FOREIGN\_KEY\_CHECKS=0;**

Now, we are able to truncate tables. After execution, **re-enable foreign key** checks as given below:

1. **SET FOREIGN\_KEY\_CHECKS=1;**

## How to truncate all tables in MySQL?

The TRUNCATE statement in MySQL will delete only one table at a time. If we want to delete more than one table, we need to execute the separate TRUNCATE statement. The below example shows how to truncate multiple tables in MySQL:

1. **TRUNCATE TABLE** table\_name1;
2. **TRUNCATE TABLE** table\_name2;
3. **TRUNCATE TABLE** table\_name3;

We can also use the below SQL query that generates several TRUNCATE TABLE commands at once using the table names in our database:

1. **SELECT Concat('TRUNCATE TABLE ', TABLE\_NAME)**
2. **FROM INFORMATION\_SCHEMA.TABLES**
3. **WHERE table\_schema = 'database\_name';**

## MySQL DESCRIBE TABLE

DESCRIBE means to show the information in detail. Since we have tables in MySQL, so we will use the **DESCRIBE command to show the structure of our table**, such as column names, constraints on column names, etc. The **DESC** command is a short form of the DESCRIBE command. Both DESCRIBE and DESC command are equivalent and case sensitive.

### Syntax

The following are the syntax to display the table structure:

1. {DESCRIBE | **DESC**} table\_name;

**We can use the following steps to show all columns of the table:**

**Step 1:** Login into the MySQL database server.

**Step 2:** Switch to a specific database.

**Step 3:** Execute the DESCRIBE statement.

Let us understand it with the help of an example that explains how to show columns of the table in the selected database.

## Login to the MySQL Database

The first step is to login to the database server using the **username** and **password**. We should see the output as below image:

1. >mysql -u root -p
2. Enter **password**: \*\*\*\*\*
3. mysql>

```
MySQL 8.0 Command Line Client
Enter password: *****
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 8
Server version: 8.0.19 MySQL Community Server - GPL

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

## Switch to a Specific Database

The next step is to open a particular database from which you want to display the table information using the following query. After the execution of a query, we should see the below output:

1. mysql> USE mysqltestdb;

```
MySQL 8.0 Command Line Client
mysql> USE mysqltestdb;
Database changed
mysql> _
```

## Execute DESCRIBE Statement

It is the last step to display the table information. Before executing the DESCRIBE statement, we can optionally display all the tables stored in our selected database with the [\*\*SHOW TABLES\*\* statement](#):

1. mysql> SHOW TABLES;

```
MySQL 8.0 Command Line Client
mysql> USE mysqltestdb;
Database changed
mysql> SHOW TABLES;
+-----+
| Tables_in_mysqltestdb |
+-----+
| address_book
| branches
| contact
| contacts
| customer
| duplicate_table
| employee
| myset_test
| numeric_tests
| orders
+-----+
```

**For example**, if we want to show a **customer table's structure**, execute the below statement. After successful execution, it will give the output as below image:

1. mysql> DESCRIBE customer;

```
MySQL 8.0 Command Line Client
mysql> DESCRIBE customer;
+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key  | Default | Extra |
+-----+-----+-----+-----+-----+
| cust_id | int    | NO   | PRI   | NULL    |       |
| name    | varchar(35) | YES  |       | NULL    |       |
| occupation | varchar(25) | YES  |       | NULL    |       |
| age     | int    | YES  |       | NULL    |       |
+-----+-----+-----+-----+-----+
4 rows in set (0.03 sec)
```

We can also use the DESC statement for practice, which is a shorthand of the DESCRIBE command. See the below output:

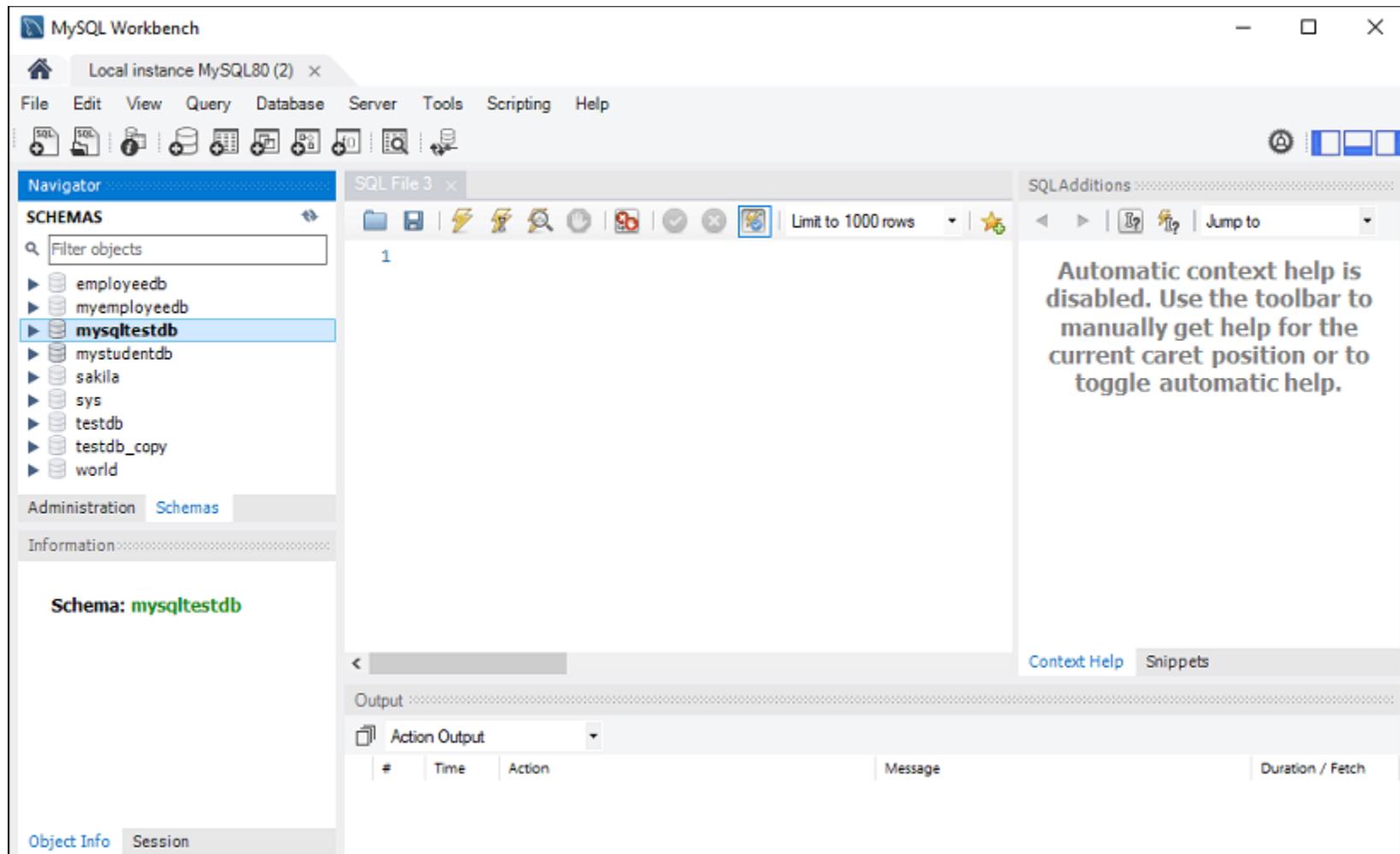
```

MySQL 8.0 Command Line Client
mysql> DESC customer;
+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key  | Default | Extra |
+-----+-----+-----+-----+-----+
| cust_id | int    | NO   | PRI   | NULL    |       |
| name     | varchar(35) | YES  |       | NULL    |       |
| occupation | varchar(25) | YES  |       | NULL    |       |
| age      | int    | YES  |       | NULL    |       |
+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)

```

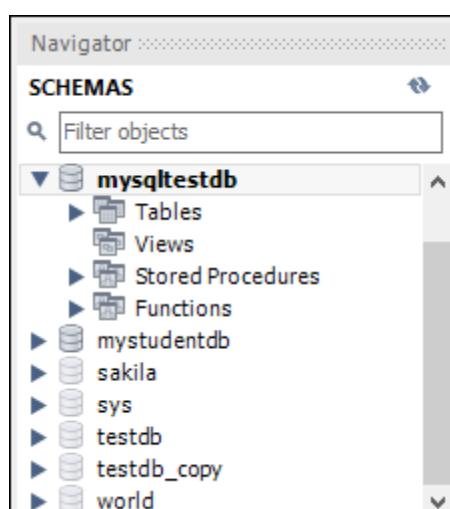
## How to display table information in MySQL Workbench?

To display the column information of the table in [MySQL Workbench](#), we first need to launch the Workbench tool and login with the username and password to the [MySQL](#) database server. We will get the following screen:

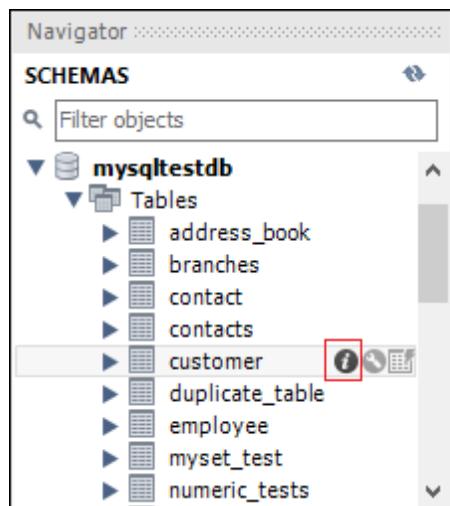


Now do the following steps to show the table information:

1. Go to the **Navigation tab** and click on the **Schema menu**. Here, we can see all the previously created databases. Select any database under the Schema menu, for example, **mysqltestdb**. It will pop up the multiple options that can be shown in the following image.



2. Next, click on the "**Tables**" that shows all tables stored in the mysqltestdb database. Select a table whose column information you want to display. Then, mouse over on that table, it will show **three icons**. See the below image:



Now, click the **icon (i)** shown in the red rectangular box. It will display the following image:

Column	Type	Default Value	Nullable	Character Set	Collation	Privileges
age	int		YES			select,insert,update,references
cust_id	int		NO			select,insert,update,references
name	varchar(35)		YES	utf8mb4	utf8mb4_0900_...	select,insert,update,references
occupation	varchar(25)		YES	utf8mb4	utf8mb4_0900_...	select,insert,update,references

Finally, click on the "**Columns**" menu to display the table structure.

## MySQL SHOW COLUMNS Command

MySQL also allows the SHOW COLUMNS command to display table structure. It is a more flexible way to get columns information of a table.

### Syntax:

The following are the syntax of the SHOW COLUMNS command:

1. mysql> SHOW COLUMNS **FROM** table\_name;

**For example**, if we execute the below query, we will get all columns information of a table in a particular database:

1. mysql> SHOW COLUMNS **FROM** customer;

```
MySQL 8.0 Command Line Client
mysql> SHOW COLUMNS FROM customer;
+-----+-----+-----+-----+-----+
| Field | Type  | Null | Key  | Default | Extra |
+-----+-----+-----+-----+-----+
| cust_id | int   | NO   | PRI  | NULL    |       |
| name    | varchar(35) | YES  |       | NULL    |       |
| occupation | varchar(25) | YES  |       | NULL    |       |
| age     | int   | YES  |       | NULL    |       |
+-----+-----+-----+-----+-----+
4 rows in set (0.01 sec)
```

If we want to show the **columns information of a table from another database** or not available in the current database, we can use the following query:

1. mysql> SHOW COLUMNS **FROM** database\_name.table\_name;
- 2.
3. OR
- 4.
5. mysql> SHOW COLUMNS **FROM** table\_name **IN** database\_name;

In the below image, we can see that we had used the mysqltestdb database. But we had displayed the column's information of a table from another database without switching to the current database.

```
MySQL 8.0 Command Line Client
mysql> USE mysqltestdb;
Database changed
mysql> SHOW COLUMNS FROM mystudentdb.student_info;
+-----+-----+-----+-----+-----+
| Field | Type  | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| stud_id | int   | NO   | PRI  | NULL    |       |
| stud_code | varchar(15) | YES  |       | NULL    |       |
| stud_name | varchar(35) | YES  |       | NULL    |       |
| subject | varchar(25) | YES  |       | NULL    |       |
| marks | int   | YES  |       | NULL    |       |
| phone | varchar(15) | YES  |       | NULL    |       |
+-----+-----+-----+-----+-----+
6 rows in set (0.00 sec)

mysql> SHOW COLUMNS FROM student_info IN mystudentdb;
+-----+-----+-----+-----+-----+
| Field | Type  | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| stud_id | int   | NO   | PRI  | NULL    |       |
| stud_code | varchar(15) | YES  |       | NULL    |       |
| stud_name | varchar(35) | YES  |       | NULL    |       |
| subject | varchar(25) | YES  |       | NULL    |       |
| marks | int   | YES  |       | NULL    |       |
| phone | varchar(15) | YES  |       | NULL    |       |
+-----+-----+-----+-----+-----+
```

If we want to display the more column information, we need to add **FULL** keyword with the SHOW TABLES statement as follows:

1. mysql> SHOW **FULL** COLUMNS **FROM** table\_name;

**For example**, the below SQL query lists all columns of the **student\_info** table in the **mystudentdb** database:

1. mysql> SHOW **FULL** COLUMNS **FROM** student\_info;

After execution, we can see that this command adds the **collation**, **privileges**, **default**, and **comment** columns to the result set.

```
MySQL 8.0 Command Line Client
mysql> SHOW FULL COLUMNS FROM student_info;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| Field | Type  | Collation | Null | Key | Default | Extra | Privileges | Comment |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| stud_id | int   | utf8mb4_0900_ai_ci | NO   | PRI  | NULL    |       | select,insert,update,references |
| stud_code | varchar(15) | utf8mb4_0900_ai_ci | YES  |       | NULL    |       | select,insert,update,references |
| stud_name | varchar(35) | utf8mb4_0900_ai_ci | YES  |       | NULL    |       | select,insert,update,references |
| subject | varchar(25) | utf8mb4_0900_ai_ci | YES  |       | NULL    |       | select,insert,update,references |
| marks | int   | NULL      | YES  |       | NULL    |       | select,insert,update,references |
| phone | varchar(15) | utf8mb4_0900_ai_ci | YES  |       | NULL    |       | select,insert,update,references |
+-----+-----+-----+-----+-----+-----+-----+-----+
6 rows in set (0.00 sec)
```

## MySQL EXPLAIN

The EXPLAIN keyword is synonymous to the DESCRIBE statement, which is **used to obtain information about how MySQL executes the queries**. It can work with INSERT, SELECT, DELETE, UPDATE, and REPLACE queries. From **MySQL 8.0.19** and later versions, it can also work with TABLE statements. When we use this keyword in queries, it will process the statement and provide the information about how tables are joined, the order of the table, estimated partitions and rows.

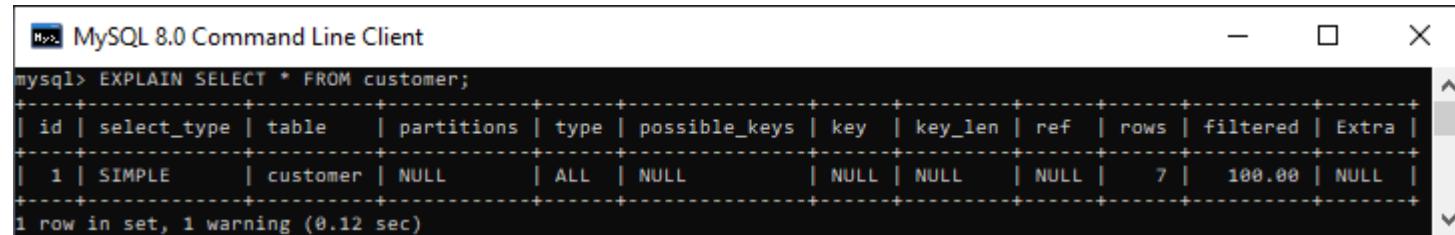
### Example

If we want to show the execution plan of a **SELECT statement**, we can use the query as below:

1. mysql> EXPLAIN **SELECT \* FROM** customer;

#### Output:

This query produces the following information:



The screenshot shows the MySQL 8.0 Command Line Client window. The command entered is 'EXPLAIN SELECT \* FROM customer;'. The output is a table with the following data:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	customer	NULL	ALL	NULL	NULL	NULL	NULL	7	100.00	NULL

1 row in set, 1 warning (0.12 sec)

## MySQL DROP Table

MYSQL uses a Drop Table statement to delete the existing table. This statement removes the complete data of a table along with the whole structure or definition permanently from the database. So, you must be very careful while removing the table because we cannot recover the lost data after deleting it.

### Syntax

The following are the syntax to remove the table in MySQL:

1. mysql> **DROP TABLE** table\_name;
2. OR,
3. mysql> **DROP TABLE** schema\_name.table\_name;

The full syntax of DROP TABLE statement in MySQL is:

1. **DROP [ TEMPORARY ] TABLE [ IF EXISTS ] table\_name [ RESTRICT | CASCADE ];**

The above syntax used many parameters or arguments. Let us discuss each in detail:

Parameter	Description
<b>Name</b>	
TEMPORARY	It is an optional parameter that specifies to delete the temporary tables only.
table_name	It specifies the name of the table which we are going to remove from the database.
IF EXISTS	It is optional, which is used with the DROP TABLE statement to remove the tables only if it exists in the database.
RESTRICT and CASCADE	Both are optional parameters that do not have any impact or effect on this statement. They are included in the syntax for future versions of MySQL.

**NOTE:** It is to be noted that you must have a **DROP** privileges to execute the **DROP TABLE** statement in the MySQL.

### Example

This example specifies how we can drop an existing table from the database. Suppose our database contains a table "orders" as shown in the image below:

```
MySQL 8.0 Command Line Client
mysql> SELECT * FROM orders;
+-----+-----+-----+
| order_id | prod_name | price |
+-----+-----+-----+
| 1 | Laptop | 80000 |
| 2 | Mouce | 3000 |
| 3 | Desktop | 50000 |
| 4 | Iphone | 50000 |
+-----+-----+-----+
4 rows in set (0.00 sec)

mysql>
```

To delete the above table, we need to run the following statement:

1. mysql> **DROP TABLE** orders;

It will remove the table permanently. We can also check the table is present or not as shown in the below output:

```
MySQL 8.0 Command Line Client
mysql> DROP TABLE orders;
Query OK, 0 rows affected (0.95 sec)

mysql> SELECT * FROM orders;
ERROR 1146 (42S02): Table 'mystudentdb.orders' doesn't exist
mysql>
```

If we try to delete a table that does not exist in the database, we will get an error message as given below:

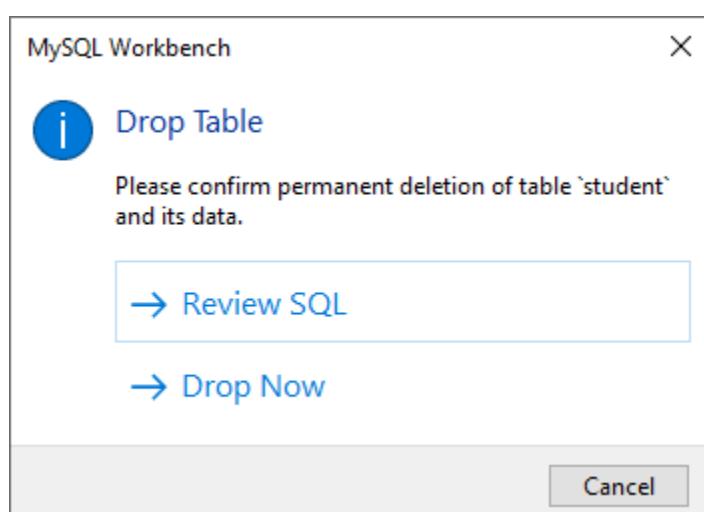
```
MySQL 8.0 Command Line Client
mysql> DROP TABLE orders;
ERROR 1051 (42S02): Unknown table 'mystudentdb.orders'
mysql>
```

If we use the IF EXISTS clause with the DROP TABLE statement, MySQL gives the warning message which can be shown in the below output:

```
MySQL 8.0 Command Line Client
mysql> DROP TABLE IF EXISTS orders;
Query OK, 0 rows affected, 1 warning (0.04 sec)
```

## How to DROP table in Workbench

1. To delete a table, you need to choose the table, right-click on it, and select the Drop Table option. The following screen appears:



2. Select **Drop Now** option in the popup window to delete the table from the database instantly.

## MySQL DROP Multiple Table

Sometimes we want to delete more than one table from the database. In that case, we have to use the table names and separate them by using the comma operator. The following statement can be used to remove multiple tables:

1. **DROP TABLE** IF EXISTS table\_name1, table\_name2, **table**, ...., table\_nameN;

## MySQL TRUNCATE Table vs. DROP Table

You can also use the DROP TABLE command to delete the complete table, but it will remove complete table data and structure both. You need to re-create the table again if you have to store some data. But in the case of TRUNCATE TABLE, it removes only table data, not structure. You don't need to re-create the table again because the table structure already exists.

# MySQL Temporary Table

MySQL has a feature to create a special table called a **Temporary Table** that allows us to **keep temporary data**. We can reuse this table several times in a particular session. It is available in MySQL for the user from **version 3.23**, and above so if we use an older version, this table cannot be used. This table is visible and accessible only for the **current session**. MySQL deletes this table automatically as long as the current session is closed or the user terminates the connection. We can also use the **DROP TABLE** command for removing this table explicitly when the user is not going to use it.

If we use a **PHP script** to run the code, this table removes automatically as long as the script has finished its execution. If the user is connected with the server through the MySQL client, then this table will exist until the user closes the MySQL client program or terminates the connection or removed the table manually.

A temporary table provides a very useful and flexible feature that allows us to achieve complex tasks quickly, such as when we query data that requires a single **SELECT statement** with **JOIN** clauses. Here, the user can use this table to keep the output and performs another query to process it.

A temporary table in **MySQL** has many features, which are given below:

- MySQL uses the CREATE TEMPORARY TABLE statement to create a temporary table.
- This statement can only be used when the MySQL server has the CREATE TEMPORARY TABLES privilege.
- It can be visible and accessible to the client who creates it, which means two different clients can use the temporary tables with the same name without conflicting with each other. It is because this table can only be seen by that client who creates it. Thus, the user cannot create two temporary tables with the same name in the same session.
- A temporary table in MySQL will be dropped automatically when the user closes the session or terminates the connection manually.
- A temporary table can be created by the user with the same name as a normal table in a database. For example, if the user creates a temporary table with the name student, then the existing student table cannot be accessible. So, the user performs any query against the student table, is now going to refer to the temporary student table. When the user removes a temporary table, the permanent student table becomes accessible again.

## Syntax of Creating Temporary Table

In MySQL, the syntax of creating a temporary table is the same as the syntax of creating a normal table statement except the TEMPORARY keyword. Let us see the following statement which creates the temporary table:

1. mysql> **CREATE TEMPORARY TABLE** table\_name (
2.   column\_1, column\_2, ..., table\_constraints
3. );

If the user wants to create a temporary table whose structure is the same as an existing table in the database, then the above statement cannot be used. Instead, we use the syntax as given below:

1. Mysql> **CREATE TEMPORARY TABLE** temporary\_table\_name **SELECT \* FROM** original\_table\_name **LIMIT 0;**

## MySQL Temporary Table Example

Let us understand how we can create a temporary table in MySQL. Execute the following statement that creates a temporary table in the selected database:

1. mysql> **CREATE TEMPORARY TABLE** Students( student\_name **VARCHAR**(40) NOT NULL, total\_marks **DECIMAL**(12 ,2) NOT NULL **DEFAULT** 0.00, total\_subjects **INT** UNSIGNED NOT NULL **DEFAULT** 0);

We can see the below image:

```
MySQL 8.0 Command Line Client
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 52
Server version: 8.0.19 MySQL Community Server - GPL

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use mystudentdb;
Database changed
mysql> CREATE TEMPORARY TABLE Students( student_name VARCHAR(40) NOT NULL, total_marks DECIMAL(12,2) NOT NULL DEFAULT 0.00,
total_subjects INT UNSIGNED NOT NULL DEFAULT 0);
Query OK, 0 rows affected (0.00 sec)

mysql>
```

Next, we need to insert values in the temporary table:

1. mysql>**INSERT INTO** Students(student\_name, total\_marks, total\_subjects) **VALUES** ('Joseph', 150.75, 2), ('Peter', 180.75, 2);

After executing the above statement, it will give the below output:

```
MySQL 8.0 Command Line Client
Server version: 8.0.19 MySQL Community Server - GPL

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use mystudentdb;
Database changed
mysql> CREATE TEMPORARY TABLE Students( student_name VARCHAR(40) NOT NULL, total_marks DECIMAL(12,2) NOT NULL DEFAULT 0.00,
total_subjects INT UNSIGNED NOT NULL DEFAULT 0);
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO Students(student_name, total_marks, total_subjects) VALUES ('Joseph', 150.75, 2), ('Peter', 180.75, 2);
Query OK, 2 rows affected (0.05 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql>
```

Now, run the following query to get the result:

1. mysql> **SELECT \* FROM** Students;

After the successful execution of the above statement, we will get the output as below:

```
MySQL 8.0 Command Line Client

mysql> SELECT * FROM Students;
+-----+-----+-----+
| student_name | total_marks | total_subjects |
+-----+-----+-----+
| Joseph      |    150.75  |          2 |
| Peter       |    180.75  |          2 |
+-----+-----+-----+
2 rows in set (0.00 sec)

mysql>
```

It is to be noted that when we run a **SHOW TABLES** command, then our temporary table will not be shown on the list. Also, if we close the current session and then will execute the SELECT statement, we will get a message saying that no data available in the database, and even the temporary table will not exist.

## A Temporary Table whose structure is based on a normal table

In this example, we are going to create a temporary table whose structure is based on the already available tables in the database.

Suppose our database has the following table as permanent:

```

MySQL> SELECT * FROM customer;
+-----+-----+-----+
| cust_id | cust_name | city      | occupation |
+-----+-----+-----+
| 1       | Peter     | London    | Business   |
| 2       | Joseph    | Texas     | Doctor     |
| 3       | Mark      | New Delhi | Engineer  |
| 4       | Michael   | Newyork   | Scientist |
+-----+-----+-----+
4 rows in set (0.00 sec)

MySQL> SELECT * FROM orders;
+-----+-----+-----+
| order_id | prod_name | price |
+-----+-----+-----+
| 1         | Laptop    | 80000 |
| 2         | Mouce     | 3000  |
| 3         | Desktop   | 50000 |
| 4         | Iphone    | 50000 |
+-----+-----+-----+
4 rows in set (0.00 sec)

MySQL> -

```

Here, the structure of a temporary table is created by using the SELECT statement and merge two tables using the INNER JOIN clause and sorts them based on the price. Write the following statement in the MySQL prompt:

1. **CREATE TEMPORARY TABLE** temp\_customers
2. **SELECT** c.cust\_name, c.city, o.prod\_name, o.price
3. **FROM** orders o
4. **INNER JOIN** customer c **ON** c.cust\_id = o.order\_id
5. **ORDER BY** o.price **DESC**;

When we execute the above statement, we will get the following message:

```

MySQL> CREATE TEMPORARY TABLE temp_customers SELECT c.cust_name, c.city, o.prod_name, o.price FROM orders o
INNER JOIN customer c ON c.cust_id = o.order_id ORDER BY o.price DESC;
Query OK, 4 rows affected (0.04 sec)
Records: 4  Duplicates: 0  Warnings: 0

```

Now, run the below command to see the temporary table:

1. mysql> **SELECT \* FROM** temp\_customers;

```

MySQL> SELECT * FROM temp_customers;
+-----+-----+-----+
| cust_name | city      | prod_name | price |
+-----+-----+-----+
| Peter     | London    | Laptop    | 80000 |
| Mark      | New Delhi | Desktop   | 50000 |
| Michael   | Newyork   | Iphone    | 50000 |
| Joseph    | Texas     | Mouce     | 3000  |
+-----+-----+-----+
4 rows in set (0.00 sec)

```

We can also perform queries from the above temporary table "**temp\_customers**" similar to the querying data from a permanent table. The following query explains it more clearly:

1. Mysql> **SELECT** cust\_name, prod\_name, price **FROM** temp\_customers;

After executing the above statement, it will give the output as below:

```

MySQL> SELECT cust_name, prod_name, price FROM temp_customers;
+-----+-----+-----+
| cust_name | prod_name | price |
+-----+-----+-----+
| Peter     | Laptop    | 80000 |
| Mark      | Desktop   | 50000 |
| Michael   | Iphone    | 50000 |
| Joseph    | Mouce     | 3000  |
+-----+-----+-----+
4 rows in set (0.00 sec)

```

**NOTE:** It is noted that we can use **IF NOT EXISTS** keyword to avoid the "table already exists" error.

## How to Drop Temporary Table in MySQL

MySQL allows us to remove the temporary table using the **DROP TABLE** statement. But, it's a good practice to use the **TEMPORARY** keyword with the **DROP TABLE** statement. This keyword helps us to avoid the mistake of deleting a permanent table when the temporary table and permanent table have the same name in the current session. So, it is recommended to use the following query for removing the temporary table:

1. mysql> **DROP TEMPORARY TABLE** table\_name;

This query will not remove a permanent table of the database that means it only deletes a temporary table. If we try to delete a permanent table with this statement, it will throw an error message saying that you are deleting a table is unknown. For example, if we want to remove the above temporary table "temp\_customers", we need to use the following statement:

1. mysql> **DROP TEMPORARY TABLE** top\_customers;

## MySQL Copy/Clone/Duplicate Table

MySQL copy or clone table is a feature that allows us to create a **duplicate table of an existing table**, including the table structure, indexes, constraints, default values, etc. Copying data of an existing table into a new table is very useful in a situation like backing up data in table failure. It is also advantageous when we need to test or perform something without affecting the original table, for example, replicating the production data for testing.

We can copy an existing table to a new table using the **CREATE TABLE** and **SELECT** statement, as shown below:

1. **CREATE TABLE** new\_table\_name
2. **SELECT** column1, column2, column3
3. **FROM** existing\_table\_name;

From the above, first, it creates a new table that indicates in the CREATE TABLE statement. Second, the result set of a **SELECT statement** defines the structure of a new table. Finally, **MySQL** fills data getting from the SELECT statement to the newly created table.

If there is a need to copy only partial data from an existing table to a new table, use **WHERE clause** with the SELECT statement as shown below:

1. **CREATE TABLE** new\_table\_name
2. **SELECT** column1, column2, column3
3. **FROM** existing\_table\_name
4. **WHERE** condition;

We have to ensure that the table we are going to create should not already exist in our database. The **IF NOT EXISTS** clause in MySQL allows us to check whether a table exists in the database or not before creating a new table. So, the below statement explains it more clearly:

1. **CREATE TABLE** IF NOT EXISTS new\_table\_name
2. **SELECT** column1, column2, column3
3. **FROM** existing\_table\_name
4. **WHERE** condition;

It is to be noted that this statement only copies the table and its data. It doesn't copy all dependent objects of the table, such as indexes, triggers, primary key constraints, foreign key constraints, etc. So the **command of copying data along with its dependent objects** from an existing to the new table can be written as the following statements:

1. **CREATE TABLE** IF NOT EXISTS new\_table\_name LIKE existing\_table\_name;
- 2.
3. **INSERT** new\_table\_name **SELECT \* FROM** existing\_table\_name;

In the above, we can see that we need to execute two statements for copying data along with structure and constraints. The first command creates a new table **new\_table\_name** that duplicates the **existing\_table\_name**, and the second command adds data from the existing table to the new\_table\_name.

## MySQL Copy/Clone Table Example

Let us demonstrate how we can create a duplicate table with the help of an example. First, we are going to create a table named "**original\_table**" using the below statement:

1. **CREATE TABLE** original\_table (
2. Id **int PRIMARY KEY** NOT NULL,
3. **Name varchar(45)** NOT NULL,
4. Product **varchar(45) DEFAULT** NULL,
5. Country **varchar(25) DEFAULT** NULL,

```
6. Year int NOT NULL  
7. );
```

Next, it is required to add values to this table. Execute the below statement:

1. **INSERT INTO** original\_table( Id, Name, Product, Country, Year)
2. **VALUES** (1, 'Stephen', 'Computer', 'USA', 2015),
3. (2, 'Joseph', 'Laptop', 'India', 2016),
4. (3, 'John', 'TV', 'USA', 2016),
5. (4, 'Donald', 'Laptop', 'England', 2015),
6. (5, 'Joseph', 'Mobile', 'India', 2015),
7. (6, 'Peter', 'Mouse', 'England', 2016);

Next, execute the SELECT statement to display the records:

1. mysql> **SELECT \* FROM** original\_table;

We will get the output, as shown below:

```
MySQL 8.0 Command Line Client  
mysql> CREATE TABLE original_table (  
    -> Id int PRIMARY KEY NOT NULL,  
    -> Name varchar(45) NOT NULL,  
    -> Product varchar(45) DEFAULT NULL,  
    -> Country varchar(25) DEFAULT NULL,  
    -> Year int NOT NULL  
    -> );  
Query OK, 0 rows affected (1.55 sec)  
  
mysql> INSERT INTO original_table( Id, Name, Product, Country, Year)  
    -> VALUES (1, 'Stephen', 'Computer', 'USA', 2015),  
    -> (2, 'Joseph', 'Laptop', 'India', 2016),  
    -> (3, 'John', 'TV', 'USA', 2016),  
    -> (4, 'Donald', 'Laptop', 'England', 2015),  
    -> (5, 'Joseph', 'Mobile', 'India', 2015),  
    -> (6, 'Peter', 'Mouse', 'England', 2016);  
Query OK, 6 rows affected (0.14 sec)  
Records: 6  Duplicates: 0  Warnings: 0  
  
mysql> SELECT * FROM original_table;  
+----+----+----+----+  
| Id | Name  | Product | Country | Year |  
+----+----+----+----+  
| 1  | Stephen | Computer | USA     | 2015 |  
| 2  | Joseph  | Laptop   | India   | 2016 |  
| 3  | John    | TV       | USA     | 2016 |  
| 4  | Donald  | Laptop   | England | 2015 |  
| 5  | Joseph  | Mobile   | India   | 2015 |  
| 6  | Peter   | Mouse   | England | 2016 |  
+----+----+----+----+
```

Now, execute the following statement that copies data from the existing table "original\_table" to a new table named "**duplicate\_table**" in the selected database.

1. **CREATE TABLE** IF NOT EXISTS **duplicate\_table**
2. **SELECT \* FROM** original\_table;

After the successful execution, we can verify the table data using the SELECT statement. See the below output:

```
MySQL 8.0 Command Line Client  
mysql> CREATE TABLE IF NOT EXISTS duplicate_table  
    -> SELECT * FROM original_table;  
Query OK, 6 rows affected (1.78 sec)  
Records: 6  Duplicates: 0  Warnings: 0  
  
mysql> SELECT * FROM duplicate_table;  
+----+----+----+----+  
| Id | Name  | Product | Country | Year |  
+----+----+----+----+  
| 1  | Stephen | Computer | USA     | 2015 |  
| 2  | Joseph  | Laptop   | India   | 2016 |  
| 3  | John    | TV       | USA     | 2016 |  
| 4  | Donald  | Laptop   | England | 2015 |  
| 5  | Joseph  | Mobile   | India   | 2015 |  
| 6  | Peter   | Mouse   | England | 2016 |  
+----+----+----+----+
```

Sometimes there is a need to copy only **partial data** from an existing table to a new table. In that case, we can use the WHERE clause with the SELECT statement as follows:

1. **CREATE TABLE** IF NOT EXISTS `duplicate_table`
2. **SELECT \* FROM** `original_table` **WHERE** `Year = '2016'`;

This statement creates a duplicate table that contains data for the **year 2016** only. We can verify the table using a SELECT statement, as shown below:

The screenshot shows the MySQL 8.0 Command Line Client interface. The command line displays two SQL statements: `CREATE TABLE IF NOT EXISTS duplicate_table` followed by a multi-line comment block containing `-> SELECT * FROM original_table WHERE Year='2016';`. The response shows "Query OK, 3 rows affected (0.99 sec)". Below this, the command `SELECT * FROM duplicate_table;` is run, and the result is displayed in a table format. The table has columns `Id`, `Name`, `Product`, `Country`, and `Year`. The `Year` column for all three rows is highlighted with a red box.

Id	Name	Product	Country	Year
2	Joseph	Laptop	India	2016
3	John	TV	USA	2016
6	Peter	Mouse	England	2016

Suppose there is a need to copy an existing table along with all dependent objects associated with the table, execute the two statements that are given below:

1. `mysql> CREATE TABLE` `duplicate_table` `LIKE` `original_table`;
2. AND,
3. `mysql> INSERT` `duplicate_table` `SELECT * FROM` `original_table`;

The screenshot shows the MySQL 8.0 Command Line Client interface. It displays three SQL statements: `CREATE TABLE` `duplicate_table` `LIKE` `original_table`; `INSERT` `duplicate_table` `SELECT * FROM` `original_table`; and `SELECT * FROM` `duplicate_table`. The first statement returns "Query OK, 0 rows affected (0.88 sec)". The second statement returns "Query OK, 6 rows affected (0.24 sec)". The third statement shows the results of the `SELECT` query on the `duplicate_table`, which contains 6 rows of data from the `original_table`.

Id	Name	Product	Country	Year
1	Stephen	Computer	USA	2015
2	Joseph	Laptop	India	2016
3	John	TV	USA	2016
4	Donald	Laptop	England	2015
5	Joseph	Mobile	India	2015
6	Peter	Mouse	England	2016

### Let us see how we can copy a table to a different database through an example.

Suppose there is a situation to copy a table from a different database. In that case, we need to execute the below statements:

1. **CREATE TABLE** `destination_db.new_table_name`
2. `LIKE` `source_db.existing_table_name`;
- 3.
4. **INSERT** `destination_db.new_table_name`
5. **SELECT \* FROM** `source_db.existing_table_name`;

In the above, the first command creates a new table in the selected(destination) database by cloning the existing table from the source database. The second command copies data from the existing table to the new table in the selected database.

### The following demonstration explains it more clearly.

Suppose we have two databases named "**mysqltestdb**" and "**mystudentdb**" on the MySQL Server. The `mytestdb` database contains a table named "original\_table" that have the following data:

```

MySQL 8.0 Command Line Client
mysql> SELECT * FROM mysqltestdb.original_table;
+----+-----+-----+-----+-----+
| Id | Name  | Product | Country | Year |
+----+-----+-----+-----+-----+
| 1  | Stephen | Computer | USA    | 2015 |
| 2  | Joseph  | Laptop   | India   | 2016 |
| 3  | John    | TV       | USA    | 2016 |
| 4  | Donald  | Laptop   | England | 2015 |
| 5  | Joseph  | Mobile   | India   | 2015 |
| 6  | Peter   | Mouse    | England | 2016 |
+----+-----+-----+-----+-----+

```

Now, we are going to copy this table into another database named mystudentdb using the following statement:

1. **CREATE TABLE** mystudentdb.duplicate\_table
2. **LIKE** mysqltestdb.original\_table;
- 3.
4. **INSERT** mystudentdb.duplicate\_table
5. **SELECT \* FROM** mysqltestdb.original\_table;

After successful execution, we can verify the table in mystudentdb database using the below command:

1. mysql> **SELECT \* FROM** mystudentdb.duplicate\_table;

In the below output, we can see that the table is successfully copied into one database to another database.

```

MySQL 8.0 Command Line Client
mysql> CREATE TABLE mystudentdb.duplicate_table
-> LIKE mysqltestdb.original_table;
Query OK, 0 rows affected (1.04 sec)

mysql> INSERT mystudentdb.duplicate_table
-> SELECT * FROM mysqltestdb.original_table;
Query OK, 6 rows affected (0.11 sec)
Records: 6  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM mystudentdb.duplicate_table;
+----+-----+-----+-----+-----+
| Id | Name  | Product | Country | Year |
+----+-----+-----+-----+-----+
| 1  | Stephen | Computer | USA    | 2015 |
| 2  | Joseph  | Laptop   | India   | 2016 |
| 3  | John    | TV       | USA    | 2016 |
| 4  | Donald  | Laptop   | England | 2015 |
| 5  | Joseph  | Mobile   | India   | 2015 |
| 6  | Peter   | Mouse    | England | 2016 |
+----+-----+-----+-----+-----+

```

## MySQL REPAIR TABLE

### How to Fix a Corrupted Table in MySQL?

MySQL Repair Table allows us to repair or fix the corrupted table. The repair table in MySQL provides **support only for selected storage engines, not for all**. It is to ensure that we have a few privileges like **SELECT** and **INSERT** to use this statement. Normally, we should never use the repair table until disastrous things happen with the table. This statement rarely gets all data from the **MyISAM** table. Therefore, we need to find why our table is corrupted to eliminate the use of this statement.

When we execute the REPAIR TABLE statement, it first checks the table that we are going to repair is required an upgradation or not. If required, it will perform upgradation with the same rules as CHECK TABLE ... FOR UPGRADE statement works. It is always good to keep our table's backup before performing the "table repair" option because it might cause a loss of our data.

### Syntax

The following is the syntax to repair a corrupted table in MySQL:

1. REPAIR [NO\_WRITE\_TO\_BINLOG | **LOCAL**]
2. **TABLE** tbl\_name [, tbl\_name] ...
3. [QUICK] [EXTENDED] [USE\_FRM]

Let us discuss the use of each option in detail.

**NO\_WRITE\_TO\_BINLOG or LOCAL:** It's a place where the server is responsible for writing the REPAIR TABLE statements for the replication slaves. We can optionally specify the optional NO\_WRITE\_TO\_BINLOG/LOCAL keyword to suppress the logging.

**QUICK:** The quick option allows the REPAIR TABLE statement for repairing only the index file. It does not allow to repair of the data file. This type of repair gives the same result as the **myisamchk --recover -quick** command works.

**EXTENDED:** Instead of creating the index row by row, this option allows MySQL to create one index at a time with sorting. This type of repair gives the same result as the **myisamchk --safe-recover** command works.

**USE\_FRM:** This option is used when the .MYI index file is not found or if its header is corrupted. The USE-FRM option informs MySQL to do not trust the information present in this file header and re-create it by using the information provided from the data dictionary. This type of repair cannot work with the **myisamchk** command.

## Storage Engine and Partitioning Support with Repair Table

We have mentioned earlier that the repair table does not work for all storage engines. It supports only MyISAM, ARCHIVE, and CSV tables. The **repair table statement does not support views**.

We can also use the repair table statement for partitioned tables. But, here, we cannot use the USE\_FRM option with this statement. If we want to repair multiple partitions, we can use the **ALTER TABLE ... REPAIR PARTITION** statement.

## MySQL REPAIR TABLE Example

Let us understand the working of the repair table statement in MySQL through example. First, we need to create a new table named **vehicle** in the selected database as follows:

1. **CREATE TABLE** vehicle (
2.   vehicle\_no **VARCHAR**(18) **PRIMARY KEY**,
3.   model\_name **VARCHAR**(45),
4.   cost\_price **DECIMAL**(10,2 ),
5.   sell\_price **DECIMAL**(10,2 )
6. );

Next, we will insert some data into this table with the below statement:

1. mysql> **INSERT INTO** vehicle (vehicle\_no, model\_name, cost\_price, sell\_price)
2. **VALUES**('S2001', 'Scorpio', 950000, 1000000),
3. ('M3000', 'Mercedes', 2500000, 3000000),
4. ('R0001', 'Rolls Royas', 75000000, 85000000);

Next, execute the below statement to verify the data:

1. mysql> **SELECT \* FROM** vehicle;

We should get the below result:

vehicle_no	model_name	cost_price	sell_price
M3000	Mercedes	2500000.00	3000000.00
R0001	Rolls Royas	75000000.00	85000000.00
S2001	Scorpio	950000.00	1000000.00

Next, we will execute the below statement to check the storage engine of the vehicle table:

1. mysql> **SELECT** table\_name, engine
2. **FROM** information\_schema.tables
3. **WHERE** table\_name = 'vehicle';

After executing the statement, we should get the below output:

```

MySQL 8.0 Command Line Client
mysql> SELECT table_name, engine
-> FROM information_schema.tables
-> WHERE table_name = 'vehicle';
+-----+-----+
| TABLE_NAME | ENGINE |
+-----+-----+
| vehicle    | InnoDB |
+-----+-----+
1 row in set (0.11 sec)

```

Here we can see that the storage engine of the vehicle table is InnoDB. Therefore, if we create the repair table using the below query for this storage engine, MySQL issued an error:

1. mysql> REPAIR **TABLE** vehicle;

See the below output:

```

MySQL 8.0 Command Line Client
mysql> REPAIR TABLE vehicle;
+-----+-----+
| Table      | Op      | Msg_type | Msg_text          |
+-----+-----+
| testdb.vehicle | repair | note     | The storage engine for the table doesn't support repair |
+-----+-----+
1 row in set (0.07 sec)

```

To remove this error, we first need to alter the table storage engine to MyISAM with the following query and then used the repair table statement.

1. mysql> **ALTER TABLE** vehicle ENGINE = 'MyISAM';
2. //Now, use the repair **table** query
3. mysql> REPAIR **TABLE** vehicle;

We will get the below output:

```

MySQL 8.0 Command Line Client
mysql> ALTER TABLE vehicle ENGINE = 'MyISAM';
Query OK, 3 rows affected (1.47 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> REPAIR TABLE vehicle;
+-----+-----+
| Table      | Op      | Msg_type | Msg_text          |
+-----+-----+
| testdb.vehicle | repair | status   | OK               |
+-----+-----+
1 row in set (0.34 sec)

```

**In this output, we can see that the REPAIR TABLE statement contains the following columns in the result set:**

SN	Column Name	Descriptions
1.	Table	This column indicates the name of the table.
2.	Op	This column always contains repair word whether the storage engine supports or not with the statement.
3.	Msg_type	This column can be either status, error, info, note, or warning.
4.	Msg_text	This column consists of the informational message.

Let us see another example to use a repair table statement with any QUICK, EXTENDED or USE\_FRM options. Thus, we will first create another table named **memberships** and stored this table in the "**MyISAM**" storage engine instead of the default one InnoDB.

1. **CREATE TABLE** memberships (
2. id **INT AUTO\_INCREMENT PRIMARY KEY**,
3. **name VARCHAR(55) NOT NULL**,
4. **email VARCHAR(55) NOT NULL**,
5. **plan VARCHAR(45) NOT NULL**,
6. **validity\_date DATE NOT NULL**

7. ) ENGINE = MyISAM;

We will insert some data into this table with the below statement:

1. mysql> **INSERT INTO** memberships (**name**, email, plan, validity\_date)
2. **VALUES('Stephen', 'stephen@javatpoint.com', 'Gold', '2020-06-13'),**
3. **('Jenifer', 'jenifer@javatpoint.com', 'Platinum', '2020-06-10'),**
4. **('david', 'david@javatpoint.com', 'Silver', '2020-06-15');**

Next, execute the SELECT statement to verify the data. We will get the below result:

```
MySQL 8.0 Command Line Client
mysql> SELECT * FROM memberships;
+----+-----+-----+-----+
| id | name | email        | plan   | validity_date |
+----+-----+-----+-----+
| 1  | Stephen | stephen@javatpoint.com | Gold   | 2020-06-13    |
| 2  | Jenifer | jenifer@javatpoint.com | Platinum | 2020-06-10    |
| 3  | david   | david@javatpoint.com | Silver  | 2020-06-15    |
+----+-----+-----+-----+
3 rows in set (0.01 sec)
```

Since we have created the MyISAM storage engine table, the repair table statement does not issue any error. See the below statement:

1. mysql> **REPAIR TABLE** memberships **QUICK EXTENDED**;

We should get the output as follows:

```
MySQL 8.0 Command Line Client
mysql> REPAIR TABLE memberships QUICK EXTENDED;
+-----+-----+-----+
| Table      | Op     | Msg_type | Msg_text |
+-----+-----+-----+
| testdb.memberships | repair | status   | OK       |
+-----+-----+-----+
1 row in set (0.09 sec)
```

If we use the REPAIR TABLE statement with the table that does not exist in our selected database, MySQL gives an error message. See the below statement:

1. mysql> **REPAIR TABLE** service\_memberships **QUICK EXTENDED**;

After execution, we will get the following output:

```
MySQL 8.0 Command Line Client
mysql> REPAIR TABLE service_memberships QUICK EXTENDED;
+-----+-----+-----+
| Table      | Op     | Msg_type | Msg_text |
+-----+-----+-----+
| testdb.service_memberships | repair | Error    | Table 'testdb.service_memberships' doesn't exist |
| testdb.service_memberships | repair | status   | Operation failed |
+-----+-----+-----+
```

In this article, we have learned how to repair the corrupted table in MySQL using the Repair Table statement. This statement works only for certain storage engines. Thus, before using this query, we first check the table storage engine supports it or not. If it is not supported, we need to change it into MyISAM, ARCHIVE, or CSV. It is always good to keep our table's backup before performing the "table repair" query because it might cause a loss of our data.

## MySQL Add/Delete Column

A column is a series of cells in a table that may contain text, numbers, and images. Every column stores one value for each row in a table. In this section, we are going to discuss how to add or delete columns in an existing table.

### How can we add a column in MySQL table?

MySQL allows the **ALTER TABLE ADD COLUMN** command to add a new column to an existing table. The following are the syntax to do this:

1. **ALTER TABLE** table\_name
2. **ADD COLUMN** column\_name column\_definition [**FIRST|AFTER** existing\_column];

In the above,

- First, we need to specify the table name.
- Next, after the ADD COLUMN clause, we have to specify the name of a new column along with its definition.
- Finally, we need to specify the FIRST or AFTER keyword. The FIRST Keyword is used to add the column as the first column of the table. The AFTER keyword is used to add a new column after the existing column. If we have not provided these keywords, MySQL adds the new column as the last column in the table by default.

Sometimes it is required to add **multiple columns** into the existing table. Then, we can use the syntax as follows:

1. **ALTER TABLE** table\_name
2. **ADD COLUMN** column\_name1 column\_definition [**FIRST|AFTER** existing\_column],
3. **ADD COLUMN** column\_name2 column\_definition [**FIRST|AFTER** existing\_column];

## MySQL ADD COLUMN Example

Let us understand it with the help of various examples. Here, we will create a table named "**Test**" using the following statements:

1. **CREATE TABLE** Test (
2.   Stude\_id **int AUTO\_INCREMENT PRIMARY KEY**,
3.   **Name varchar(55) NOT NULL**
4. );

The table structure looks like the below image:

The screenshot shows the MySQL 8.0 Command Line Client interface. The command line displays the creation of a table 'Test' with two columns: 'ID' (auto-increment primary key) and 'Name' (varchar(55) NOT NULL). The 'DESCRIBE Test;' command is run, showing the table structure with columns ID and Name.

```
MySQL 8.0 Command Line Client
mysql> CREATE TABLE Test (
    -> ID int AUTO_INCREMENT PRIMARY KEY,
    -> Name varchar(55) NOT NULL
    -> );
Query OK, 0 rows affected (0.81 sec)

mysql> DESCRIBE Test;
+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key  | Default | Extra       |
+-----+-----+-----+-----+-----+
| ID    | int    | NO   | PRI  | NULL    | auto_increment |
| Name  | varchar | NO   |      | NULL    |              |
+-----+-----+-----+-----+-----+
```

After creating a table, we want to add a new column named City to the Test table. Since we have not specified the new column position explicitly after the column name, MySQL will add it as the last column.

1. **ALTER TABLE** Test
2. **ADD COLUMN** City **VARCHAR(30) NOT NULL**;

Next, we want to add a new column named **Phone\_number** to the **Test** table. This time, we will explicitly specify the new column position so that MySQL adds the column to the specified place.

1. **ALTER TABLE** Test
2. **ADD COLUMN** Phone\_number **VARCHAR(20) NOT NULL AFTER Name**;

In the below output, we can see that the two columns are added successfully at the specified position.

```

MySQL 8.0 Command Line Client
mysql> ALTER TABLE Test
    -> ADD COLUMN City VARCHAR(30) NOT NULL;
Query OK, 0 rows affected (0.77 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> ALTER TABLE Test
    -> ADD COLUMN Phone_number VARCHAR(20) NOT NULL AFTER Name;
Query OK, 0 rows affected (1.42 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> DESCRIBE Test;
+-----+-----+-----+-----+
| Field      | Type       | Null | Key | Default | Extra           |
+-----+-----+-----+-----+
| ID          | int         | NO   | PRI  | NULL    | auto_increment |
| Name        | varchar(5) | NO   |      | NULL    |                 |
| Phone_number | varchar(20) | NO   |      | NULL    |                 |
| City        | varchar(30) | NO   |      | NULL    |                 |
+-----+-----+-----+-----+

```

Let us add some data into the Test table using the [INSERT statement](#) as follows:

1. **INSERT INTO** Test( **Name**, Phone\_number, City)
2. **VALUES** ('Peter', '34556745362', 'California'),
3. ('Mike', '983635674562', 'Texas');

It will look like this.

```

MySQL 8.0 Command Line Client
mysql> INSERT INTO Test( Name, Phone_number, City)
    -> VALUES ('Peter', '34556745362', 'California'),
    -> ('Mike', '983635674562', 'Texas');
Query OK, 2 rows affected (0.14 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM Test;
+-----+-----+-----+-----+
| Stude_id | Name    | Phone_number | City      |
+-----+-----+-----+-----+
|      1   | Peter   | 34556745362 | California |
|      2   | Mike    | 983635674562 | Texas     |
+-----+-----+-----+-----+

```

Suppose we want to add more than one column ,(**Branch, Email**) in the Test table. In that case, execute the statement as follows:

1. **ALTER TABLE** Test
2. **ADD COLUMN** Branch **VARCHAR(30) DEFAULT NULL After Name**,
3. **ADD COLUMN** Email **VARCHAR(20) DEFAULT NULL AFTER Phone\_number;**

It is to note that columns Branch and Email are assigned to default value **NULL**. However, the Test table already has data so that MySQL will use null values for those new columns.

We can verify the record in the Test table as below:

```

MySQL 8.0 Command Line Client
mysql> ALTER TABLE Test
    -> ADD COLUMN Branch VARCHAR(30) DEFAULT NULL After Name,
    -> ADD COLUMN Email VARCHAR(20) DEFAULT NULL AFTER Phone_number;
Query OK, 0 rows affected (2.37 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM Test;
+-----+-----+-----+-----+-----+-----+
| Stude_id | Name    | Branch | Phone_number | Email   | City      |
+-----+-----+-----+-----+-----+-----+
|      1   | Peter   | NULL   | 34556745362 | NULL   | California |
|      2   | Mike    | NULL   | 983635674562 | NULL   | Texas     |
+-----+-----+-----+-----+-----+-----+

```

If we accidentally add a new column with the existing column name, MySQL will **throw an error**. For example, execute the below statement that issues an error:

1. **ALTER TABLE** Test
2. **ADD COLUMN** City **VARCHAR(30) NOT NULL;**

We will get the following error message.

```
MySQL 8.0 Command Line Client
mysql> ALTER TABLE Test
-> ADD COLUMN City VARCHAR(30) NOT NULL;
ERROR 1060 (42S21): Duplicate column name 'City'
```

## How can we rename a column in MySQL table?

MySQL allows the **ALTER TABLE CHANGE COLUMN** statement to change the old column with a new name. The following are the syntax to do this:

1. **ALTER TABLE** table\_name
2. **CHANGE COLUMN** old\_column\_name new\_column\_name column\_definition [**FIRST|AFTER** existing\_column];

In the above,

- o First, we need to specify the table name.
- o Next, after the CHANGE COLUMN clause, we have to specify the old column name and new column name along with its definition. We must have to specify the column definition even it will not change.
- o Finally, we need to specify the FIRST or AFTER keyword. It is optional that specified when we need to change the column name at the specific position.

### MySQL RENAME COLUMN Example

This example shows how we can change the column name in the MySQL table:

1. **ALTER TABLE** Test
2. **CHANGE COLUMN** Phone\_number Mobile\_number
3. **varchar**(20) NOT NULL;

This statement will change the column name **Phone\_number** with the new name **Mobile\_number** in the Test table. The below output explains it more clearly.

```
Select MySQL 8.0 Command Line Client
mysql> ALTER TABLE Test
-> CHANGE COLUMN Phone_number Mobile_number
-> varchar(20) NOT NULL;
Query OK, 0 rows affected (0.31 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM Test;
+-----+-----+-----+-----+-----+-----+
| Stude_id | Name | Branch | Mobile_number | Email | City |
+-----+-----+-----+-----+-----+-----+
|      1 | Peter |    NULL | 34556745362 |   NULL | California |
|      2 | Mike  |    NULL | 983635674562 |   NULL | Texas      |
+-----+-----+-----+-----+-----+-----+
```

## How can we drop a column from MySQL table?

Sometimes, we want to remove single or multiple columns from the table. MySQL allows the **ALTER TABLE DROP COLUMN** statement to delete the column from the table. The following are the syntax to do this:

1. **ALTER TABLE** table\_name **DROP COLUMN** column\_name;

In the above,

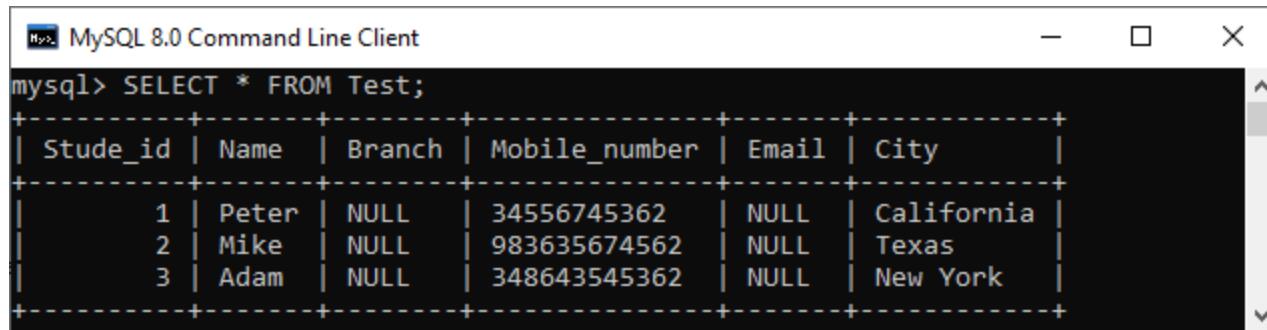
- o First, we need to specify the **table name** from which we want to remove the column.
- o Next, after the **DROP COLUMN** clause, we have to specify the column name that we want to delete from the table. It is to note that the COLUMN keyword is optional in the DROP COLUMN clause.

If we want to remove **multiple columns** from the table, execute the following statements:

1. **ALTER TABLE** table\_name
2. **DROP COLUMN** column\_1,
3. **DROP COLUMN** column\_2,
4. ....;

## MySQL DROP COLUMN Example

This example explains how we can delete a column from the MySQL table. Here, we will take a table "**Test**" that we have created earlier and look like the below image:

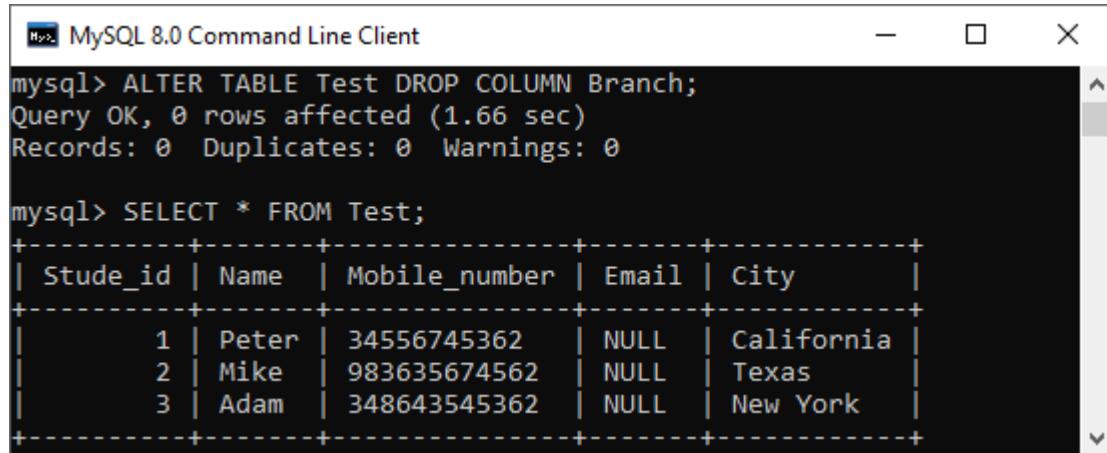


```
MySQL 8.0 Command Line Client
mysql> SELECT * FROM Test;
+-----+-----+-----+-----+-----+-----+
| Stude_id | Name | Branch | Mobile_number | Email | City
+-----+-----+-----+-----+-----+-----+
| 1 | Peter | NULL | 34556745362 | NULL | California |
| 2 | Mike | NULL | 983635674562 | NULL | Texas |
| 3 | Adam | NULL | 348643545362 | NULL | New York |
+-----+-----+-----+-----+-----+-----+
```

Suppose we want to delete a column name "**Branch**" from the Test table. To do this, execute the below statement:

1. **ALTER TABLE Test DROP COLUMN Branch;**

After successful execution, we can verify the result below where a column Branch is deleted from the table:



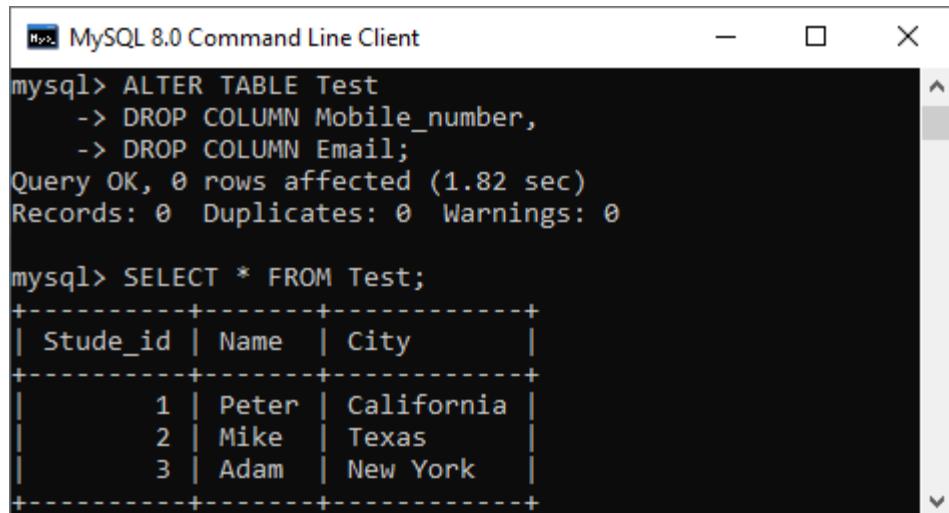
```
MySQL 8.0 Command Line Client
mysql> ALTER TABLE Test DROP COLUMN Branch;
Query OK, 0 rows affected (1.66 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM Test;
+-----+-----+-----+-----+-----+
| Stude_id | Name | Mobile_number | Email | City
+-----+-----+-----+-----+-----+
| 1 | Peter | 34556745362 | NULL | California |
| 2 | Mike | 983635674562 | NULL | Texas |
| 3 | Adam | 348643545362 | NULL | New York |
+-----+-----+-----+-----+-----+
```

In some cases, it is required to remove multiple columns from the table. To do this, we need to execute the below statement:

1. **ALTER TABLE Test**
2. **DROP COLUMN Mobile\_number,**
3. **DROP COLUMN Email;**

The command will delete both columns. We can verify it using the queries given in the below image.



```
MySQL 8.0 Command Line Client
mysql> ALTER TABLE Test
-> DROP COLUMN Mobile_number,
-> DROP COLUMN Email;
Query OK, 0 rows affected (1.82 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM Test;
+-----+-----+-----+
| Stude_id | Name | City
+-----+-----+-----+
| 1 | Peter | California |
| 2 | Mike | Texas |
| 3 | Adam | New York |
+-----+-----+-----+
```

### Remember the following key points before deleting a column from the table:

MySQL works with relational databases where the schema of one table can depend on the columns of another table. So when we remove a column from one table, it will affect all dependent tables also. Consider the below points while removing column:

- When we remove columns from a table, it will affect all associated objects such as triggers, stored procedures, and views. Suppose we delete a column that is referencing in the trigger. After removing the column, the trigger becomes invalid.
- The dropped column depends on other applications code, must also be changed, which takes time and effort.
- When we remove a column from the large table, it will affect the database's performance during removal time.

## MySQL Show Columns

Columns in the table are a series of cells that can store text, numbers, and images. Every column stores one value for each row in a table. When we work with the MySQL server, it is common to display the column information from a particular table. In this section, we are going to discuss how to display or list columns in an existing table.

### MySQL provides two ways for displaying the column information:

1. MySQL SHOW COLUMNS Statement
2. MySQL DESCRIBE Statement

Let us discuss both in detail.

## MySQL SHOW COLUMNS Statement

SHOW COLUMNS statement in MySQL is a more flexible way to display the column information in a given table. It can also support views. Using this statement, we will get only that column information for which we have some privilege.

### Syntax

The following is a syntax to display the column information in a specified table:

1. SHOW [EXTENDED] [**FULL**] {COLUMNS | FIELDS}
2. [{**FROM** | IN} table\_name]
3. [{**FROM** | IN} db\_name]
4. [LIKE 'pattern' | **WHERE** expr]

Let's discuss the syntax parameters in detail.

The **EXTENDED** is an optional keyword to display the information, including hidden columns. MySQL uses hidden columns internally that are not accessible by users.

The **FULL** is also an optional keyword to display the column information, including collation, comments, and the privileges we have for each column.

The **table\_name** is the name of a table from which we are going to show column information.

The **db\_name** is the name of a database containing a table from which we will show column information.

The **LIKE** or **WHERE** clause is used to display only the matched column information in a given table.

**We can also use the alternative of table\_name FROM db\_name syntax as db\_name.tbl\_name. Therefore, the below statements are equivalent:**

1. SHOW COLUMNS **FROM** mytable\_name **FROM** mydb\_name;
2. OR,
3. SHOW COLUMNS **FROM** mydb\_name.mytable\_name;

## SHOW COLUMNS Statement Example

Let us understand how this statement works in [MySQL](#) through various examples.

Suppose we have a table named **student\_info** in a sample database that contains the data as follows:

```

MySQL 8.0 Command Line Client
mysql> SELECT * FROM student_info;
+----+----+----+----+----+----+
| stud_id | stud_code | stud_name | subject | marks | phone |
+----+----+----+----+----+----+
| 1 | 101 | Mark | English | 68 | 34545693537 |
| 2 | 102 | Joseph | Physics | 70 | 98765435659 |
| 3 | 103 | John | Maths | 70 | 97653269756 |
| 4 | 104 | Barack | Maths | 90 | 87698753256 |
| 5 | 105 | Rinky | Maths | 85 | 67531579757 |
| 6 | 106 | Adam | Science | 92 | 79642256864 |
| 7 | 107 | Andrew | Science | 83 | 56742437579 |
| 8 | 108 | Brayan | Science | 85 | 75234165670 |
| 10 | 110 | Alexandar | Biology | 67 | 2347346438 |
+----+----+----+----+----+----+
9 rows in set (0.01 sec)

```

Next, if we want to get the columns information of this table, we can use the statement as follows:

1. mysql> SHOW COLUMNS **FROM** student\_info;

We will see the below output:

```

MySQL 8.0 Command Line Client
mysql> SHOW COLUMNS FROM student_info;
+-----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| stud_id | int | NO | PRI | NULL |       |
| stud_code | varchar(15) | YES |     | NULL |       |
| stud_name | varchar(35) | YES |     | NULL |       |
| subject | varchar(25) | YES |     | NULL |       |
| marks | int | YES |     | NULL |       |
| phone | varchar(15) | YES |     | NULL |       |
+-----+-----+-----+-----+-----+-----+
6 rows in set (1.65 sec)

```

If we want to filter the columns of a table, we need to use the LIKE or [WHERE clause](#) in the statement. See the below query:

1. mysql> SHOW COLUMNS **FROM** student\_info **LIKE** 's%';

This query shows the column information that starts with the letter S only. See the below output:

```

MySQL 8.0 Command Line Client
mysql> SHOW COLUMNS FROM student_info LIKE 's%';
+-----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| stud_id | int | NO | PRI | NULL |       |
| stud_code | varchar(15) | YES |     | NULL |       |
| stud_name | varchar(35) | YES |     | NULL |       |
| subject | varchar(25) | YES |     | NULL |       |
+-----+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)

```

If we want to display hidden column information, we need to add the FULL keyword to the SHOW COLUMNS statement as follows:

1. mysql> SHOW **FULL** COLUMNS **FROM** student\_info;

It returns the below output that displays all columns information of the student\_info table in the sample database.

```

MySQL 8.0 Command Line Client
mysql> SHOW FULL COLUMNS FROM student_info;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| Field | Type | Collation | Null | Key | Default | Extra | Privileges | Comment |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| stud_id | int | utf8mb4_0900_ai_ci | NO | PRI | NULL |       | select,insert,update,references |
| stud_code | varchar(15) | utf8mb4_0900_ai_ci | YES |     | NULL |       | select,insert,update,references |
| stud_name | varchar(35) | utf8mb4_0900_ai_ci | YES |     | NULL |       | select,insert,update,references |
| subject | varchar(25) | utf8mb4_0900_ai_ci | YES |     | NULL |       | select,insert,update,references |
| marks | int | NULL | YES |     | NULL |       | select,insert,update,references |
| phone | varchar(15) | utf8mb4_0900_ai_ci | YES |     | NULL |       | select,insert,update,references |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
6 rows in set (0.00 sec)

```

The **SHOW COLUMNS** statement provides the below information for each column in a given table:

**Field:** It indicates the name of the column in a given table.

**Type:** It indicates the data type of each column.

**Collation:** It is used to sequence the order of a specific character set. Here it indicates the string column for non-binary values and NULL for other columns. We will see this column only when we use the FULL keyword.

**Null:** It indicates the nullability of a column. If a column can store NULL values, it returns YES. And if a column cannot store NULL value, it contains NO value.

**Key:** It indicates the indexing of the columns as PRI, UNI, and MUL. Let us understand this field in detail.

- If we have not specified any key, it means the column is not indexed. Otherwise, index as a secondary column in a multiple-column.
- If the column is specified as a PRI, it means the column is a PRIMARY KEY or one of the fields in a multiple-column PRIMARY KEY.
- If the column is specified as a UNI, it means the column contains a UNIQUE index.
- If the column is specified as a MUL, it means the column is the first column of a non-unique index where we can use a given value multiple times.
- If the column is specified by more than one key-value, this field displays the key which has the highest priority (the key priority is in the order of PRI, UNI, and MUL).

**Default:** It indicates the default value to the column. If the column includes no DEFAULT clause or has an explicit NULL default, it contains a NULL value.

**Extra:** It indicates the additional information related to a given column. This field is non-empty in the following cases:

- If the column is specified with the AUTO\_INCREMENT attribute, its value is filled with auto\_increment.
- If the column is specified with TIMESTAMP or DATETIME that have the ON UPDATE CURRENT\_TIMESTAMP attribute, its value is filled with on update CURRENT\_TIMESTAMP.
- For the generated columns, its value filled with VIRTUAL GENERATED or VIRTUAL STORED.
- If the column contains an expression default value, its value is filled with DEFAULT\_GENERATED.

**Privileges:** It indicates the privileges that we have for the column. We will see this column only when we use the FULL keyword.

**Comment:** It indicates the comment that we have included in the column definition. We will see this column only when we use the FULL keyword.

## MySQL DESCRIBE Statement

DESCRIBE statement in MySQL is also provides information similar to the SHOW COLUMNS command.

### Syntax

The following is the syntax to display the column information in a given table:

1. {DESCRIBE | DESC} table\_name;

In this syntax, the **DESCRIBE** and **DESC** clause return the same result.

### Example

If we want to show column information of **students\_info** table, we can execute the below statement.

1. mysql> DESCRIBE students\_info;

After successful execution, it will give the output as below image:

```

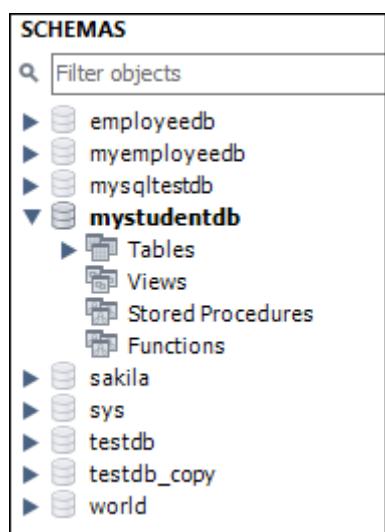
MySQL 8.0 Command Line Client
mysql> DESCRIBE student_info;
+-----+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| stud_id | int    | NO   | PRI  | NULL    |       |
| stud_code | varchar(15) | YES  |       | NULL    |       |
| stud_name | varchar(35) | YES  |       | NULL    |       |
| subject | varchar(25) | YES  |       | NULL    |       |
| marks | int    | YES  |       | NULL    |       |
| phone | varchar(15) | YES  |       | NULL    |       |
+-----+-----+-----+-----+-----+-----+
6 rows in set (0.00 sec)

```

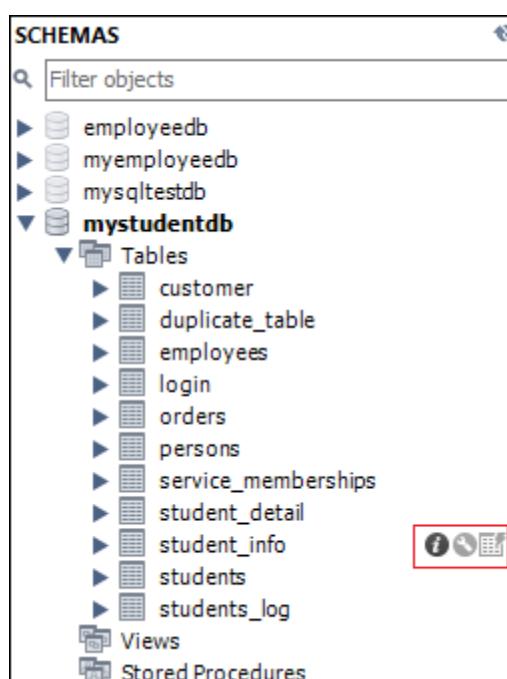
## How to display column information in MySQL Workbench?

We first launch the tool and log in with the username and password to display the given table's column information in MySQL Workbench. Now, we need to do the following steps to show the column information:

1. Go to the **Navigation tab** and click on the **Schema menu** where all the previously created databases available. Select your desired database (for example, **mstudentdb**). It will pop up the following options.



2. Click on the **Tables** that show all tables stored in the **mysqltestdb** database. Select a table whose column information you want to display. Then, mouse over on that table, it will show **three icons**. See the below image:



3. Click the **icon (i)** shown in the red rectangular box. We should get the screen as follows:

Info	Columns	Indexes	Triggers	Foreign keys	Partitions	Grants	DDL
Local instance MySQL80 (2)							
<b>mystudentdb.student_info</b>							
<b>Table Details</b>							
Engine:	InnoDB						
Row format:	Dynamic						
Column count:	6						
Table rows:	8						
AVG row length:	2048						
Data length:	16.0 KiB						
Index length:	0.0 bytes						
Max data length:	0.0 bytes						
Data free:	0.0 bytes						

4. Finally, click on the "**Columns**" menu. We can see the column information as like below output.

Column	Type	Default Value	Nullable	Character Set	Collation	Privileges
marks	int		YES			select,insert,update,references
phone	varchar(15)		YES	utf8mb4	utf8mb4_0900_...	select,insert,update,references
stud_code	varchar(15)		YES	utf8mb4	utf8mb4_0900_...	select,insert,update,references
stud_id	int		NO			select,insert,update,references
stud_name	varchar(35)		YES	utf8mb4	utf8mb4_0900_...	select,insert,update,references
subject	varchar(25)		YES	utf8mb4	utf8mb4_0900_...	select,insert,update,references

## MySQL Rename Column

Sometimes our column name is non-meaningful, so it is required to rename or change the column's name. MySQL provides a useful syntax that can rename one or more columns in the table. Few privileges are essential before renaming the column, such as ALTER and DROP statement privileges.

**MySQL can rename the column name in two ways:**

1. Using the CHANGE statement
2. Using the RENAME statement

### Using the CHANGE Statement:

The following are the syntax that illustrates the column rename using the CHANGE statement:

1. **ALTER TABLE** table\_name
2. **CHANGE COLUMN** old\_column\_name new\_column\_name Data Type;

In this syntax, we can see that we may require re-specification of all the column attributes. This syntax can also allow us to change the column's data types. But, sometimes the CHANGE statement might have the following disadvantages:

- o All information of column attributes might not be available to the application for renaming.
- o There is a risk of accidental data type change that might result in the application's data loss.

### Example

Let us understand how the CHANGE statement works in [MySQL](#) to rename a column through the various examples. Suppose we have a table named **balance** that contains the data as follows:

```
MySQL 8.0 Command Line Client
mysql> SELECT * FROM balance;
+----+-----+-----+
| id | account_num | balance |
+----+-----+-----+
| 1  | 1030       | 50000.00 |
| 2  | 2035       | 230000.00 |
| 3  | 5564       | 125000.00 |
| 4  | 4534       | 8000.00   |
| 5  | 7648       | 45000.00 |
+----+-----+-----+
5 rows in set (0.00 sec)
```

Due to some reason, we need to change the **column name account\_num along with its data type**. In that case, we first check the structure of the table using the **DESCRIBE** statement as follows:

```
MySQL 8.0 Command Line Client
mysql> DESCRIBE balance;
+-----+-----+-----+-----+-----+-----+
| Field | Type  | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id    | int   | NO   | PRI | NULL    | auto_increment |
| account_num | int | YES  |     | NULL    |           |
| balance | float(10,2) | YES |     | NULL    |           |
+-----+-----+-----+-----+-----+-----+
3 rows in set (0.13 sec)
```

In this image, we can see that the data type of column name **account\_num** is **int**. And we want to change this column name as **account\_no** and its data type as **int to varchar**. Thus, we can execute the below statement to do this:

1. mysql> **ALTER TABLE** balance
2. **CHANGE COLUMN** account\_num account\_no **VARCHAR(25)**;

After executing the above command, we can verify it by using the DESCRIBE statement again. In the below image, the column name **account\_num** and its data type have changed successfully.

```
MySQL 8.0 Command Line Client
mysql> ALTER TABLE balance
-> CHANGE COLUMN account_num account_no VARCHAR(25);
Query OK, 5 rows affected (1.97 sec)
Records: 5  Duplicates: 0  Warnings: 0

mysql> DESCRIBE balance;
+-----+-----+-----+-----+-----+-----+
| Field | Type  | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id    | int   | NO   | PRI | NULL    | auto_increment |
| account_no | varchar(25) | YES  |     | NULL    |           |
| balance | float(10,2) | YES |     | NULL    |           |
+-----+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

## Using the RENAME Statement:

To remove the drawback of a **CHANGE** statement, MySQL proposed the following syntax that illustrates the changing of the column name using a **RENAME** statement:

1. **ALTER TABLE** table\_name
2. **RENAME COLUMN** old\_column\_name **TO** new\_column\_name;

## Example

Let us understand how the **RENAME** statement works in MySQL to change the column name through the various examples. Suppose we have a table named **customer** that contains the following data:

The screenshot shows the MySQL 8.0 Command Line Client interface. A query is run: `SELECT * FROM customer;`. The result is a table with columns: id, customer\_name, account, and email. The data rows are: (1, Stephen, 1030, stephen@javatpoint.com), (2, Jenifer, 2035, jenifer@javatpoint.com), (3, Mathew, 5564, mathew@javatpoint.com), (4, Smith, 4534, smith@javatpoint.com), and (5, david, 7648, david@javatpoint.com). A message at the bottom says "5 rows in set (0.00 sec)".

Suppose we want to change the column name account with **account\_no** without changing its data types. We can do this by executing the below statement:

1. mysql> **ALTER TABLE** customer RENAME **COLUMN** account **to** account\_no;

After executing the above command, we can verify it by using the **SELECT statement** again. In the below image, the column name account has changed successfully.

The screenshot shows the MySQL 8.0 Command Line Client interface. First, an `ALTER TABLE customer RENAME COLUMN account to account_no;` command is run. It returns "Query OK, 0 rows affected (0.12 sec)" and "Records: 0 Duplicates: 0 Warnings: 0". Then, a `SELECT * FROM customer;` command is run, which returns the same table structure and data as the first screenshot, confirming the column name change.

## Renaming Multiple Columns

MySQL also allows us to change the multiple column names within a single statement. If we want to rename multiple column names, we might use the below syntax:

1. **ALTER TABLE** table\_name
2. **CHANGE** old\_column\_name1 new\_column\_name1 Data Type,
3. **CHANGE** old\_column\_name2 new\_column\_name2 Data Type,
4. ...
5. ...
6. **CHANGE** old\_column\_nameN new\_column\_nameN Data Type;

OR

1. **ALTER TABLE** table\_name
2. **RENAME COLUMN** old\_column\_name1 **TO** new\_column\_name1,
3. **RENAME COLUMN** old\_column\_name2 **TO** new\_column\_name2,
4. ...
5. ...
6. **RENAME COLUMN** old\_column\_nameN **TO** new\_column\_nameN;

## Example

Suppose we want to change **column names id and customer\_name** from the **customer table**. To change multiple column names within a single statement, we can use the statement as follows:

1. mysql> **ALTER TABLE** customer
2. **CHANGE** id cust\_id **int**,
3. **CHANGE** customer\_name cust\_name **varchar(45)**;

After executing the above command, we can verify it by using the **SELECT statement** again. In the below image, the column name id and customer\_name have changed successfully:

```
MySQL 8.0 Command Line Client
mysql> ALTER TABLE customer
-> CHANGE id cust_id int,
-> CHANGE customer_name cust_name varchar(45);
Query OK, 5 rows affected (1.39 sec)
Records: 5  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM customer;
+-----+-----+-----+-----+
| cust_id | cust_name | account_no | email
+-----+-----+-----+-----+
|      1 | Stephen   |      1030 | stephen@javatpoint.com
|      2 | Jenifer   |      2035 | jenifer@javatpoint.com
|      3 | Mathew    |      5564 | mathew@javatpoint.com
|      4 | Smith     |      4534 | smith@javatpoint.com
|      5 | david     |      7648 | david@javatpoint.com
+-----+-----+-----+-----+
```

Let us again change the currently modifying column name through the RENAME COLUMN statement as follows:

1. mysql> **ALTER TABLE** customer
2. RENAME **COLUMN** cust\_id **TO** id,
3. RENAME **COLUMN** cust\_name **TO** customer\_name;

After executing the above command, we can verify it by using the DESCRIBE statement again. In the below image, the **column name cust\_id and cust\_name** have changed successfully:

```
MySQL 8.0 Command Line Client
mysql> ALTER TABLE customer
-> RENAME COLUMN cust_id TO id,
-> RENAME COLUMN cust_name TO customer_name;
Query OK, 0 rows affected (0.19 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> DESCRIBE customer;
+-----+-----+-----+-----+-----+-----+
| Field   | Type    | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id      | int     | NO   | PRI | NULL    |       |
| customer_name | varchar(45) | YES  |     | NULL    |       |
| account_no | int     | YES  |     | NULL    |       |
| email    | varchar(55) | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
```

In this article, we have learned an introduction of the MySQL RENAME column and how to change the column name in a specified table, along with a query example for better understanding.

## MySQL View

A view is a database object that has no values. Its contents are based on the base table. It contains rows and columns similar to the real table. In MySQL, the View is a **virtual table** created by a query by joining one or more tables. It is operated similarly to the base table but does not contain any data of its own. The View and table have one main difference that the views are definitions built on top of other tables (or views). If any changes occur in the underlying table, the same changes reflected in the View also.

[MySQL](#) allows us to create a view in mainly two ways:

1. MySQL Command line client
2. MySQL Workbench

Let us discuss both in detail.

## MySQL Command Line Client

We can create a new view by using the **CREATE VIEW** and **SELECT** statement. [SELECT statements](#) are used to take data from the source table to make a VIEW.

## Syntax

Following is the syntax to create a view in MySQL:

1. **CREATE [OR REPLACE] VIEW** view\_name **AS**
2. **SELECT** columns
3. **FROM** tables
4. [**WHERE** conditions];

## Parameters:

The view syntax contains the following parameters:

**OR REPLACE:** It is optional. It is used when a VIEW already exists. If you do not specify this clause and the VIEW already exists, the CREATE VIEW statement will return an error.

**view\_name:** It specifies the name of the VIEW that you want to create in MySQL.

**WHERE conditions:** It is also optional. It specifies the conditions that must be met for the records to be included in the VIEW.

## Example

Let us understand it with the help of an example. Suppose our database has a table **course**, and we are going to create a view based on this table. Thus, the below example will create a VIEW name "**trainer**" that creates a virtual table made by taking data from the table courses.

1. **CREATE VIEW** trainer **AS**
2. **SELECT** course\_name, trainer
3. **FROM** courses;

Once the execution of the CREATE VIEW statement becomes successful, MySQL will create a view and stores it in the database.

MySQL 8.0 Command Line Client

```
mysql> SHOW TABLES;
+-----+
| Tables_in_testdb |
+-----+
| contact          |
| courses          |
| customer         |
+-----+
3 rows in set (0.13 sec)

mysql> CREATE VIEW trainer AS
    -> SELECT course_name, trainer
    -> FROM courses;
Query OK, 0 rows affected (0.33 sec)
```

## To see the created VIEW

We can see the created view by using the following syntax:

1. **SELECT \* FROM** view\_name;

Let's see how it looks the created VIEW:

1. **SELECT \* FROM** trainer;

MySQL 8.0 Command Line Client

```
mysql> SELECT * FROM trainer;
+-----+-----+
| course_name | trainer |
+-----+-----+
| Java        | Mike   |
| Python      | James  |
| Android     | Robin  |
| Hadoop      | Stephen|
| Testing     | Micheal|
+-----+-----+
5 rows in set (0.05 sec)
```

**NOTE:** It is essential to know that a view does not store the data physically. When we execute the **SELECT** statement for the view, MySQL uses the query specified in the view's definition and produces the output. Due to this feature, it is sometimes referred to as a virtual table.

## MySQL Update VIEW

In MYSQL, the **ALTER VIEW** statement is used to modify or update the already created VIEW without dropping it.

### Syntax:

Following is the syntax used to update the existing view in MySQL:

1. **ALTER VIEW** view\_name **AS**
2. **SELECT** columns
3. **FROM** table
4. **WHERE** conditions;

### Example:

The following example will alter the already created VIEW name "trainer" by adding a new column.

1. **ALTER VIEW** trainer **AS**
2. **SELECT** id, course\_name, trainer
3. **FROM** courses;

Once the execution of the **ALTER VIEW** statement becomes successful, MySQL will update a view and stores it in the database. We can see the altered view using the **SELECT** statement, as shown in the output:

The screenshot shows the MySQL 8.0 Command Line Client interface. The command line displays two SQL statements: an **ALTER VIEW** statement to modify the 'trainer' view, and a **SELECT** statement to retrieve data from the modified view. The output shows the view was successfully altered and now contains the 'course\_name' column, which is populated with values like Java, Python, Android, Hadoop, and Testing, along with the 'id' and 'trainer' columns.

```
MySQL 8.0 Command Line Client
mysql> ALTER VIEW trainer AS
    -> SELECT id, course_name, trainer
    -> FROM courses;
Query OK, 0 rows affected (0.22 sec)

mysql> SELECT * FROM trainer;
+---+-----+-----+
| id | course_name | trainer |
+---+-----+-----+
| 1  | Java        | Mike   |
| 2  | Python       | James  |
| 3  | Android      | Robin  |
| 4  | Hadoop       | Stephen|
| 5  | Testing      | Micheal|
+---+-----+-----+
```

## MySQL Drop VIEW

We can drop the existing VIEW by using the **DROP VIEW** statement.

### Syntax:

The following is the syntax used to delete the view:

1. **DROP VIEW** [IF EXISTS] view\_name;

### Parameters:

**view\_name:** It specifies the name of the VIEW that we want to drop.

**IF EXISTS:** It is optional. If we do not specify this clause and the VIEW doesn't exist, the **DROP VIEW** statement will return an error.

### Example:

Suppose we want to delete the view "**trainer**" that we have created above. Execute the below statement:

1. **DROP VIEW** trainer;

After successful execution, it is required to verify the view is available or not as below:

```
MySQL 8.0 Command Line Client
mysql> DROP VIEW trainer;
Query OK, 0 rows affected (0.18 sec)

mysql> SELECT * FROM trainer;
ERROR 1146 (42S02): Table 'testdb.trainer' doesn't exist
```

## MySQL Create View with JOIN Clause

Here, we will see the complex example of view creation that involves multiple tables and uses a **join** clause.

Suppose we have two sample table as shown below:

Table: course			Table: contact		
id	course_name	trainer	id	email	mobile
1	Java	Mike	1	mike@javatpoint.com	4354657678987
2	Python	James	2	james@javatpoint.com	3434676587767
3	Android	Robin	3	robin@javatpoint.com	8987674541123
4	Hadoop	Stephen	4	stephen@javatpoint.com	6767645458795
5	Testing	Micheal	5	micheal@javatpoint.com	2345476779874

Now execute the below statement that will create a view Trainer along with the join statement:

1. **CREATE VIEW** Trainer
2. **AS SELECT** c.course\_name, c.trainer, t.email
3. **FROM** courses c, contact t
4. **WHERE** c.id = t.id;

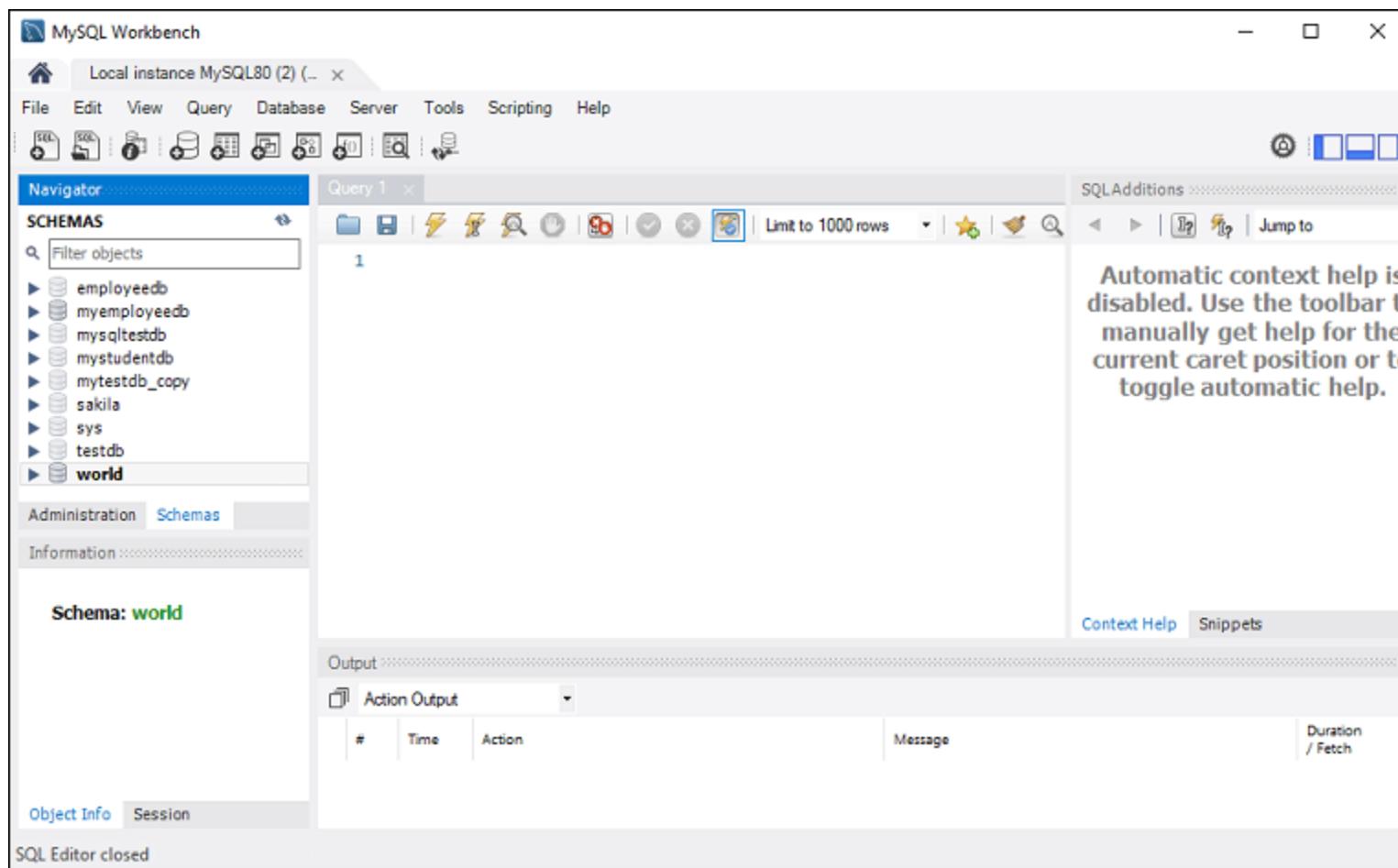
We can verify the view using the SELECT statement shown in the below image:

```
MySQL 8.0 Command Line Client
mysql> CREATE VIEW Trainer
-> AS SELECT c.course_name, c.trainer, t.email
-> FROM courses c, contact t
-> WHERE c.id= t.id;
Query OK, 0 rows affected (0.29 sec)

mysql> SELECT * FROM Trainer;
+-----+-----+-----+
| course_name | trainer | email      |
+-----+-----+-----+
| Java        | Mike    | mike@javatpoint.com
| Python       | James   | james@javatpoint.com
| Android     | Robin   | robin@javatpoint.com
| Hadoop      | Stephen | stephen@javatpoint.com
| Testing     | Micheal | micheal@javatpoint.com
+-----+-----+-----+
5 rows in set (0.00 sec)
```

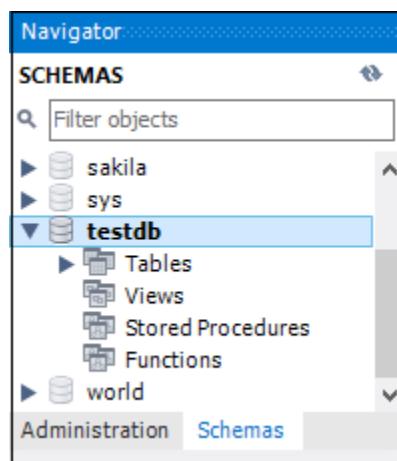
## Create View using MySQL Workbench

To create a view in the database using this tool, we first need to launch the [MySQL Workbench](#) and log in with the **username** and **password** to the MySQL server. It will show the following screen:

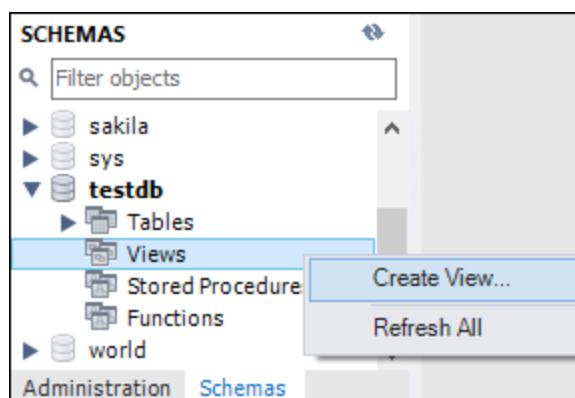


Now do the following steps for database deletion:

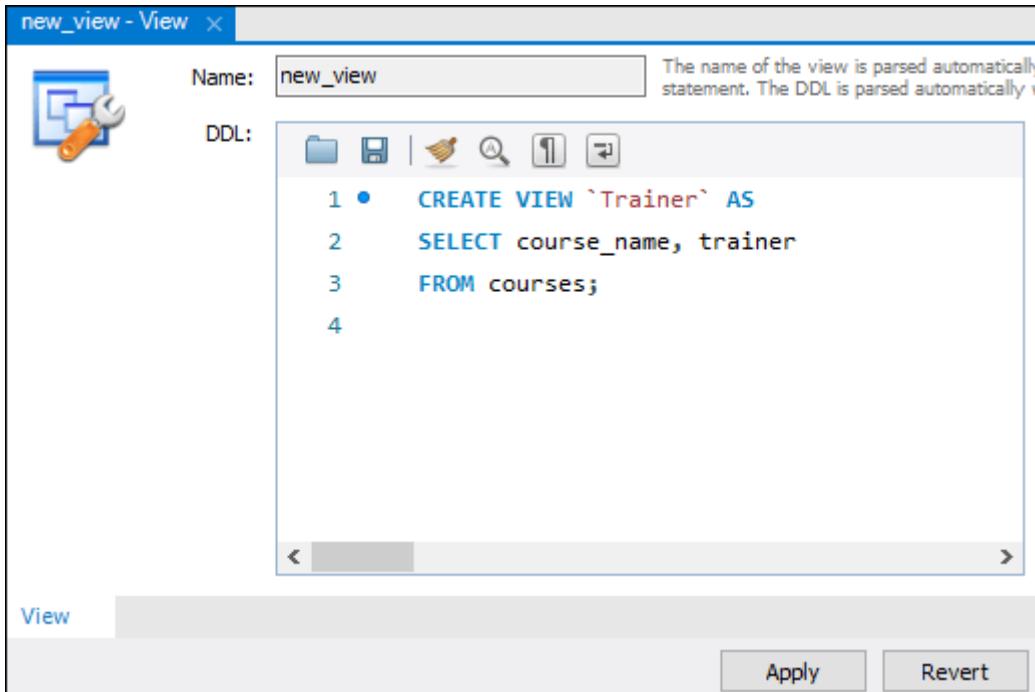
1. Go to the Navigation tab and click on the **Schema menu**. Here, we can see all the previously created databases. Select any database under the Schema menu, for example, **testdb**. It will pop up the option that can be shown in the following screen.



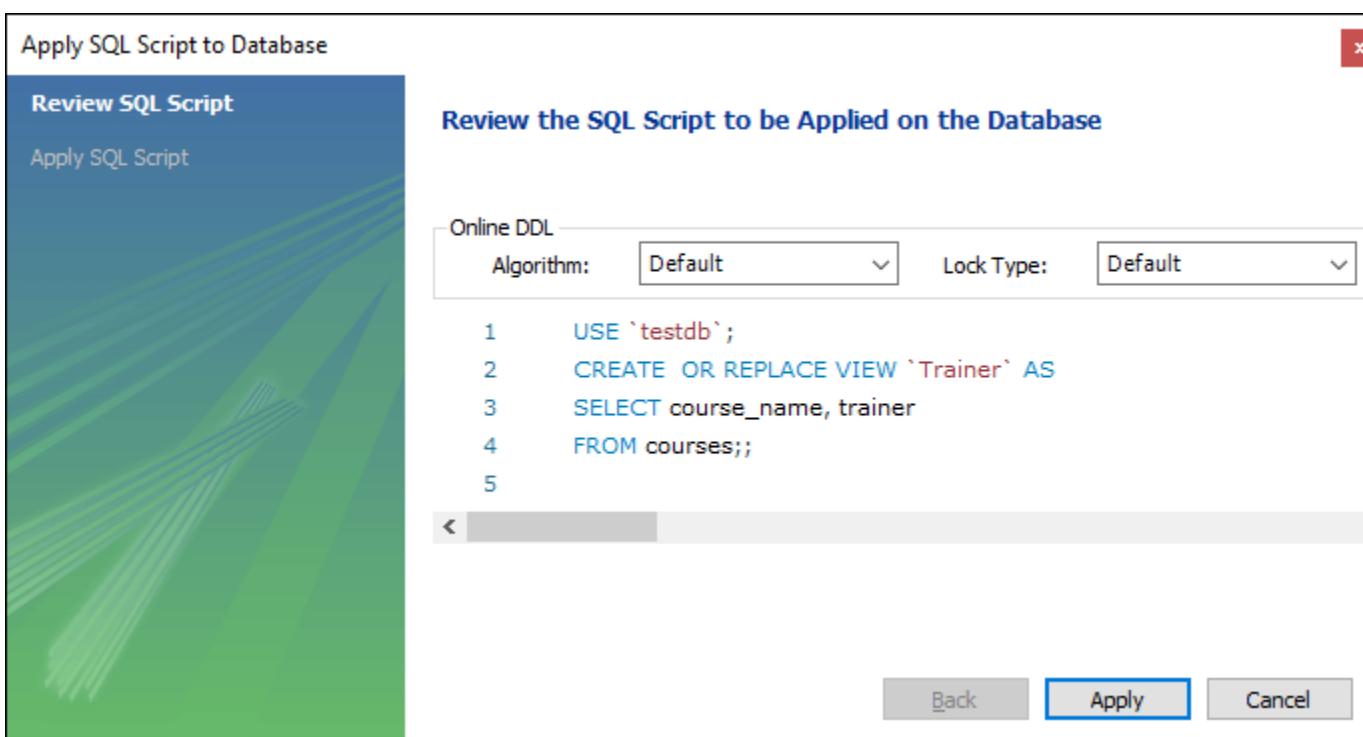
2. Next, we need to right-click on the view option, and a new pop up screen will come:



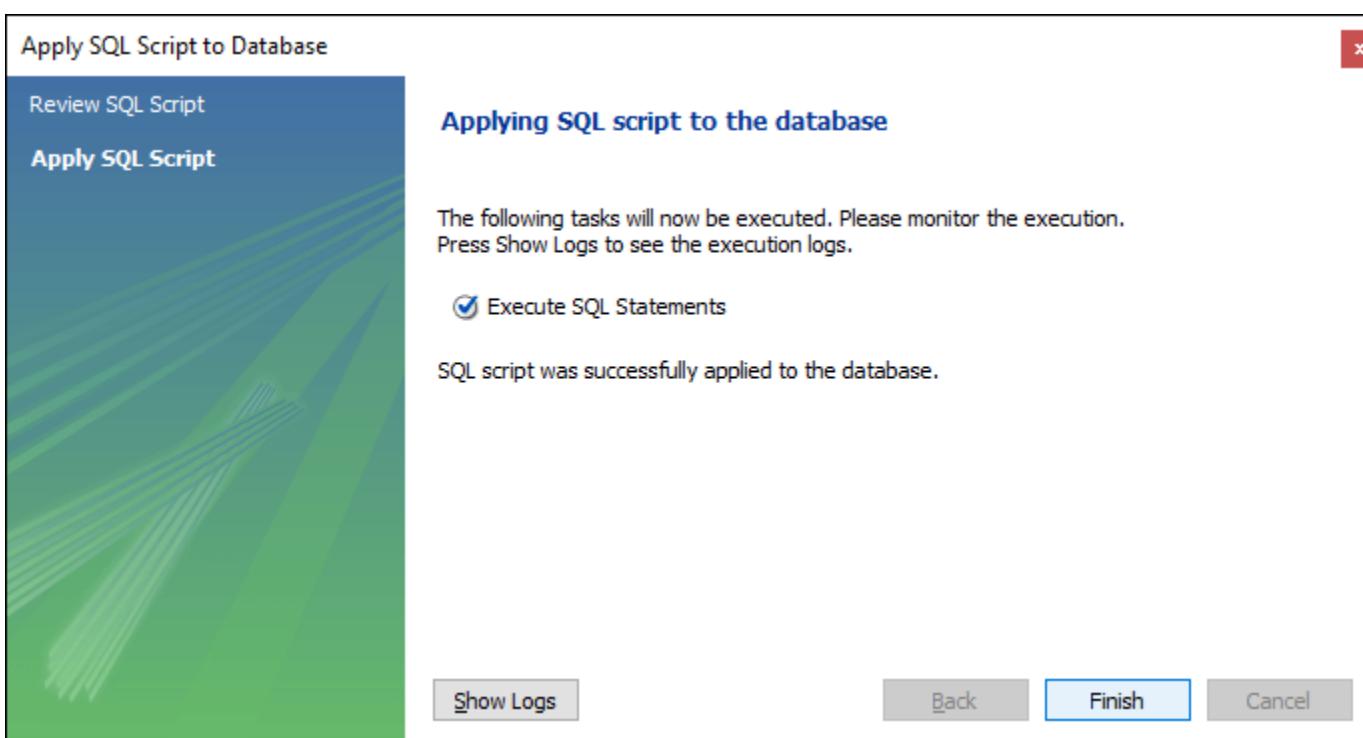
3. As soon as we select the "**Create View**" option, it will give the below screen where we can write our own view.



4. After completing the script's writing, click on the **Apply** button, we will see the following screen:



5. In this screen, we will review the script and click the **Apply** button on the database



6. Finally, click on the **Finish** button to complete the view creation. Now, we can verify the view as below:

The screenshot shows the MySQL Workbench interface. On the left, the Navigator pane displays the 'SCHEMAS' tree with 'testdb' selected. Under 'testdb', there are 'Tables' and 'Views'. A 'trainer' view is selected, highlighted with a blue border. The main workspace shows a SQL editor with the query `SELECT * FROM testdb.trainer;` and a Result Grid displaying the following data:

	course_name	trainer
▶	Java	Mike
	Python	James
	Android	Robin
	Hadoop	Stephen
	Testing	Micheal

## Why we use View?

MySQL view provides the following advantages to the user:

### Simplify complex query

It allows the user to simplify complex queries. If we are using the complex query, we can create a view based on it to use a simple SELECT statement instead of typing the complex query again.

### Increases the Re-usability

We know that View simplifies the complex queries and converts them into a single line of code to use VIEWS. Such type of code makes it easier to integrate with our application. This will eliminate the chances of repeatedly writing the same formula in every query, making the code reusable and more readable.

### Help in Data Security

It also allows us to show only authorized information to the users and hide essential data like personal and banking information. We can limit which information users can access by authoring only the necessary data to them.

### Enable Backward Compatibility

A view can also enable the backward compatibility in legacy systems. Suppose we want to split a large table into many smaller ones without affecting the current applications that reference the table. In this case, we will create a view with the same name as the real table so that the current applications can reference the view as if it were a table.

## MySQL Table Locking

A lock is a mechanism associated with a table used to restrict the unauthorized access of the data in a table. **MySQL allows a client session to acquire a table lock explicitly to cooperate with other sessions to access the table's data.** MySQL also allows table locking to prevent it from unauthorized modification into the same table during a specific period.

A session in MySQL can acquire or release locks on the table only for itself. Therefore, one session cannot acquire or release table locks for other sessions. It is to note that we must have a TABLE LOCK and SELECT privileges for table locking.

Table Locking in MySQL is mainly **used to solve concurrency problems**. It will be used while running a transaction, i.e., first read a value from a table (database) and then write it into the table (database).

[MySQL](#) provides **two types of locks** onto the table, which are:

**READ LOCK:** This lock allows a user to only read the data from a table.

**WRITE LOCK:** This lock allows a user to do both reading and writing into a table.

It is to note that the default storage engine used in MySQL is InnoDB. The InnoDB storage engine does not require table locking manually because MySQL automatically uses row-level locking for InnoDB tables. Therefore, we can do multiple transactions on the same table simultaneously to read and write operations without making each other wait. All other storage engines use table locking in MySQL.

Before understanding the table locking concept, first, we will create a new table named "**info\_table**" using the statement as follows:

```
1. CREATE TABLE info_table (
2.     Id INT NOT NULL AUTO_INCREMENT,
3.     Name VARCHAR(50) NOT NULL,
4.     Message VARCHAR(80) NOT NULL,
5.     PRIMARY KEY (Id)
6. );
```

## MySQL LOCK TABLES Statement

The following is the syntax that allows us to acquire a table lock explicitly:

```
1. LOCK TABLES table_name [READ | WRITE];
```

In the above syntax, we have specified the **table name** on which we want to acquire a lock after the **LOCK TABLES** keywords. We can specify the **lock type**, either READ or WRITE.

We can also lock more than one table in MySQL by using a list of comma-separated table's names with lock types. See the below syntax:

```
1. LOCK TABLES tab_name1 [READ | WRITE],
2.         tab_name2 [READ | WRITE],..... ;
```

## MySQL UNLOCK TABLES Statement

The following is the syntax that allows us **to release a lock** for a table in MySQL:

```
1. mysql> UNLOCK TABLES;
```

## LOCK TYPES

Let us understand the lock types in detail.

### READ Locks

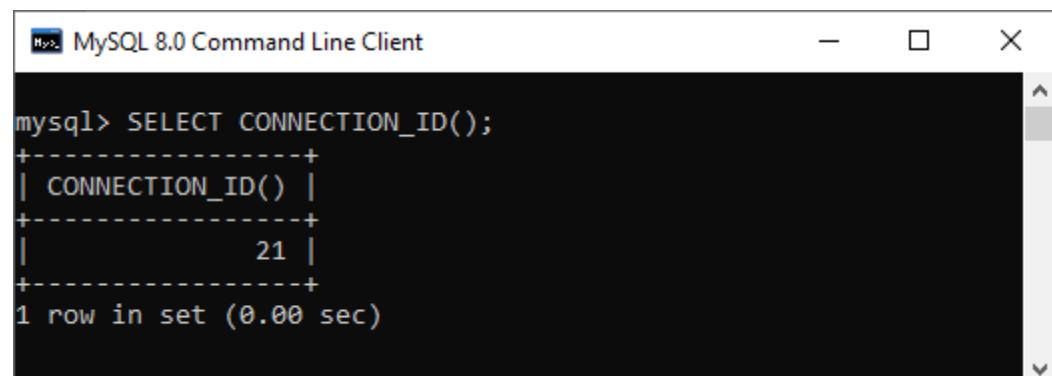
The following are the features of the READ lock:

- At the same time, MySQL allows multiple sessions to acquire a READ lock for a table. And all other sessions can read the table without acquiring the lock.
- If the session holds the READ lock on a table, they cannot perform a write operation on it. It is because the READ lock can only read data from the table. All other sessions that do not acquire a READ lock are not able to write data into the table without releasing the READ lock. The write operations go into the waiting states until we have released the READ lock.
- When the session is terminated normally or abnormally, MySQL implicitly releases all types of locks on to the table. This feature is also relevant for the WRITE lock.

Let us take an example to see how READ locks work in MySQL with the given scenario. We will first connect to the database and use the **CONNECTION\_ID()** function that gives the current connection id in the first session as follows:

```
1. mysql> SELECT CONNECTION_ID();
```

See the below output:



The screenshot shows the MySQL 8.0 Command Line Client window. The command `SELECT CONNECTION_ID();` is entered in the prompt. The output shows a single row with the value 21 in the column labeled `CONNECTION_ID()`. The footer of the window indicates `1 row in set (0.00 sec)`.

CONNECTION_ID()
21

Next, we will insert few rows into the **info\_table** using the below statement:

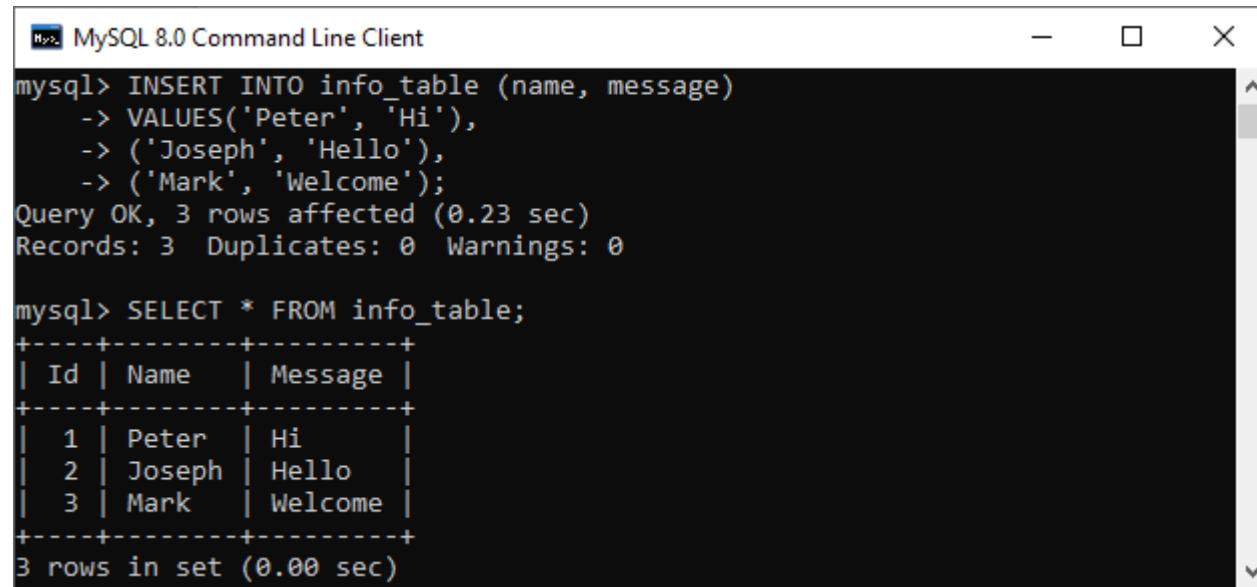
```
1. mysql> INSERT INTO info_table (name, message)
```

2. **VALUES**('Peter', 'Hi'),
3. ('Joseph', 'Hello'),
4. ('Mark', 'Welcome');

Now, verify the data into the table using the below statement:

1. mysql> **SELECT \* FROM** info\_table;

We should see the output as follows:



```
MySQL 8.0 Command Line Client
mysql> INSERT INTO info_table (name, message)
-> VALUES('Peter', 'Hi'),
-> ('Joseph', 'Hello'),
-> ('Mark', 'Welcome');
Query OK, 3 rows affected (0.23 sec)
Records: 3 Duplicates: 0 Warnings: 0

mysql> SELECT * FROM info_table;
+----+-----+-----+
| Id | Name  | Message |
+----+-----+-----+
| 1  | Peter | Hi     |
| 2  | Joseph | Hello   |
| 3  | Mark  | Welcome |
+----+-----+-----+
3 rows in set (0.00 sec)
```

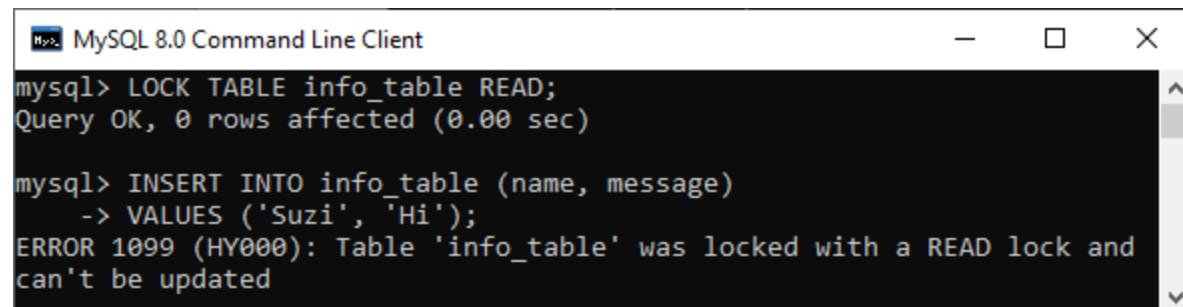
Now, we will execute the **LOCK TABLE** statement to acquire a lock onto the table:

1. mysql> **LOCK TABLE** info\_table **READ**;

After that, we will try to insert a new record into the info\_table as follows:

1. mysql> **INSERT INTO** info\_table (**name**, message)
2. **VALUES** ('Suzi', 'Hi');

We will get the below output where MySQL issues the following message "**Table 'info\_table' was locked with a READ lock and can't be updated**".

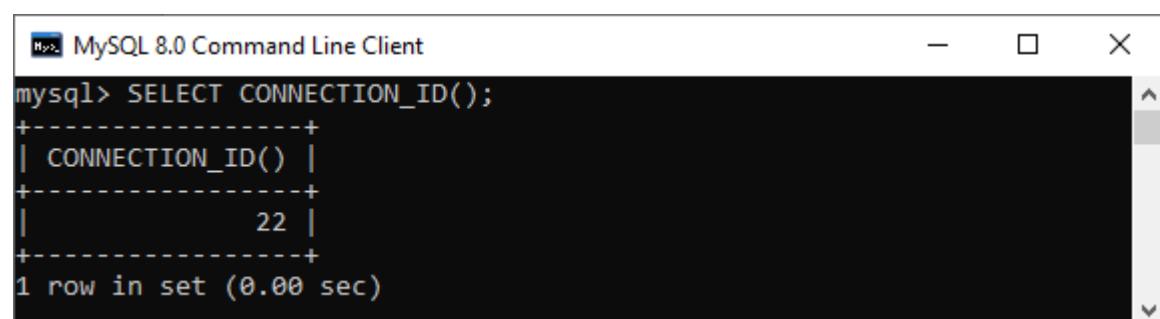


```
MySQL 8.0 Command Line Client
mysql> LOCK TABLE info_table READ;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO info_table (name, message)
-> VALUES ('Suzi', 'Hi');
ERROR 1099 (HY000): Table 'info_table' was locked with a READ lock and
can't be updated
```

Thus, we can see that once the READ lock is acquired on to the table, we cannot write data to the table in the same session.

Now, we will check how the READ lock work from a different session. First, we will connect to the database and see the connection id:



```
MySQL 8.0 Command Line Client
mysql> SELECT CONNECTION_ID();
+-----+
| CONNECTION_ID() |
+-----+
|      22       |
+-----+
1 row in set (0.00 sec)
```

Next, we will query data from the info\_table that returns the output as follows:

```

MySQL 8.0 Command Line Client
mysql> SELECT * FROM info_table;
+----+----+-----+
| Id | Name | Message |
+----+----+-----+
| 1  | Peter | Hi      |
| 2  | Joseph | Hello   |
| 3  | Mark  | Welcome |
| 4  | Jenni | Hey    |
+----+----+-----+
4 rows in set (0.00 sec)

```

Then, insert some rows into this table as follows:

1. mysql> **INSERT INTO** info\_table (**name**, message)
2. **VALUES** ('Stephen', 'Hello');

We should see the output as follows:

Time	Action	Message	Duration / Fetch
14:15:26	INSERT INTO info_table (name, message) VALUES ('Stephen', 'Hell...')	Running...	?

In the above output, we can see that the insert operation from the second session is in the **waiting state**. It is due to the READ lock, which is already acquired on the table by the first session and has not been released yet.

We can see the detailed information about them using the **SHOW PROCESSLIST** statement in the first session. See the below output:

ID	User	Host	db	Command	Time	State	Info
4	event_scheduler	localhost	NULL	Daemon	1091832	Waiting on empty queue	NULL
29	root	localhost:61420	employeedb	Sleep	81		NULL
30	root	localhost:61421	testdb	Query	0	starting	SHOW PROCESSLIST
31	root	localhost:61422	testdb	Query	21	Waiting for table metadata lock	INSERT INTO info_table (name, message) VALU...

Finally, we need to release the lock by using the **UNLOCK TABLES** statement in the first session. Now, we are able to execute the INSERT operation in the second session.

## Write Locks

The following are the features of a WRITE lock:

- o It is the session that holds the lock of a table and can read and write data both from the table.
- o It is the only session that accesses the table by holding a lock. And all other sessions cannot access the data of the table until the WRITE lock is released.

Let us take an example to see how WRITE locks works in MySQL with the given scenario. In the first session, we will acquire a WRITE lock using the below statement:

1. mysql> **LOCK TABLE** info\_table **WRITE**;

Then, we will insert a new record into the info\_table as follows:

1. mysql> **INSERT INTO** info\_table (**name**, message)
2. **VALUES** ('Stephen', 'How R U');

The above statement worked. Now, we can verify the output using the SELECT statement:

```

MySQL 8.0 Command Line Client
mysql> LOCK TABLE info_table WRITE;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO info_table (name, message)
    -> VALUES ('Stephen', 'How R U');
Query OK, 1 row affected (0.03 sec)

mysql> SELECT * FROM info_table;
+----+-----+-----+
| Id | Name | Message |
+----+-----+-----+
| 1  | Peter | Hi      |
| 2  | Joseph | Hello   |
| 3  | Mark  | Welcome |
| 8  | Stephen | How R U |
+----+-----+-----+
4 rows in set (0.00 sec)

```

Again, we will attempt to access (read/write) the table from the second session:

1. **INSERT INTO** info\_table (**name**, message)
2. **VALUES** ('George', 'Welcome');
- 3.
4. **SELECT \* FROM** info\_table;

We can see that these operations are put into a waiting state. See the detailed information about them using the SHOW PROCESSLIST statement:

<b>Id</b>	<b>User</b>	<b>Host</b>	<b>db</b>	<b>Command</b>	<b>Time</b>	<b>State</b>	<b>Info</b>
4	event_scheduler	localhost	NULL	Daemon	1095111	Waiting on empty queue	NULL
29	root	localhost:61420	employeedb	Sleep	67		NULL
30	root	localhost:61421	testdb	Query	74	Waiting for table metadata lock	SELECT * FROM info_table LIMIT 0, 1000
34	root	localhost:61455	testdb	Sleep	403		NULL
35	root	localhost:61458	testdb	Query	36	Waiting for table metadata lock	INSERT INTO info_table (name, message)...

Finally, we will release the lock from the first session. Now, we can execute the pending operations.

## Read vs. Write Lock

- Read lock is similar to "**shared**" locks because multiple threads can acquire it at the same time.
- Write lock is an "**exclusive**" locks because another thread cannot read it.
- We cannot provide read and write locks both on the table at the same time.
- Read lock has a **low priority** than Write lock, which ensures that updates are made as soon as possible.

## MySQL Lock Account

A lock is a mechanism used to prevent unauthorized modifications into our database. It is essential to the security of our database. In this article, we are going to learn how to use the **CREATE USER... ACCOUNT LOCK** and **ALTER TABLE... ACCOUNT LOCK** statements for locking the user accounts in the MySQL server.

We can lock the user accounts by using the [CREATE USER... ACCOUNT LOCK](#) statement as follows:

1. **CREATE USER** account\_name IDENTIFIED BY 'password' ACCOUNT LOCK;

The **ACCOUNT LOCK clause** specifies the initial locking state for a new user account. If we do not specify this clause with the statement, then a newly created user is stored in an unlocked state by default. If we have enabled the **validate\_password** plugin during user creation, we cannot create an account without a password, even if it is locked.

[MySQL](#) also allows us to provide the lock on an existing user account by using the **ALTER USER... ACCOUNT LOCK** statement as follows:

1. **ALTER USER** account\_name IDENTIFIED BY 'password' ACCOUNT LOCK;

The account locking state remains unchanged if we do not specify the ACCOUNT LOCK clause with the statement.

MySQL uses the **account\_locked column of the mysql.user system table** to store the account locking state. We can use the **SHOW CREATE USER** statement to validate whether the account is unlocked or locked. If this column value is **Y**, it means the account is locked. If it contains **N**, it means the account is unlocked.

If we will try to access the locked account, the attempt fails, and MySQL issues an error that writes the below message to the error log:

1. Access denied for user 'user\_name'@'host\_name'.
2. An account is locked.

## MySQL User Account Locking Examples

Let us understand the working of locking user accounts through examples:

### 1. Using ACCOUNT LOCK clause for locking a new user account

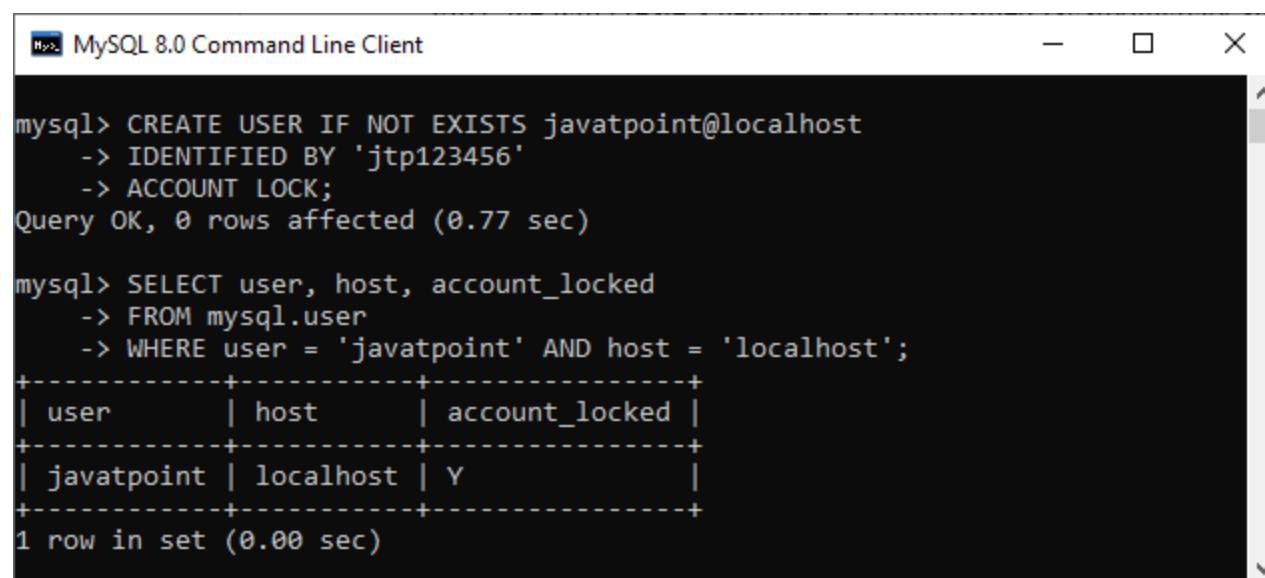
First, we will create a new user account named **javatpoint@localhost** in the locked state using the below statement:

1. mysql> **CREATE USER** IF NOT EXISTS javatpoint@localhost
2. IDENTIFIED BY 'jtp123456'
3. ACCOUNT LOCK;

Next, we will execute the below statement to show the user account and its status:

1. mysql> **SELECT** user, host, account\_locked
2. **FROM** mysql.user
3. **WHERE** user = 'javatpoint' AND host = 'localhost';

We should get the below output:



```
MySQL 8.0 Command Line Client

mysql> CREATE USER IF NOT EXISTS javatpoint@localhost
    -> IDENTIFIED BY 'jtp123456'
    -> ACCOUNT LOCK;
Query OK, 0 rows affected (0.77 sec)

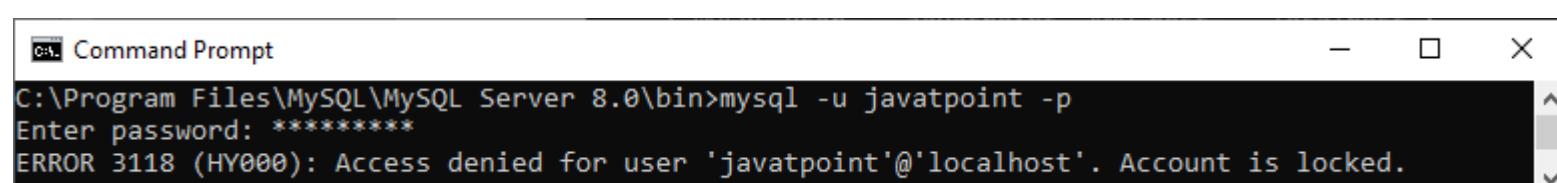
mysql> SELECT user, host, account_locked
    -> FROM mysql.user
    -> WHERE user = 'javatpoint' AND host = 'localhost';
+-----+-----+-----+
| user      | host      | account_locked |
+-----+-----+-----+
| javatpoint | localhost | Y              |
+-----+-----+-----+
1 row in set (0.00 sec)
```

IN this output, we can see that the **account\_locked** column in the **mysql.user** system table indicates Y. It means the username **javatpoint** is locked on the server.

If we try to access the user account **javatpoint** to connect to the MySQL Server, the attempt fails, and we will receive an error:

1. mysql -u javatpoint -p
2. Enter **password**: \*\*\*\*\*

Here is the error message:



```
Command Prompt

C:\Program Files\MySQL\MySQL Server 8.0\bin>mysql -u javatpoint -p
Enter password: *****
ERROR 3118 (HY000): Access denied for user 'javatpoint'@'localhost'. Account is locked.
```

### 2. MySQL account locking for an existing user account

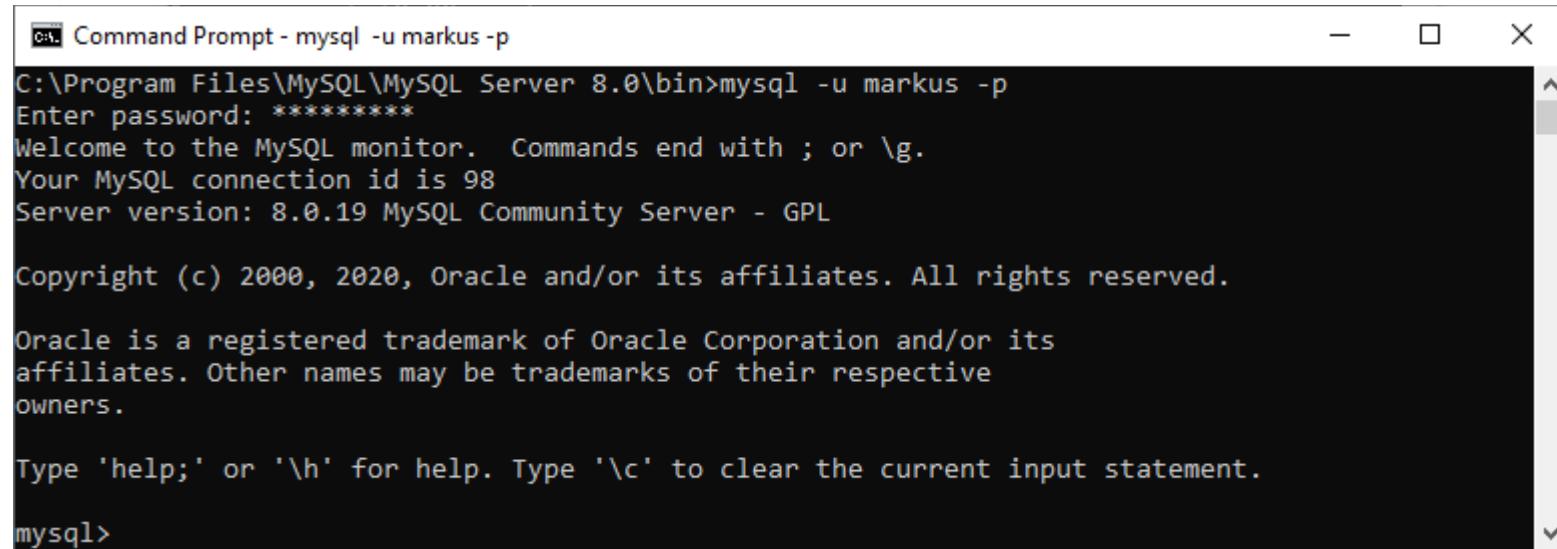
We can understand it by creating a new user account named **markus@localhost** using the below statement:

1. mysql> **CREATE USER** IF NOT EXISTS markus@localhost
2. IDENTIFIED BY 'mark12345';

Next, we will log in to the MySQL server with a newly created user account **markus@localhost** as follows:

1. mysql -u markus -p
2. Enter **password**: \*\*\*\*\*

We will get the below output that means the user account markus@localhost is login successfully.



```
C:\ Command Prompt - mysql -u markus -p
C:\Program Files\MySQL\MySQL Server 8.0\bin>mysql -u markus -p
Enter password: *****
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 98
Server version: 8.0.19 MySQL Community Server - GPL

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

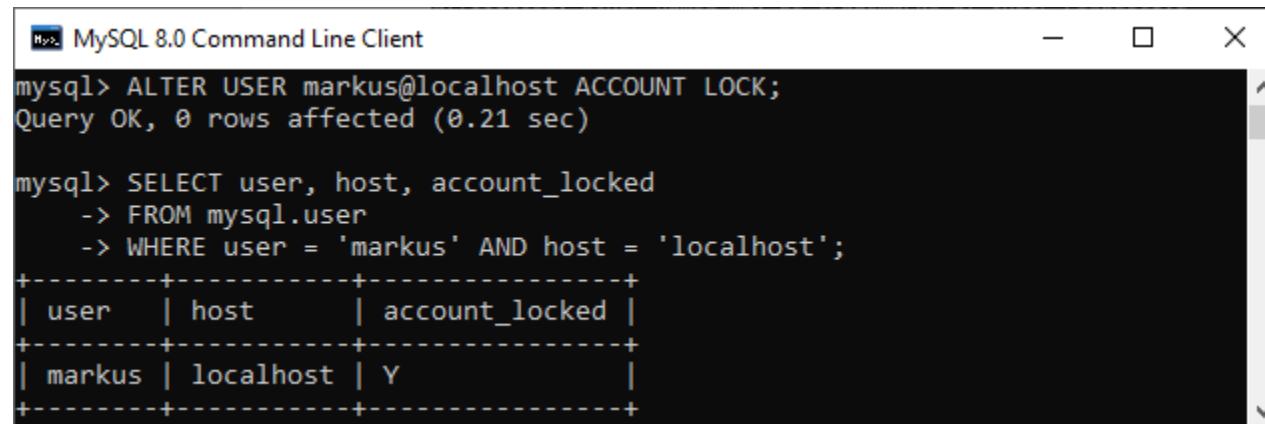
Now, we will use the **ALTER TABLE LOCK ACCOUNT** statement to lock this user account as follows:

1. mysql> **ALTER USER** markus@localhost **ACCOUNT LOCK**;

Again, we will execute the below statement to show the user status:

1. mysql> **SELECT** user, host, account\_locked
2. **FROM** mysql.user
3. **WHERE** user = 'markus' AND host = 'localhost';

We can see the below output that indicates user account markus was locked successfully:



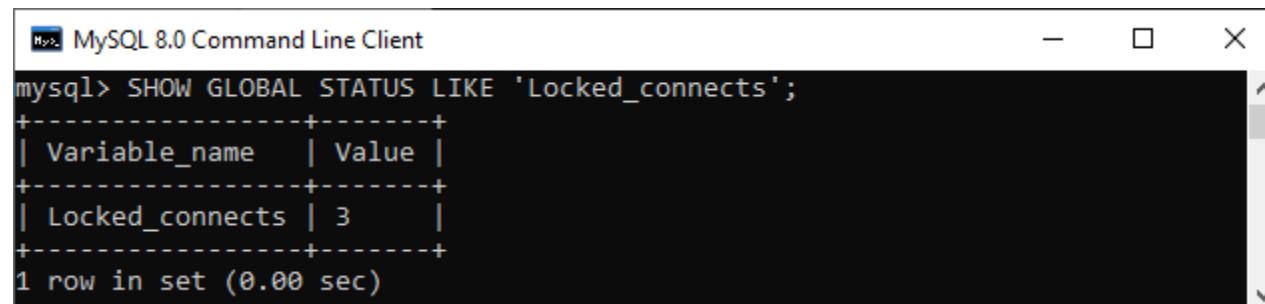
```
MySQL 8.0 Command Line Client
mysql> ALTER USER markus@localhost ACCOUNT LOCK;
Query OK, 0 rows affected (0.21 sec)

mysql> SELECT user, host, account_locked
    -> FROM mysql.user
    -> WHERE user = 'markus' AND host = 'localhost';
+-----+-----+
| user | host   | account_locked |
+-----+-----+
| markus | localhost | Y           |
+-----+-----+
```

If we want to show the number of attempts to connect to the MySQL Server of locked accounts, we need the `locked_connects` variables. Each time we try to connect the locked user account, MySQL increases this variable's status by 1. See the below command:

1. mysql> **SHOW GLOBAL STATUS LIKE 'Locked\_connects'**;

After execution, we will get this output that shows we have tried three times to connect the locked user account:



```
MySQL 8.0 Command Line Client
mysql> SHOW GLOBAL STATUS LIKE 'Locked_connects';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| Locked_connects | 3      |
+-----+-----+
1 row in set (0.00 sec)
```

## MySQL Unlock Account

Unlock is a mechanism that allows the user to **release all locks** or any specific lock associated with the account. In this article, we will learn how to unlock the user accounts in the MySQL server.

When the **CREATE USER... UNLOCK** statement creates a new user account, the new user stored as a locked state.

If we want to release a lock from the existing user account, we need to use the **ALTER USER... ACCOUNT UNLOCK** statement as follows:

1. **ALTER USER** [IF EXISTS] user\_account\_name **ACCOUNT UNLOCK**;

In this syntax, we have to first specify the **user account name** that we want to release a lock after the ALTER USER keyword. Next, we need to provide the **ACCOUNT UNLOCK** clause next to the user name. It is to note the **IF EXISTS** option can also be used to unlock the account only if it has existed in the server.

MySQL also allows us **to unlock multiple user accounts** at the same time by using the below statement:

1. **ALTER USER [IF EXISTS]**
2. **user\_account\_name1, user\_account\_name2, ...**
3. **ACCOUNT UNLOCK;**

In this syntax, we need to provide a list of comma-separated user name for unlocking multiple accounts within a single query. If we do not specify the ACCOUNT UNLOCK clause with the statement, the account unlocking state remains unchanged.

MySQL uses the **account\_locked column of the mysql.user system table** to store the account locking state. We can use the **SHOW CREATE USER** statement to validate whether the account is unlocked or locked. If this column value is **Y**, it means the account is locked. If it contains **N**, it means the account is unlocked.

If we will try to connect to the account without unlocking, MySQL issues an error that writes the below message to the error log:

1. Access denied **for user 'user\_name'@'host\_name'**.
2. An account **is** locked.

## MySQL User Account Unlocking Examples

Let us understand how to unlock the user accounts through examples. First, we will create a new user account named **javatpoint@localhost** in the locked state using the below statement:

1. **mysql> CREATE USER IF NOT EXISTS javatpoint@localhost**
2. **IDENTIFIED BY 'jtp123456'**
3. **ACCOUNT LOCK;**

Next, we will execute the below statement to show the user account and its status:

1. **mysql> SELECT user, host, account\_locked**
2. **FROM mysql.user**
3. **WHERE user = 'javatpoint' AND host = 'localhost';**

We should get the below output:

The screenshot shows the MySQL 8.0 Command Line Client window. The command line displays two SQL statements. The first statement creates a user 'javatpoint'@'localhost' with a password 'jtp123456' and locks the account. The second statement selects the 'user', 'host', and 'account\_locked' columns from the 'mysql.user' table for the user 'javatpoint' on 'localhost'. The output shows one row where 'account\_locked' is 'Y', indicating the account is locked.

```
MySQL 8.0 Command Line Client - MySQL [(none)]>

mysql> CREATE USER IF NOT EXISTS javatpoint@localhost
      -> IDENTIFIED BY 'jtp123456'
      -> ACCOUNT LOCK;
Query OK, 0 rows affected (0.77 sec)

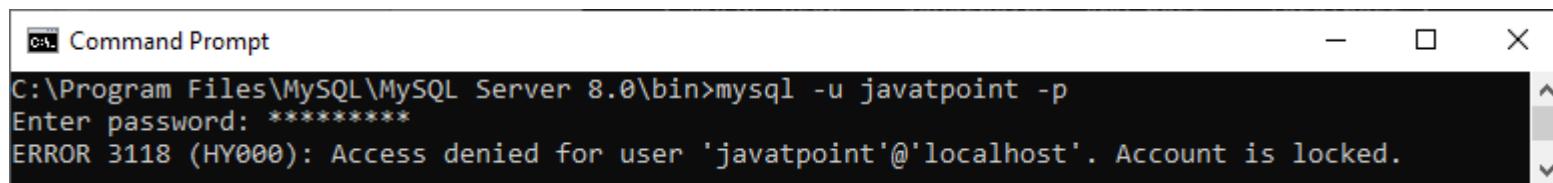
mysql> SELECT user, host, account_locked
      -> FROM mysql.user
      -> WHERE user = 'javatpoint' AND host = 'localhost';
+-----+-----+-----+
| user    | host     | account_locked |
+-----+-----+-----+
| javatpoint | localhost | Y           |
+-----+-----+-----+
1 row in set (0.00 sec)
```

In this output, we can see that the **account\_locked** column in the **mysql.user** system table indicates **Y**. It means the username **javatpoint** is locked on the server.

If we try to connect with this account without unlocking in the MySQL Server, it returns the following error:

1. **mysql -u javatpoint -p**
2. Enter **password: \*\*\*\*\***

Here is the error message:

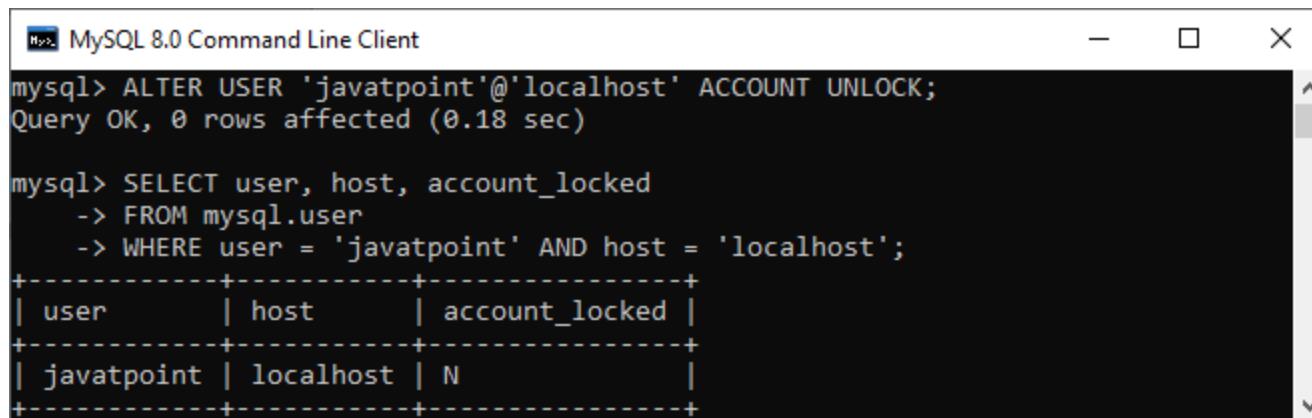


```
Command Prompt
C:\Program Files\MySQL\MySQL Server 8.0\bin>mysql -u javatpoint -p
Enter password: *****
ERROR 3118 (HY000): Access denied for user 'javatpoint'@'localhost'. Account is locked.
```

Thus, we can use the **ALTER USER** statement to unlock the account before accessing it as follows:

1. mysql> **ALTER USER 'javatpoint'@'localhost' ACCOUNT UNLOCK;**

In the below output, we can see that the account\_locked column status is N. It means the user account javatpoint does not have any lock.



```
MySQL 8.0 Command Line Client
mysql> ALTER USER 'javatpoint'@'localhost' ACCOUNT UNLOCK;
Query OK, 0 rows affected (0.18 sec)

mysql> SELECT user, host, account_locked
    -> FROM mysql.user
    -> WHERE user = 'javatpoint' AND host = 'localhost';
+-----+-----+-----+
| user | host | account_locked |
+-----+-----+-----+
| javatpoint | localhost | N |
+-----+-----+-----+
```

In this article, we have learned how we can use the **ALTER TABLE ACCOUNT UNLOCK** statement to release a lock from an existing user account.

## MySQL Queries

# MySQL Queries

A list of commonly used MySQL queries to create database, use database, create table, insert record, update record, delete record, select record, truncate table and drop table are given below.

## 1) MySQL Create Database

MySQL create database is used to create database. For example

1. **create database db1;**

[More Details...](#)

## 2) MySQL Select/Use Database

MySQL use database is used to select database. For example

1. **use db1;**

[More Details...](#)

## 3) MySQL Create Query

MySQL create query is used to create a table, view, procedure and function. For example:

1. **CREATE TABLE** customers
2. **(id int(10),**
3. **name varchar(50),**
4. **city varchar(50),**
5. **PRIMARY KEY** (id )
6. **);**

[More Details...](#)

## 4) MySQL Alter Query

MySQL alter query is used to add, modify, delete or drop columns of a table. Let's see a query to add column in customers table:

1. **ALTER TABLE** customers
2. **ADD** age **varchar**(50);

[More Details...](#)

## 5) MySQL Insert Query

MySQL insert query is used to insert records into table. For example:

1. **insert into** customers **values**(101,'rahul','delhi');

[More Details...](#)

## 6) MySQL Update Query

MySQL update query is used to update records of a table. For example:

1. **update** customers **set name='bob'**, city='london' **where** id=101;

[More Details...](#)

## 7) MySQL Delete Query

MySQL update query is used to delete records of a table from database. For example:

1. **delete from** customers **where** id=101;

[More Details...](#)

## 8) MySQL Select Query

Oracle select query is used to fetch records from database. For example:

1. **SELECT \* from** customers;

[More Details...](#)

## 9) MySQL Truncate Table Query

MySQL update query is used to truncate or remove records of a table. It doesn't remove structure. For example:

1. **truncate table** customers;

[More Details...](#)

## 10) MySQL Drop Query

MySQL drop query is used to drop a table, view or database. It removes structure and data of a table if you drop table. For example:

1. **drop table** customers;

[More Details...](#)

# MySQL Constraints

The constraint in MySQL is used to specify the rule that allows or restricts what values/data will be stored in the table. They provide a suitable method to ensure data accuracy and integrity inside the table. It also helps to limit the type of data that will be inserted inside the table. If any interruption occurs between the constraint and data action, the action is failed.

## Types of MySQL Constraints

Constraints in MySQL is classified into two types:

1. **Column Level Constraints:** These constraints are applied only to the single column that limits the type of particular column data.
2. **Table Level Constraints:** These constraints are applied to the entire table that limits the type of data for the whole table.

## How to create constraints in MySQL

We can define the constraints during a table created by using the CREATE TABLE statement. MySQL also uses the ALTER TABLE statement to specify the constraints in the case of the existing table schema.

### Syntax

The following are the syntax to create a constraints in table:

1. **CREATE TABLE** new\_table\_name (
2. col\_name1 datatype **constraint**,
3. col\_name2 datatype **constraint**,
4. col\_name3 datatype **constraint**,
5. ....
6. );

## Constraints used in MySQL

The following are the most common constraints used in the MySQL:

- o NOT NULL
- o CHECK
- o DEFAULT
- o PRIMARY KEY
- o AUTO\_INCREMENT
- o UNIQUE
- o INDEX
- o ENUM
- o FOREIGN KEY

Let us discuss each of these constraints in detail.

### NOT NULL Constraint

This constraint specifies that the column cannot have NULL or empty values. The below statement creates a table with NOT NULL constraint.

1. mysql> **CREATE TABLE** Student(Id **INTEGER**, LastName TEXT NOT NULL, FirstName TEXT NOT NULL, City **VARCHAR(35)**);

Execute the queries listed below to understand how it works:

1. mysql> **INSERT INTO** Student **VALUES**(1, 'Hanks', 'Peter', 'New York');
- 2.
3. mysql> **INSERT INTO** Student **VALUES**(2, NULL, 'Amanda', 'Florida');

### Output

```
MySQL 8.0 Command Line Client
mysql> CREATE TABLE Student(Id INTEGER, LastName TEXT NOT NULL, FirstName TEXT NOT NULL, City VARCHAR(35));
Query OK, 0 rows affected (2.08 sec)

mysql> INSERT INTO Student VALUES(1, 'Hanks', 'Peter', 'New York');
Query OK, 1 row affected (0.15 sec)

mysql> INSERT INTO Student VALUES(2, NULL, 'Amanda', 'Florida');
ERROR 1048 (23000): Column 'LastName' cannot be null
```

In the above image, we can see that the first INSERT query executes correctly, but the second statement fails and gives an error that says column LastName cannot be null.

### UNIQUE Constraint

This constraint ensures that all values inserted into the column will be unique. It means a column cannot store duplicate values. MySQL allows us to use more than one column with UNIQUE constraint in a table. The below statement creates a table with a UNIQUE constraint:

1. mysql> **CREATE TABLE** ShirtBrands(Id **INTEGER**, BrandName **VARCHAR**(40) **UNIQUE**, Size **VARCHAR**(30));

Execute the queries listed below to understand how it works:

1. mysql> **INSERT INTO** ShirtBrands(Id, BrandName, Size) **VALUES**(1, 'Pantaloons', 38), (2, 'Cantabil', 40);
- 2.
3. mysql> **INSERT INTO** ShirtBrands(Id, BrandName, Size) **VALUES**(1, 'Raymond', 38), (2, 'Cantabil', 40);

### Output

In the below output, we can see that the first INSERT query executes correctly, but the second statement fails and gives an error that says: Duplicate entry 'Cantabil' for key BrandName.

The screenshot shows the MySQL 8.0 Command Line Client interface. The command line displays the following SQL statements and their results:

```
mysql> CREATE TABLE ShirtBrands(Id INTEGER, BrandName VARCHAR(40) UNIQUE, Size VARCHAR(30));
Query OK, 0 rows affected (0.88 sec)

mysql> INSERT INTO ShirtBrands(Id, BrandName, Size) VALUES(1, 'Pantaloons', 38), (2, 'Cantabil', 40);
Query OK, 2 rows affected (0.26 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> INSERT INTO ShirtBrands(Id, BrandName, Size) VALUES(3, 'Raymond', 38), (4, 'Cantabil', 40);
ERROR 1062 (23000): Duplicate entry 'Cantabil' for key 'shirtbrands.BrandName'
```

### CHECK Constraint

It controls the value in a particular column. It ensures that the inserted value in a column must be satisfied with the given condition. In other words, it determines whether the value associated with the column is valid or not with the given condition.

Before the version 8.0.16, MySQL uses the limited version of this constraint syntax, as given below:

1. **CHECK** (expr)

After the version 8.0.16, MySQL uses the CHECK constraints for all storage engines i.e., table constraint and column constraint, as given below:

1. **[CONSTRAINT [symbol]] CHECK** (expr) [[NOT] ENFORCED]

Let us understand how a CHECK constraint works in MySQL. For example, the following statement creates a table "Persons" that contains CHECK constraint on the "Age" column. The CHECK constraint ensures that the inserted value in a column must be satisfied with the given condition means the Age of a person should be greater than or equal to 18:

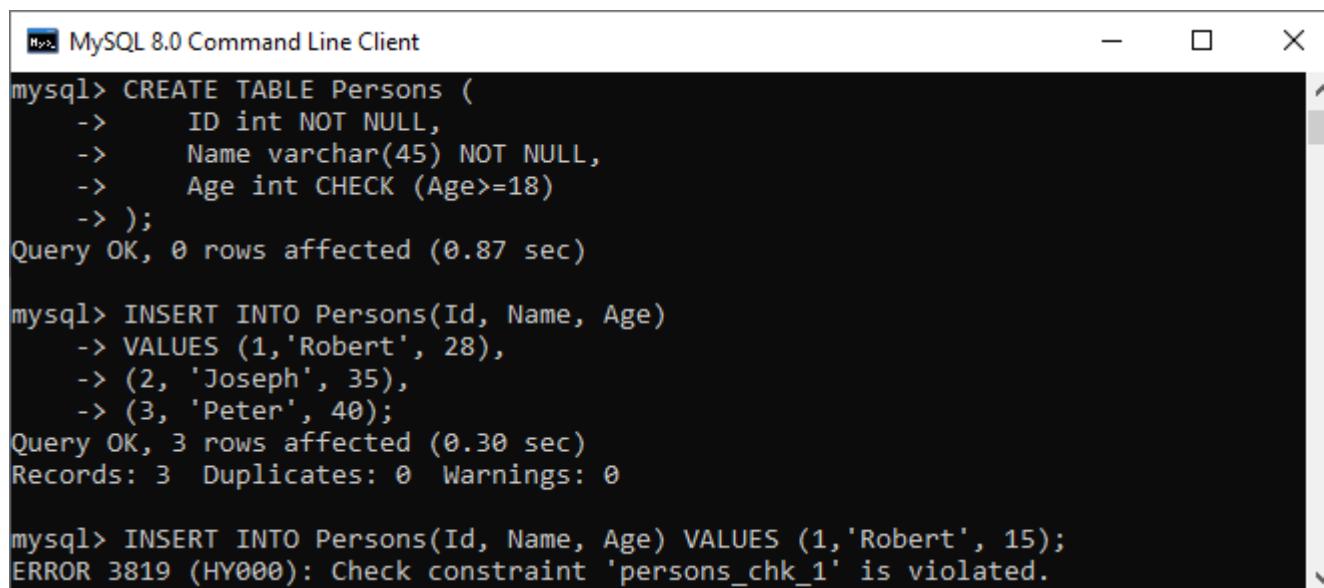
1. mysql> **CREATE TABLE** Persons (
2. ID **int** NOT NULL,
3. Name **varchar**(45) NOT NULL,
4. Age **int CHECK** (Age>=18)
5. );

Execute the listed queries to insert the values into the table:

1. mysql> **INSERT INTO** Persons(Id, Name, Age)
2. **VALUES** (1,'Robert', 28), (2, 'Joseph', 35), (3, 'Peter', 40);
- 3.
4. mysql> **INSERT INTO** Persons(Id, Name, Age) **VALUES** (1,'Robert', 15);

### Output

In the below output, we can see that the first INSERT query executes successfully, but the second statement fails and gives an error that says: CHECK constraint is violated for key Age.



```
MySQL 8.0 Command Line Client
mysql> CREATE TABLE Persons (
->     ID int NOT NULL,
->     Name varchar(45) NOT NULL,
->     Age int CHECK (Age>=18)
-> );
Query OK, 0 rows affected (0.87 sec)

mysql> INSERT INTO Persons(Id, Name, Age)
-> VALUES (1, 'Robert', 28),
-> (2, 'Joseph', 35),
-> (3, 'Peter', 40);
Query OK, 3 rows affected (0.30 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> INSERT INTO Persons(Id, Name, Age) VALUES (1, 'Robert', 15);
ERROR 3819 (HY000): Check constraint 'persons_chk_1' is violated.
```

## DEFAULT Constraint

This constraint is used to set the default value for the particular column where we have not specified any value. It means the column must contain a value, including NULL.

For example, the following statement creates a table "Persons" that contains DEFAULT constraint on the "City" column. If we have not specified any value to the City column, it inserts the default value:

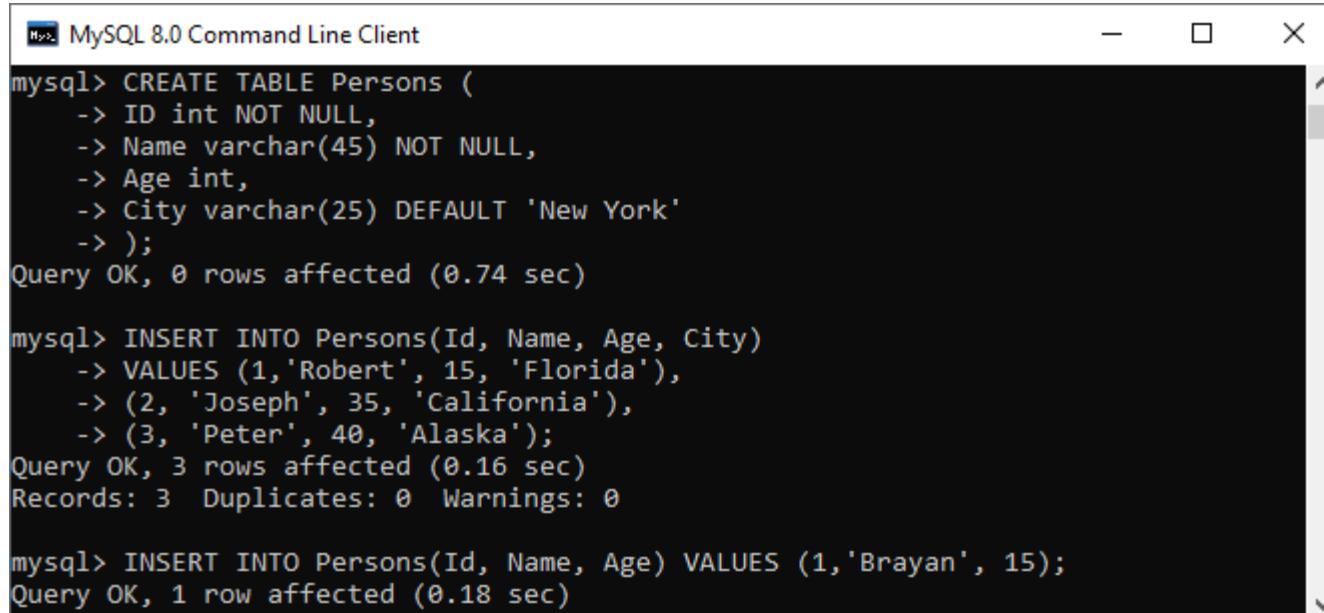
1. mysql> **CREATE TABLE** Persons (
2.   ID **int** NOT NULL,
3.   Name **varchar**(45) NOT NULL,
4.   Age **int**,
5.   City **varchar**(25) **DEFAULT** 'New York'
6. );

Execute the listed queries to insert the values into the table:

1. mysql> **INSERT INTO** Persons(Id, Name, Age, City)
2. **VALUES** (1, 'Robert', 15, 'Florida'),
3. (2, 'Joseph', 35, 'California'),
4. (3, 'Peter', 40, 'Alaska');
- 5.
6. mysql> **INSERT INTO** Persons(Id, Name, Age) **VALUES** (1, 'Brayan', 15);

## Output

In the below output, we can see that the first insert query that contains all fields executes successfully, while the second insert statement does not contain the "City" column but also executed successfully. It is because it has a default value.



```
MySQL 8.0 Command Line Client
mysql> CREATE TABLE Persons (
->     ID int NOT NULL,
->     Name varchar(45) NOT NULL,
->     Age int,
->     City varchar(25) DEFAULT 'New York'
-> );
Query OK, 0 rows affected (0.74 sec)

mysql> INSERT INTO Persons(Id, Name, Age, City)
-> VALUES (1, 'Robert', 15, 'Florida'),
-> (2, 'Joseph', 35, 'California'),
-> (3, 'Peter', 40, 'Alaska');
Query OK, 3 rows affected (0.16 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> INSERT INTO Persons(Id, Name, Age) VALUES (1, 'Brayan', 15);
Query OK, 1 row affected (0.18 sec)
```

Now, executes the following statement to validate the default value for the 4th column:

1. mysql> **SELECT \* FROM** Persons;

We can see that it works perfectly. It means default value "New York" stored automatically in the City column.

```
MySQL 8.0 Command Line Client
mysql> SELECT * FROM Persons;
+----+-----+-----+-----+
| ID | Name | Age | City |
+----+-----+-----+-----+
| 1  | Robert | 15 | Florida |
| 2  | Joseph | 35 | California |
| 3  | Peter | 40 | Alaska |
| 1  | Brayian | 15 | New York |
+----+-----+-----+-----+
4 rows in set (0.00 sec)
```

## PRIMARY KEY Constraint

This constraint is used to identify each record in a table uniquely. If the column contains primary key constraints, then it cannot be null or empty. A table may have duplicate columns, but it can contain only one primary key. It always contains unique value into a column.

The following statement creates a table "Person" and explains the use of this primary key more clearly:

1. **CREATE TABLE** Persons (
2. ID **int** NOT NULL **PRIMARY KEY**,
3. **Name varchar**(45) NOT NULL,
4. Age **int**,
5. City **varchar**(25));

Next, use the insert query to store data into a table:

1. **INSERT INTO** Persons(Id, **Name**, Age, City)
2. **VALUES** (1,'Robert', 15, 'Florida'),
3. (2, 'Joseph', 35, 'California'),
4. (3, 'Peter', 40, 'Alaska');
- 5.
6. **INSERT INTO** Persons(Id, **Name**, Age, City)
7. **VALUES** (1,'Stephen', 15, 'Florida');

## Output

In the below output, we can see that the first insert query executes successfully. While the second insert statement fails and gives an error that says: Duplicate entry for the primary key column.

```
MySQL 8.0 Command Line Client
mysql> CREATE TABLE Persons (
-> ID int NOT NULL PRIMARY KEY,
-> Name varchar(45) NOT NULL,
-> Age int,
-> City varchar(25));
Query OK, 0 rows affected (0.98 sec)

mysql> INSERT INTO Persons(Id, Name, Age, City)
-> VALUES (1,'Robert', 15, 'Florida'),
-> (2, 'Joseph', 35, 'California'),
-> (3, 'Peter', 40, 'Alaska');
Query OK, 3 rows affected (0.17 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> INSERT INTO Persons(Id, Name, Age, City)
-> VALUES (1,'Stephen', 15, 'Florida');
ERROR 1062 (23000): Duplicate entry '1' for key 'persons.PRIMARY'
```

## AUTO\_INCREMENT Constraint

This constraint automatically generates a unique number whenever we insert a new record into the table. Generally, we use this constraint for the primary key field in a table.

We can understand it with the following example where the id column going to be auto-incremented in the Animal table:

1. mysql> **CREATE TABLE** Animals(
2. id **int** NOT NULL AUTO\_INCREMENT,
3. **name CHAR**(30) NOT NULL,
4. **PRIMARY KEY** (id);

Next, we need to insert the values into the "Animals" table:

1. mysql> **INSERT INTO** Animals (**name**) **VALUES**
2. ('Tiger'),('Dog'),('Penguin'),
3. ('Camel'),('Cat'),('Ostrich');

Now, execute the below statement to get the table data:

1. mysql> **SELECT \* FROM** Animals;

#### Output

In the output, we can see that I have not specified any value for the auto-increment column, so MySQL automatically generates a unique number in the sequence order for this field.

The screenshot shows the MySQL 8.0 Command Line Client window. The command history is as follows:

```
mysql> CREATE TABLE Animals(
    -> id int NOT NULL AUTO_INCREMENT,
    -> name CHAR(30) NOT NULL,
    -> PRIMARY KEY (id));
Query OK, 0 rows affected (0.83 sec)

mysql> INSERT INTO Animals (name) VALUES
    -> ('Tiger'),('Dog'),('Penguin'),
    -> ('Camel'),('Cat'),('Ostrich');
Query OK, 6 rows affected (0.18 sec)
Records: 6  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM Animals;
+----+-----+
| id | name  |
+----+-----+
| 1  | Tiger |
| 2  | Dog   |
| 3  | Penguin |
| 4  | Camel |
| 5  | Cat   |
| 6  | Ostrich |
+----+-----+
6 rows in set (0.00 sec)
```

#### ENUM Constraint

The ENUM data type in MySQL is a string object. It allows us to limit the value chosen from a list of permitted values in the column specification at the time of table creation. It is short for enumeration, which means that each column may have one of the specified possible values. It uses numeric indexes (1, 2, 3...) to represent string values.

The following illustration creates a table named "shirts" that contains three columns: id, name, and size. The column name "size" uses the ENUM data type that contains small, medium, large, and x-large sizes.

1. mysql> **CREATE TABLE** Shirts (
2. id **INT PRIMARY KEY** AUTO\_INCREMENT,
3. **name** **VARCHAR**(35),
4. **size** **ENUM('small', 'medium', 'large', 'x-large')**
5. );

Next, we need to insert the values into the "Shirts" table using the below statements:

1. mysql> **INSERT INTO** Shirts(id, **name**, **size**)
2. **VALUES** (1,'t-shirt', 'medium'),
3. (2, 'casual-shirt', 'small'),
4. (3, 'formal-shirt', 'large');

Now, execute the SELECT statement to see the inserted values into the table:

1. mysql> **SELECT \* FROM** Shirts;

#### Output

We will get the following output:

```

MySQL 8.0 Command Line Client
mysql> CREATE TABLE Shirts (
->     id INT PRIMARY KEY AUTO_INCREMENT,
->     name VARCHAR(35),
->     size ENUM('small', 'medium', 'large', 'x-large')
-> );
Query OK, 0 rows affected (1.41 sec)

mysql> INSERT INTO Shirts(id, name, size)
-> VALUES (1,'t-shirt', 'medium'),
-> (2, 'casual-shirt', 'small'),
-> (3, 'formal-shirt', 'large');
Query OK, 3 rows affected (0.19 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM Shirts;
+---+-----+-----+
| id | name      | size   |
+---+-----+-----+
| 1  | t-shirt    | medium |
| 2  | casual-shirt | small  |
| 3  | formal-shirt | large  |
+---+-----+-----+
3 rows in set (0.00 sec)

```

## INDEX Constraint

This constraint allows us to create and retrieve values from the table very quickly and easily. An index can be created using one or more than one column. It assigns a ROWID for each row in that way they were inserted into the table.

The following illustration creates a table named "shirts" that contains three columns: id, name, and size.

1. mysql> **CREATE TABLE** Shirts (
2.   id **INT PRIMARY KEY** AUTO\_INCREMENT,
3.   **name VARCHAR**(35),
4.   **size ENUM**('small', 'medium', 'large', 'x-large')
5. );

Next, we need to insert the values into the "Shirts" table using the below statements:

1. mysql> **INSERT INTO** Shirts(id, **name, size**)
2. **VALUES** (1,'t-shirt', 'medium'),
3. (2, 'casual-shirt', 'small'),
4. (3, 'formal-shirt', 'large');

Now, execute this statement for creating index:

1. mysql> **CREATE INDEX** idx\_name **ON** Shirts(**name**);

We can use the query below to retrieve the data using the index column:

1. mysql> **SELECT \* FROM** Shirts **USE INDEX**(idx\_name);

## Output

The following output appears:

```

MySQL 8.0 Command Line Client
mysql> CREATE INDEX idx_name ON Shirts(name);
Query OK, 0 rows affected (0.78 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM Shirts USE INDEX(idx_name);
+---+-----+-----+
| id | name      | size   |
+---+-----+-----+
| 1  | t-shirt    | medium |
| 2  | casual-shirt | small  |
| 3  | formal-shirt | large  |
+---+-----+-----+
3 rows in set (0.00 sec)

```

## Foreign Key Constraint

This constraint is used to link two tables together. It is also known as the referencing key. A foreign key column matches the primary key field of another table. It means a foreign key field in one table refers to the primary key field of another table.

Let us consider the structure of these tables: Persons and Orders.

#### Table: Persons

1. **CREATE TABLE** Persons (
2. Person\_ID **int** NOT NULL **PRIMARY KEY**,
3. Name **varchar**(45) NOT NULL,
4. Age **int**,
5. City **varchar**(25)
6. );

#### Table: Orders

1. **CREATE TABLE** Orders (
2. Order\_ID **int** NOT NULL **PRIMARY KEY**,
3. Order\_Num **int** NOT NULL,
4. Person\_ID **int**,
5. **FOREIGN KEY** (Person\_ID) **REFERENCES** Persons(Person\_ID)
6. );

In the above table structures, we can see that the "Person\_ID" field in the "Orders" table points to the "Person\_ID" field in the "Persons" table. The "Person\_ID" is the PRIMARY KEY in the "Persons" table, while the "Person\_ID" column of the "Orders" table is a FOREIGN KEY.

#### Output

Our table contains the following data:

```
MySQL 8.0 Command Line Client
mysql> SELECT * FROM Persons;
+-----+-----+-----+-----+
| Person_ID | Name    | Age   | City   |
+-----+-----+-----+-----+
|      1 | Stephen |    15 | Florida|
|      2 | Joseph  |    35 | California|
|      3 | Peter   |    40 | Alaska  |
+-----+-----+-----+-----+
3 rows in set (0.00 sec)

mysql> SELECT * FROM Orders;
+-----+-----+-----+
| Order_ID | Order_Num | Person_ID |
+-----+-----+-----+
|      1 |      5544 |         2 |
|      2 |      3322 |         3 |
|      3 |      2135 |         2 |
|      4 |      3432 |         1 |
+-----+-----+-----+
4 rows in set (0.00 sec)
```

## MySQL INSERT Statement

MySQL INSERT statement is used to store or add data in MySQL table within the database. We can perform insertion of records in two ways using a single query in MySQL:

1. Insert record in a single row
2. Insert record in multiple rows

#### Syntax:

The below is generic syntax of **SQL INSERT INTO** command to insert a single record in MySQL table:

1. **INSERT INTO** table\_name ( field1, field2,...fieldN )
2. **VALUES**
3. ( value1, value2,...valueN );

In the above syntax, we first have to specify the table name and list of comma-separated columns. Second, we provide the list of values corresponding to columns name after the VALUES clause.

**NOTE:** Field name is optional. If we want to specify partial values, the field name is mandatory. It also ensures that the column name and values should be the same. Also, the position of columns and corresponding values must be the same.

If we want to insert **multiple records** within a single command, use the following statement:

1. **INSERT INTO** table\_name **VALUES**
2. ( value1, value2,...valueN )
3. ( value1, value2,...valueN )
4. ....
5. ( value1, value2,...valueN );

In the above syntax, all rows should be separated by commas in the value fields.

## MySQL INSERT Example

Let us understand how **INSERT statements** work in MySQL with the help of multiple examples. First, create a table "**People**" in the database using the following command:

1. **CREATE TABLE** People(
2. id **int** NOT NULL AUTO\_INCREMENT,
3. **name varchar**(45) NOT NULL,
4. occupation **varchar**(35) NOT NULL,
5. age **int**,
6. **PRIMARY KEY** (id)
7. );

**1.** If we want to store single records for all fields, use the syntax as follows:

1. **INSERT INTO** People (id, **name**, occupation, age)
2. **VALUES** (101, 'Peter', 'Engineer', 32);

**2.** If we want to store multiple records, use the following statements where we can either specify all field names or don't specify any field.

1. **INSERT INTO** People **VALUES**
2. (102, 'Joseph', 'Developer', 30),
3. (103, 'Mike', 'Leader', 28),
4. (104, 'Stephen', 'Scientist', 45);

**3.** If we want to store records without giving all fields, we use the following **partial field** statements. In such case, it is mandatory to specify field names.

1. **INSERT INTO** People (**name**, occupation)
2. **VALUES** ('Stephen', 'Scientist'), ('Bob', 'Actor');

In the below output, we can see that all INSERT statements have successfully executed and stored the value in a table correctly.

```
mysql> CREATE TABLE People(
    -> id int NOT NULL AUTO_INCREMENT,
    -> name varchar(45) NOT NULL,
    -> occupation varchar(35) NOT NULL,
    -> age int,
    -> PRIMARY KEY (id)
    -> );
Query OK, 0 rows affected (0.74 sec)

mysql> INSERT INTO People (id, name, occupation, age)
    -> VALUES (101, 'Peter', 'Engineer', 32);
Query OK, 1 row affected (0.21 sec)

mysql> INSERT INTO People VALUES
    -> (102, 'Joseph', 'Developer', 30),
    -> (103, 'Mike', 'Leader', 28),
    -> (104, 'Stephen', 'Scientist', 45);
Query OK, 3 rows affected (0.27 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> INSERT INTO People (name, occupation)
    -> VALUES ('Stephen', 'Scientist'), ('Bob', 'Actor');
Query OK, 2 rows affected (0.10 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

We can use the below syntax to show the records of the **People** table:

1. mysql> **SELECT \* FROM** People;

We will get the output as follows:

```
mysql> SELECT * FROM People;
+----+-----+-----+----+
| id | name | occupation | age |
+----+-----+-----+----+
| 101 | Peter | Engineer | 32 |
| 102 | Joseph | Developer | 30 |
| 103 | Mike | Leader | 28 |
| 104 | Stephen | Scientist | 45 |
| 105 | Stephen | Scientist | NULL |
| 106 | Bob | Actor | NULL |
+----+-----+-----+----+
```

## Inserting Date in MySQL Table:

We can also use the **INSERT STATEMENT** to add the date in MySQL table. MySQL provides several data types for storing dates such as DATE, TIMESTAMP, DATETIME, and YEAR. The **default format** of the date in MySQL is **YYYY-MM-DD**.

This format has the below descriptions:

- **YYYY:** It represents the four-digit year, like 2020.
- **MM:** It represents the two-digit month, like 01, 02, 03, and 12.
- **DD:** It represents the two-digit day, like 01, 02, 03, and 31.

Following is the basic syntax to insert date in MySQL table:

1. **INSERT INTO** table\_name (column\_name, column\_date) **VALUES** ('DATE: Manual Date', '2008-7-04');

If we want to insert a date in the mm/dd/yyyy format, it is required to use the below statement:

1. **INSERT INTO** table\_name **VALUES** (STR\_TO\_DATE(date\_value, format\_specifier));

## MySQL UPDATE Query

MySQL UPDATE query is a DML statement used to modify the data of the MySQL table within the database. In a real-life scenario, records are changed over a period of time. So, we need to make changes in the values of the tables also. To do so, it is required to use the UPDATE query.

The UPDATE statement is used with the **SET** and **WHERE clauses**. The SET clause is used to change the values of the specified column. We can update single or multiple columns at a time.

## Syntax

Following is a generic syntax of UPDATE command to modify data into the MySQL table:

1. **UPDATE** table\_name
2. **SET** column\_name1 = new-value1,
3.       column\_name2=new-value2, ...
4. **[WHERE Clause]**

## Parameter Explanation

The description of parameters used in the syntax of the UPDATE statement is given below:

Parameter	Descriptions
table_name	It is the name of a table in which we want to perform updation.
column_name	It is the name of a column in which we want to perform updation with the new value using the SET clause. If there is a need to update multiple columns, separate the columns with a comma operator by specifying the value in each column.
WHERE Clause	It is optional. It is used to specify the row name in which we are going to perform updation. If we omit this clause, MySQL updates all rows.

### Note:

- o This statement can update values in a single table at a time.
- o We can update single or multiple columns altogether with this statement.
- o Any condition can be specified by using the WHERE clause.
- o WHERE clause is very important because sometimes we want to update only a single row, and if we omit this clause, it accidentally updates all rows of the table.

The UPDATE command supports these modifiers in MySQL:

**LOW\_PRIORITY:** This modifier instructs the statement to delay the UPDATE command's execution until no other clients reading from the table. It takes effects only for the storage engines that use only table-level locking.

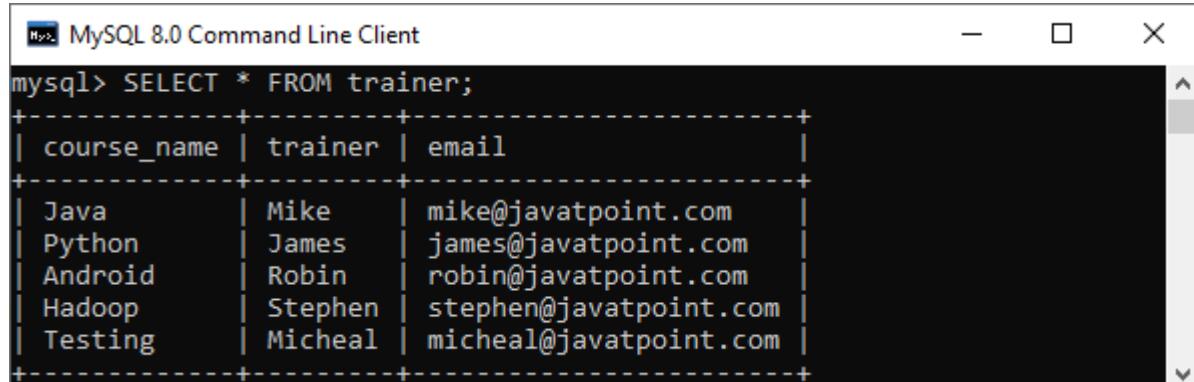
**IGNORE:** This modifier allows the statement to do not abort the execution even if errors occurred. If it finds **duplicate-key** conflicts, the rows are not updated.

Therefore, the full syntax of **UPDATE statement** is given below:

1. **UPDATE** [LOW\_PRIORITY] [IGNORE] table\_name
2. **SET** column\_assignment\_list
3. **[WHERE condition]**

## Example:

Let us understand the UPDATE statement with the help of various examples. Suppose we have a table "**trainer**" within the "**testdb**" database. We are going to update the data within the "trainer" table.



The screenshot shows the MySQL 8.0 Command Line Client interface. A terminal window displays the following SQL query and its results:

```
mysql> SELECT * FROM trainer;
+-----+-----+-----+
| course_name | trainer | email
+-----+-----+-----+
| Java        | Mike    | mike@javatpoint.com
| Python      | James   | james@javatpoint.com
| Android     | Robin   | robin@javatpoint.com
| Hadoop      | Stephen | stephen@javatpoint.com
| Testing     | Micheal | micheal@javatpoint.com
+-----+-----+-----+
```

## Update Single Column

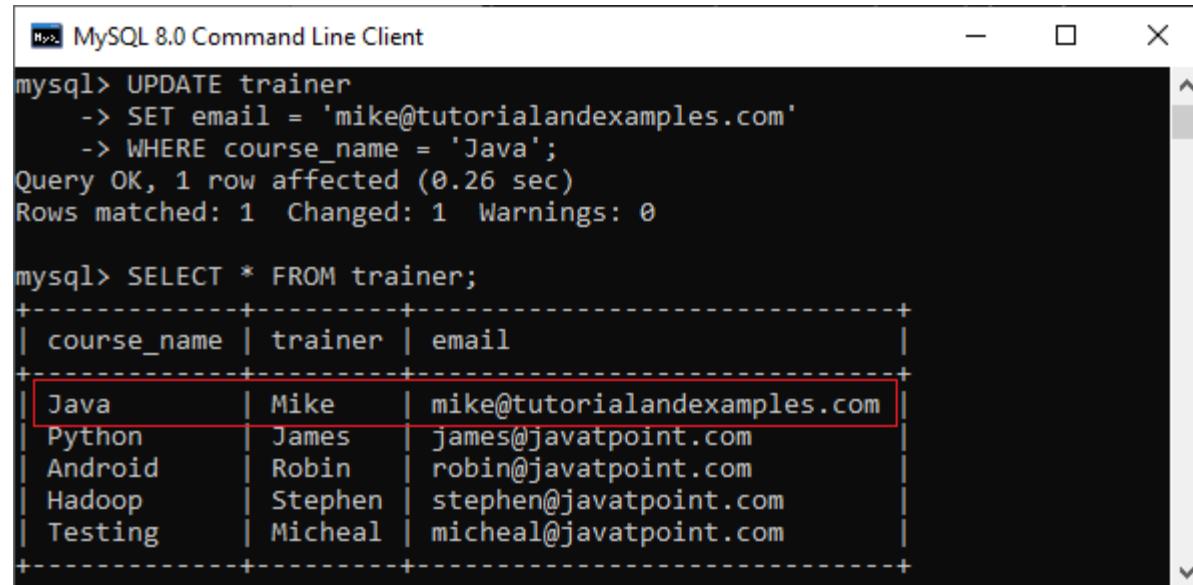
This query will update the **email id of Java** course with the new id as follows:

1. **UPDATE** trainer
2. **SET** email = 'mike@tutorialandexamples.com'
3. **WHERE** course\_name = 'Java';

After successful execution, we will verify the table using the below statement:

1. **SELECT \* FROM** trainer;

In the output, we can see that our table is updated as per our conditions.

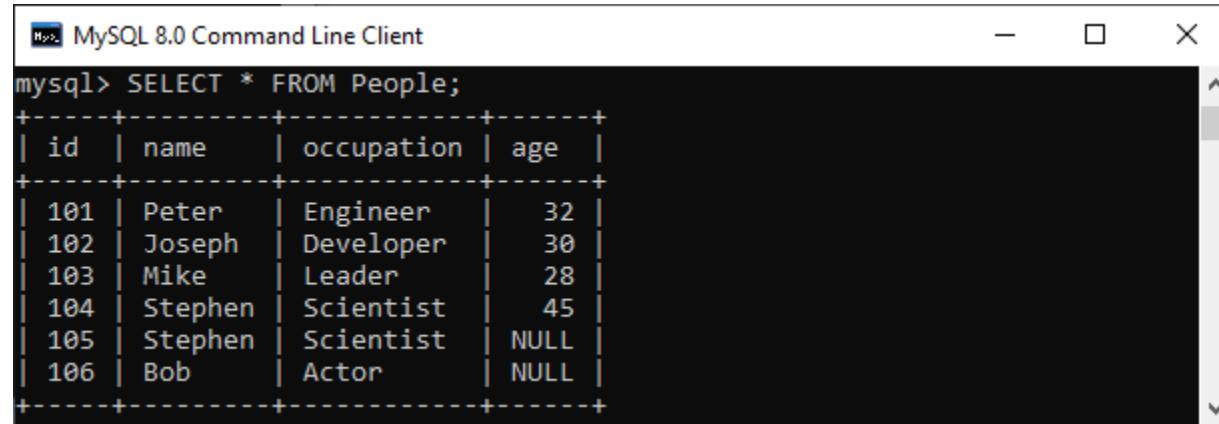


```
MySQL 8.0 Command Line Client
mysql> UPDATE trainer
-> SET email = 'mike@tutorialandexamples.com'
-> WHERE course_name = 'Java';
Query OK, 1 row affected (0.26 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT * FROM trainer;
+-----+-----+-----+
| course_name | trainer | email
+-----+-----+-----+
| Java        | Mike    | mike@tutorialandexamples.com
| Python      | James   | james@javatpoint.com
| Android     | Robin   | robin@javatpoint.com
| Hadoop      | Stephen | stephen@javatpoint.com
| Testing     | Micheal | micheal@javatpoint.com
+-----+-----+-----+
```

### Update Multiple Columns

The UPDATE statement can also be used to update multiple columns by specifying a comma-separated list of columns. Suppose we have a table as below:

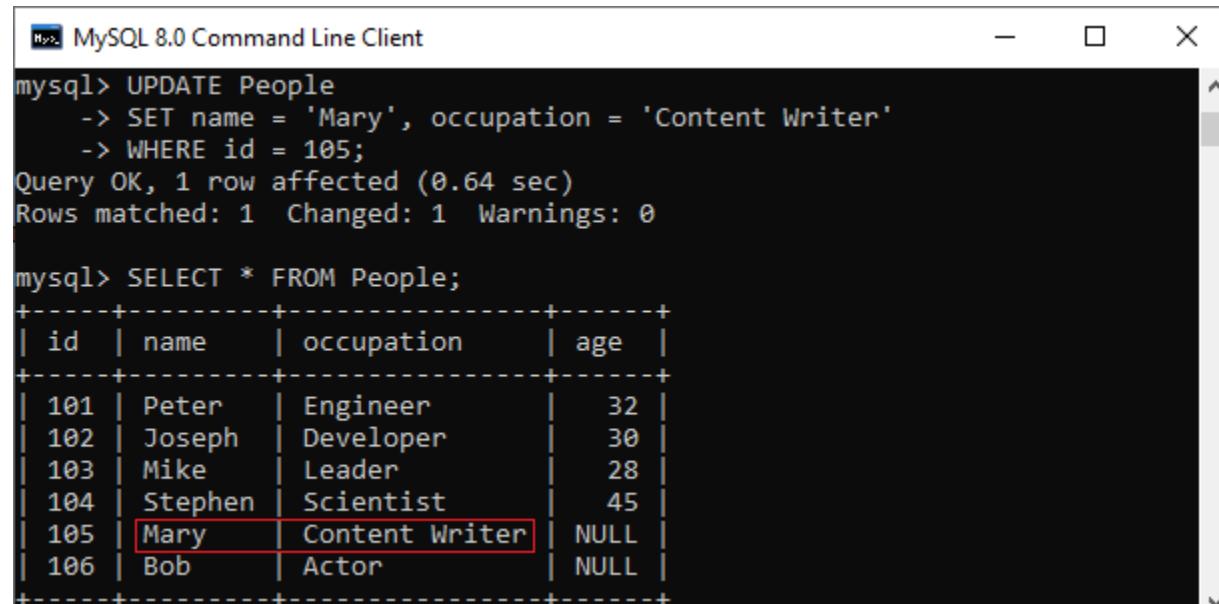


```
MySQL 8.0 Command Line Client
mysql> SELECT * FROM People;
+----+-----+-----+-----+
| id | name | occupation | age |
+----+-----+-----+-----+
| 101 | Peter | Engineer   | 32 |
| 102 | Joseph | Developer  | 30 |
| 103 | Mike   | Leader     | 28 |
| 104 | Stephen | Scientist  | 45 |
| 105 | Stephen | Scientist  | NULL |
| 106 | Bob    | Actor      | NULL |
+----+-----+-----+-----+
```

This statement explains will update the **name** and **occupation** whose **id = 105** in the **People** table as follows:

1. **UPDATE** People
2. **SET name = 'Mary'**, occupation = 'Content Writer'
3. **WHERE** id = 105;

We can verify the output below:



```
MySQL 8.0 Command Line Client
mysql> UPDATE People
-> SET name = 'Mary', occupation = 'Content Writer'
-> WHERE id = 105;
Query OK, 1 row affected (0.64 sec)
Rows matched: 1  Changed: 1  Warnings: 0

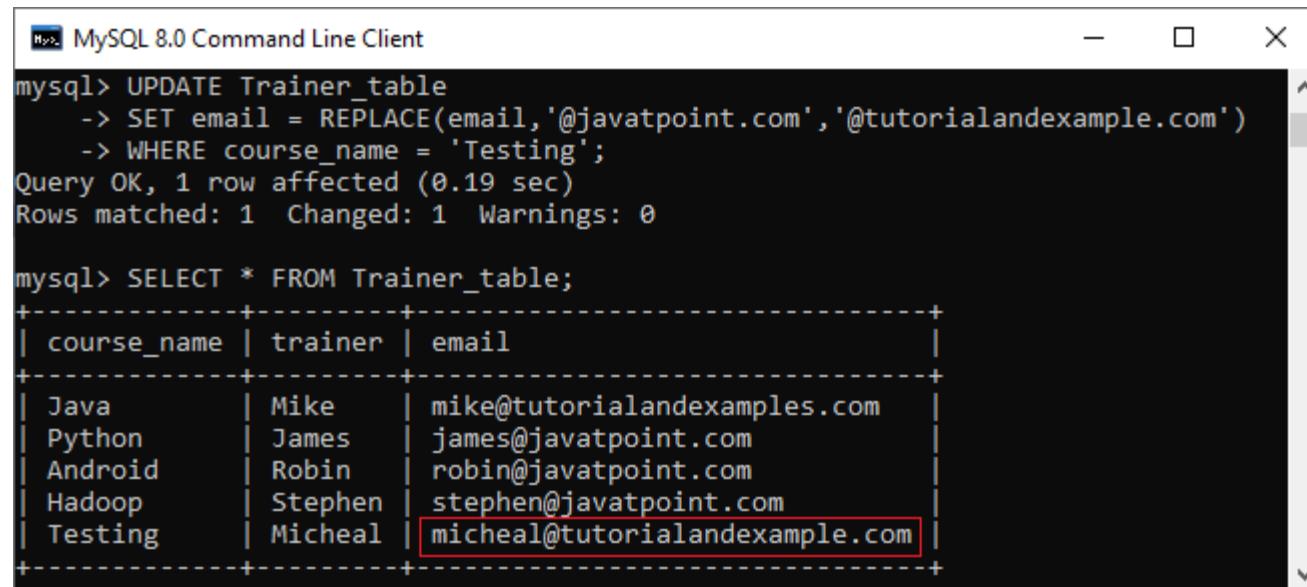
mysql> SELECT * FROM People;
+----+-----+-----+-----+
| id | name | occupation | age |
+----+-----+-----+-----+
| 101 | Peter | Engineer   | 32 |
| 102 | Joseph | Developer  | 30 |
| 103 | Mike   | Leader     | 28 |
| 104 | Stephen | Scientist  | 45 |
| 105 | Mary   | Content Writer | NULL |
| 106 | Bob    | Actor      | NULL |
+----+-----+-----+-----+
```

### UPDATE Statement to Replace String

We can also use the UPDATE statement in MySQL to change the string name in the particular column. The following example updates the domain parts of emails of **Android course**:

1. **UPDATE** Trainer\_table
2. **SET** email = **REPLACE**(email,'@javatpoint.com','@tutorialandexample.com')
3. **WHERE** course\_name = 'Testing';

It will give the following output:



```
MySQL 8.0 Command Line Client
mysql> UPDATE Trainer_table
    -> SET email = REPLACE(email,'@javatpoint.com','@tutorialandexample.com')
    -> WHERE course_name = 'Testing';
Query OK, 1 row affected (0.19 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT * FROM Trainer_table;
+-----+-----+-----+
| course_name | trainer | email
+-----+-----+-----+
| Java        | Mike    | mike@tutorialandexamples.com
| Python       | James   | james@javatpoint.com
| Android      | Robin   | robin@javatpoint.com
| Hadoop       | Stephen | stephen@javatpoint.com
| Testing      | Micheal | micheal@tutorialandexample.com
+-----+-----+-----+
```

## MySQL DELETE Statement

MySQL DELETE statement is used to remove records from the MySQL table that is no longer required in the database. **This query in MySQL deletes a full row from the table and produces the count of deleted rows.** It also allows us to delete more than one record from the table within a single query, which is beneficial while removing large numbers of records from a table. By using the delete statement, we can also remove data based on conditions.

**Once we delete the records using this query, we cannot recover it.** Therefore before deleting any records from the table, it is recommended to **create a backup of your database**. The database backups allow us to restore the data whenever we need it in the future.

### Syntax:

The following are the syntax that illustrates how to use the DELETE statement:

1. **DELETE FROM** table\_name **WHERE** condition;

In the above statement, we have to first specify the table name from which we want to delete data. Second, we have to specify the condition to delete records in the **WHERE clause**, which is optional. If we omit the WHERE clause into the statement, this query will remove whole records from the database table.

If we want to delete records from multiple tables using a single DELETE query, we must add the **JOIN clause** with the DELETE statement.

If we want to delete all records from a table without knowing the count of deleted rows, we must use the **TRUNCATE TABLE** statement that gives better performance.

Let us understand how the DELETE statement works in MySQL through various examples.

## MySQL DELETE Statement Examples

Here, we are going to use the "**Employees**" and "**Payment**" tables for the demonstration of the DELETE statement. Suppose the Employees and Payment tables contain the following data:

The image shows two separate MySQL Command Line Client windows. The top window displays the output of the query `SELECT * FROM Employees;`, which lists 7 rows of employee data. The bottom window displays the output of the query `SELECT * FROM Payment;`, which lists 6 rows of payment data.

```

MySQL 8.0 Command Line Client
mysql> SELECT * FROM Employees;
+-----+-----+-----+-----+
| emp_id | name   | birthdate | gender | hire_date |
+-----+-----+-----+-----+
| 101    | Bryan  | 1988-08-12 | M      | 2015-08-26 |
| 102    | Joseph | 1978-05-12 | M      | 2014-10-21 |
| 103    | Mike   | 1984-10-13 | M      | 2017-10-28 |
| 104    | Daren  | 1979-04-11 | F      | 2006-11-01 |
| 105    | Marie  | 1990-02-11 | F      | 2018-10-12 |
| 106    | Marco  | 1988-04-11 | M      | 2010-10-12 |
| 107    | Antonio | 1982-02-15 | M      | 2005-10-12 |
+-----+-----+-----+-----+
7 rows in set (0.00 sec)

MySQL 8.0 Command Line Client
mysql> SELECT * FROM Payment;
+-----+-----+-----+-----+
| payment_id | emp_id | amount | payment_date |
+-----+-----+-----+-----+
| 301        | 101    | 1200   | 2015-09-15 |
| 302        | 101    | 1200   | 2015-09-30 |
| 303        | 103    | 1500   | 2015-10-15 |
| 304        | 103    | 1500   | 2015-10-30 |
| 305        | 102    | 1800   | 2015-09-15 |
| 306        | 102    | 1800   | 2015-09-30 |
+-----+-----+-----+-----+
6 rows in set (0.00 sec)

```

If we want to delete an employee whose **emp\_id** is **107**, we should use the **DELETE** statement with the **WHERE** clause. See the below query:

1. mysql> **DELETE FROM** Employees **WHERE** emp\_id=107;

After the execution of the query, it will return the output as below image. Once the record is deleted, verify the table using the **SELECT** statement:

The image shows a MySQL Command Line Client window. It first executes the `DELETE FROM Employees WHERE emp_id=107;` query, which returns "Query OK, 1 row affected (0.10 sec)". Then it runs the `SELECT * FROM Employees;` query again, which now shows only 6 rows of data, confirming that the employee with `emp_id 107` has been deleted.

```

MySQL 8.0 Command Line Client
mysql> DELETE FROM Employees WHERE emp_id=107;
Query OK, 1 row affected (0.10 sec)

mysql> SELECT * FROM Employees;
+-----+-----+-----+-----+
| emp_id | name   | birthdate | gender | hire_date |
+-----+-----+-----+-----+
| 101    | Bryan  | 1988-08-12 | M      | 2015-08-26 |
| 102    | Joseph | 1978-05-12 | M      | 2014-10-21 |
| 103    | Mike   | 1984-10-13 | M      | 2017-10-28 |
| 104    | Daren  | 1979-04-11 | F      | 2006-11-01 |
| 105    | Marie  | 1990-02-11 | F      | 2018-10-12 |
| 106    | Marco  | 1988-04-11 | M      | 2010-10-12 |
+-----+-----+-----+-----+
6 rows in set (0.00 sec)

```

If we want to delete all records from the table, there is no need to use the **WHERE** clause with the **DELETE** statement. See the below code and output:

The image shows a MySQL Command Line Client window. It first executes the `DELETE FROM Employees;` query, which returns "Query OK, 6 rows affected (0.12 sec)". Then it runs the `SELECT * FROM Employees;` query again, which returns "Empty set (0.00 sec)", indicating that the table is now empty.

```

MySQL 8.0 Command Line Client
mysql> DELETE FROM Employees;
Query OK, 6 rows affected (0.12 sec)

mysql> SELECT * FROM Employees;
Empty set (0.00 sec)

```

In the above output, we can see that after removing all rows, the **Employees** table will be empty. It means no records available in the selected table.

## MySQL DELETE and LIMIT Clause

MySQL Limit clause is used to restrict the count of rows returns from the result set, rather than fetching the whole records in the table. Sometimes we want to limit the number of rows to be deleted from the table; in that case, we will use the **LIMIT** clause as follows:

1. **DELETE FROM** table\_name
2. **WHERE** condition
3. **ORDER BY** colm1, colm2, ...

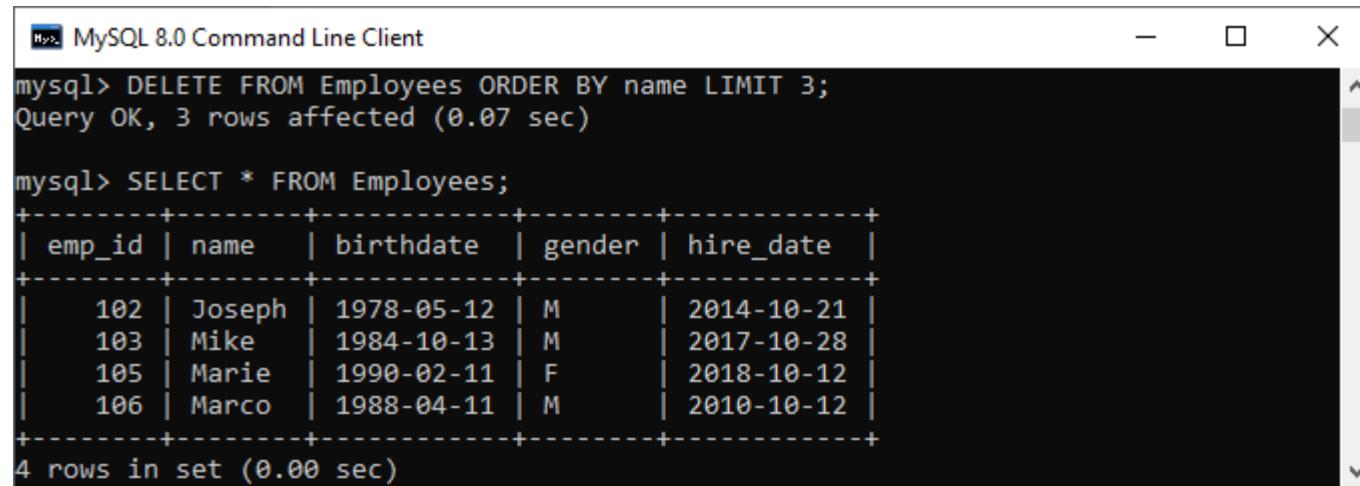
4. LIMIT row\_count;

It is to note that the order of rows in a MySQL table is unspecified. Therefore, we should always use the **ORDER BY** clause while using the LIMIT clause.

**For example**, the following query first sorts the employees according to their names alphabetically and deletes the first three employees from the table:

1. mysql> **DELETE FROM** Employees **ORDER BY** name LIMIT 3;

It will give the below output:



```
MySQL 8.0 Command Line Client
mysql> DELETE FROM Employees ORDER BY name LIMIT 3;
Query OK, 3 rows affected (0.07 sec)

mysql> SELECT * FROM Employees;
+-----+-----+-----+-----+
| emp_id | name   | birthdate | gender | hire_date |
+-----+-----+-----+-----+
| 102    | Joseph | 1978-05-12 | M      | 2014-10-21 |
| 103    | Mike   | 1984-10-13 | M      | 2017-10-28 |
| 105    | Marie  | 1990-02-11 | F      | 2018-10-12 |
| 106    | Marco  | 1988-04-11 | M      | 2010-10-12 |
+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

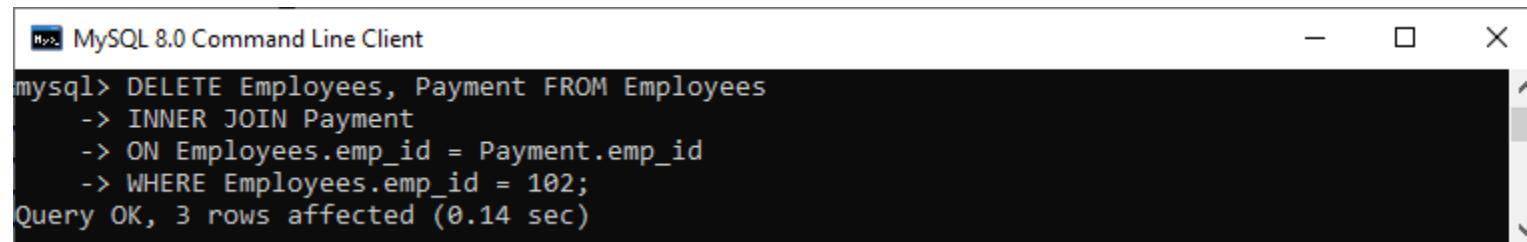
## MySQL DELETE and JOIN Clause

The JOIN clause is used to add the two or more tables in MySQL. We will add the JOIN clause with the DELETE statement whenever we want to delete records from multiple tables within a single query. See the below query:

1. mysql> **DELETE** Employees, Payment **FROM** Employees
2. **INNER JOIN** Payment
3. **ON** Employees.emp\_id = Payment.emp\_id
4. **WHERE** Employees.emp\_id = 102;

### Output:

After execution, we will see the output as below image:



```
MySQL 8.0 Command Line Client
mysql> DELETE Employees, Payment FROM Employees
   -> INNER JOIN Payment
   -> ON Employees.emp_id = Payment.emp_id
   -> WHERE Employees.emp_id = 102;
Query OK, 3 rows affected (0.14 sec)
```

To read more information about the DELETE statement with the JOIN clause, [click here](#).

## MySQL SELECT Statement

The SELECT statement in MySQL is used to **fetch data from one or more tables**. We can retrieve records of all fields or specified fields that match specified criteria using this statement. It can also work with various scripting languages such as [PHP](#), [Ruby](#), and many more.

## SELECT Statement Syntax

It is the most commonly used [SQL](#) query. The general syntax of this statement to fetch data from tables are as follows:

1. **SELECT** field\_name1, field\_name 2,... field\_nameN
2. **FROM** table\_name1, table\_name2...
3. **[WHERE]** condition]
4. **[GROUP BY]** field\_name(s)]
5. **[HAVING]** condition]
6. **[ORDER BY]** field\_name(s)]
7. **[OFFSET M ][LIMIT N];**

## Syntax for all fields:

1. **SELECT \* FROM** tables [**WHERE** conditions]
2. [**GROUP BY** fieldName(s)]
3. [**HAVING** condition]
4. [**ORDER BY** fieldName(s)]
5. [**OFFSET M ][LIMIT N;**]

## Parameter Explanation

The SELECT statement uses the following parameters:

Parameter Name	Descriptions
field_name(s) or *	It is used to specify one or more columns to returns in the result set. The asterisk (*) returns all fields of a table.
table_name(s)	It is the name of tables from which we want to fetch data.
WHERE	It is an optional clause. It specifies the condition that returned the matched records in the result set.
GROUP BY	It is optional. It collects data from multiple records and grouped them by one or more columns.
HAVING	It is optional. It works with the GROUP BY clause and returns only those rows whose condition is TRUE.
ORDER BY	It is optional. It is used for sorting the records in the result set.
OFFSET	It is optional. It specifies to which row returns first. By default, It starts with zero.
LIMIT	It is optional. It is used to limit the number of returned records in the result set.

**NOTE:** It is to note that MySQL always evaluates the **FROM** clause first, and then the **SELECT** clause will be evaluated.

## MySQL SELECT Statement Example:

Let us understand how SELECT command works in [MySQL](#) with the help of various examples. Suppose we have a table named **employee\_detail** that contains the following data:

ID	Name	Email	Phone	City	Working_hours
1	Peter	peter@javatpoint.com	49562959223	Texas	12
2	Suzi	suzi@javatpoint.com	70679834522	California	10
3	Joseph	joseph@javatpoint.com	09896765374	Alaska	14
4	Alex	alex@javatpoint.com	97335737548	Los Angeles	9
5	Mark	mark@javatpoint.com	78765645643	Washington	12
6	Stephen	stephen@javatpoint.com	986345793248	New York	10

1. If we want to retrieve a **single column from the table**, we need to execute the below query:

1. mysql> **SELECT Name FROM employee\_detail;**

We will get the below output where we can see only one column records.

MySQL 8.0 Command Line Client

```
mysql> SELECT Name FROM employee_detail;
```

Name
Alex
Joseph
Mark
Peter
Stephen
Suzi

2. If we want to query **multiple columns from the table**, we need to execute the below query:

1. mysql> **SELECT Name, Email, City FROM employee\_detail;**

We will get the below output where we can see the name, email, and city of employees.

MySQL 8.0 Command Line Client

```
mysql> SELECT Name, Email, City FROM employee_detail;
```

Name	Email	City
Peter	peter@javatpoint.com	Texas
Suzi	suzi@javatpoint.com	California
Joseph	joseph@javatpoint.com	Alaska
Alex	alex@javatpoint.com	Los Angeles
Mark	mark@javatpoint.con	Washington
Stephen	stephen@javatpoint.com	New York

3. If we want to fetch data from **all columns of the table**, we need to use all column's names with the select statement. Specifying all column names is not convenient to the user, so MySQL uses an **asterisk (\*)** to retrieve all column data as follows:

1. mysql> **SELECT \* FROM employee\_detail;**

We will get the below output where we can see all columns of the table.

MySQL 8.0 Command Line Client

```
mysql> SELECT * FROM employee_detail;
```

ID	Name	Email	Phone	City	Working_hours
1	Peter	peter@javatpoint.com	49562959223	Texas	12
2	Suzi	suzi@javatpoint.com	70679834522	California	10
3	Joseph	joseph@javatpoint.com	09896765374	Alaska	14
4	Alex	alex@javatpoint.com	97335737548	Los Angeles	9
5	Mark	mark@javatpoint.con	78765645643	Washington	12
6	Stephen	stephen@javatpoint.com	986345793248	New York	10

4. Here, we use the **SUM function** with the **HAVING clause** in the SELECT command to get the employee name, city, and total working hours. Also, it uses the **GROUP BY clause** to group them by the Name column.

1. **SELECT Name, City, SUM(working\_hours) AS "Total working hours"**
2. **FROM employee\_detail**
3. **GROUP BY Name**
4. **HAVING SUM(working\_hours) > 5;**

It will give the below output:

```

MySQL 8.0 Command Line Client
mysql> SELECT Name, City, SUM(working_hours) AS "Total working hours"
-> FROM employee_detail
-> GROUP BY name
-> HAVING SUM(working_hours) > 5;
+-----+-----+-----+
| Name | City | Total working hours |
+-----+-----+-----+
| Alex | Los Angeles | 9 |
| Joseph | Alaska | 14 |
| Mark | Washington | 12 |
| Peter | Texas | 12 |
| Stephen | New York | 10 |
| Suzi | California | 10 |
+-----+-----+-----+

```

5. MySQL SELECT statement can also be used to retrieve records from multiple tables by using a **JOIN statement**. Suppose we have a table named "**customer**" and "**orders**" that contains the following data:

**Table: customer**

cust_id	cust_name	city	occupation
1	Peter	London	Business
2	Joseph	Texas	Doctor
3	Mark	New Delhi	Engineer
4	Michael	New York	Scientist
5	Alexandar	Maxico	Student

**Table: orders**

order_id	prod_name	order_num	order_date
1	Laptop	5544	2020-02-01
2	Mouse	3322	2020-02-11
3	Desktop	2135	2020-01-05
4	Mobile	3432	2020-02-22
5	Antivirus	5648	2020-03-10

Execute the following SQL statement that returns the matching records from both tables using the [INNER JOIN query](#):

1. **SELECT** cust\_name, city, order\_num, order\_date
2. **FROM** customer **INNER JOIN** orders
3. **ON** customer.cust\_id = orders.order\_id
4. **WHERE** order\_date < '2020-04-30'
5. **ORDER BY** cust\_name;

After successful execution of the query, we will get the output as follows:

```

MySQL 8.0 Command Line Client
mysql> SELECT cust_name, city, order_num, order_date
-> FROM customer INNER JOIN orders
-> ON customer.cust_id = orders.order_id
-> WHERE order_date < '2020-04-30'
-> ORDER BY cust_name;
+-----+-----+-----+-----+
| cust_name | city | order_num | order_date |
+-----+-----+-----+-----+
| Alexander | Maxico | 5648 | 2020-03-10 |
| Joseph | Texas | 3322 | 2020-02-11 |
| Mark | New Delhi | 2135 | 2020-01-05 |
| Michael | New York | 3432 | 2020-02-22 |
| Peter | London | 5544 | 2020-02-01 |
+-----+-----+-----+-----+

```

## MySQL REPLACE

The REPLACE statement in MySQL is an extension of the SQL Standard. This statement works the same as the INSERT statement, except that if an old row matches the new record in the table for a PRIMARY KEY or a UNIQUE index, this command deleted the old row before the new row is added.

This statement is required when we want to update the existing records into the table to keep them updated. If we use the standard insert query for this purpose, it will give a Duplicate entry for PRIMARY KEY or a UNIQUE key error. In this case, we will use the REPLACE statement to perform our task. The REPLACE command requires one of the two **possible** actions take place:

- o If no matching value is found with the existing data row, then a standard INSERT statement is performed.
- o
- o If the duplicate record found, the replace command will delete the existing row and then adds the new record in the table.

In the REPLACE statement, the updation performed in two steps. First, it will delete the existing record, and then the newly updated record is added, similar to a standard INSERT command. Thus, we can say that the REPLACE statement performs two standard functions, **DELETE** and **INSERT**.

## Syntax

The following are the syntax of **REPLACE** statement in MySQL:

1. **REPLACE [INTO] table\_name(column\_list)**
2. **VALUES(value\_list);**

## MySQL REPLACE Example

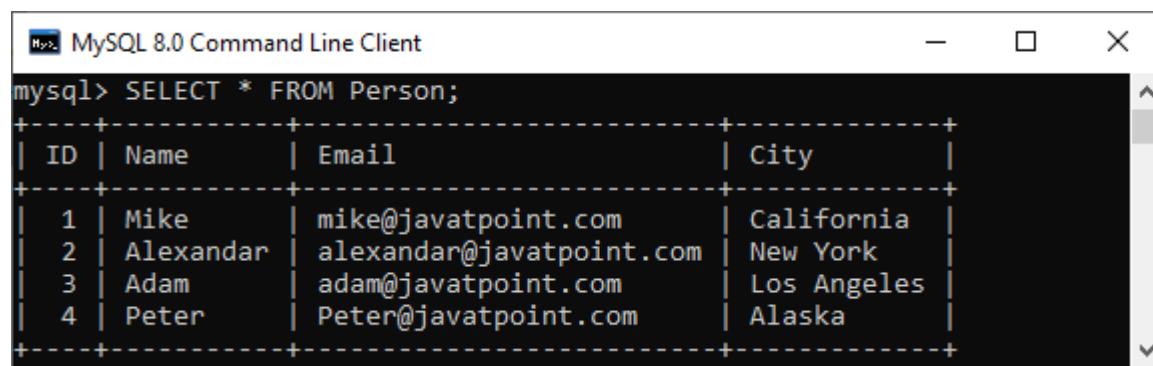
Let us understand the working of the REPLACE statement in MySQL with the help of an example. First, we are going to create a table named "**Person**" using the following statement:

1. **CREATE TABLE Person (**
2. **ID int AUTO\_INCREMENT PRIMARY KEY,**
3. **Name varchar(45) DEFAULT NULL,**
4. **Email varchar(45) DEFAULT NULL UNIQUE,**
5. **City varchar(25) DEFAULT NULL**
6. **);**

Next, we need to fill the record into the table using the **INSERT** statement as below:

1. **INSERT INTO Person(ID, Name, Email, City)**
2. **VALUES (1, 'Mike', 'mike@javatpoint.com', 'California'),**
3. **(2, 'Alexandar', 'alexandar@javatpoint.com', 'New York'),**
4. **(3, 'Adam', 'adam@javatpoint.com', 'Los Angeles'),**
5. **(4, 'Peter', 'Peter@javatpoint.com', 'Alaska');**

Execute the **SELECT statement** to verify the records that can be shown in the below output:



```
MySQL 8.0 Command Line Client
mysql> SELECT * FROM Person;
+----+-----+-----+-----+
| ID | Name | Email           | City      |
+----+-----+-----+-----+
| 1  | Mike  | mike@javatpoint.com | California |
| 2  | Alexandar | alexandar@javatpoint.com | New York   |
| 3  | Adam  | adam@javatpoint.com | Los Angeles |
| 4  | Peter | Peter@javatpoint.com | Alaska     |
+----+-----+-----+-----+
```

After verifying the data into a table, we can replace any old row with the new row using the REPLACE statement. Execute the below statement that updates the **city of a person whose id is 4**.

1. **REPLACE INTO Person (id, city)**
2. **VALUES(4, 'Amsterdam');**

After the successful execution of the above statement, it is required to query the data of the table Person again to verify the replacement.

```

MySQL 8.0 Command Line Client
mysql> REPLACE INTO Person (id, city)
-> VALUES(4,'Amsterdam');
Query OK, 2 rows affected (0.20 sec)

mysql> SELECT * FROM Person;
+----+-----+-----+-----+
| ID | Name | Email | City |
+----+-----+-----+-----+
| 1  | Mike  | mike@javatpoint.com | California |
| 2  | Alexandar | alexandar@javatpoint.com | New York |
| 3  | Adam  | adam@javatpoint.com | Los Angeles |
| 4  | NULL  | NULL   | Amsterdam |
+----+-----+-----+-----+

```

The value in the **name** and **email** columns are **NULL** now. It is because the REPLACE statement works as follows:

- o This statement first tries to insert a new row into the Person table. But the insertion of a new row is failed because the id = 4 already exists in the table.
- o So this statement first delete the row whose id = 4 and then insert a new row with the same id and city as Amsterdam. Since we have not specified the value for the name and email column, it was set to NULL.

## MySQL REPLACE statement to update a row

We can use the following REPLACE statement to update a row data into a table:

1. **REPLACE INTO table**
2. **SET** column1 = value1, column2 = value2;

The above syntax is similar to the **UPDATE statement** except for the REPLACE keyword. It is to note that we cannot use the **WHERE clause** with this statement.

Execute the below example that uses the REPLACE statement to update the city of the person named **Mike** from **California** to **Birmingham**.

1. **REPLACE INTO** Person
2. **SET** ID = 1,
3. **Name** = 'Mike',
4. **City** = 'Birmingham';

After verification of the table, we can see the following output:

```

MySQL 8.0 Command Line Client
mysql> REPLACE INTO Person
-> SET ID = 1,
->      Name = 'Mike',
->      City = 'Birmingham';
Query OK, 2 rows affected (0.28 sec)

mysql> SELECT * FROM Person;
+----+-----+-----+-----+
| ID | Name | Email | City |
+----+-----+-----+-----+
| 1  | Mike  | NULL  | Birmingham |
| 2  | Alexandar | alexandar@javatpoint.com | New York |
| 3  | Adam  | adam@javatpoint.com | Los Angeles |
| 4  | NULL  | NULL   | Amsterdam |
+----+-----+-----+-----+

```

If we have not specified the column's value in the **SET clause**, this command works like the **UPDATE statement**, which means the REPLACE statement will use the default value of that column.

## MySQL REPLACE to insert data from the SELECT statement.

We can use the following REPLACE INTO statement to inserts data into a table with the data returns from a query.

1. **REPLACE INTO** table1(column\_list)
2. **SELECT** column\_list
3. **FROM** table2
4. **WHERE** condition;

It is to note that the above REPLACE query is similar to the **INSERT INTO SELECT** statement. Execute the below example that uses the REPLACE INTO statement to copy a row within the same table.

1. **REPLACE INTO** Person(**Name**, **City**)
2. **SELECT Name**, **City**
3. **FROM** Person **WHERE** id = 2;

After verification of the table, we will get the following output. In this output, we can see that the copy of a row within the same table is successfully added.

The screenshot shows the MySQL 8.0 Command Line Client interface. The command `REPLACE INTO Person(Name, City)` is run with a subquery `-> SELECT Name, City` and a condition `-> FROM Person WHERE id = 2;`. The response indicates `Query OK, 1 row affected (0.15 sec)`, `Records: 1 Duplicates: 0 Warnings: 0`. Following this, the command `SELECT * FROM Person;` is run, displaying the table's contents:

ID	Name	Email	City
1	Mike	NULL	Birmingham
2	Alexandar	alexandar@javatpoint.com	New York
3	Adam	adam@javatpoint.com	Los Angeles
4	NULL	NULL	Amsterdam
5	Alexandar	NULL	New York

## MySQL INSERT ON DUPLICATE KEY UPDATE

The Insert on Duplicate Key Update statement is the extension of the INSERT statement in MySQL. When we specify the ON DUPLICATE KEY UPDATE clause in a SQL statement and a row would cause duplicate error value in a **UNIQUE or PRIMARY KEY** index column, then updation of the existing row occurs.

In other words, when we insert new values into the table, and it causes duplicate row in a UNIQUE OR PRIMARY KEY column, we will get an error message. However, if we use ON DUPLICATE KEY UPDATE clause in a **SQL** statement, it will update the old row with the new row values, whether it has a unique or primary key column.

**For example**, if column col1 is defined as UNIQUE and contains the value 10 into the table tab1, we will get a similar effect after executing the below two statements:

1. mysql> **INSERT INTO** tab1 (col1, col2, col3) **VALUES** (10,20,30) **ON DUPLICATE KEY UPDATE** col3=col3+1;
- 2.
3. mysql> **UPDATE** tab1 **SET** col3=col3+1 **WHERE** col1=1;

It makes sure that if the inserted row matched with more than one unique index into the table, then the ON DUPLICATE KEY statement only updates the **first matched unique index**. Therefore, it is not recommended to use this statement on tables that contain more than one unique index.

If the table contains AUTO\_INCREMENT primary key column and the ON DUPLICATE KEY statement tries to insert or update a row, the `Last_Insert_ID()` function returns its AUTO\_INCREMENT value.

The following are the syntax of **Insert on Duplicate Key Update** statement in MySQL:

1. **INSERT INTO** table (column\_names)
2. **VALUES** (data)
3. **ON DUPLICATE KEY UPDATE**
4. column1 = expression, column2 = expression...;

In this syntax, we can see that the **INSERT statement** only adds the ON DUPLICATE KEY UPDATE clause with a **column-value pair** assignment whenever it finds duplicate rows. The working of ON DUPLICATE KEY UPDATE clause first tries to insert the new values into the row, and if an error occurs, it will update the existing row with the new row values.

The **VALUES()** function only used in this clause, and it does not have any meaning in any other context. It returns the column values from the INSERT portion and particularly useful for multi-rows inserts.

MySQL gives the **number of affected-rows** with ON DUPLICATE KEY UPDATE statement based on the given action:

- o If we insert the new row into a table, it returns one affected-rows.

- o If we update the existing row into a table, it returns two affected-rows.
- o If we update the existing row using its current values into the table, it returns the number of affected-rows 0.

## MySQL INSERT ON DUPLICATE KEY Example

Let us understand the working of the INSERT ON DUPLICATE KEY UPDATE clause in MySQL with the help of an example.

First, create a table named "Student" using the below statement:

1. **CREATE TABLE** Student (
2. Stud\_ID **int** AUTO\_INCREMENT **PRIMARY KEY**,
3. Name **varchar**(45) **DEFAULT** NULL,
4. Email **varchar**(45) **DEFAULT** NULL,
5. City **varchar**(25) **DEFAULT** NULL
6. );

Next, insert the data into the table. Execute the following statement:

1. **INSERT INTO** Student(Stud\_ID, Name, Email, City)
2. **VALUES** (1,'Stephen', 'stephen@javatpoint.com', 'Texax'),
3. (2, 'Joseph', 'Joseph@javatpoint.com', 'Alaska'),
4. (3, 'Peter', 'Peter@javatpoint.com', 'california');

Execute the **SELECT** statement to verify the insert operation:

1. **SELECT \* FROM** Student;

We will get the output as below where we have **three** rows into the table:

```

MySQL 8.0 Command Line Client
mysql> CREATE TABLE Student (
    -> Stud_ID int AUTO_INCREMENT PRIMARY KEY,
    -> Name varchar(45) DEFAULT NULL,
    -> Email varchar(45) DEFAULT NULL,
    -> City varchar(25) DEFAULT NULL
    -> );
Query OK, 0 rows affected (0.54 sec)

mysql> INSERT INTO Student(Stud_ID, Name, Email, City)
    -> VALUES (1,'Stephen', 'stephen@javatpoint.com', 'Texax'),
    -> (2, 'Joseph', 'Joseph@javatpoint.com', 'Alaska'),
    -> (3, 'Peter', 'Peter@javatpoint.com', 'california');
Query OK, 3 rows affected (0.16 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM Student;
+-----+-----+-----+-----+
| Stud_ID | Name      | Email            | City       |
+-----+-----+-----+-----+
|      1 | Stephen   | stephen@javatpoint.com | Texax     |
|      2 | Joseph    | Joseph@javatpoint.com | Alaska    |
|      3 | Peter     | Peter@javatpoint.com | california |
+-----+-----+-----+-----+

```

Again, add one more row into the table using the below query:

1. **INSERT INTO** Student(Stud\_ID, Name, Email, City)
2. **VALUES** (4,'John', 'john@javatpoint.com', 'New York');

The above statement will add row successfully because it does not have any duplicate values.

```

MySQL 8.0 Command Line Client
mysql> INSERT INTO Student(Stud_ID, Name, Email, City)
-> VALUES (4, 'John', 'john@javatpoint.com', 'New York');
Query OK, 1 row affected (0.19 sec)

mysql> SELECT * FROM Student;
+-----+-----+-----+-----+
| Stud_ID | Name   | Email      | City    |
+-----+-----+-----+-----+
| 1      | Stephen | stephen@javatpoint.com | Texax   |
| 2      | Joseph  | Joseph@javatpoint.com | Alaska  |
| 3      | Peter   | Peter@javatpoint.com  | califonia |
| 4      | John    | john@javatpoint.com  | New York |
+-----+-----+-----+-----+

```

Finally, we are going to add a row with a duplicate value in the **Stud\_ID** column:

1. **INSERT INTO** Student(Stud\_ID, **Name**, Email, City)
2. **VALUES** (4, 'John', 'john@javatpoint.com', 'New York')
3. **ON DUPLICATE KEY UPDATE** City = 'California';

MySQL gives the following message after successful execution of the above query:

1. Query OK, 2 **rows** affected.

In the below out, we can see that the **row id=4** already exists. So the query only updates the **City New York** with **California**.

```

MySQL 8.0 Command Line Client
mysql> INSERT INTO Student(Stud_ID, Name, Email, City)
-> VALUES (4, 'John', 'john@javatpoint.com', 'New York')
-> ON DUPLICATE KEY UPDATE City = 'California';
Query OK, 2 rows affected (0.14 sec)

mysql> SELECT * FROM Student;
+-----+-----+-----+-----+
| Stud_ID | Name   | Email      | City    |
+-----+-----+-----+-----+
| 1      | Stephen | stephen@javatpoint.com | Texax   |
| 2      | Joseph  | Joseph@javatpoint.com | Alaska  |
| 3      | Peter   | Peter@javatpoint.com  | Boston  |
| 4      | John    | john@javatpoint.com  | California |
+-----+-----+-----+-----+

```

## MySQL INSERT IGNORE

Insert Ignore statement in MySQL has a special feature that ignores the **invalid rows** whenever we are inserting single or multiple rows into a table. We can understand it with the following explanation, where a table contains a primary key column.

The primary key column cannot stores duplicate values into a table. **For example, student\_roll\_number** should always be unique for every student. Similarly, the **employee\_id** in the company should always be distinct into the employee table. When we try to insert a duplicate record into a table with a primary key column, it produces an error message. However, if we use the **INSERT IGNORE** statement to add duplicate rows into a table with a primary key column, MySQL does not produce any error. This statement is preferred when we are trying to insert records in bulk, and resulting errors can interrupt the execution process. As a result, it does not store any record into a table. In such a case, the **INSERT IGNORE** statement only generates the warnings.

Below are the cases where an **INSERT IGNORE** statement avoids error:

- o When we will try to insert a duplicate key where the column of a table has a PRIMARY or UNIQUE KEY constraint.
- o When we will try to add a NULL value where the column of a table has a NOT NULL constraint.
- o When we will try to insert a record to a partitioned table where the entered values do not match the format of listed partitions.

## Syntax

The following are a syntax to use the **INSERT IGNORE** statement in MySQL:

1. **INSERT IGNORE INTO** table\_name (column\_names)
2. **VALUES** ( value\_list), ( value\_list) ....;

## MySQL INSERT IGNORE Example

Let us understand how `INSERT IGNORE` statement works in [MySQL](#) with the help of an example. First, we need to create a table named "**Student**" using the following statement:

1. **CREATE TABLE** Student (
2. `Stud_ID int AUTO_INCREMENT PRIMARY KEY,`
3. `Name varchar(45) DEFAULT NULL,`
4. `Email varchar(45) NOT NULL UNIQUE,`
5. `City varchar(25) DEFAULT NULL`
6. );

The **UNIQUE** constraint ensures that we cannot insert duplicate values into the **email column**. Next, it is required to insert the records into the table. We can execute the below statement to add data into a table:

1. **INSERT INTO** Student(`Stud_ID, Name, Email, City`)
2. **VALUES** (1, 'Stephen', 'stephen@javatpoint.com', 'Texax'),
3. (2, 'Joseph', 'Joseph@javatpoint.com', 'Alaska'),
4. (3, 'Peter', 'Peter@javatpoint.com', 'california');

Finally, execute the **SELECT** statement to verify the insert operation:

1. **SELECT \* FROM** Student;

We can see the below output where we have **three** rows into the table:

The screenshot shows the MySQL 8.0 Command Line Client interface. The command line area contains the following SQL statements and their results:

```
mysql> CREATE TABLE Student (
    -> Stud_ID int AUTO_INCREMENT PRIMARY KEY,
    -> Name varchar(45) DEFAULT NULL,
    -> Email varchar(45) NOT NULL UNIQUE,
    -> City varchar(25) DEFAULT NULL
    -> );
Query OK, 0 rows affected (0.88 sec)

mysql> INSERT INTO Student(Stud_ID, Name, Email, City)
    -> VALUES (1, 'Stephen', 'stephen@javatpoint.com', 'Texax'),
    -> (2, 'Joseph', 'Joseph@javatpoint.com', 'Alaska'),
    -> (3, 'Peter', 'Peter@javatpoint.com', 'california');
Query OK, 3 rows affected (0.18 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM Student;
+-----+-----+-----+-----+
| Stud_ID | Name      | Email            | City   |
+-----+-----+-----+-----+
|      1 | Stephen   | stephen@javatpoint.com | Texax |
|      2 | Joseph    | Joseph@javatpoint.com | Alaska |
|      3 | Peter     | Peter@javatpoint.com | califonia |
+-----+-----+-----+-----+
```

Let us execute the below statement that will try to add two records into the table:

1. **INSERT INTO** Student(`Stud_ID, Name, Email, City`)
2. **VALUES** (4, 'Donald', 'donald@javatpoint.com', 'New York'),
3. (5, 'Joseph', 'Joseph@javatpoint.com', 'Chicago');

It will produce an error: **ERROR 1062 (23000): Duplicate entry 'Joseph@javatpoint.com' for key 'student.Email'** because of the email violates the **UNIQUE** constraint.

Now, let us see what happened if we use the `INSERT IGNORE` statement into the above query:

1. **INSERT IGNORE INTO** Student(`Stud_ID, Name, Email, City`)
2. **VALUES** (4, 'Donald', 'donald@javatpoint.com', 'New York'),
3. (5, 'Joseph', 'Joseph@javatpoint.com', 'Chicago');

MySQL will produce a message: one row added, and the other row was ignored.

1. 1 row affected, 1 warning(s): 1062 Duplicate entry **for key** email.
2. Records: 2 Duplicates: 1 Warning: 1

We can see the detailed warning using the **SHOW WARNINGS** command:

```
MySQL 8.0 Command Line Client
mysql> SHOW WARNINGS;
+-----+-----+
| Level | Code | Message
+-----+-----+
| Warning | 1062 | Duplicate entry 'Joseph@javatpoint.com' for key 'student.Email' |
+-----+-----+
```

Thus, we can say that if we use the **INSERT IGNORE** statement, MySQL gives a warning instead of issuing an error.

## MySQL INSERT IGNORE and STRICT mode

In MySQL, STRICT MODE handles **invalid or missing values** that are going to be added into a table using **INSERT OR UPDATE** statement. If the strict mode is **ON**, and we are trying to add invalid values into a table using the [INSERT statement](#), the statement is aborted, and we will get an error message.

However, if we use the **INSERT IGNORE** command, MySQL produces a warning message instead of throwing an error. Also, this statement tries to truncate values to make them valid before inserting it into the table.

Let us understand it with the help of an example. First, we are going to create a table named "**Test**" using the below statement:

1. **CREATE TABLE** Test (
2. ID **int AUTO\_INCREMENT PRIMARY KEY**,
3. Name **varchar(5) NOT NULL**
4. );

In the above table, the name column only accepts the string whose length is less than or equal to **five characters**. Now, execute the below statement to insert the records into a table.

1. **INSERT INTO** Test(**Name**)
2. **VALUES ('Peter'), ('John')**;

We can see that the specified name validates the name column constraint so it will be added successfully. Execute the **SELECT** statement to verify the result. It will give the output as below:

```
MySQL 8.0 Command Line Client
mysql> CREATE TABLE Test (
    -> ID int AUTO_INCREMENT PRIMARY KEY,
    -> Name varchar(5) NOT NULL
    -> );
Query OK, 0 rows affected (0.87 sec)

mysql> INSERT INTO Test(Name)
    -> VALUES ('Peter'), ('John');
Query OK, 2 rows affected (0.14 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM Test;
+----+-----+
| ID | Name  |
+----+-----+
| 1  | Peter |
| 2  | John  |
+----+-----+
```

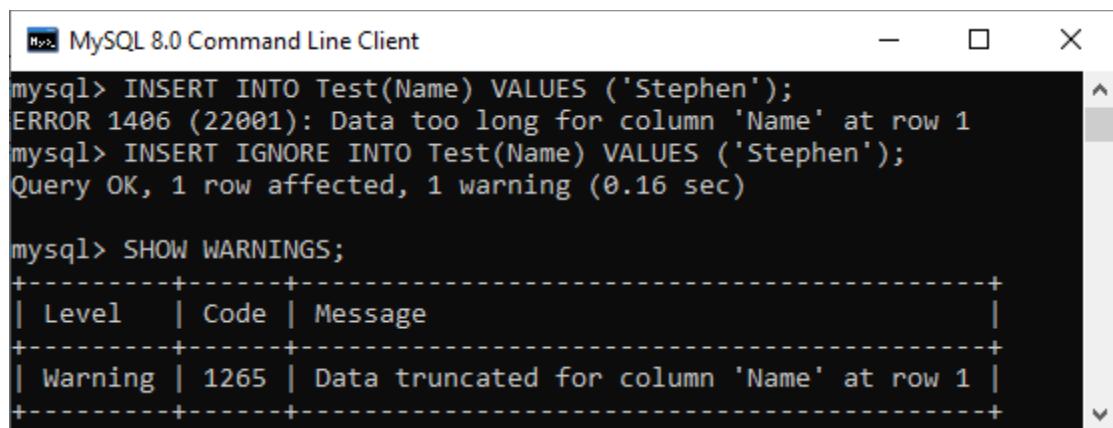
Next, insert the name whose length is greater than five:

1. **INSERT INTO** Test(**Name**) **VALUES ('Stephen')**;

MySQL does not add values and gives an error message because the strict mode is ON. However, if we use the **INSERT IGNORE** statement to insert the same string, it will give the warning message instead of throwing an error.

1. **INSERT IGNORE INTO** Test(**Name**) **VALUES ('Stephen')**;

Finally, we have executed the **SHOW WARNINGS** command to check the warning message. The below output explains it more clearly that shows MySQL tries to truncate data before inserting it into a table.



```
MySQL 8.0 Command Line Client
mysql> INSERT INTO Test(Name) VALUES ('Stephen');
ERROR 1406 (22001): Data too long for column 'Name' at row 1
mysql> INSERT IGNORE INTO Test(Name) VALUES ('Stephen');
Query OK, 1 row affected, 1 warning (0.16 sec)

mysql> SHOW WARNINGS;
+-----+-----+
| Level | Code | Message
+-----+-----+
| Warning | 1265 | Data truncated for column 'Name' at row 1 |
+-----+-----+
```

## MySQL INSERT INTO SELECT

Sometimes we want to insert data of one table into the other table in the same or different database. It is not very easy to enter these data using the INSERT query manually. We can optimize this process with the use of MySQL INSERT INTO SELECT query. It allows us to populate the MySQL tables quickly. This section will cover the INSERT INTO SELECT command, syntax, and its use cases.

The **INSERT INTO SELECT statement**

in **MySQL**

allows us to insert values into a table where data comes from a SELECT query. In other words, **this query copies data from one table and inserts them in the other table**. We must consider the following point before using this statement:

- o The data types in source and target tables must be the same.
- o The existing records in the target table should be unaffected.

The INSERT INTO SELECT command is advantageous when we need to copy data from one table to another table or to summarize data from more than one table into a single table.

### Syntax

Earlier, we have used the **INSERT command** for adding single or multiple records into a table along with listing column values in the **VALUES** clause as follows:

1. **INSERT INTO** table\_name (column\_list)
2. **VALUES** (value\_list);

The following is the basic syntax that illustrates the use of the INSERT INTO SELECT command in MySQL. If we want to copy all data from one table into another table, we can use the below statement:

1. **INSERT INTO** table\_name2
2. **SELECT \* FROM** table\_name1
3. **WHERE** condition;

From the **MySQL version 8.0.19**, we can use a **TABLE statement** in place of SELECT query to get the same output, as shown below:

1. **INSERT INTO** table2 **TABLE** table1;

Here, TABLE **table1** is equivalent to **SELECT \* FROM table1**. When we want to add all records from the source table into the target table without filtering the values, it is used.

If we want to copy only some columns from one table to another table, we can use the below statement:

1. **INSERT INTO** table\_name2 (column\_list)
2. **SELECT** column\_list
3. **FROM** table\_name1
4. **WHERE** condition;

In this syntax, we have used a SELECT statement instead of using the VALUES clause. Here SELECT command retrieves values from one or more tables.

## Parameter Explanation

The INSERT INTO SELECT statement uses the following parameters:

**table\_name1**: It is the name of a source table.

**table\_name2**: It is the name of a target table where we will copy source table data.

**column\_list**: It represents the column names of the table.

**condition**: it is used to filter the table data.

## MySQL INSERT INTO SELECT Example

Let us understand how the INSERT INTO SELECT statement works in MySQL with the help of an example. First, we need to create a table named "**person**" using the statement given below:

1. **CREATE TABLE** person (
2. id **int** AUTO\_INCREMENT **PRIMARY KEY**,
3. **name varchar**(45) NOT NULL,
4. email **varchar**(45) NOT NULL,
5. city **varchar**(25) NOT NULL
6. );

Next, we will insert values into the table. We can execute the below statement to add data into a table:

1. **INSERT INTO** person (id, **name**, email, city)
2. **VALUES** (1,'Stephen', 'stephen@javatpoint.com', 'Texas'),
3. (2, 'Joseph', 'Joseph@javatpoint.com', 'Alaska'),
4. (3, 'Peter', 'Peter@javatpoint.com', 'Texas'),
5. (4,'Donald', 'donald@javatpoint.com', 'New York'),
6. (5, 'Kevin', 'kevin@javatpoint.com', 'Texas');

We can verify the data by executing the SELECT statement:

1. **SELECT \* FROM** person;

After executing the query, we can see the below output where we have five rows into the table:

The screenshot shows the MySQL 8.0 Command Line Client window. The command entered is 'SELECT \* FROM person;'. The resulting table has four columns: id, name, email, and city. The data is as follows:

id	name	email	city
1	Stephen	stephen@javatpoint.com	Texas
2	Joseph	Joseph@javatpoint.com	Alaska
3	Peter	Peter@javatpoint.com	Texas
4	Donald	donald@javatpoint.com	New York
5	Kevin	kevin@javatpoint.com	Texas

Suppose we want to insert a person's name who belongs to Texas City into another table. The following query is used to search all person who locates in Texas:

1. **SELECT** name, email, city
2. **FROM** person
3. **WHERE** city = 'Texas';

Now, we will create another table named **person\_info** that have a same number of column, and data types in the same order as of the above table:

1. **CREATE TABLE** person\_info (
2. person\_id **int** AUTO\_INCREMENT **PRIMARY KEY**,
3. person\_name **varchar**(45) NOT NULL,
4. email **varchar**(45) NOT NULL,
5. city **varchar**(25) NOT NULL
6. );

Second, we will use the **INSERT INTO SELECT** statement to insert persons located in Texas from the **person table** into the **person\_info table**:

1. **INSERT INTO** person\_info (person\_name, email, city)
2. **SELECT** name, email, city
3. **FROM** person
4. **WHERE** city = 'Texas';

After executing this statement, we can verify the insert operation using the **SELECT** query. We will get the below output where all persons located in the Texas City inserted successfully.

The screenshot shows the MySQL 8.0 Command Line Client window. The command entered is 'SELECT \* FROM person\_info;'. The resulting table has four columns: person\_id, person\_name, email, and city. The data is as follows:

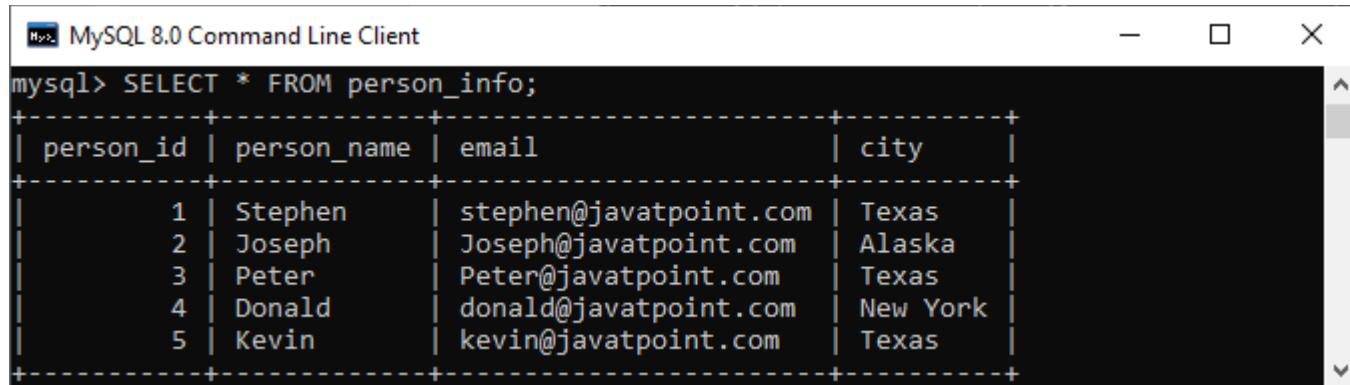
person_id	person_name	email	city
1	Stephen	stephen@javatpoint.com	Texas
2	Peter	Peter@javatpoint.com	Texas
3	Kevin	kevin@javatpoint.com	Texas

3 rows in set (0.00 sec)

Suppose we want to insert all person's table data into the **person\_info table** without filtering any values; we can do this using the below statement:

1. **INSERT INTO** person\_info **Table** person;

Execute the SELECT statement to verify the data. Here is the output:



The screenshot shows the MySQL 8.0 Command Line Client window. The command entered is `SELECT * FROM person_info;`. The output is a table with four columns: person\_id, person\_name, email, and city. The data is as follows:

person_id	person_name	email	city
1	Stephen	stephen@javatpoint.com	Texas
2	Joseph	Joseph@javatpoint.com	Alaska
3	Peter	Peter@javatpoint.com	Texas
4	Donald	donald@javatpoint.com	New York
5	Kevin	kevin@javatpoint.com	Texas

## MySQL Indexes

# How to Create Index in MySQL

An index is a data structure that allows us to add indexes in the existing table. It enables you to improve the faster retrieval of records on a database table. It creates an **entry** for each value of the indexed columns. We use it to quickly find the record without searching each row in a database table whenever the table is accessed. We can create an index by using one or more **columns** of the table for efficient access to the records.

When a table is created with a primary key or unique key, it automatically creates a special index named **PRIMARY**. We called this index as a clustered index. All indexes other than PRIMARY indexes are known as a non-clustered index or secondary index.

## Need for Indexing in MySQL

Suppose we have a contact book that contains names and mobile numbers of the user. In this contact book, we want to find the mobile number of Martin Williamson. If the contact book is an unordered format means the name of the contact book is not sorted alphabetically, we need to go over all pages and read every name until we will not find the desired name that we are looking for. This type of searching name is known as sequential searching.

To find the name and contact of the user from table **contactbooks**, generally, we used to execute the following query:

1. mysql> **SELECT** mobile\_number **FROM** contactbooks **WHERE** first\_name = 'Martin' AND last\_name = 'Taybu';

This query is very simple and easy. Although it finds the phone number and name of the user fast, the database searches entire rows of the table until it will not find the rows that you want. Assume, the contactbooks table contains **millions** of rows, then, without an index, the data retrieval takes a lot of time to find the result. In that case, the database indexing plays an important role in returning the desired result and improves the overall performance of the query.

## MySQL CREATE INDEX Statement

Generally, we create an index at the time of table creation in the database. The following statement creates a table with an index that contains two columns col2 and col3.

1. mysql> **CREATE TABLE** t\_index(
2.   col1 **INT PRIMARY KEY**,
3.   col2 **INT NOT NULL**,
4.   col3 **INT NOT NULL**,
5.   col4 **VARCHAR**(20),
6.   **INDEX** (col2,col3)
7. );

If we want to add index in table, we will use the CREATE INDEX statement as follows:

1. mysql> **CREATE INDEX** [index\_name] **ON** [table\_name] (**column** names)

In this statement, **index\_name** is the name of the index, **table\_name** is the name of the table to which the index belongs, and the **column\_names** is the list of columns.

Let us add the new index for the column col4, we use the following statement:

1. mysql> **CREATE INDEX** ind\_1 **ON** t\_index(col4);

By default, MySQL allowed index type **BTREE** if we have not specified the type of index. The following table shows the different types of an index based on the storage engine of the table.

SN	Storage Engine	Index Type
1.	InnoDB	BTREE
2.	Memory/Heap	HASH, BTREE
3.	MYISAM	BTREE

### Example

In this example, we are going to create a table **student** and perform the CREATE INDEX statement on that table.

#### Table Name: student

studentid	firstname	lastname	class	age
2	Mark	Boucher	EE	22
3	Michael	Clark	CS	18
4	Peter	Fleming	CS	22
5	Virat	Kohli	EC	23
6	Martin	Taybu	EE	24
7	John	Tucker	CS	25
NULL	NULL	NULL	NULL	NULL

Now, execute the following statement to return the result of the student whose **class** is **CS branch**:

1. mysql> **SELECT** studentid, firstname, lastname **FROM** student **WHERE** class = 'CS';

This statement will give the following output:

studentid	firstname	lastname
1	Ricky	Ponting
3	Michael	Clark
4	Peter	Fleming
7	John	Tucker
NULL	NULL	NULL

In the above table, we can see the four rows that are indicating the students whose class is the CS branch.

If you want to see how MySQL performs this query internally, execute the following statement:

1. mysql> **EXPLAIN** **SELECT** studentid, firstname, lastname **FROM** student **WHERE** class = 'CS';

You will get the output below. Here, MySQL scans the whole table that contains seven rows to find the student whose class is the CS branch.

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	student	NULL	ALL	NULL	NULL	NULL	NULL	7	14.29	Using where

Now, let us create an index for a class column using the following statement.

1. mysql> **CREATE INDEX** class **ON** student (class);

After executing the above statement, the index is created successfully. Now, run the below statement to see how MySQL internally performs this query.

1. mysql> **EXPLAIN** **SELECT** studentid, firstname, lastname **FROM** student **WHERE** class = 'CS';

The above statement gives output, as shown below:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	student	NULL	ref	class	class	42	const	4	100.00	NULL

In this output, MySQL finds four rows from the class index without scanning the whole table. Hence, it increases the speed of retrieval of records on a database table.

If you want to **show** the indexes of a table, execute the following statement:

1. mysql> **SHOW INDEXES** **FROM** student;

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type
student	0	PRIMARY	1	studentid	A	4	NULL	NULL		BTREE
student	0	studentid_UNIQUE	1	studentid	A	4	NULL	NULL		BTREE
student	1	class	1	class	A	3	NULL	NULL		BTREE

It will give the following output.

## MySQL Drop Index

MySQL allows a DROP INDEX statement to remove the existing index from the table. To delete an index from a table, we can use the following query:

1. mysql>**DROP INDEX** index\_name **ON** table\_name [algorithm\_option | lock\_option];

If we want to delete an index, it requires two things:

- o First, we have to specify the name of the index that we want to remove.
- o Second, name of the table from which your index belongs.

The Drop Index syntax contains two optional options, which are Algorithm and Lock for reading and writing the tables during the index modifications. Let us explain both in detail:

### Algorithm Option

The algorithm\_option enables us to specify the specific algorithm for removing the index in a table. The syntax of **algorithm\_option** are as follows:

1. Algorithm [=] {**DEFAULT** | **INPLACE** | **COPY**}

The Drop Index syntax supports mainly two algorithms which are **INPLACE** and **COPY**.

**COPY:** This algorithm allows us to copy one table into another new table row by row and then DROP Index statement performed on this new table. On this table, we cannot perform an INSERT and UPDATE statement for data manipulation.

**INPLACE:** This algorithm allows us to rebuild a table instead of copy the original table. We can perform all data manipulation operations on this table. On this table, [MySQL](#) issues an exclusive metadata lock during the index removal.

**Note:** If you not defined the algorithm clause, MySQL uses the INPLACE algorithm. If INPLACE is not supported, it uses the COPY algorithm. The DEFAULT algorithm works the same as without using any algorithm clause with the Drop index statement.

### Lock Option

This clause enables us to control the level of concurrent reads and writes during the index removal. The syntax of **lock\_option** are as follows:

1. LOCK [=] {**DEFAULT**|**NONE**|**SHARED**|**EXCLUSIVE**}

In the syntax, we can see that the lock\_option contains **four modes** that are **DEFAULT**, **NONE**, **SHARED**, and **EXCLUSIVE**. Now, we are going to discuss all the modes in detail:

**SHARED:** This mode supports only concurrent reads, not concurrent writes. When the concurrent reads are not supported, it gives an error.

**DEFAULT:** This mode can have the maximum level of concurrency for a specified algorithm. It will enable concurrent reads and writes if supported otherwise enforces exclusive mode.

**NONE:** You have concurrent read and write if this mode is supported. Otherwise, it gives an error.

**EXCLUSIVE:** This mode enforces exclusive access.

### Example

First, execute the following command to show the indexes available in the table.

1. mysql> SHOW INDEXES **FROM** student;

It will give the following output.

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type
student	0	PRIMARY	1	studentid	A	4	NULL	NULL		BTREE
student	0	studentid_UNIQUE	1	studentid	A	4	NULL	NULL		BTREE
student	1	class	1	class	A	3	NULL	NULL		BTREE

In the output, we can see that there are three indexes available. Now, execute the following statement to removes the **class** index from table **student**.

1. mysql> **DROP INDEX** class **ON** student;

Again, execute the SHOW INDEXES statement to verify the index is removed or not. After performing this statement, we will get the following output, where only two indexes are available.

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type
student	0	PRIMARY	1	studentid	A	7	NULL	NULL		BTREE
student	0	studentid_UNIQUE	1	studentid	A	7	NULL	NULL		BTREE

### Example using Algorithm and Lock

The following statement drops the **age** index form the student table using an algorithm and a lock option.

1. mysql> **DROP INDEX** age **ON** student **ALGORITHM** = INPLACE **LOCK** = **DEFAULT**;

### MySQL Drop PRIMARY Key Index

In some cases, the table contains a PRIMARY index that was created whenever you create a table with a primary key or unique key. In that case, we need to execute the following command because the PRIMARY is a reserved word.

1. mysql> **DROP INDEX PRIMARY ON** table\_name;

To remove the primary key index from the student table, execute the following statement:

1. mysql> **DROP INDEX PRIMARY ON** student;

## MySQL Show Indexes

We can get the index information of a table using the Show Indexes statement. This statement can be written as:

1. mysql> **SHOW INDEXES FROM** table\_name;

In the above syntax, we can see that if we want to get the index of a table, it requires to specify the **table\_name** after the **FROM** keyword. After the successful execution of the statement, it will return the index information of a table in the current database.

If we want to get the index information of a table in a different database or database to which you are not connected, [MySQL](#) allows us to specify the database name with the Show Indexes statement. The following statement explains it more clearly:

1. mysql> **SHOW INDEXES FROM** table\_name **IN** database\_name;

The above statement can also be written as:

1. mysql> **SHOW INDEXES FROM** database\_name.table\_name;

**Note:** It is noted that **Index** and **Keys** both are synonyms of **Indexes**, and **IN** is the synonyms of **FROM** keyword. Therefore, we can also write the Show Indexes statement with these synonyms as below:

1. mysql> **SHOW INDEXES IN** table\_name **FROM** database\_name;

OR,

1. mysql> **SHOW KEYS FROM** table\_name **IN** database\_name;

The SHOW INDEX query returns the following fields/information:

**Table:** It contains the name of the table.

**Non\_unique:** It returns 1 if the index contains duplicates. Otherwise, it returns 0.

**Key\_name:** It is the name of an index. If the table contains a primary key, the index name is always PRIMARY.

**Seq\_in\_index:** It is the sequence number of the column in the index that starts from 1.

**Column\_name:** It contains the name of a column.

**Collation:** It gives information about how the column is sorted in the index. It contains values where **A** represents ascending, **D** represents descending, and **Null** represents not sorted.

**Cardinality:** It gives an estimated number of unique values in the index table where the higher cardinality represents a greater chance of using indexes by MySQL.

**Sub\_part:** It is a prefix of the index. It has a NULL value if all the column of the table is indexed. When the column is partially indexed, it will return the number of indexed characters.

**Packed:** It tells how the key is packed. Otherwise, it returns NULL.

**NULL:** It contains **blank** if the column does not have NULL value; otherwise, it returns YES.

**Index\_type:** It contains the name of the index method like BTREE, HASH, RTREE, FULLTEXT, etc.

**Comment:** It contains the index information when they are not described in its column. For example, when the index is disabled, it returns disabled.

**Index\_column:** When you create an index with **comment** attributes, it contains the comment for the specified index.

**Visible:** It contains YES if the index is visible to the query optimizer, and if not, it contains NO.

**Expression:** MySQL 8.0 supports **functional key parts** that affect both **expression** and **column\_name** columns. We can understand it more clearly with the below points:

- o For functional parts, the expression column represents expression for the key part, and column\_name represents NULL.
- o For the non-functional part, the expression represents NULL, and column\_name represents the column indexed by the key part.

## MySQL SHOW INDEX Example

Here, we are going to create a table **student\_info** that contains the student id, name, age, mobile number, and email details. Execute the following command to create a table:

```
1. CREATE TABLE `student_info` (
2.   `studentid` int NOT NULL AUTO_INCREMENT,
3.   `name` varchar(45) DEFAULT NULL,
4.   `age` varchar(3) DEFAULT NULL,
5.   `mobile` varchar(20) DEFAULT NULL,
6.   `email` varchar(25) DEFAULT NULL,
7.   PRIMARY KEY (`studentid`),
8.   UNIQUE KEY `email_UNIQUE` (`email`)
9. )
```

Next, we create an index on this table by the following command:

1. mysql> **CREATE INDEX** mobile **ON** student\_info (mobile) INVISIBLE;
- 2.
3. mysql> **CREATE INDEX** name **ON** student\_info (**name**) COMMENT 'Student Name';

Now, execute the following command that returns the all index information from the student\_info table:

1. mysql> **SHOW INDEXES** **FROM** student\_info;

We will get the output below:

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible	Expression
student_info	0	PRIMARY	1	studentid	A	0	HULL	HULL		BTREE			YES	NULL
student_info	0	email_UNIQUE	1	email	A	0	HULL	HULL	YES	BTREE			YES	NULL
student_info	1	mobile	1	mobile	A	0	HULL	HULL	YES	BTREE			NO	NULL
student_info	1	name	1	name	A	0	HULL	HULL	YES	BTREE	Student Name		YES	NULL

## Filter Index Information

We can filter the index information using **where clause**. The following statement can be used to filter the index information:

1. Mysql> SHOW INDEXES **FROM** table\_name **where** condition;

### Example

If you want to get only **invisible** indexes of the student\_info table, execute the following command:

1. mysql> SHOW INDEXES **FROM** student\_info **WHERE** visible = 'NO';

It will give the following output:

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible	Expression
student_info	1	mobile	1	mobile	A	0	NULL	NULL	YES	BTREE			NO	NULL

## MySQL UNIQUE INDEX

Indexing is a process to find an unordered list into an ordered list that allows us to retrieve records faster. **It creates an entry for each value that appears in the index columns.** It helps in maximizing the query's efficiency while searching on tables in MySQL. Without indexing, we need to scan the whole table to find the relevant information. The working of MySQL indexing is similar to the book index.

Generally, we use the primary key constraint to enforce the uniqueness value of one or more columns. But, we can use only one primary key for each table. So if we want to make multiple sets of columns with unique values, the primary key constraint will not be used.

MySQL allows another constraint called the **UNIQUE INDEX** to enforce the uniqueness of values in one or more columns. We can create more than one UNIQUE index in a single table, which is not possible with the primary key constraint.

### Syntax

The following is a generic syntax used to create a unique index in MySQL table:

1. **CREATE UNIQUE INDEX** index\_name
2. **ON** table\_name (index\_column1, index\_column2,...);

MySQL allows another approach to enforcing the uniqueness value in one or more columns using the **UNIQUE Key** statement. We can read more information about the **UNIQUE KEY** [here](#).

If we use a **UNIQUE** constraint in the table, MySQL automatically creates a **UNIQUE** index behind the scenes. The following statement explains how to create a unique constraint when we create a table.

1. **CREATE TABLE** table\_name(
2. col1 col\_definition,
3. col2 col\_definition,
4. ...
5. **[CONSTRAINT** constraint\_name]
6. **UNIQUE** Key (column\_name(s))
7. );

**NOTE:** It is recommended to use the constraint name while creating a table. If we omit the constraint name, MySQL generates a name for this column automatically.

### UNIQUE Index and NULL

NULL values in MySQL considers distinct values similar to other databases. Hence, we can store multiple NULL values in the **UNIQUE** index column. This feature of MySQL sometimes reported as a bug, but it is not a bug.

## MySQL UNIQUE Index Examples

Let us understand it with the help of an example. Suppose we want to manage the employee details in a database application where we need email columns unique. Execute the following statement that creates a table "**Employee\_Detail**" with a **UNIQUE** constraint:

1. **CREATE TABLE** Employee\_Detail(
2. ID **int** AUTO\_INCREMENT **PRIMARY KEY**,
3. Name **varchar**(45),
4. Email **varchar**(45),
5. Phone **varchar**(15),
6. City **varchar**(25),
7. **UNIQUE KEY** unique\_email (Email)
8. );

If we execute the below statement, we can see that MySQL created a UNIQUE index for **Email** column of Employee\_Detail table:

1. SHOW INDEXES **FROM** Employee\_Detail;

In the below screen, we can see that the Email column is created as a unique index.

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type
employee_detail	0	PRIMARY	1	ID	A	0	NULL	NULL		BTREE
employee_detail	0	unique_email	1	Email	A	0	NULL	NULL	YES	BTREE

Next, we are going to insert records to the table using the following statements:

1. **INSERT INTO** Employee\_Detail(ID, Name, Email, Phone, City)
2. **VALUES** (1, 'Peter', 'peter@javatpoint.com', '49562959223', 'Texas'),
3. (2, 'Suzi', 'suzi@javatpoint.com', '70679834522', 'California'),
4. (3, 'Joseph', 'joseph@javatpoint.com', '09896765374', 'Alaska');

The above statement executed successfully because all columns are unique. If we insert a record whose email is **suzi@javatpoint.com**, we will get the duplicate error message.

1. mysql> **INSERT INTO** Employee\_Detail(ID, Name, Email, Phone, City)
2. **VALUES** (2, 'Suzi', 'suzi@javatpoint.com', '70679834522', 'Texas');

The following output explains all of the above steps more clearly:

```

MySQL 8.0 Command Line Client

mysql> CREATE TABLE Employee_Detail(
->     ID int AUTO_INCREMENT PRIMARY KEY,
->     Name varchar(45),
->     Email varchar(45),
->     Phone varchar(15),
->     City varchar(25),
->     UNIQUE KEY unique_email (Email)
-> );
Query OK, 0 rows affected (1.08 sec)

mysql> INSERT INTO Employee_Detail(ID, Name, Email, Phone, City)
-> VALUES (1, 'Peter', 'peter@javatpoint.com', '49562959223', 'Texas'),
-> (2, 'Suzi', 'suzi@javatpoint.com', '70679834522', 'California'),
-> (3, 'Joseph', 'joseph@javatpoint.com', '09896765374', 'Alaska');
Query OK, 3 rows affected (0.72 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> INSERT INTO Employee_Detail(ID, Name, Email, Phone, City)
-> VALUES (2, 'Suzi', 'suzi@javatpoint.com', '70679834522', 'Texas');
ERROR 1062 (23000): Duplicate entry '2' for key 'employee_detail.PRIMARY'

```

Suppose we want the **Name** and **Phone** of the Employee\_Detail table is also unique. In this case, we will use the below statement to create a UNIQUE index for those columns:

1. **CREATE UNIQUE INDEX** index\_name\_phone
2. **ON** Employee\_Detail (**Name**, Phone);

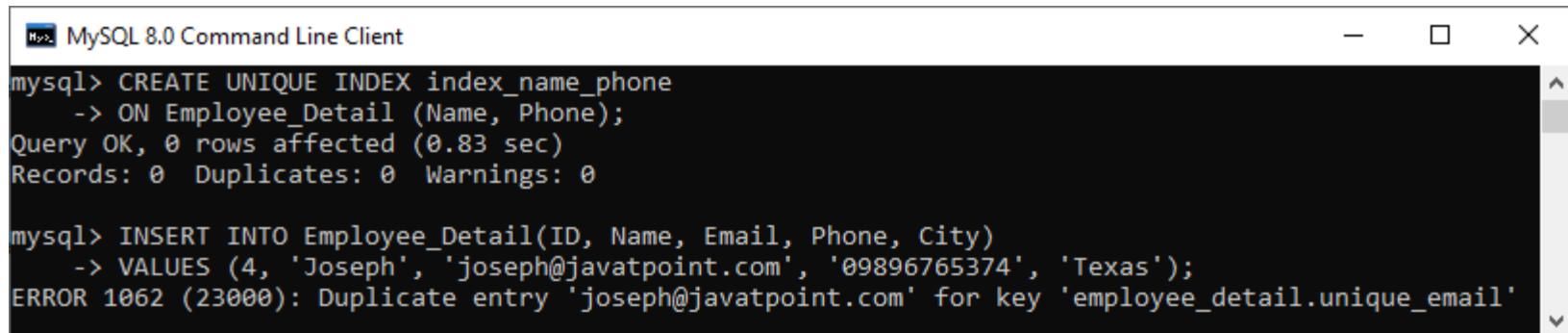
If we execute the **SHOW INDEX** statement again, we can see that MySQL created a UNIQUE index **index\_name\_phone** for name and phone columns also.

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type
employee_detail	0	PRIMARY	1	ID	A	3	NULL	NULL		BTREE
employee_detail	0	unique_email	1	Email	A	3	NULL	NULL	YES	BTREE
employee_detail	0	index_name_phone	1	Name	A	3	NULL	NULL	YES	BTREE
employee_detail	0	index_name_phone	2	Phone	A	3	NULL	NULL	YES	BTREE

Adding this record into the table produces an error. It is because of the combination of a name and phone already exists.

1. mysql> **INSERT INTO** Employee\_Detail(ID, **Name**, Email, Phone, City)
2. **VALUES** (4, 'Joseph', 'joseph@javatpoint.com', '09896765374', 'Texas');

Look into this output:



The screenshot shows a terminal window titled "MySQL 8.0 Command Line Client". The command "CREATE UNIQUE INDEX index\_name\_phone" is run, followed by "ON Employee\_Detail (Name, Phone);". The response indicates "Query OK, 0 rows affected (0.83 sec)". Below this, another "INSERT INTO" statement is run with values (4, 'Joseph', 'joseph@javatpoint.com', '09896765374', 'Texas'). The response shows an "ERROR 1062 (23000): Duplicate entry 'joseph@javatpoint.com' for key 'employee\_detail.unique\_email'" message.

## MySQL Clustered Index

An index is a separate data structure that allows us to add indexes in the existing table. It enables you to improve the faster retrieval of records on a database table. It creates an entry for each value of the indexed columns.

A clustered index is actually a table where the data for the rows are stored. It defines the order of the table data based on the key values that can be sorted in only one way. In the database, each table can have only one clustered index. In a relational database, if the table column contains a primary key or unique key, MySQL allows you to create a clustered index named **PRIMARY** based on that specific column.

### Characteristics

The essential characteristics of a clustered index are as follows:

- It helps us to store data and indexes at the same time.
- It stores data in only one way based on the key values.
- Key lookup.
- They are scan and index seek.
- Clustered index always use one or more column for creating an index.

### Advantages

The main advantages of the clustered index are as follows:

- It helps us to maximize the cache hits and minimizes the page transfer.
- It is an ideal option for range or group with max, min, and count queries.
- At the start of the range, it uses a location mechanism for finding an index entry.

### Disadvantages

The main disadvantages of the clustered index are as follows:

- It contains many insert records in a non-sequential order.
- It creates many constant page splits like data pages or index pages.
- It always takes a long time to update the records.
- It needs extra work for SQL queries, such as insert, updates, and deletes.

### Clustered Index on InnoDB Tables

MySQL InnoDB table must have a clustered index. The InnoDB table uses a clustered index for optimizing the speed of most common lookups and DML (Data Manipulation Language) operations like INSERT, UPDATE, and DELETE command.

When the primary key is defined in an InnoDB table, MySQL always uses it as a clustered index named **PRIMARY**. If the table does not contain a primary key column, MySQL searches for the **unique key**. In the unique key, all columns are **NOT NULL** and use it as a clustered index. Sometimes, the table does not have a primary key nor unique key, then MySQL

internally creates hidden clustered index **GEN\_CLUST\_INDEX** that contains the values of row id. Thus, there is only one clustered index in the InnoDB table.

The indexes other than the PRIMARY Indexes (clustered indexes) are known as a secondary index or non-clustered indexes. In the MySQL InnoDB tables, every record of the non-clustered index has primary key columns for both row and columns. MySQL uses this primary key value for searching a row in the clustered index or secondary index.

## Example

In the below statement, the PRIMARY KEY is a clustered index.

```
1. CREATE TABLE `student_info` (
2.   `studentid` int NOT NULL AUTO_INCREMENT,
3.   `name` varchar(45) DEFAULT NULL,
4.   `age` varchar(3) DEFAULT NULL,
5.   `mobile` varchar(20) DEFAULT NULL,
6.   `email` varchar(25) DEFAULT NULL,
7.   PRIMARY KEY (`studentid`), //clustered index
8.   UNIQUE KEY `email_UNIQUE` (`email`)
9. )
```

## Difference between MySQL Clustered and Non-Clustered Index

The difference between clustered and non-clustered index is the most famous question in the database related interviews. Both indexes have the same physical structure and are stored as a BTREE structure in the MySQL server database. In this section, we are going to explain the most popular differences between them.

Indexing in MySQL is a process that helps us to return the requested data from the table very fast. If the table does not have an index, it scans the whole table for the requested data. MySQL allows two different types of Indexing:

1. Clustered Index
2. Non-Clustered Index

Let us first discuss clustered and non-clustered indexing in brief.

### What is a Clustered Index?

A clustered index is a table where the data for the rows are stored. It defines the order of the table data based on the key values that can be sorted in only one direction. In the database, each table can contain only one clustered index. In a relational database, if the table column contains a primary key or unique key, MySQL allows you to create a clustered index named **PRIMARY** based on that specific column.

#### Keep Watching

Competitive questions on Structures in Hindi  
00:00/03:34

## Example

The following example explains how the clustered index created in MySQL:

```
1. CREATE TABLE Student
2. ( post_id INT NOT NULL AUTO_INCREMENT, user_id INT NOT NULL,
3.   CONSTRAINT Post_PK
4.   PRIMARY KEY (user_id, post_id), //clustered index
5.   CONSTRAINT post_id_UQ
6.   UNIQUE (post_id)
7. ) ENGINE = InnoDB ;
```

## Characteristics

Following are the essential characteristics of a clustered index:

- o It enables us to store data and indexes together.
- o It stores data in only one way based on the key values.

- o Key lookup.
- o It supports index scan and index seek data operations.
- o Clustered index always uses one or more columns for creating an index.

## What is a Non-Clustered Index?

The indexes other than PRIMARY indexes (clustered indexes) called a non-clustered index. The non-clustered indexes are also known as secondary indexes. The non-clustered index and table data are both stored in different places. It is not able to sort (ordering) the table data. The non-clustered indexing is the same as a book where the content is written in one place, and the index is at a different place. MySQL allows a table to store one or more than one non-clustered index. The non-clustered indexing improves the performance of the queries which uses keys without assigning primary key.

### Example

1. //It will **create** non-clustered **index**
2. **CREATE** NonClustered **INDEX** index\_name **ON** table\_name (column\_name **ASC**);

## Characteristics

Following are the essential characteristics of a non-clustered index:

- o It stores only key values.
- o It allows accessing secondary data that has pointers to the physical rows.
- o It helps in the operation of an index scan and seeks.
- o A table can contain one or more than one non-clustered index.
- o The non-clustered index row stores the value of a non-clustered key and row locator.

## Clustered VS Non-Clustered Index

Let us see some of the popular differences between clustered and non-clustered indexes through the tabular form:

Parameter	Clustered Index	Non-Clustered Index
Definition	A clustered index is a table where the data for the rows are stored. In a relational database, if the table column contains a primary key, MySQL automatically creates a clustered index named <b>PRIMARY</b> .	The indexes other than PRIMARY indexes (clustered indexes) called a non-clustered index. The non-clustered indexes are also known as secondary indexes.
Use for	It can be used to sort the record and store the index in physical memory.	It creates a logical ordering of data rows and uses pointers for accessing the physical data files.
Size	Its size is large.	Its size is small in comparison to a clustered index.
Data Accessing	It accesses the data very fast.	It has slower accessing power in comparison to the clustered index.
Storing Method	It stores records in the leaf node of an index.	It does not store records in the leaf node of an index that means it takes extra space for data.
Additional Disk Space	It does not require additional reports.	It requires an additional space to store the index separately.

Type of Key	It uses the primary key as a clustered index.	It can work with unique constraints that act as a composite key.
Contains in Table	A table can only one clustered index.	A table can contain one or more than a non-clustered index.
Index Id	A clustered index always contains an index id of 0.	A non-clustered index always contains an index id>0.

### MySQL Clauses

## MySQL WHERE Clause

MySQL WHERE Clause is used with SELECT, INSERT, UPDATE and DELETE clause to filter the results. It specifies a specific position where you have to do the operation.

### Syntax:

1. **WHERE** conditions;

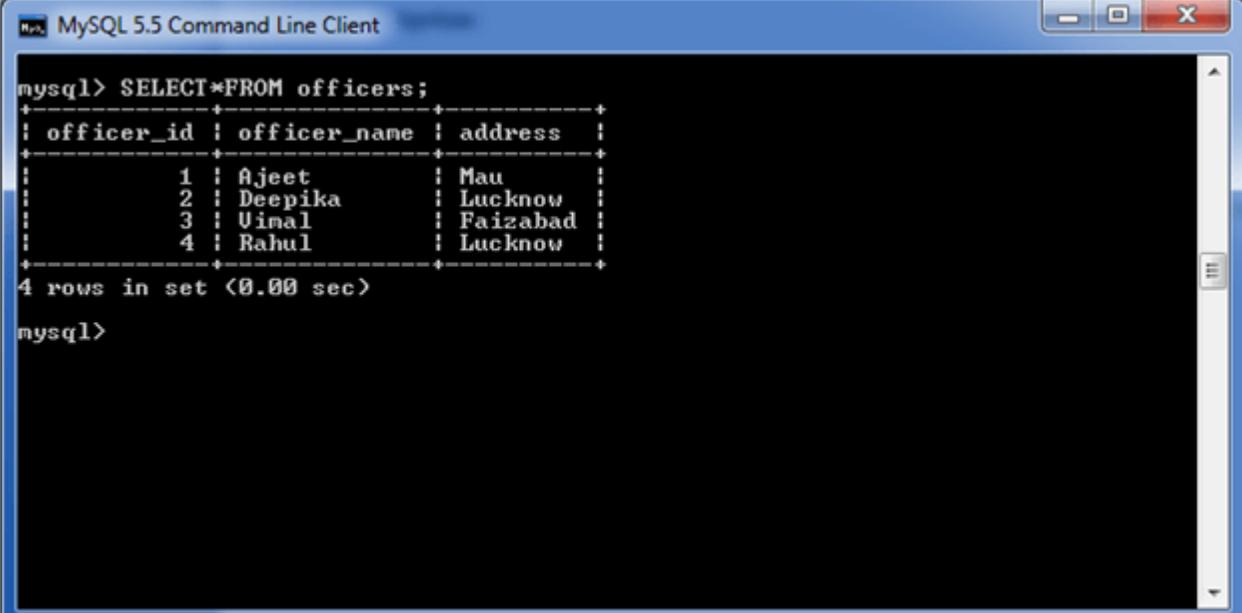
### Parameter:

conditions: It specifies the conditions that must be fulfilled for records to be selected.

## MySQL WHERE Clause with single condition

Let's take an example to retrieve data from a table "officers".

### Table structure:



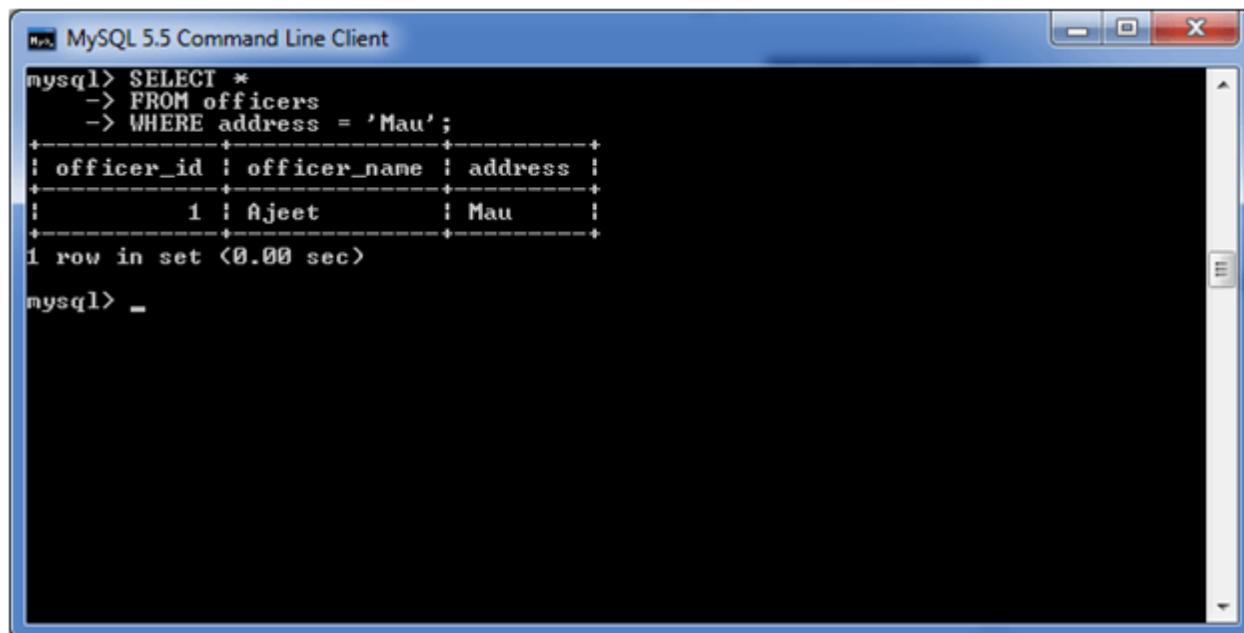
```
MySQL 5.5 Command Line Client
mysql> SELECT * FROM officers;
+-----+-----+-----+
| officer_id | officer_name | address |
+-----+-----+-----+
| 1 | Ajeeet | Mau |
| 2 | Deepika | Lucknow |
| 3 | Uinal | Faizabad |
| 4 | Rahul | Lucknow |
+-----+-----+-----+
4 rows in set (0.00 sec)

mysql>
```

### Execute this query:

1. **SELECT \***
2. **FROM** officers
3. **WHERE** address = 'Mau';

### Output:



MySQL 5.5 Command Line Client

```
mysql> SELECT *  
-> FROM officers  
-> WHERE address = 'Mau';  
+-----+-----+-----+  
| officer_id | officer_name | address |  
+-----+-----+-----+  
| 1 | Ajeet | Mau |  
+-----+-----+-----+  
1 row in set (0.00 sec)  
  
mysql>
```

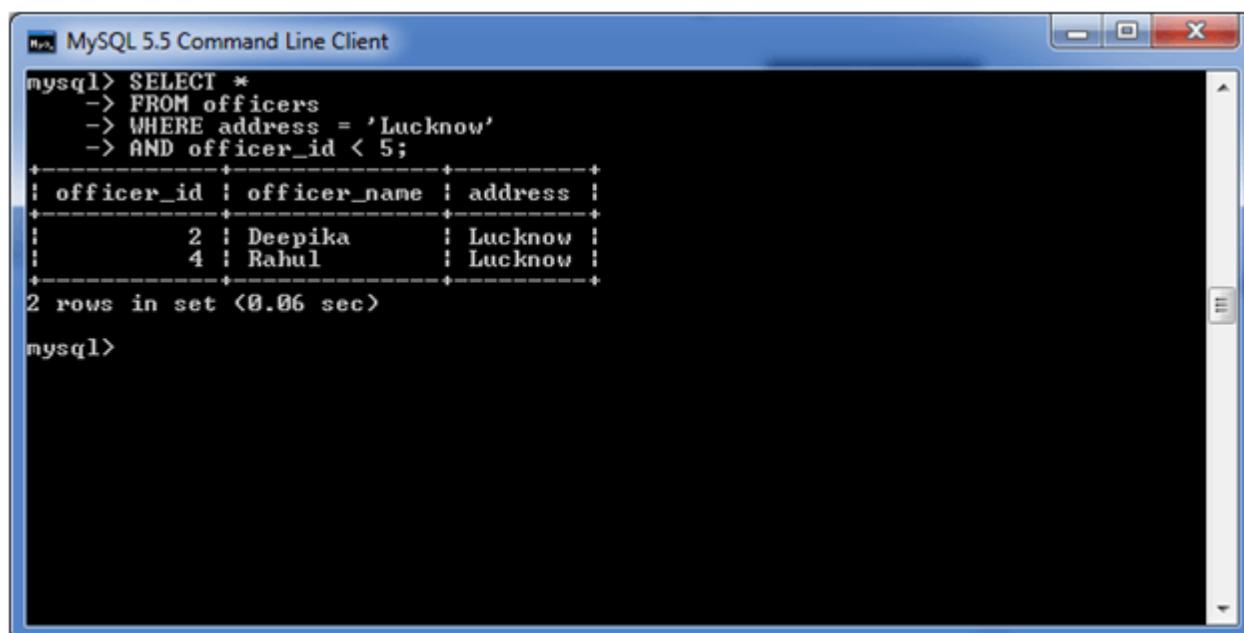
## MySQL WHERE Clause with AND condition

In this example, we are retrieving data from the table "officers" with AND condition.

**Execute the following query:**

1. **SELECT \***
2. **FROM officers**
3. **WHERE address = 'Lucknow'**
4. **AND officer\_id < 5;**

**Output:**



MySQL 5.5 Command Line Client

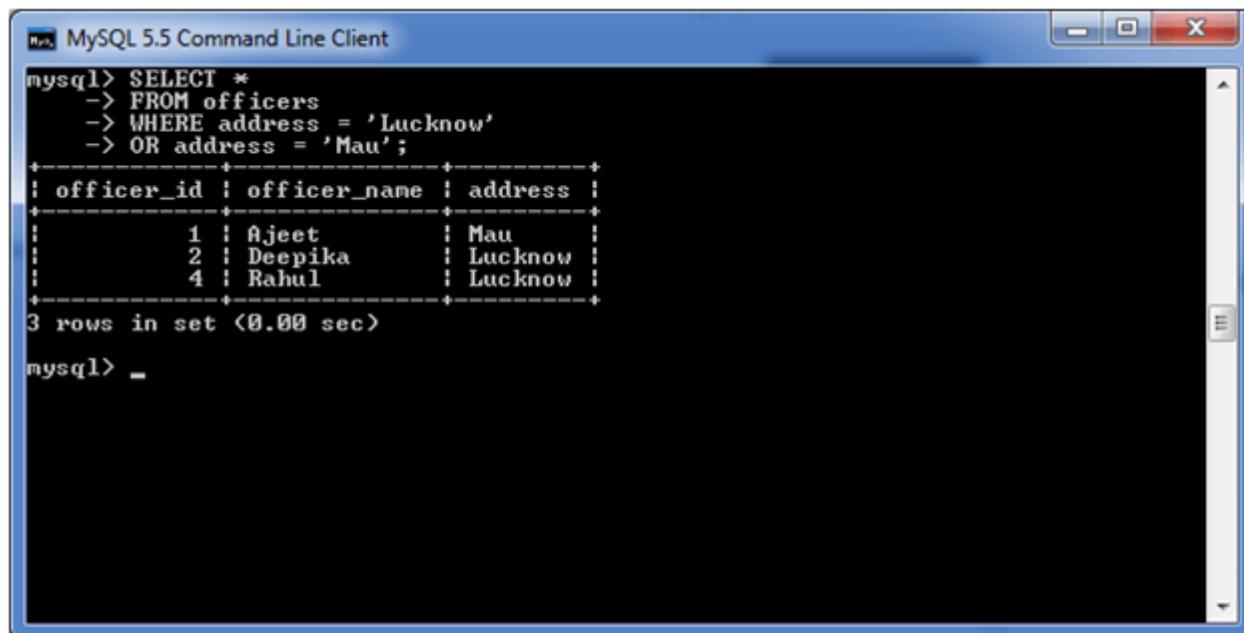
```
mysql> SELECT *  
-> FROM officers  
-> WHERE address = 'Lucknow'  
-> AND officer_id < 5;  
+-----+-----+-----+  
| officer_id | officer_name | address |  
+-----+-----+-----+  
| 2 | Deepika | Lucknow |  
| 4 | Rahul | Lucknow |  
+-----+-----+-----+  
2 rows in set (0.06 sec)  
  
mysql>
```

## WHERE Clause with OR condition

**Execute the following query:**

1. **SELECT \***
2. **FROM officers**
3. **WHERE address = 'Lucknow'**
4. **OR address = 'Mau';**

**Output:**



MySQL 5.5 Command Line Client

```
mysql> SELECT *  
-> FROM officers  
-> WHERE address = 'Lucknow'  
-> OR address = 'Mau';  
+-----+  
| officer_id | officer_name | address |  
+-----+  
| 1          | Ajeet        | Mau     |  
| 2          | Deepika      | Lucknow |  
| 4          | Rahul         | Lucknow |  
+-----+  
3 rows in set (0.00 sec)  
mysql> _
```

## MySQL WHERE Clause with combination of AND & OR conditions

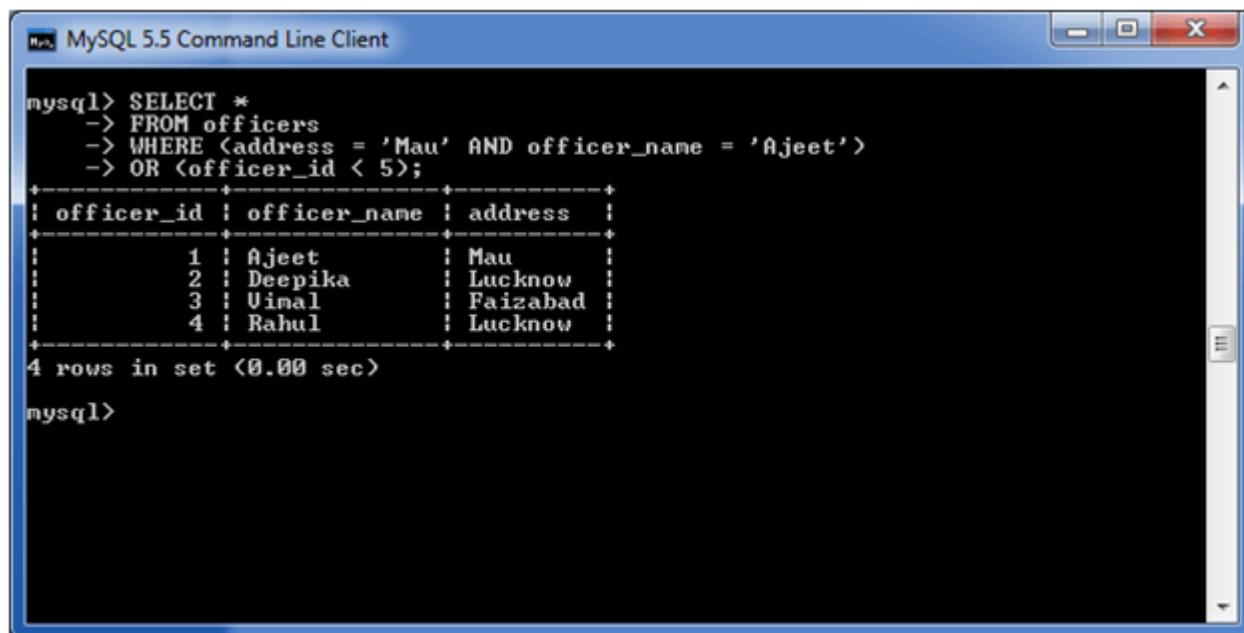
You can also use the AND & OR conditions altogether with the WHERE clause.

**See this example:**

**Execute the following query:**

1. **SELECT \***
2. **FROM** officers
3. **WHERE** (address = 'Mau' AND officer\_name = 'Ajeet')
4. **OR** (officer\_id < 5);

**Output:**



MySQL 5.5 Command Line Client

```
mysql> SELECT *  
-> FROM officers  
-> WHERE address = 'Mau' AND officer_name = 'Ajeet'  
-> OR officer_id < 5;  
+-----+  
| officer_id | officer_name | address |  
+-----+  
| 1          | Ajeet        | Mau     |  
| 2          | Deepika      | Lucknow |  
| 3          | Vimal         | Faizabad |  
| 4          | Rahul         | Lucknow |  
+-----+  
4 rows in set (0.00 sec)  
mysql> _
```

## MySQL Distinct Clause

MySQL DISTINCT clause is used to remove duplicate records from the table and fetch only the unique records. The DISTINCT clause is only used with the SELECT statement.

**Syntax:**

1. **SELECT DISTINCT** expressions
2. **FROM** tables
3. **[WHERE conditions];**

## Parameters

**expressions:** specify the columns or calculations that you want to retrieve.

**tables:** specify the name of the tables from where you retrieve records. There must be at least one table listed in the FROM clause.

**WHERE conditions:** It is optional. It specifies the conditions that must be met for the records to be selected.

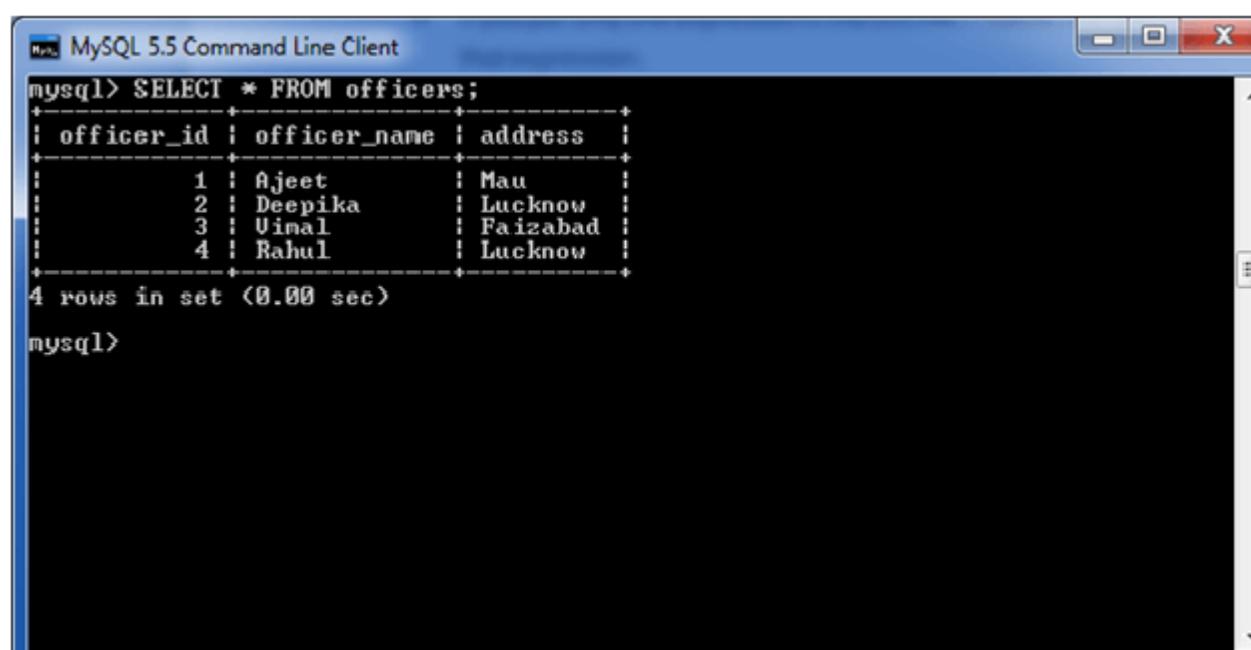
**Note:**

- If you put only one expression in the DISTINCT clause, the query will return the unique values for that expression.
- If you put more than one expression in the DISTINCT clause, the query will retrieve unique combinations for the expressions listed.
- In MySQL, the DISTINCT clause doesn't ignore NULL values. So if you are using the DISTINCT clause in your SQL statement, your result set will include NULL as a distinct value.

## MySQL DISTINCT Clause with single expression

If you use a single expression then the MySQL DISTINCT clause will return a single field with unique records (no duplicate record).

**See the table:**

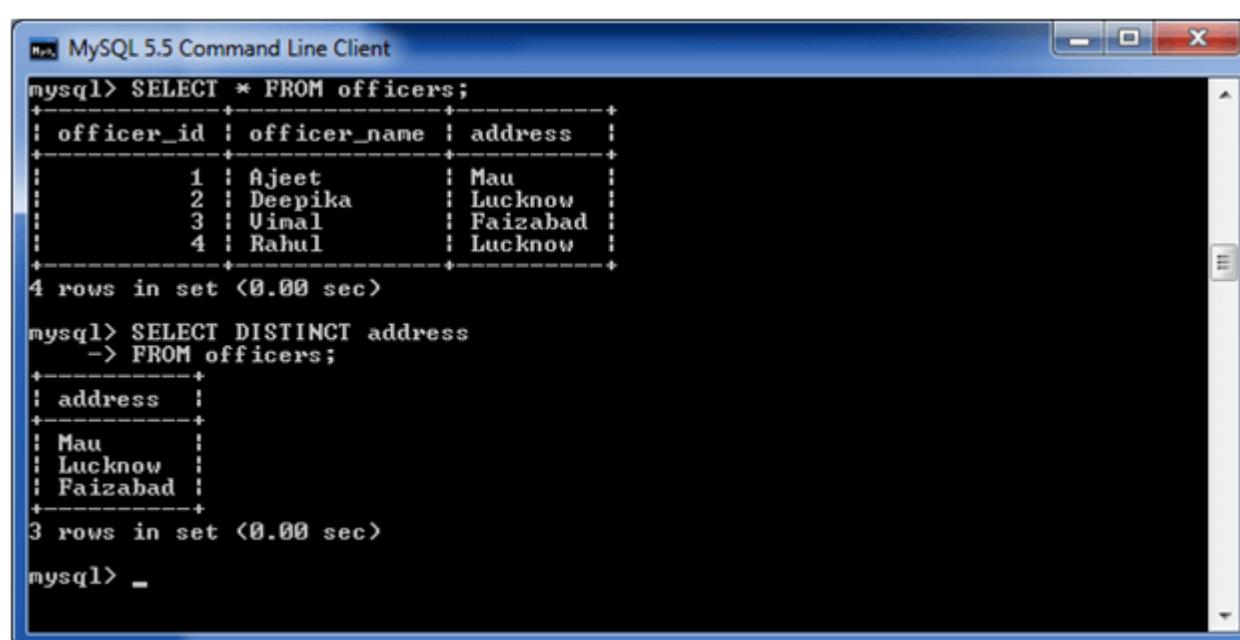


```
mysql> SELECT * FROM officers;
+-----+-----+-----+
| officer_id | officer_name | address |
+-----+-----+-----+
| 1 | Ajeeet | Mau |
| 2 | Deepika | Lucknow |
| 3 | Uimal | Faizabad |
| 4 | Rahul | Lucknow |
+-----+-----+-----+
4 rows in set (0.00 sec)

mysql>
```

**Use the following query:**

1. **SELECT DISTINCT** address
2. **FROM** officers;



```
mysql> SELECT * FROM officers;
+-----+-----+-----+
| officer_id | officer_name | address |
+-----+-----+-----+
| 1 | Ajeeet | Mau |
| 2 | Deepika | Lucknow |
| 3 | Uimal | Faizabad |
| 4 | Rahul | Lucknow |
+-----+-----+-----+
4 rows in set (0.00 sec)

mysql> SELECT DISTINCT address
-> FROM officers;
+-----+
| address |
+-----+
| Mau |
| Lucknow |
| Faizabad |
+-----+
3 rows in set (0.00 sec)

mysql>
```

## MySQL DISTINCT Clause with multiple expressions

If you use multiple expressions with DISTINCT Clause then MySQL DISTINCT clause will remove duplicates from more than one field in your SELECT statement.

**Use the following query:**

1. **SELECT DISTINCT** officer\_name, address
2. **FROM** officers;

```

MySQL 5.5 Command Line Client
+-----+
| address |
+-----+
| Mau      |
| Lucknow |
| Faizabad|
+-----+
3 rows in set <0.00 sec>

mysql> SELECT DISTINCT officer_name, address
-> FROM officers;
+-----+-----+
| officer_name | address  |
+-----+-----+
| Ajeet        | Mau      |
| Deepika      | Lucknow |
| Vimal        | Faizabad|
| Rahul        | Lucknow |
+-----+-----+
4 rows in set <0.00 sec>

mysql>

```

## MySQL FROM Clause

The MySQL FROM Clause is used to select some records from a table. It can also be used to retrieve records from multiple tables using JOIN condition.

### Syntax:

1. **FROM** table1
2. [ { **INNER** JOIN | **LEFT** [OUTER] JOIN| **RIGHT** [OUTER] JOIN } table2
3. **ON** table1.column1 = table2.column1 ]

### Parameters

**table1** and **table2**: specify tables used in the MySQL statement. The two tables are joined based on table1.column1 = table2.column1.

### Note:

- o If you are using the FROM clause in a MySQL statement then at least one table must have been selected.
- o If you are using two or more tables in the MySQL FROM clause, these tables are generally joined using INNER or OUTER joins.

## MySQL FROM Clause: Retrieve data from one table

The following query specifies how to retrieve data from a single table.

### Use the following Query:

1. **SELECT \***
2. **FROM** officers
3. **WHERE** officer\_id <= 3;

```

MySQL 5.5 Command Line Client
+-----+
| address |
+-----+
| Mau      |
| Lucknow |
| Faizabad|
+-----+
3 rows in set <0.00 sec>

mysql> SELECT *
-> FROM officers
-> WHERE officer_id <= 3;
+-----+-----+
| officer_id | officer_name | address  |
+-----+-----+
| 1          | Ajeet        | Mau      |
| 2          | Deepika      | Lucknow |
| 3          | Vimal        | Faizabad|
+-----+-----+
3 rows in set <0.00 sec>

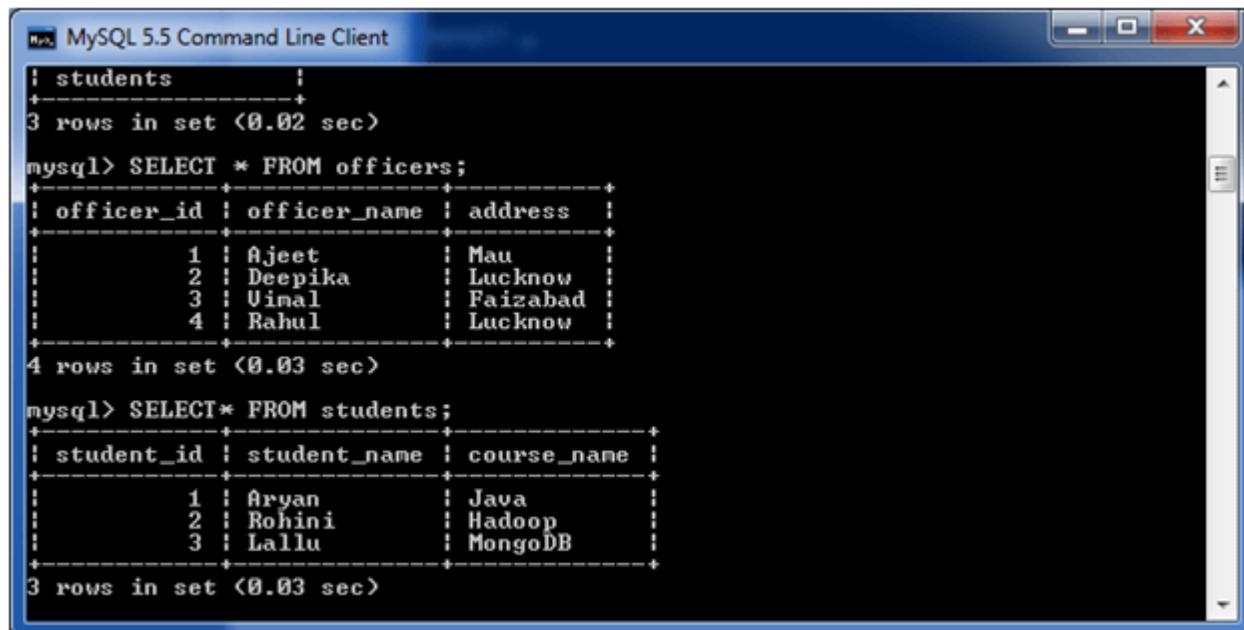
mysql>

```

## MySQL FROM Clause: Retrieve data from two tables with inner join

Let's take an example to retrieve data from two tables using INNER JOIN.

Here, we have two tables "officers" and "students".



The screenshot shows the MySQL 5.5 Command Line Client interface. It displays the results of two SELECT statements. The first statement, 'SELECT \* FROM officers;', returns 4 rows with columns officer\_id, officer\_name, and address. The second statement, 'SELECT \* FROM students;', returns 3 rows with columns student\_id, student\_name, and course\_name. The data is presented in tabular form with column headers and row numbers.

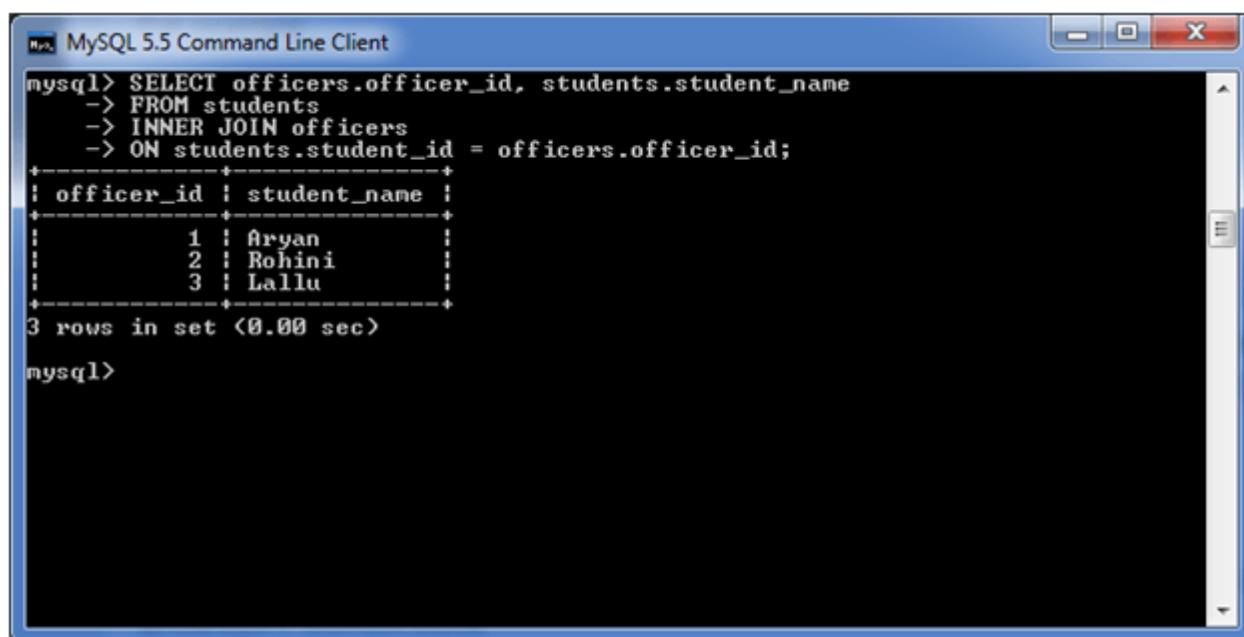
officers		
officer_id	officer_name	address
1	Ajeet	Mau
2	Deepika	Lucknow
3	Uinal	Faizabad
4	Rahul	Lucknow

students		
student_id	student_name	course_name
1	Aryan	Java
2	Rohini	Hadoop
3	Lallu	MongoDB

Execute the following query:

1. **SELECT** officers.officer\_id, students.student\_name
2. **FROM** students
3. **INNER JOIN** officers
4. **ON** students.student\_id = officers.officer\_id;



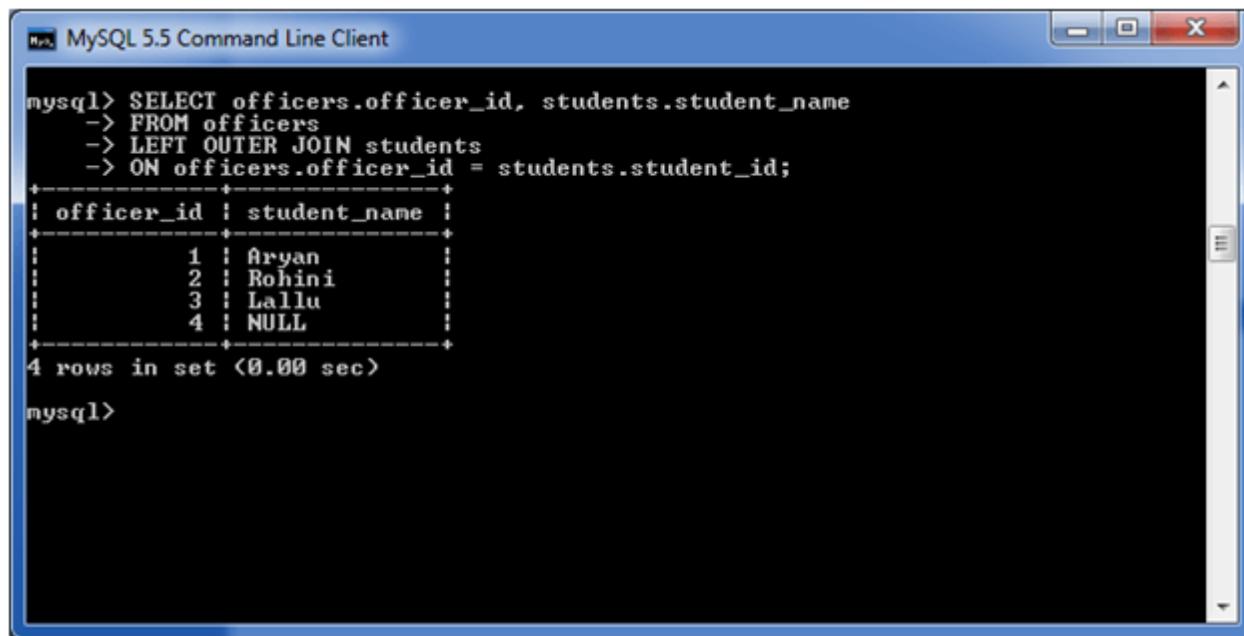
The screenshot shows the MySQL 5.5 Command Line Client interface. It displays the results of a complex SELECT query involving an INNER JOIN. The query selects officer\_id and student\_name from the students table, joining it with the officers table on the condition that students.student\_id equals officers.officer\_id. The result shows 3 rows with columns officer\_id and student\_name.

officer_id	student_name
1	Aryan
2	Rohini
3	Lallu

## MySQL FROM Clause: Retrieve data from two tables using outer join

Execute the following query:

1. **SELECT** officers.officer\_id, students.student\_name
2. **FROM** officers
3. **LEFT OUTER JOIN** students
4. **ON** officers.officer\_id = students.student\_id;



The screenshot shows the MySQL 5.5 Command Line Client window. The query executed is:

```
mysql> SELECT officers.officer_id, students.student_name
-> FROM officers
-> LEFT OUTER JOIN students
-> ON officers.officer_id = students.student_id;
```

The result set is:

officer_id	student_name
1	Aryan
2	Rohini
3	Lallu
4	NULL

4 rows in set (0.00 sec)

mysql>

## MySQL ORDER BY Clause

The MySQL ORDER BY Clause is used to sort the records in ascending or descending order.

### Syntax:

1. **SELECT** expressions
2. **FROM** tables
3. [**WHERE** conditions]
4. **ORDER BY** expression [ **ASC** | **DESC** ];

### Parameters

**expressions:** It specifies the columns that you want to retrieve.

**tables:** It specifies the tables, from where you want to retrieve records. There must be at least one table listed in the FROM clause.

**WHERE conditions:** It is optional. It specifies conditions that must be fulfilled for the records to be selected.

**ASC:** It is optional. It sorts the result set in ascending order by expression (default, if no modifier is provided).

**DESC:** It is also optional. It sorts the result set in descending order by expression.

**Note:** You can use MySQL ORDER BY clause in a **SELECT statement**, **SELECT LIMIT statement**, and **DELETE LIMIT statement**.

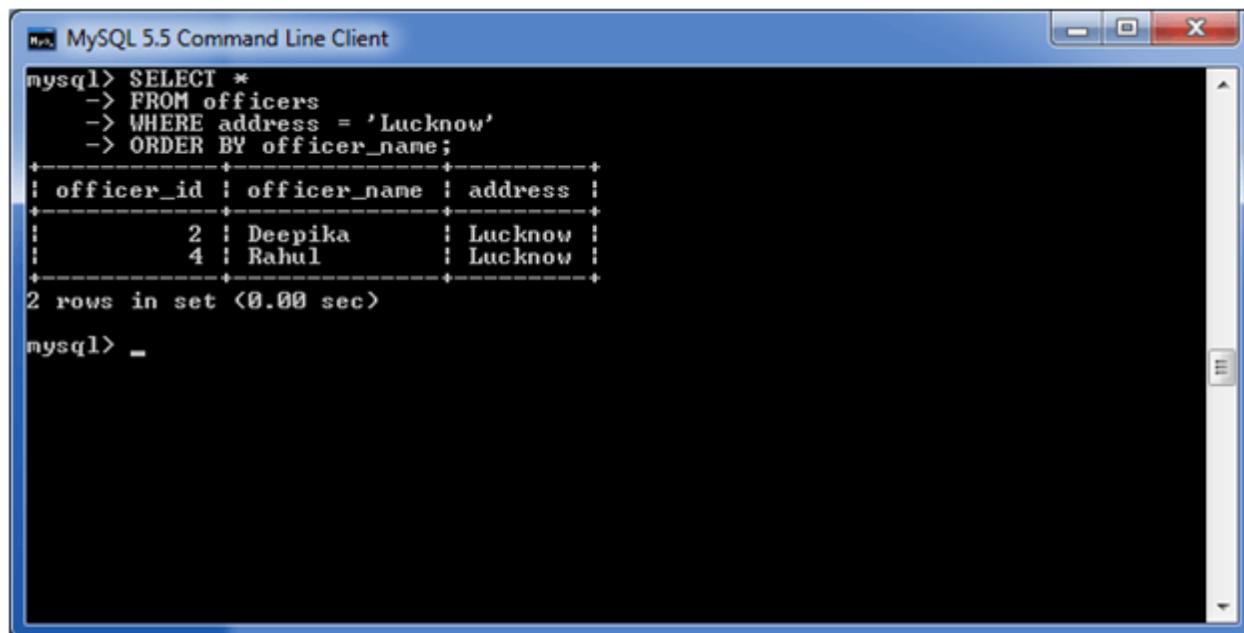
## MySQL ORDER BY: without using ASC/DESC attribute

If you use MySQL ORDER BY clause without specifying the ASC and DESC modifier then by default you will get the result in ascending order.

### Execute the following query:

1. **SELECT \***
2. **FROM** officers
3. **WHERE** address = 'Lucknow'
4. **ORDER BY** officer\_name;

### Output:



MySQL 5.5 Command Line Client

```
mysql> SELECT *  
-> FROM officers  
-> WHERE address = 'Lucknow'  
-> ORDER BY officer_name;  
+-----+-----+  
| officer_id | officer_name | address |  
+-----+-----+  
| 2 | Deepika | Lucknow |  
| 4 | Rahul | Lucknow |  
+-----+-----+  
2 rows in set (0.00 sec)  
mysql> _
```

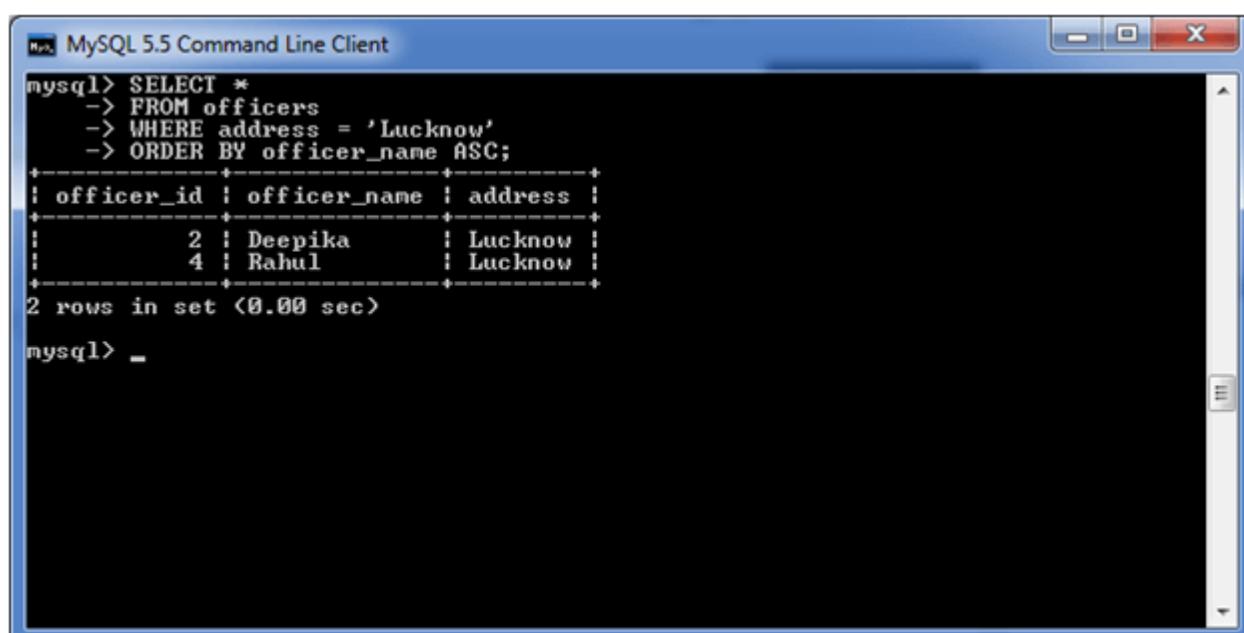
## MySQL ORDER BY: with ASC attribute

Let's take an example to retrieve the data in ascending order.

Execute the following query:

1. **SELECT \***
2. **FROM officers**
3. **WHERE address = 'Lucknow'**
4. **ORDER BY officer\_name **ASC**;**

Output:

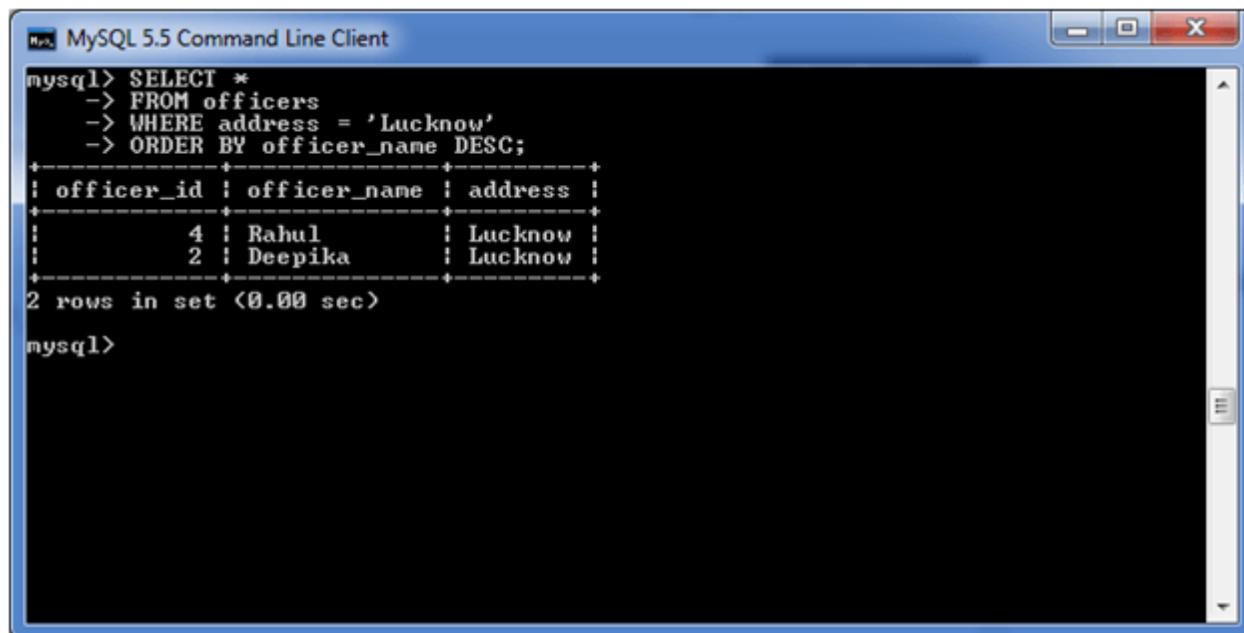


MySQL 5.5 Command Line Client

```
mysql> SELECT *  
-> FROM officers  
-> WHERE address = 'Lucknow'  
-> ORDER BY officer_name ASC;  
+-----+-----+  
| officer_id | officer_name | address |  
+-----+-----+  
| 2 | Deepika | Lucknow |  
| 4 | Rahul | Lucknow |  
+-----+-----+  
2 rows in set (0.00 sec)  
mysql> _
```

## MySQL ORDER BY: with DESC attribute

1. **SELECT \***
2. **FROM officers**
3. **WHERE address = 'Lucknow'**
4. **ORDER BY officer\_name **DESC**;**



MySQL 5.5 Command Line Client

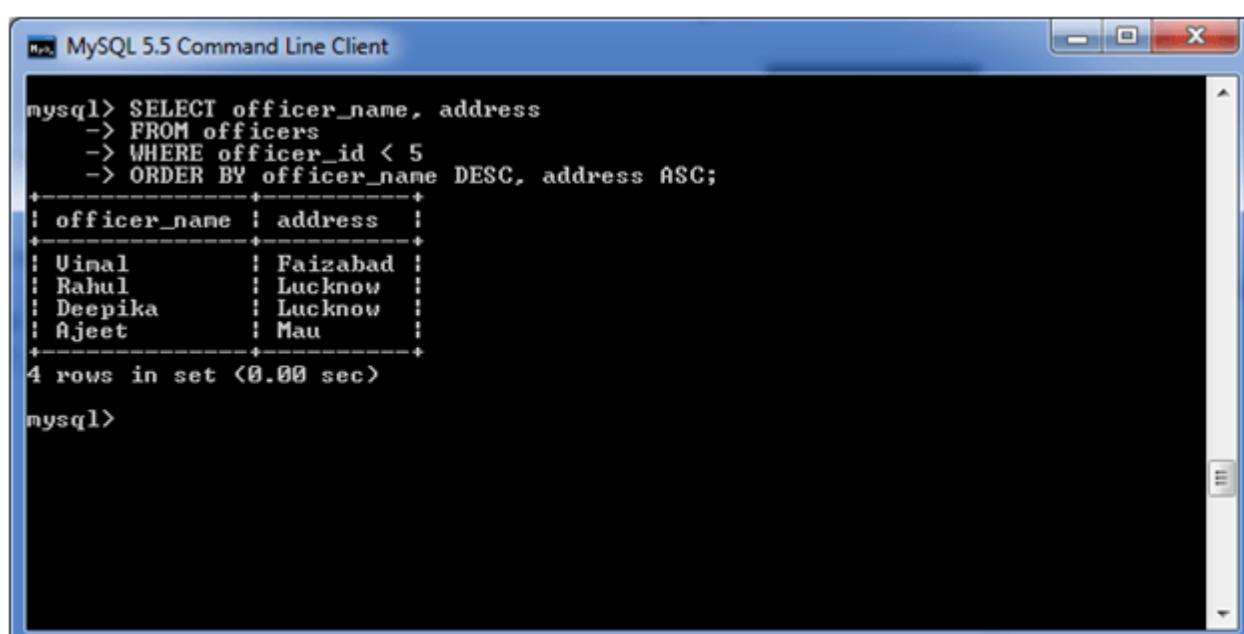
```
mysql> SELECT *  
-> FROM officers  
-> WHERE address = 'Lucknow'  
-> ORDER BY officer_name DESC;  
+-----+-----+  
| officer_id | officer_name | address |  
+-----+-----+  
| 4 | Rahul | Lucknow |  
| 2 | Deepika | Lucknow |  
+-----+  
2 rows in set <0.00 sec>  
mysql>
```

## MySQL ORDER BY: using both ASC and DESC attributes

Execute the following query:

1. **SELECT** officer\_name, address
2. **FROM** officers
3. **WHERE** officer\_id < 5
4. **ORDER BY** officer\_name **DESC**, address **ASC**;

Output:



MySQL 5.5 Command Line Client

```
mysql> SELECT officer_name, address  
-> FROM officers  
-> WHERE officer_id < 5  
-> ORDER BY officer_name DESC, address ASC;  
+-----+-----+  
| officer_name | address |  
+-----+-----+  
| Vimal | Faizabad |  
| Rahul | Lucknow |  
| Deepika | Lucknow |  
| Ajeet | Mau |  
+-----+  
4 rows in set <0.00 sec>  
mysql>
```

## MySQL GROUP BY Clause

The MySQL GROUP BY Clause is used to collect data from multiple records and group the result by one or more column. It is generally used in a SELECT statement.

You can also use some aggregate functions like COUNT, SUM, MIN, MAX, AVG etc. on the grouped column.

Syntax:

1. **SELECT** expression1, expression2, ... expression\_n,
2. aggregate\_function (expression)
3. **FROM** tables
4. [**WHERE** conditions]
5. **GROUP BY** expression1, expression2, ... expression\_n;

## Parameters

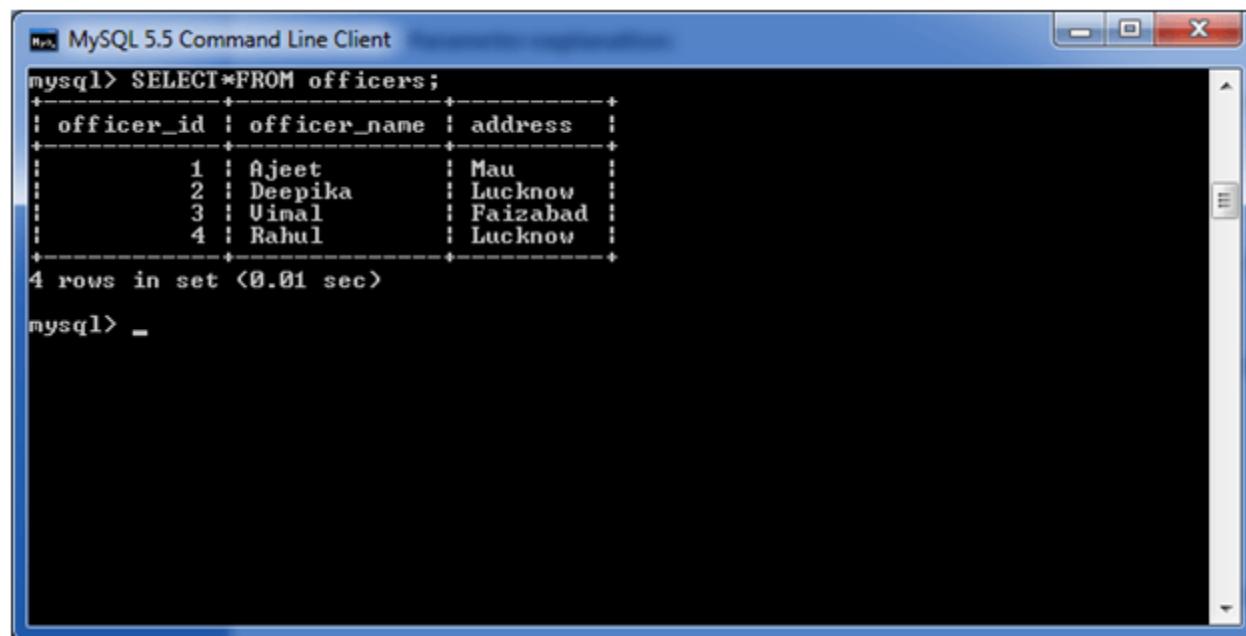
**expression1, expression2, ... expression\_n:** It specifies the expressions that are not encapsulated within an aggregate function and must be included in the GROUP BY clause.

**aggregate\_function:** It specifies a function such as SUM, COUNT, MIN, MAX, or AVG etc. tables: It specifies the tables, from where you want to retrieve the records. There must be at least one table listed in the FROM clause.

**WHERE conditions:** It is optional. It specifies the conditions that must be fulfilled for the records to be selected.

## (i) MySQL GROUP BY Clause with COUNT function

Consider a table named "officers" table, having the following records.



The screenshot shows the MySQL 5.5 Command Line Client window. The command entered is `SELECT * FROM officers;`. The output displays the following table:

officer_id	officer_name	address
1	Ajeet	Mau
2	Deepika	Lucknow
3	Uinal	Faizabad
4	Rahul	Lucknow

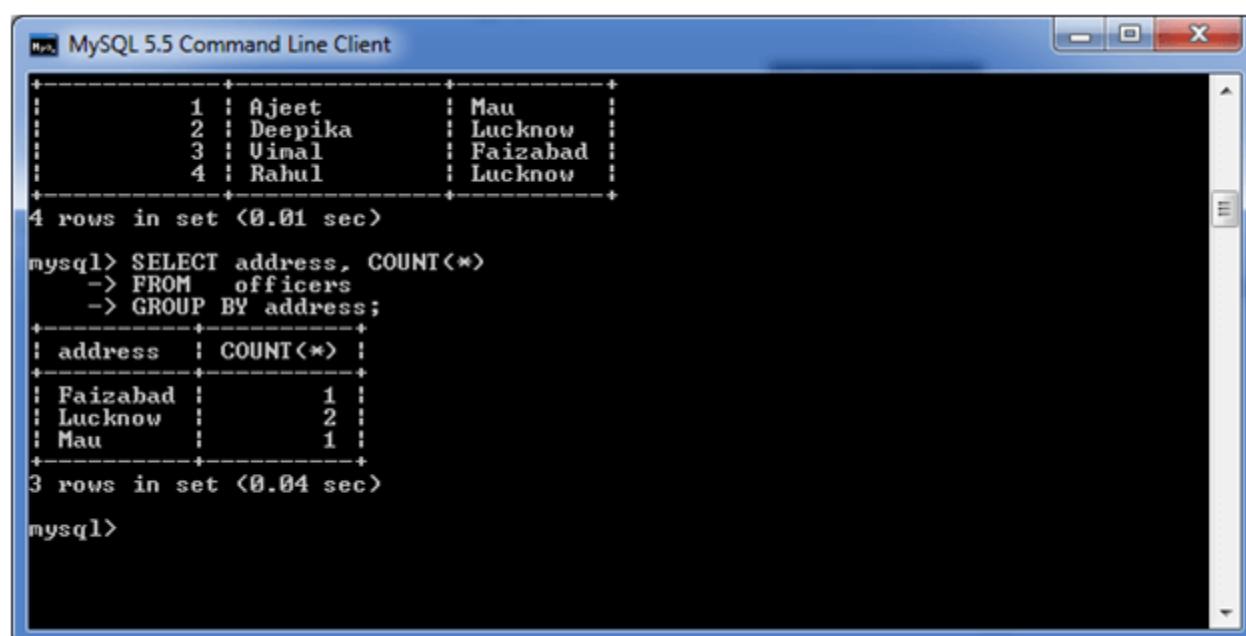
4 rows in set (0.01 sec)

Now, let's count repetitive number of cities in the column address.

**Execute the following query:**

1. **SELECT** address, **COUNT(\*)**
2. **FROM** officers
3. **GROUP BY** address;

**Output:**



The screenshot shows the MySQL 5.5 Command Line Client window. The commands executed are:

```
mysql> SELECT address, COUNT(*)  
-> FROM officers  
-> GROUP BY address;
```

The output shows the count of occurrences for each address:

address	COUNT(*)
Faizabad	1
Lucknow	2
Mau	1

3 rows in set (0.04 sec)

## (ii) MySQL GROUP BY Clause with SUM function

Let's take a table "employees" table, having the following data.

```

MySQL 5.5 Command Line Client
-> (3, 'Milan', '2015-01-25', 9),
-> (1, 'Ajeet', '2015-01-26', 12),
-> (3, 'Milan', '2015-01-26', 9);
Query OK, 10 rows affected (0.06 sec)
Records: 10  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM employees;
+-----+-----+-----+-----+
| emp_id | emp_name | working_date | working_hours |
+-----+-----+-----+-----+
| 1      | Ajeet    | 2015-01-24   | 12            |
| 2      | Ayan     | 2015-01-24   | 10            |
| 3      | Milan    | 2015-01-24   | 9             |
| 4      | Ruchi    | 2015-01-24   | 6             |
| 1      | Ajeet    | 2015-01-25   | 12            |
| 2      | Ayan     | 2015-01-25   | 10            |
| 4      | Ruchi    | 2015-01-25   | 6             |
| 3      | Milan    | 2015-01-25   | 9             |
| 1      | Ajeet    | 2015-01-26   | 12            |
| 3      | Milan    | 2015-01-26   | 9             |
+-----+-----+-----+-----+
10 rows in set (0.00 sec)

mysql>

```

Now, the following query will GROUP BY the example using the SUM function and return the emp\_name and total working hours of each employee.

**Execute the following query:**

1. **SELECT** emp\_name, **SUM**(working\_hours) **AS** "Total working hours"
2. **FROM** employees
3. **GROUP BY** emp\_name;

**Output:**

```

MySQL 5.5 Command Line Client
+-----+-----+-----+-----+
| 1      | Ajeet    | 2015-01-26   | 12            |
| 3      | Milan    | 2015-01-26   | 9             |
+-----+-----+-----+-----+
10 rows in set (0.00 sec)

mysql>
mysql> SELECT emp_name, SUM(working_hours) AS "Total working hours"
-> FROM employees
-> GROUP BY name;
ERROR 1054 (42S22): Unknown column 'name' in 'group statement'
mysql>
mysql> SELECT emp_name, SUM(working_hours) AS "Total working hours"
-> FROM employees
-> GROUP BY emp_name;
+-----+-----+
| emp_name | Total working hours |
+-----+-----+
| Ajeet    | 36                |
| Ayan     | 20                |
| Milan    | 27                |
| Ruchi    | 12                |
+-----+-----+
4 rows in set (0.00 sec)

mysql>

```

### (iii) MySQL GROUP BY Clause with MIN function

The following example specifies the minimum working hours of the employees form the table "employees".

**Execute the following query:**

1. **SELECT** emp\_name, **MIN**(working\_hours) **AS** "Minimum working hour"
2. **FROM** employees
3. **GROUP BY** emp\_name;

**Output:**

```

MySQL 5.5 Command Line Client
-> GROUP BY emp_name;
+-----+
| emp_name | Total working hours |
+-----+
| Ajeet    |            36 |
| Ayan     |             20 |
| Milan    |             27 |
| Ruchi   |             12 |
+-----+
4 rows in set <0.00 sec>

mysql> SELECT emp_name, MIN(working_hours) AS "Minimum working hour"
-> FROM employees
-> GROUP BY emp_name;
+-----+
| emp_name | Minimum working hour |
+-----+
| Ajeet    |             12 |
| Ayan     |              10 |
| Milan    |               9 |
| Ruchi   |               6 |
+-----+
4 rows in set <0.00 sec>

mysql> _

```

#### (iv) MySQL GROUP BY Clause with MAX function

The following example specifies the maximum working hours of the employees form the table "employees".

**Execute the following query:**

1. **SELECT** emp\_name, **MAX** (working\_hours) **AS** "Minimum working hour"
2. **FROM** employees
3. **GROUP BY** emp\_name;

**Output:**

```

MySQL 5.5 Command Line Client
-> GROUP BY emp_name;
+-----+
| emp_name | Minimum working hour |
+-----+
| Ajeet    |             12 |
| Ayan     |              10 |
| Milan    |               9 |
| Ruchi   |               6 |
+-----+
4 rows in set <0.00 sec>

mysql> SELECT emp_name, MAX(working_hours) AS "Minimum working hour"
-> FROM employees
-> GROUP BY emp_name;
+-----+
| emp_name | Minimum working hour |
+-----+
| Ajeet    |             12 |
| Ayan     |              10 |
| Milan    |               9 |
| Ruchi   |               6 |
+-----+
4 rows in set <0.00 sec>

mysql> _

```

#### (v) MySQL GROUP BY Clause with AVG function

The following example specifies the average working hours of the employees form the table "employees".

**Execute the following query:**

1. **SELECT** emp\_name, **AVG**(working\_hours) **AS** "Average working hour"
2. **FROM** employees
3. **GROUP BY** emp\_name;

**Output:**

```

mysql> SELECT emp_name, MIN(working_hours) AS "Minimum working hour"
    -> FROM employees
    -> GROUP BY emp_name;
+-----+-----+
| emp_name | Minimum working hour |
+-----+-----+
| Ajeet     |          12           |
| Ayan      |          10           |
| Milan     |           9           |
| Ruchi     |           6           |
+-----+-----+
4 rows in set <0.00 sec>

mysql> SELECT emp_name, AVG(working_hours) AS "Average working hour"
    -> FROM employees
    -> GROUP BY emp_name;
+-----+-----+
| emp_name | Average working hour |
+-----+-----+
| Ajeet     |      12.0000        |
| Ayan      |      10.0000        |
| Milan     |      9.0000         |
| Ruchi     |      6.0000          |
+-----+-----+
4 rows in set <0.00 sec>

mysql>

```

## MySQL HAVING Clause

MySQL HAVING Clause is used with GROUP BY clause. It always returns the rows where condition is TRUE.

### Syntax:

1. **SELECT** expression1, expression2, ... expression\_n,
2. aggregate\_function (expression)
3. **FROM** tables
4. [ **WHERE** conditions]
5. **GROUP BY** expression1, expression2, ... expression\_n
6. **HAVING** condition;

### Parameters

**aggregate\_function:** It specifies any one of the aggregate function such as SUM, COUNT, MIN, MAX, or AVG.

**expression1, expression2, ... expression\_n:** It specifies the expressions that are not encapsulated within an aggregate function and must be included in the GROUP BY clause.

**WHERE conditions:** It is optional. It specifies the conditions for the records to be selected.

**HAVING condition:** It is used to restrict the groups of returned rows. It shows only those groups in result set whose conditions are TRUE.

## HAVING Clause with SUM function

Consider a table "employees" table having the following data.

```

mysql> CREATE TABLE employees (
    -> emp_id INT,
    -> emp_name VARCHAR(20),
    -> working_date DATE,
    -> working_hours INT
    -> );
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO employees VALUES
    -> (1, 'Ajeet', '2015-01-24', 12),
    -> (2, 'Ayan', '2015-01-24', 10),
    -> (3, 'Milan', '2015-01-24', 9),
    -> (4, 'Ruchi', '2015-01-24', 6),
    -> (1, 'Ajeet', '2015-01-25', 12),
    -> (2, 'Ayan', '2015-01-25', 10),
    -> (4, 'Ruchi', '2015-01-25', 6),
    -> (3, 'Milan', '2015-01-25', 9),
    -> (1, 'Ajeet', '2015-01-26', 12),
    -> (3, 'Milan', '2015-01-26', 9);
Query OK, 10 rows affected (0.06 sec)
Records: 10  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM employees;
+-----+-----+-----+-----+
| emp_id | emp_name | working_date | working_hours |
+-----+-----+-----+-----+
|     1  | Ajeet    | 2015-01-24  |          12   |
|     2  | Ayan    | 2015-01-24  |          10   |
|     3  | Milan   | 2015-01-24  |           9   |
|     4  | Ruchi   | 2015-01-24  |           6   |
|     1  | Ajeet    | 2015-01-25  |          12   |
|     2  | Ayan    | 2015-01-25  |          10   |
|     4  | Ruchi   | 2015-01-25  |           6   |
|     3  | Milan   | 2015-01-25  |           9   |
|     1  | Ajeet    | 2015-01-26  |          12   |
|     3  | Milan   | 2015-01-26  |           9   |
+-----+-----+-----+-----+
10 rows in set (0.00 sec)

mysql>

```

Here, we use the SUM function with the HAVING Clause to return the emp\_name and sum of their working hours.

**Execute the following query:**

1. **SELECT** emp\_name, **SUM**(working\_hours) **AS** "Total working hours"
2. **FROM** employees
3. **GROUP BY** emp\_name
4. **HAVING** **SUM**(working\_hours) > 5;

```

MySQL 5.5 Command Line Client
+-----+
10 rows in set <0.00 sec>

mysql> SELECT emp_name, SUM(working_hours) AS "Total working hours"
-> FROM employees
-> GROUP BY emp_name
-> HAVING SUM(working_hours) > 5;
+-----+
| emp_name | Total working hours |
+-----+
| Ajeet     |            36          |
| Ayan      |            20          |
| Milan     |            27          |
| Ruchi     |            12          |
+-----+
4 rows in set <0.00 sec>

mysql> _

```

Simply, it can also be used with COUNT, MIN, MAX and AVG functions.

## MySQL Privileges

# MySQL Grant Privilege

MySQL has a feature that provides many control options to the administrators and users on the database. We have already learned how to create a new user using **CREATE USER** statement in MySQL server. Now, we are going to learn about grant privileges to a user account. MySQL provides GRANT statements to give access rights to a user account.

## GRANT Statement

The grant statement enables system administrators to **assign privileges and roles** to the **MySQL** user accounts so that they can use the assigned permission on the database whenever required.

### Syntax

The following are the basic syntax of using the GRANT statement:

1. **GRANT** privilege\_name(s)
2. **ON** object
3. **TO** user\_account\_name;

### Parameter Explanation

In the above syntax, we can have the following parameters:

Parameter Name	Descriptions
privilege_name(s)	It specifies the access rights or grant privilege to user accounts. If we want to give multiple privileges, then use a comma operator to separate them.
object	It determines the privilege level on which the access rights are being granted. It means granting privilege to the table; then the object should be the name of the table.
user_account_name	It determines the account name of the user to whom the access rights would be granted.

## Privilege Levels

MySQL supports the following privilege levels:

Privilege Level	Syntax	Descriptions
Global	GRANT ALL ON *.* TO john@localhost;	It applies to all databases on MySQL server. We need to use *.* syntax for applying global privileges. Here, the user can query data from all databases and tables of the current server.
Database	GRANT ALL ON mydb.* TO john@localhost;	It applies to all objects in the current database. We need to use the db_name.* syntax for applying this privilege. Here, a user can query data from all tables in the given database.
Table	GRANT DELETE ON mydb.employees TO john@localhost;	It applies on all columns in a specified table. We need to use db_name.table_name syntax for assigning this privilege. Here, a user can query data from the given table of the specified database.
Column	GRANT SELECT (col1), INSERT (col1, col2), UPDATE (col2) ON mydb.mytable TO john@localhost;	It applies on a single column of a table. Here, we must have to specify the column(s) name enclosed with parenthesis for each privilege. The user can select one column, insert values in two columns, and update only one column in the given table.
Stored Routine	GRANT EXECUTE ON PROCEDURE mydb.myprocedure TO john@localhost;	It applies to stored routines (procedure and functions). It contains CREATE ROUTINE, ALTER ROUTINE, EXECUTE, and GRANT OPTION privileges. Here, a user can execute the stored procedure in the current database.
Proxy	GRANT PROXY ON root TO peter@localhost;	It enables one user to be a proxy for other users.

## GRANT Statement Example

Let us understand the GRANT privileges through the example. First, we need to create a new user named "**john@localhost**" using the following statement:

1. mysql> **CREATE USER** john@localhost **IDENTIFIED BY** 'jtp12345';

Next, execute the SHOW GRANT statement to check the privileges assigned to john@localhost using the following query:

1. mysql> **SHOW GRANTS FOR** john@localhost;

It will give the below output. Here, the **USAGE** means a user can log in to the database but does not have any privileges.

The screenshot shows a terminal window titled "MySQL 8.0 Command Line Client". The user has run the following commands:

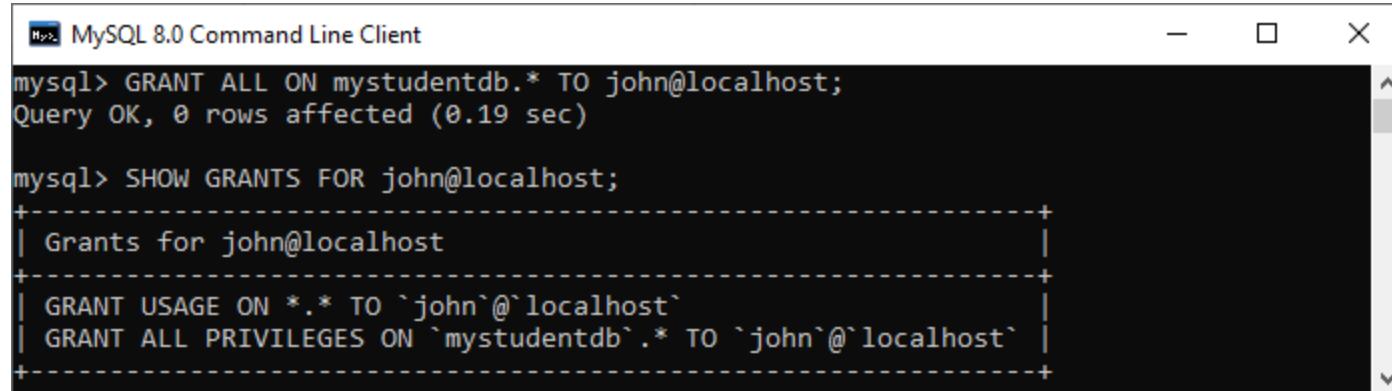
```
mysql> CREATE USER john@localhost IDENTIFIED BY 'jtp12345';
Query OK, 0 rows affected (0.14 sec)

mysql> SHOW GRANTS FOR john@localhost;
+-----+-----+
| Grants for john@localhost          |
+-----+-----+
| GRANT USAGE ON *.* TO `john`@`localhost` |
+-----+-----+
```

If we want to **assign all privileges** to all databases in the current server to john@localhost, execute the below statement:

```
1. mysql> GRANT ALL ON mystudentdb.* TO john@localhost;
```

Again, execute the SHOW GRANT statement to verify the privileges. After the successful execution, we will get the below output. Here all privileges are assigned to all databases in the current server to john@localhost.



The screenshot shows a terminal window titled "MySQL 8.0 Command Line Client". It displays two MySQL commands and their results:

```
mysql> GRANT ALL ON mystudentdb.* TO john@localhost;
Query OK, 0 rows affected (0.19 sec)

mysql> SHOW GRANTS FOR john@localhost;
+-----+-----+
| Grants for john@localhost          |
+-----+-----+
| GRANT USAGE ON *.* TO `john`@`localhost` |
| GRANT ALL PRIVILEGES ON `mystudentdb`.* TO `john`@`localhost` |
+-----+-----+
```

### Stored Routine Example

Here, the grant privileges are applied to **procedures and functions** where a user can execute the stored procedure in the current MySQL database. The EXECUTE privilege provides the ability to execute a function and procedure.

Let us understand it with the example. Suppose we have a function **calculatesalary** and want to grant **EXECUTE** privilege to a user john, run the following query:

```
1. mysql> GRANT EXECUTE ON FUNCTION calculatesalary TO john@localhost;
```

If there is a need to provide the EXECUTE privilege to all users, we must run the below command:

```
1. mysql> GRANT EXECUTE ON FUNCTION calculatesalary TO *@localhost;
```

We can choose access right from the below list on which privileges can be applied.

1. **SELECT:** It enables us to view the result set from a specified table.
2. **INSERT:** It enables us to add records in a given table.
3. **DELETE:** It enables us to remove rows from a table.
4. **CREATE:** It enables us to create tables/schemas.
5. **ALTER:** It enables us to modify tables/schemas.
6. **UPDATE:** It enables us to modify a table.
7. **DROP:** It enables us to drop a table.
8. **INDEX:** It enables us to create indexes on a table.
9. **ALL:** It enables us to give ALL permissions except GRANT privilege.
10. **GRANT:** It enables us to change or add access rights.

## MySQL Revoke Privilege

We have already learned how to give access right from grant privileges to a user account. Now, we are going to learn about revoke privileges from a user account. MySQL provides REVOKE statements to remove privileges from a user account.

### REVOKE Statement

The revoke statement enables system administrators to **revoke privileges and roles** to the MySQL user accounts so that they cannot use the assigned permission on the database in the past.

#### Syntax

The following are the basic syntax of using the REVOKE statement:

1. **REVOKE** privilege\_name(s)
2. **ON** object
3. **FROM** user\_account\_name;

#### Parameter Explanation

In the above syntax, we can have the following parameters:

Parameter Name	Descriptions
privilege_name(s)	It specifies the access rights or grant privilege that we want to revoke from user accounts.
object	It determines the privilege level on which the access rights are being granted. It means granting privilege to the table; then the object should be the name of the table.
user_account_name	It determines the account name of the user from which we want to revoke the access rights.

## Privilege Levels

MySQL supports the following privilege levels:

Privilege Level	Syntax	Descriptions
Global	REVOKE ALL, GRANT OPTION FROM john@localhost;	It applies to remove all access rights from the user on MySQL server.
Database	REVOKE ALL ON mydb.* FROM john@localhost;	It applies to revoke all privileges from objects in the current database.
Table	REVOKE DELETE ON mydb.employees FROM john@localhost;	It applies to revoke privileges from all columns in a specified table.
Column	REVOKE SELECT (col1), INSERT (col1, col2), UPDATE (col2) ON mydb.mytable FROM john@localhost;	It applies to revoke privileges from a single column of a table.
Stored Routine	REVOKE EXECUTE ON PROCEDURE/FUNCTION mydb.myprocedure FROM john@localhost;	It applies to revoke all privileges from stored routines (procedure and functions).
Proxy	REVOKE PROXY ON root FROM peter@localhost;	It enables us to revoke the proxy user.

## REVOKE Statement Example

Let us understand the REVOKE privileges through the example. First, we need to create a new user named "**john@localhost**" using the following statement:

1. mysql> **CREATE USER** john@localhost IDENTIFIED BY 'jtp12345';

Next, assign all privileges to all databases in the current server to john@localhost, using the below statement:

1. mysql> **GRANT ALL ON** mystudentdb.\* **TO** john@localhost;

Next, execute the SHOW GRANT statement to verify the privileges. In the output, we can see that all privileges are assigned to all databases in the current server to john@localhost.

```

MySQL 8.0 Command Line Client
mysql> GRANT ALL ON mystudentdb.* TO john@localhost;
Query OK, 0 rows affected (0.19 sec)

mysql> SHOW GRANTS FOR john@localhost;
+-----+-----+
| Grants for john@localhost |
+-----+-----+
| GRANT USAGE ON *.* TO `john`@`localhost` |
| GRANT ALL PRIVILEGES ON `mystudentdb`.* TO `john`@`localhost` |
+-----+-----+

```

If we want to revoke all privileges assign to the user, execute the following statement:

1. mysql> **REVOKE ALL, GRANT OPTION FROM** john@localhost;

We will get the output below where we can see that a user can log in to the database without any privileges.

```
MySQL 8.0 Command Line Client
mysql> REVOKE ALL, GRANT OPTION FROM john@localhost;
Query OK, 0 rows affected (0.25 sec)

mysql> SHOW GRANTS FOR john@localhost;
+-----+
| Grants for john@localhost |
+-----+
| GRANT USAGE ON *.* TO `john`@`localhost` |
+-----+
```

### REVOKE selected privilege from a user account

Suppose we have provided grant privilege of SELECT, INSERT, and UPDATE command on mystudentdb to the user with the following statement:

1. mysql> **GRANT SELECT, UPDATE, INSERT ON** mystudentdb.\* **TO** john@localhost;

Next, display the GRANT privilege with the following statement:

1. mysql> SHOW GRANTS **FOR** john@localhost;

Finally, execute the REVOKE statement to remove UPDATE and INSERT privilege with the below statement:

1. mysql> **REVOKE UPDATE, INSERT ON** mystudentdb.\* **FROM** john@localhost;

It will give the below output where only SELECT privilege is left.

```
MySQL 8.0 Command Line Client
mysql> GRANT SELECT, UPDATE, INSERT ON mystudentdb.* TO john@localhost;
Query OK, 0 rows affected (0.20 sec)

mysql> SHOW GRANTS FOR john@localhost;
+-----+
| Grants for john@localhost |
+-----+
| GRANT USAGE ON *.* TO `john`@`localhost` |
| GRANT SELECT, INSERT, UPDATE ON `mystudentdb`.* TO `john`@`localhost` |
+-----+
2 rows in set (0.00 sec)

mysql> REVOKE UPDATE, INSERT ON mystudentdb.* FROM john@localhost;
Query OK, 0 rows affected (0.20 sec)

mysql> SHOW GRANTS FOR john@localhost;
+-----+
| Grants for john@localhost |
+-----+
| GRANT USAGE ON *.* TO `john`@`localhost` |
| GRANT SELECT ON `mystudentdb`.* TO `john`@`localhost` |
+-----+
```

### REVOKE Proxy User Example

First, we need to grant the proxy privilege to the user whom you want using the following statement:

1. mysql> **GRANT PROXY ON** 'peter@javatpoint' **TO** 'john'@'localhost' **WITH GRANT OPTION**;

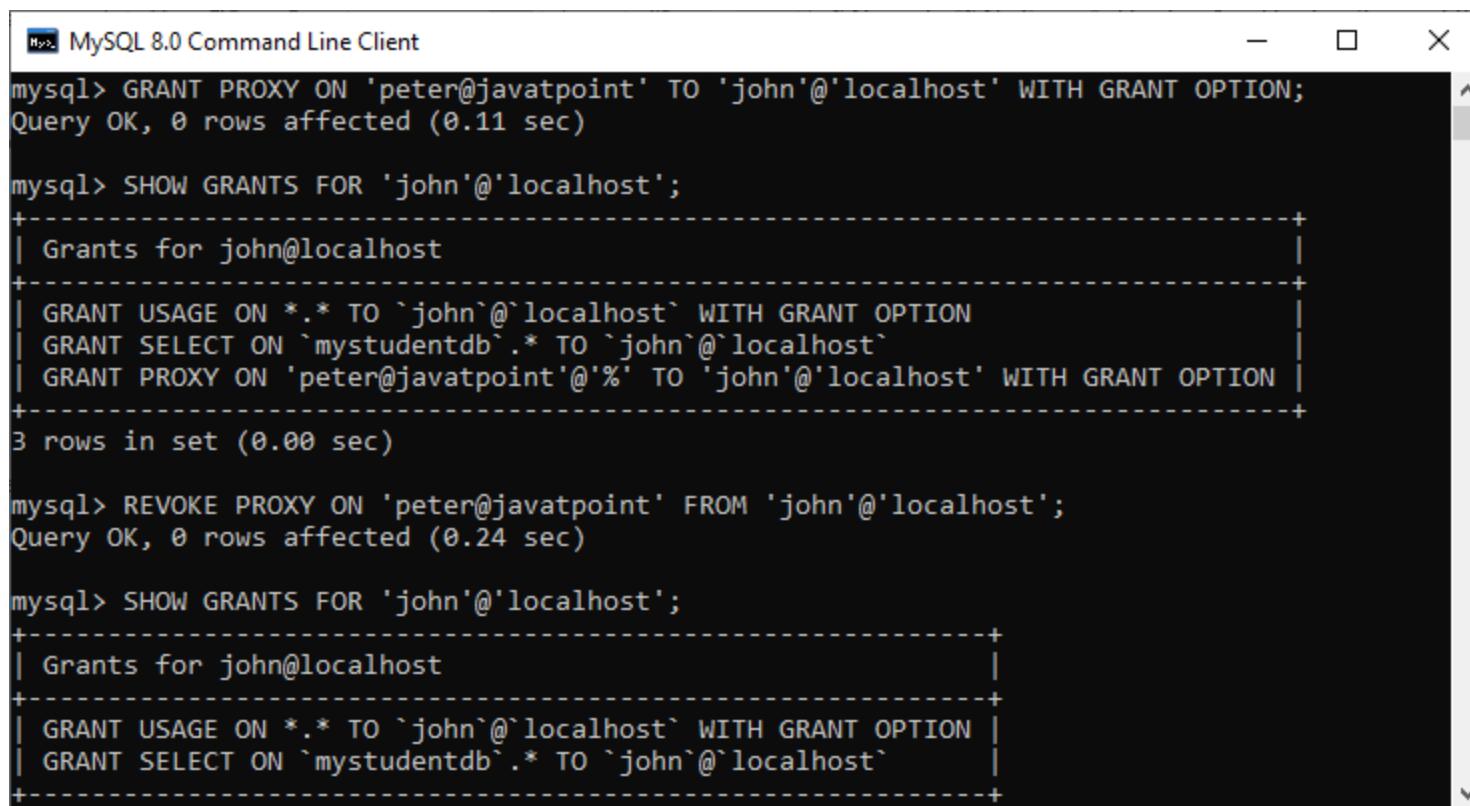
Next, display the GRANT privilege with the given statement:

1. mysql> SHOW GRANTS **FOR** 'john'@'localhost';

Finally, execute the REVOKE statement to remove proxy privilege from the user with the below statement:

1. mysql> **REVOKE PROXY ON** 'peter@javatpoint' **FROM** 'john'@'localhost';

It will give the below output where proxy privilege is revoked successfully.



```
MySQL 8.0 Command Line Client
mysql> GRANT PROXY ON 'peter@javatpoint' TO 'john'@'localhost' WITH GRANT OPTION;
Query OK, 0 rows affected (0.11 sec)

mysql> SHOW GRANTS FOR 'john'@'localhost';
+-----+
| Grants for john@localhost |
+-----+
| GRANT USAGE ON *.* TO `john`@`localhost` WITH GRANT OPTION |
| GRANT SELECT ON `mystudentdb`.* TO `john`@`localhost` |
| GRANT PROXY ON 'peter@javatpoint'@'%' TO 'john'@'localhost' WITH GRANT OPTION |
+-----+
3 rows in set (0.00 sec)

mysql> REVOKE PROXY ON 'peter@javatpoint' FROM 'john'@'localhost';
Query OK, 0 rows affected (0.24 sec)

mysql> SHOW GRANTS FOR 'john'@'localhost';
+-----+
| Grants for john@localhost |
+-----+
| GRANT USAGE ON *.* TO `john`@`localhost` WITH GRANT OPTION |
| GRANT SELECT ON `mystudentdb`.* TO `john`@`localhost` |
+-----+
```

### Revoking Privileges from Stored Routine Example

Here, the revoke privileges are applied to **procedures and functions** where we can revoke the privileges from the user who has a execute privilege in the past.

Let us understand it with the example. Suppose we have a function **calculatesalary** and want to grant **EXECUTE** privilege to a user john, run the following query:

1. mysql> **GRANT EXECUTE ON FUNCTION** calculatesalary **TO** john@localhost;

If there is a need to revoke the EXECUTE privilege to the users, we must run the below command:

1. mysql> **REVOKE EXECUTE ON FUNCTION** calculatesalary **TO** john@localhost;

We can revoke privileges from the below list on which privileges can be applied.

1. **CREATE**: It enables the user account to create databases and tables.
2. **DROP**: It allows the user account to drop databases and tables.
3. **DELETE**: It enables the user account to delete rows from a specific table.
4. **INSERT**: It allows the user account to insert rows into a specific table.
5. **SELECT**: It enables the user account to read a database.
6. **UPDATE**: It enables the user account to update table rows.

## Control Flow Function

# MySQL IF()

In this section, we are going to learn how **IF() function** works in MySQL. The IF function is one of the parts of the MySQL control flow function, which returns a value based on the given conditions. In other words, the IF function is used for validating a [function in MySQL](#). The IF function returns a value **YES** when the given condition evaluates to true and returns a **NO** value when the condition evaluates to false. It returns values either in a string or numeric form depending upon the context in which this function is used. Sometimes, this function is known as **IF-ELSE** and **IF THAN ELSE** function.

The IF function takes three expressions, where the first expression will be evaluated. If the first expression evaluates to true, not null, and not zero, it returns the second expression. If the result is false, it returns the third expression.

### Syntax

1. IF ( expression 1, expression 2, expression 3)

### Parameter

Parameter	Requirement	Descriptions
Expression 1	Required	It is a value, which is used for validation.
Expression 2	Optional	It returns a value when the condition evaluates to true.
Expression 3	Optional	It returns a value when the condition evaluates to false.

## Default Return Type

The return type of IF function can be calculated as follows:

- o If expression 2 or expression 3 are both strings or produce a string, the result is always a string.
- o If expression 2 or expression 3 gives a floating-point value, the result is always a floating-point value.
- o If expression 2 or expression 3 is an integer, the result is always an integer.

## MySQL version support

The IF function can support the following [MySQL versions](#):

- o MySQL 8.0
- o MySQL 5.7
- o MySQL 5.6
- o MySQL 5.5
- o MySQL 5.1
- o MySQL 5.0
- o MySQL 4.1
- o MySQL 4.0
- o MySQL 3.23.3

**Note:** The IF function is different from the IF statement. So do not confuse in IF function and IF statement.

Let us understand the MySQL IF function with the following examples. We can use the IF function with the SELECT statement directly.

### Example 1

1. **SELECT** IF(200>350,'YES','NO');

In the above function, the (200>350) is a condition, which is evaluated. If the condition is true, it returns a value, **YES**, and if the condition is false, it returns **NO**.

#### Output:

NO

### Example 2

1. **SELECT** IF(251 = 251,' Correct','Wrong');

In the above function, the (251 = 251) is a condition, which is evaluated. If the condition is true, it returns value **Correct**, and if the condition is false, it returns **Wrong** output.

#### Output:

Correct

### Example 3

1. **SELECT** IF(STRCMP('Rinky Ponting','Yuvraj Singh')=0, 'Correct', 'Wrong');

The above example compares the two strings. If both the string is the same, it returns **Correct**. Otherwise, the IF function returns **Wrong** output.

## Output:

Wrong

## Example 4

Here, we are going to create a table '**student**' and perform the IF function.

studentid	firstname	lastname	class	age
1	Rinky	Ponting	12	20
2	Mark	Boucher	11	22
3	Sachin	Tendulkar	10	18
4	Peter	Fleming	10	22
5	Virat	Kohli	12	23
NONE	NONE	NONE	NONE	NONE

Now, run the following [MySQL query](#). This statement returns the **last name** of the student table, in which, if the **age** is greater than 20, it returns **Mature**. Otherwise, the IF function returns **Immature**.

1. **SELECT** lastname,
2. IF(age>20,"Mature","Immature")
3. **As** Result
4. **FROM** student;

## Output:

When the above [MySQL](#) statement runs successfully, it will give the following output.

lastname	Result
Ponting	Immature
Boucher	Mature
Tendulkar	Immature
Fleming	Mature
Kohli	Mature

## MySQL IFNULL()

This section helps you to learn about the MySQL IFNULL() function. The IFNULL function is a part of the MySQL control flow function used for handling NULL values.

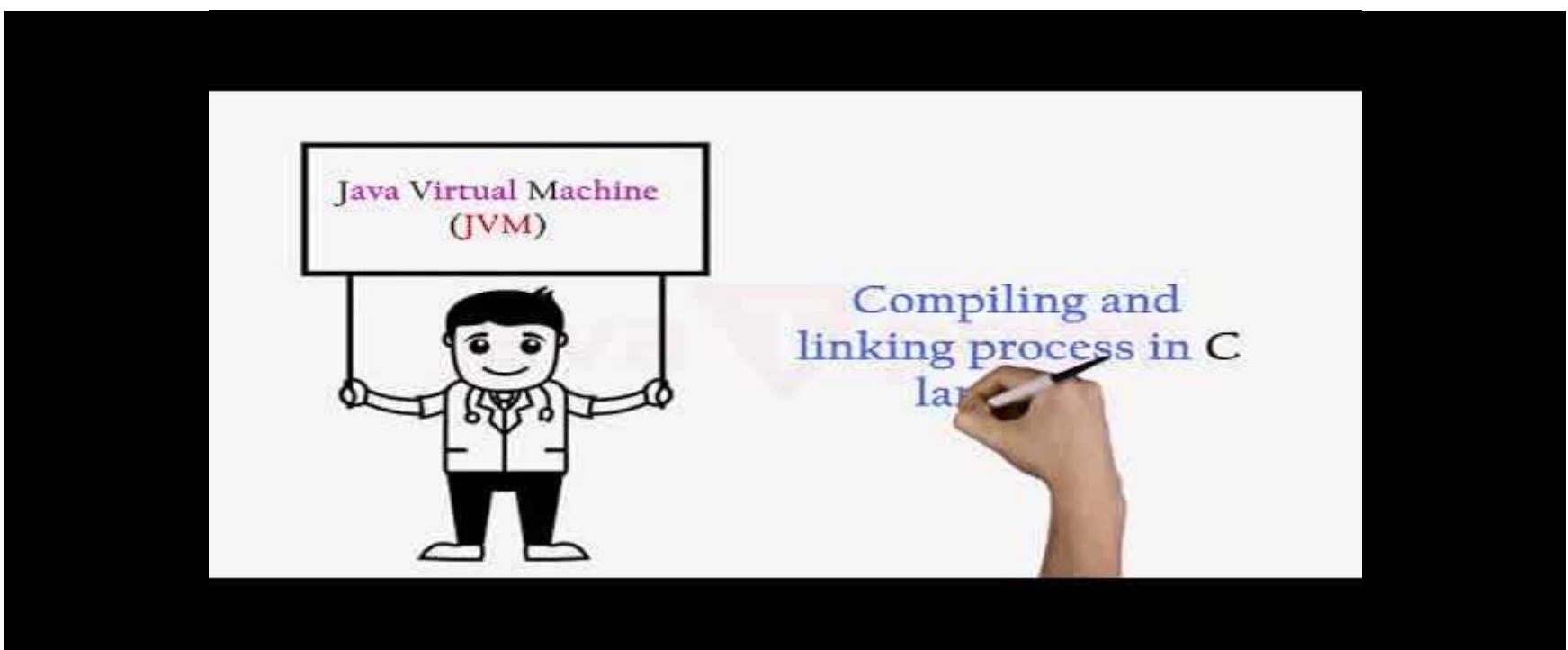
The IFNULL function accepts two expressions, and if the first expression is **not null**, it returns the first arguments. If the first expression is **null**, it returns the second argument. This function returns either string or numeric value, depending on the context where it is used.

## Syntax

We can use the IFNULL function with the following syntax:

1. IFNULL (Expression1, Expression2)

It returns expression1 when the expression1 is not null. Otherwise, it will return expression2.



## Parameters

Parameter	Requirement	Descriptions
Expression 1	Required	This expression is used to check whether it is NULL or not.
Expression 2	Required	It will return when the expression 1 is NULL.

## MySQL version support

The IFNULL function can support the following MySQL versions:

- o MySQL 8.0
- o MySQL 5.7
- o MySQL 5.6
- o MySQL 5.5
- o MySQL 5.1
- o MySQL 5.0
- o MySQL 4.1
- o MySQL 4.0

Let us understand the MySQL IFNULL() function with the following examples. We can use the IFNULL function with the **SELECT** statement directly.

### Example 1

1. **SELECT** IFNULL(0,5);

In the above function, the MySQL statement checks the first expression. If the first expression is not NULL, it will return the first expression, which is zero.

#### Output:

```
0
```

### Example 2

1. **SELECT** IFNULL("Hello", "javaTpoint");

The above MySQL statement checks the first expression. If the first expression is not NULL, it will return the first expression, which is 'Hello' value.

#### Output:

```
Hello
```

### Example 3

1. **SELECT** IFNULL(NULL,5);

The following MySQL statement checks the first expression. If the first expression is not NULL, it will return the first expression. Otherwise, it will return the second expression, which is five (5).

#### Output:

```
5
```

### Example 4

Here, we are going to create a table '**student\_contacts**' and perform the IFNULL() function.

```

1. CREATE TABLE `student_contacts` (
2.   `studentid` int unsigned NOT NULL AUTO_INCREMENT,
3.   `contactname` varchar(45) NOT NULL,
4.   `cellphone` varchar(20) DEFAULT NULL,
5.   `homephone` varchar(20) DEFAULT NULL,
6. );

```

Now, you need to insert data into a table. After inserting the values into the table, execute the following query.

```

1. SELECT
2.   contactname, cellphone, homephone
3. FROM
4.   student_contacts;

```

It will display the output that contains all rows and columns. Here, we can see that some of the contacts have only a cell phone or home phone number.

studentid	contactname	cellphone	homephone
2	Will Smith	3214356574	<b>NULL</b>
3	Johnsena	<b>NULL</b>	4563480897
4	Peter Joe	<b>NULL</b>	2123157870
5	kelly Bruke	5683128765	<b>NULL</b>
6	Freeda Pinto	4563482354	<b>NULL</b>
<b>NULL</b>	<b>NULL</b>	<b>NULL</b>	<b>NULL</b>

In the above output, we will get all contacts name whether cell phone, and home phone number is available or not. So, in that case, the IFNULL() function plays an important role.

Now, run the following MySQL query. This statement returns the **home phone** number if the cell phone is **NULL**.

```

1. SELECT
2.   contactname, IFNULL(cellphone, homephone) phone
3. FROM
4.   student_contact;

```

#### Output:

When the above MySQL statement runs successfully, it will give the following output.

contactname	phone
Will Smith	3214356574
Johnsena	4563480897
Peter Joe	2123157870
kelly Bruke	5683128765
Freeda Pinto	4563482354

**Note:** You should avoid the use of the IFNULL() function in the WHERE clause because this function reduces the performance of the query.

## MySQL NULLIF()

This section helps you to learn about the MySQL NULLIF() function. The NULLIF function is a part of the MySQL control flow function that used for **comparison** in two expressions. It also helps in preventing the division by zero error in a SQL statement.

The NULLIF function accepts two expressions, and if the first expression is equal to the second expression, it returns the **NULL**. Otherwise, it returns the first expression.

### Syntax

We can use the NULLIF function with the following syntax:

```
1. NULLIF (Expression1, Expression2)
```

It returns Null when expression1 is equal to expression2. Otherwise, it will return expression1.

## Parameter

Parameter	Requirement	Descriptions
Expression 1	Required	It specify the first expression for comparison.
Expression 2	Required	It specify the second expression for comparison.

## MySQL version support

The NULLIF function can support the following [MySQL versions](#):

- o MySQL 8.0
- o MySQL 5.7
- o MySQL 5.6
- o MySQL 5.5
- o MySQL 5.1
- o MySQL 5.0
- o MySQL 4.1
- o MySQL 4.0

Let us understand the [MySQL](#) NULLIF() function with the following examples. We can use the NULLIF function with the **SELECT** statement directly.

### Example 1

1. **SELECT** NULLIF("javaTpoint", "javaTpoint");

In the above function, the MySQL statement checks the first expression is equal to the second expression or not. If both expressions are the same, it returns NULL. Otherwise, it will return the first expression.

#### Output:

```
NULL
```

### Example 2

1. **SELECT** NULLIF("Hello", "404");

The following MySQL statement compares both expressions. If expression1 = expression2, it returns NULL. Otherwise, it will return expression1.

#### Output:

```
Hello
```

### Example 3

1. **SELECT** NULLIF(9,5);

The following MySQL statement compares both integer values. If they are equal, return NULL. Otherwise, it returns the first expression.

#### Output:

```
9
```

### Example 4

In this example, we are going to understand how NULLIF() function prevents **division by zero** error. If we run the query "SELECT 1/0", then we get an error output. So, in that case, we will use NULLIF function as below syntax.

1. **SELECT** 1/NULLIF(0,0);

#### Output:

NULL

## Example 5

Let us create a **customer** table for performing the NULLIF function. The following statement creates a customer table in your database.

```
1. CREATE TABLE 'customer' (
2.   'customer_id' INT UNSIGNED NOT NULL AUTO_INCREMENT,
3.   'cust_name' VARCHAR(45) NOT NULL,
4.   'occupation' VARCHAR(45) NOT NULL,
5.   'income' VARCHAR(15) NOT NULL,
6.   'qualification' VARCHAR(45) NOT NULL
7. );
```

Now, you need to insert data into a table. To insert values into the table, run the following command.

1. **INSERT INTO** 'myproductdb'.**'customer'** ('**cust\_name**', '**occupation**', '**income**', '**qualification**') **VALUES** ('John Miller', 'Developer', '20000', 'Btech');
2. **INSERT INTO** 'myproductdb'.**'customer'** ('**cust\_name**', '**occupation**', '**income**', '**qualification**') **VALUES** ('Mark Robert', 'Enginneer', '40000', 'Btech');
3. **INSERT INTO** 'myproductdb'.**'customer'** ('**cust\_name**', '**occupation**', '**income**', '**qualification**') **VALUES** ('Reyan Watson', 'Scientists', '60000', 'MSc');
4. **INSERT INTO** 'myproductdb'.**'customer'** ('**cust\_name**', '**occupation**', '**income**', '**qualification**') **VALUES** ('Shane Trump', 'Businessman', '10000', 'MBA');
5. **INSERT INTO** 'myproductdb'.**'customer'** ('**cust\_name**', '**occupation**', '**income**', '**qualification**') **VALUES** ('Adam Obama', 'Manager', '80000', 'MBA');
6. **INSERT INTO** 'myproductdb'.**'customer'** ('**cust\_name**', '**occupation**', '**income**', '**qualification**') **VALUES** ('Rincky Ponting', 'Cricketer', '200000', 'Btech');

After inserting the values into the table, execute the following query.

```
1. SELECT * FROM customer;
```

It will give the following table:

customer_id	cust_name	occupation	income	qualification
1	John Miller	Developer	20000	Btech
2	Mark Robert	Enginneer	40000	Btech
3	Reyan Watson	Scientists	60000	MSc
4	Shane Trump	Businessman	10000	MBA
5	Adam Obama	Manager	80000	MBA
6	Rincky Ponting	Cricketer	200000	Btech
NULL	NULL	NULL	NULL	NULL

Now, we are going to use the NULLIF function to check the **qualification** column value against the **Btech**. It means if the customer occupation is Btech, it returns NULL. Otherwise, it returns the column value.

1. **SELECT** **cust\_name**, **occupation**, **qualification**,
2. **NULLIF** (**qualification**, "Btech") **result**
3. **FROM** myproductdb.**customer**;

### Output:

When the above command executes successfully, it returns the following output.

cust_name	occupation	qualification	result
John Miller	Developer	Btech	NULL
Mark Robert	Enginneer	Btech	NULL
Reyan Watson	Scientists	MSc	MSc
Shane Trump	Businessman	MBA	MBA
Adam Obama	Manager	MBA	MBA
Rincky Ponting	Cricketer	Btech	NULL

## MySQL CASE Expression

MySQL CASE expression is a part of the control flow function that provides us to write an **if-else or if-then-else** logic to a query. This expression can be used anywhere that uses a valid program or query, such as SELECT, WHERE, ORDER BY clause, etc.

The CASE expression validates various conditions and returns the result when the first condition is **true**. Once the condition is met, it stops traversing and gives the output. If it will not find any condition true, it executes the **else block**. When the else block is not found, it returns a **NULL** value. The main goal of MySQL CASE statement is to deal with multiple IF statements in the SELECT clause.

We can use the CASE statement in two ways, which are as follows:

## 1. Simple CASE statement:

The first method is to take a value and matches it with the given statement, as shown below.

### Syntax

1. CASE value
2. WHEN [compare\_value] THEN result
3. [WHEN [compare\_value] THEN result ...]
4. [ELSE result]
5. END

It returns the result when the first **compare\_value** comparison becomes true. Otherwise, it will return the else clause.

### Example

1. mysql> **SELECT CASE 1 WHEN 1 THEN 'one' WHEN 2 THEN 'two' ELSE 'more' END;**

### Output

After the successful execution of the above command, we will see the following output.

```
mysql> SELECT CASE 1 WHEN 1 THEN 'one' WHEN 2 THEN 'two' ELSE 'more' END;
+-----+
| CASE 1 WHEN 1 THEN 'one' WHEN 2 THEN 'two' ELSE 'more' END |
+-----+
| one
+-----+
1 row in set (0.00 sec)
```

## 2. Searched CASE statement:

The second method is to consider a **search\_condition** in the **WHEN** clauses, and if it finds, return the result in the corresponding THEN clause. Otherwise, it will return the else clause. If else clause is not specified, it will return a NULL value.

### Syntax

1. CASE
2. WHEN [condition] THEN result
3. [WHEN [condition] THEN result ...]
4. [ELSE result]
5. END

### Example

1. mysql> **SELECT CASE BINARY 'B' WHEN 'a' THEN 1 WHEN 'b' THEN 2 END;**

### Output

```
mysql> SELECT CASE BINARY 'B' WHEN 'a' THEN 1 WHEN 'b' THEN 2 END;
+-----+
| CASE BINARY 'B' WHEN 'a' THEN 1 WHEN 'b' THEN 2 END |
+-----+
| NULL
+-----+
1 row in set (0.02 sec)
```

### Return Type

The CASE expression returns the result depending on the context where it is used. For example:

- If it is used in the string context, it returns the string result.
- If it is used in a numeric context, it returns the integer, float, decimal value.

## MySQL version support

The CASE statement can support the following [MySQL versions](#):

- MySQL 8.0
- MySQL 5.7
- MySQL 5.6
- MySQL 5.5
- MySQL 5.1
- MySQL 5.0
- MySQL 4.1
- MySQL 4.0
- MySQL 3.23.3

Let us create a table '**students**' and perform the CASE statement on this table.

	studentid	firstname	lastname	class	age
▶	1	Ricky	Ponting	CS	20
	2	Mark	Boucher	EE	22
	3	Michael	Clark	CS	18
	4	Peter	Fleming	CS	22
*	5	Virat	Kohli	EC	23
		NULl	NULl	NULl	NULl

In the above table, we can see that the **class column** contains the short form of the student's department. That's why we are going to change the short form of the department with the full form. Now, execute the following query to do this operation.

1. **SELECT** studentid, firstname,
2. **CASE** class
3. **WHEN 'CS' THEN 'Computer Science'**
4. **WHEN 'EC' THEN 'Electronics and Communication'**
5. **ELSE 'Electrical Engineering'**
6. **END AS** department **from** students;

After the successful execution of the above query, we will get the following output. Here, we can see that the **department** column contains full form instead of a short form.

studentid	firstname	department
1	Ricky	Computer Science
2	Mark	Electrical Engineering
3	Michael	Computer Science
4	Peter	Computer Science
5	Virat	Electronics and Communication

## MySQL IF Statement

The IF statement is used in stored programs that implement the basic conditional construct in MySQL. Based on a certain condition, it allows us to execute a set of SQL statements. It returns one of the three values True, False, or NULL.

We can use this statement in three ways IF-THEN, IF-THEN-ELSE, IF-THEN-ELSEIF-ELSE clauses, and can terminate with END-IF. Let us see each of these statements in detail.

### IF-THEN Statement

This statement executes a set of SQL queries based on certain conditions or expressions. The syntax of the IF-THEN statement is as follows:

1. IF condition **THEN**

2. statements;
3. **END IF;**

In the above syntax, we have to specify a condition for executing the code. If the statement evaluates to true, it will execute the statement between IF-THEN and END-IF. Otherwise, it will execute the statement following the END-IF.

## Example

The IF...ENDIF block executes with stored programs and terminates with a semicolon, as shown in the below example.

1. DELIMITER \$\$
2. **CREATE PROCEDURE** myResult(original\_rate **NUMERIC**(6,2),**OUT** discount\_rate **NUMERIC**(6,2))
3. **NO SQL**
4. **BEGIN**
5. IF (original\_rate>200) **THEN**
6.     **SET** discount\_rate=original\_rate\*.5;
7.     **END IF;**
8.     **select** discount\_rate;
9. **END\$\$**
10. DELIMITER \$\$;

Next, take two variables and set the value for both as below:

1. mysql> **set** @p = 600;
2. mysql> **set** @dp = 500;

Now, call the stored procedure function to check the output.

1. mysql> call myResult(@p, @dp)

We will get the following output:

	discount_rate
▶	300.00

## IF-THEN-ELSE Statement

If we want to execute other statements when the condition specifies in the IF block does not evaluate to true, this statement can be used. The syntax of an IF-THEN-ELSE statement is given below:

1. IF condition **THEN**
2. statements;
3. **ELSE**
4. **else**-statements;
5. **END IF;**

In the above syntax, we have to specify a condition for executing the code. If the statement evaluates to true, it will execute the statement between IF-THEN and ELSE. Otherwise, it will execute the statement following the ELSE and END-IF.

Let us modify the above **myResult()** stored procedure. So, first, remove myResult() stored procedure by using the command below:

1. Mysql> **DROP procedure** myResult;

Next, write the new code for this, as shown below:

1. DELIMITER \$\$
2. **CREATE PROCEDURE** myResult(original\_rate **NUMERIC**(6,2),**OUT** discount\_rate **NUMERIC**(6,2))
3. **NO SQL**
4. **BEGIN**

```

5.     IF (original_rate>200) THEN
6.         SET discount_rate=original_rate*.5;
7.     ELSE
8.         SET discount_rate=original_rate;
9.     END IF;
10.    select discount_rate;
11.    END$$
12. DELIMITER ;

```

Next, create **two variables** and set the value for both as below:

1. mysql> **set** @p = 150;
2. mysql> **set** @dp = 180;

Now, call the stored procedure function to get the output.

1. mysql> **call** myResult(@p, @dp)

It will give the following output:

	<b>discount_rate</b>
▶	150.00

## IF-THEN-ELSEIF-ELSE Statement

If we want to execute a statement based on multiple conditions, this statement can be used. The syntax of the IF-THEN-ELSE statement is given below:

1. IF condition **THEN**
2. statements;
3. ELSEIF elseif-condition **THEN**
4. elseif-statements;
5. ...
6. **ELSE**
7. **else**-statements;
8. **END IF;**

In the above syntax, if the condition becomes true, it will execute the IF-THEN branch. Otherwise, it will evaluate elseif-condition. When the elseif-condition becomes true, it will execute the elseif-statement. If this condition is also false, it will evaluate the next elseif-condition. Thus, here we will evaluate multiple elseif-condition, and if any condition in the IF and ELSE-IF does not becomes true, it will execute the statement of the ELSE branch.

Let us modify the above myResult() stored procedure. So, first, remove myResult() stored procedure by using the command below:

1. Mysql> **DROP procedure** myResult;

Next, write the new code for this, as shown below:

1. DELIMITER \$\$
2. **CREATE PROCEDURE** myResult(original\_rate **NUMERIC(6,2)**,**OUT** discount\_rate **NUMERIC(6,2)**)
3. **NO SQL**
4. **BEGIN**
5. IF (original\_rate>500) **THEN**
6. **SET** discount\_rate=original\_rate\*.5;
7. ELSEIF (original\_rate<=500 AND original\_rate>250) **THEN**
8. **SET** discount\_rate=original\_rate\*.8;
9. **ELSE**
10. **SET** discount\_rate=original\_rate;
11. **END IF;**
12. **select** discount\_rate;
13. **END\$\$**
14. DELIMITER ;

Next, create two variables and set the value for both as below:

1. mysql> **set** @p = 150;
2. mysql> **set** @dp = 150;

Now, call the stored procedure function to get the output.

1. mysql> call myResult(@p, @dp)

It will give the following output:

	discount_rate
▶	150.00

## MySQL Conditions

# MySQL AND Condition

The MySQL AND condition is used with SELECT, INSERT, UPDATE or DELETE statements to test two or more conditions in an individual query.

### Syntax:

1. **WHERE** condition1
2. AND condition2
3. ...
4. AND condition\_n;

### Parameter explanation:

**condition1, condition2, ... condition\_n:** Specifies all conditions that must be fulfilled for the records to be selected.

## MySQL AND Example

The following example specifies how to use the AND condition in MySQL with SELECT statement.

Consider a table "cus\_tbl", having the following data:

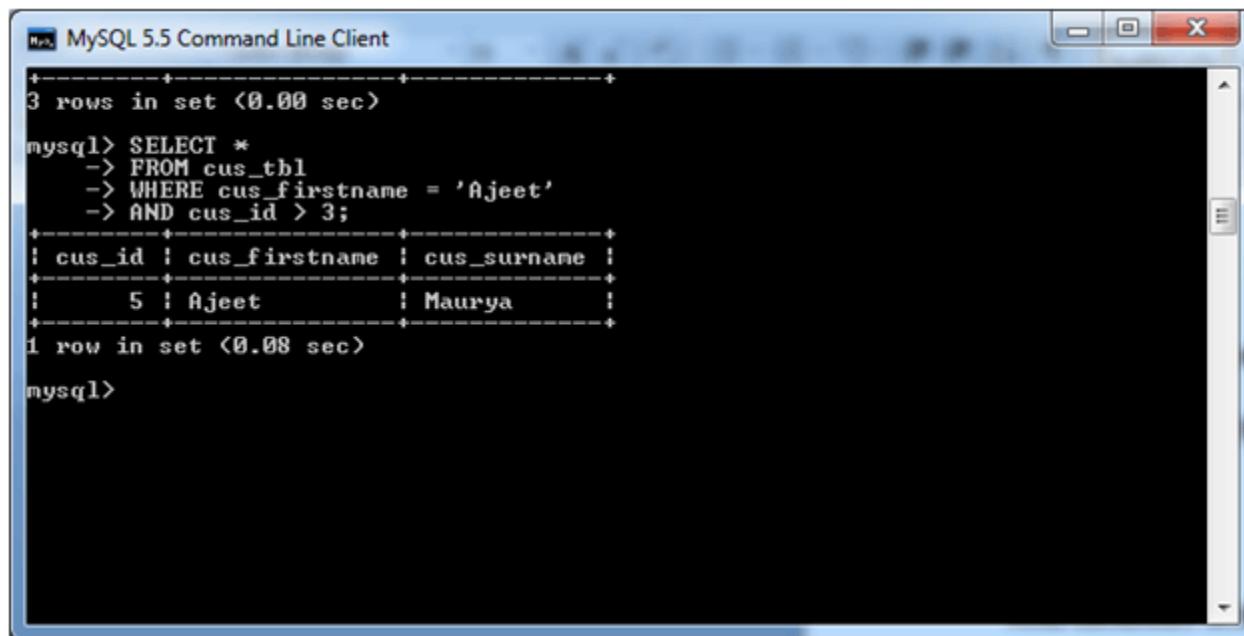
The screenshot shows the MySQL 5.5 Command Line Client interface. The session starts with a 'mysql>' prompt. The user runs a 'SELECT \* FROM cus\_tbl;' query, which returns an 'Empty set' result. Then, the user performs an 'INSERT INTO cus\_tbl' operation, defining the columns as 'cus\_id, cus\_firstname, cus\_surname'. Three rows are inserted with values: (5, 'Ajeet', 'Maurya'), (6, 'Deepika', 'Chopra'), and (7, 'Vimal', 'Jaiswal'). After the insert, a 'SELECT \* FROM cus\_tbl;' query is run again, displaying the three inserted rows in a tabular format:

cus_id	cus_firstname	cus_surname
5	Ajeet	Maurya
6	Deepika	Chopra
7	Vimal	Jaiswal

### Execute the following query:

1. **SELECT \***
2. **FROM** cus\_tbl
3. **WHERE** cus\_firstname = 'Ajeet'
4. **AND** cus\_id > 3;

### Output:



The screenshot shows the MySQL 5.5 Command Line Client window. The command entered is:

```
mysql> SELECT *  
-> FROM cus_tbl  
-> WHERE cus_firstname = 'Ajeet'  
-> AND cus_id > 3;
```

The output shows one row:

cus_id	cus_firstname	cus_surname
5	Ajeet	Maurya

1 row in set (0.08 sec)

mysql>

## MySQL OR Condition

The MySQL OR condition specifies that if you take two or more conditions then one of the conditions must be fulfilled to get the records as result.

### Syntax:

1. **WHERE** condition1
2. OR condition2
3. ...
4. OR condition\_n;

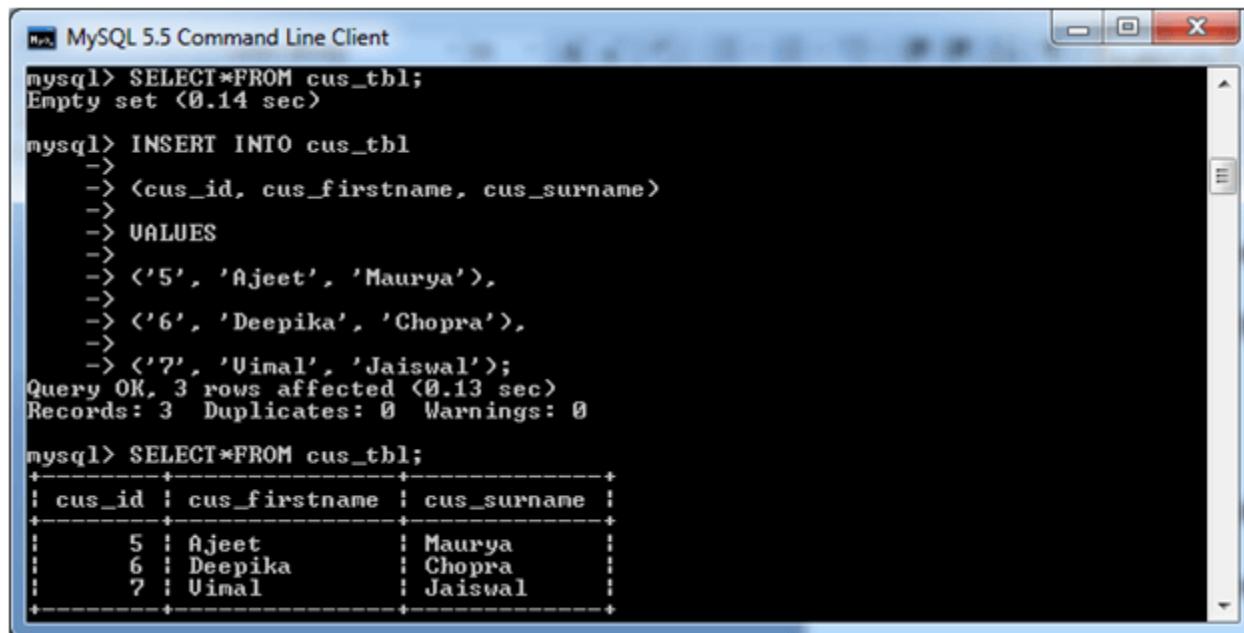
### Parameter explanation

**condition1, condition2, ... condition\_n:** Specifies all conditions that must be fulfilled for the records to be selected.

## MySQL OR Example

The following example specifies how to use the OR condition in MySQL with SELECT statement.

Consider a table "cus\_tbl", having the following data:



The screenshot shows the MySQL 5.5 Command Line Client window. The commands executed are:

```
mysql> SELECT*FROM cus_tbl;  
Empty set (0.14 sec)  
  
mysql> INSERT INTO cus_tbl  
-> <cus_id, cus_firstname, cus_surname>  
-> VALUES  
-> ('5', 'Ajeet', 'Maurya'),  
-> ('6', 'Deepika', 'Chopra'),  
-> ('7', 'Uimal', 'Jaiswal');  
Query OK, 3 rows affected (0.13 sec)  
Records: 3 Duplicates: 0 Warnings: 0  
  
mysql> SELECT*FROM cus_tbl;
```

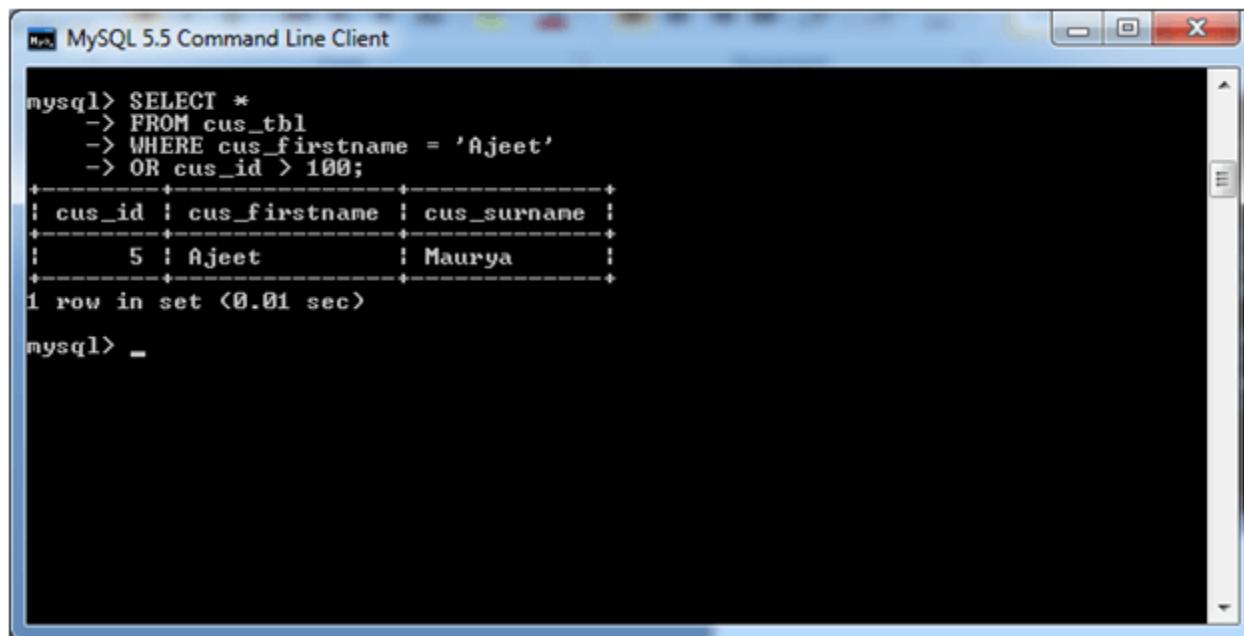
The output shows three rows:

cus_id	cus_firstname	cus_surname
5	Ajeet	Maurya
6	Deepika	Chopra
7	Uimal	Jaiswal

### Execute the following query:

1. **SELECT \***
2. **FROM** cus\_tbl
3. **WHERE** cus\_firstname = 'Ajeet'
4. **OR** cus\_id > 100;

### Output:



MySQL 5.5 Command Line Client

```
mysql> SELECT *  
-> FROM cus_tbl  
-> WHERE cus_firstname = 'Ajeet'  
-> OR cus_id > 100;  
+-----+-----+-----+  
| cus_id | cus_firstname | cus_surname |  
+-----+-----+-----+  
|      5 | Ajeet        | Maurya     |  
+-----+-----+-----+  
1 row in set (0.01 sec)  
mysql> _
```

**Note:** In the above example you can see that the second condition "cus\_id" is wrong but the query is displaying the correct result because of the OR condition.

## MySQL AND & OR condition

In MySQL, you can use AND & OR condition both together with the SELECT, INSERT, UPDATE and DELETE statement. While combine these conditions, you must be aware where to use round brackets so that the database know the order to evaluate each condition.

### Syntax:

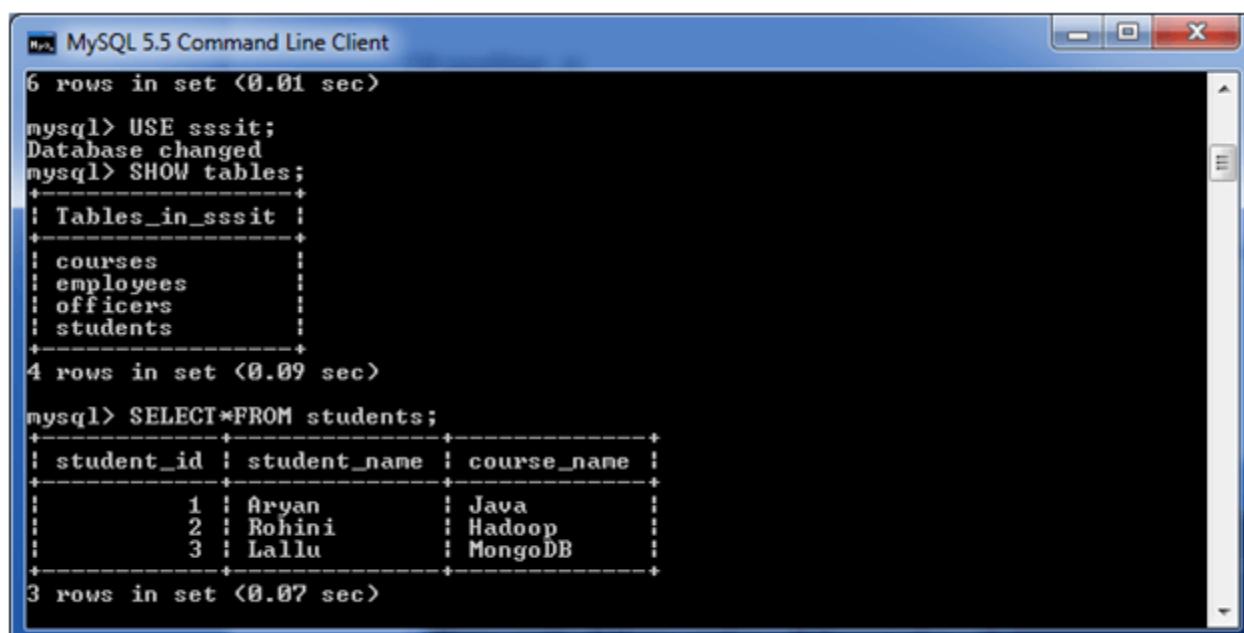
1. **WHERE** condition1
2. AND condition2
3. ...
4. OR condition\_n;

### Parameter

**condition1, condition2, ... condition\_n:** It specifies the conditions that are evaluated to determine if the records will be selected.

## MySQL AND OR Example

Consider a table "students", having the following data.

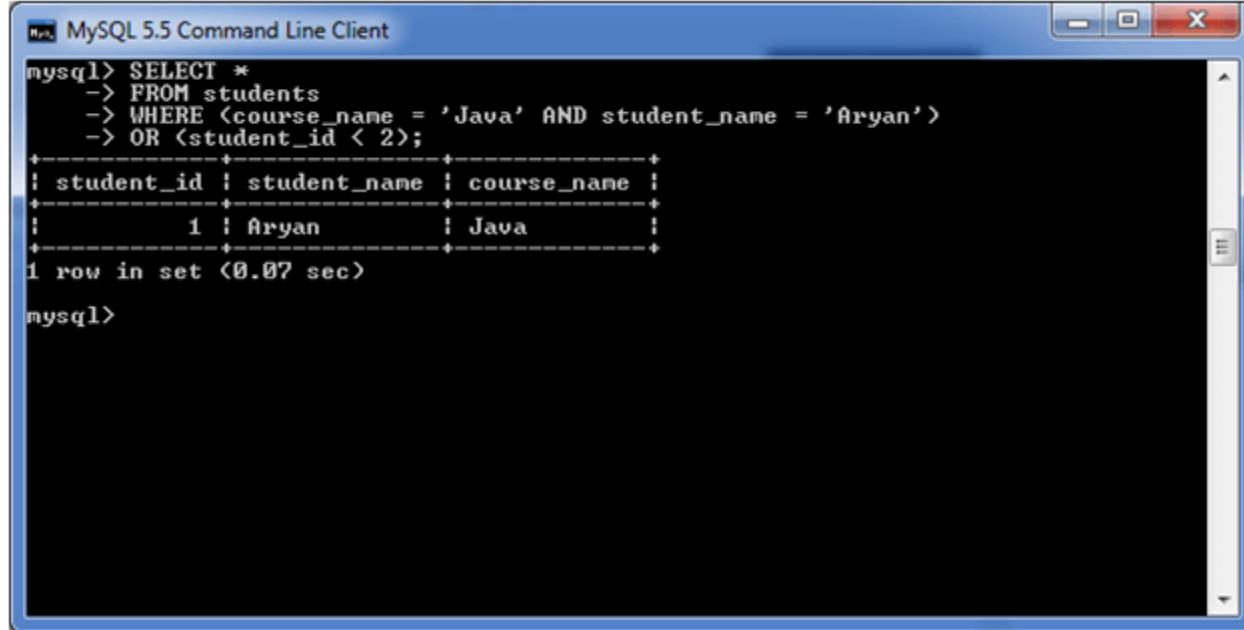


```
MySQL 5.5 Command Line Client  
6 rows in set (0.01 sec)  
  
mysql> USE sssit;  
Database changed  
mysql> SHOW tables;  
+-----+  
| Tables_in_sssit |  
+-----+  
| courses       |  
| employees     |  
| officers      |  
| students      |  
+-----+  
4 rows in set (0.09 sec)  
  
mysql> SELECT * FROM students;  
+-----+-----+-----+  
| student_id | student_name | course_name |  
+-----+-----+-----+  
|          1 | Aryan        | Java        |  
|          2 | Rohini      | Hadoop     |  
|          3 | Lallu        | MongoDB   |  
+-----+-----+-----+  
3 rows in set (0.07 sec)
```

### Execute the following query:

1. **SELECT \***
2. **FROM** students
3. **WHERE** (course\_name = 'Java' AND student\_name = 'Aryan')
4. **OR** (student\_id < 2);

## Output:



The screenshot shows the MySQL 5.5 Command Line Client window. The command entered is:

```
mysql> SELECT *  
-> FROM students  
-> WHERE course_name = 'Java' AND student_name = 'Aryan'  
-> OR student_id < 2;
```

The result set is:

student_id	student_name	course_name
1	Aryan	Java

1 row in set (0.07 sec)

mysql>

## MySQL Boolean

A Boolean is the simplest data type that always returns two possible values, either true or false. It can always use to get a confirmation in the form of YES or No value.

MySQL does not contain built-in Boolean or Bool data type. They provide a **TINYINT** data type instead of Boolean or Bool data types. MySQL considered value zero as false and non-zero value as true. If you want to use Boolean literals, use true or false that always evaluates to 0 and 1 value. The 0 and 1 represent the integer values.

Execute the following statement to see the integer values of Boolean literals:

1. Mysql> **Select TRUE, FALSE, true, false, True, False;**

After successful execution, the following result appears:

	TRUE	FALSE	true	false	True	False
▶	1	0	1	0	1	0

## MySQL Boolean Example

We can store a Boolean value in the MySQL table as an integer data type. Let us create a table student that demonstrates the use of Boolean data type in MySQL:

1. mysql> **CREATE TABLE** student (
2.   studentid **INT PRIMARY KEY** AUTO\_INCREMENT,
3.   **name VARCHAR(40)** NOT NULL,
4.   age **VARCHAR(3)**,
5.   pass **BOOLEAN**
6. );

In the above query, we can see that the pass field is defined as a Boolean when showing the definition of a table; it contains TINYINT as follows:

1. mysql> **DESCRIBE** student;

Field	Type	Null	Key	Default	Extra
studentid	int	NO	PRI	NULL	auto_increment
name	varchar(40)	NO		NULL	
age	varchar(3)	YES		NULL	
pass	tinyint(1)	YES		NULL	

Let us add two new rows in the above table with the help of following query:

1. mysql> **INSERT INTO** student(**name**, pass) **VALUES**('Peter',**true**), ('John',**false**);

When the above query executed, immediately MySQL checks for the Boolean data type in the table. If the Boolean literals found, it will be converted into integer values 0 and 1. Execute the following query to get the data from the student table:

1. Mysql> **SELECT** studentid, **name**, pass **FROM** student;

You will get the following output where the true and false literal gets converted into 0 and 1 value.

studentid	name	pass
1	Peter	1
2	John	0
NULL	NULL	NULL

Since MySQL always use TINYINT as Boolean, we can also insert any integer values into the Boolean column. Execute the following statement:

1. Mysql> **INSERT INTO** student(**name**, pass) **VALUES**('Miller',2);

You will get the following result:

studentid	name	pass
1	Peter	1
2	John	0
3	Miller	2
NULL	NULL	NULL

In some cases, you need to get the result in true and false literals. In that case, you need to execute the if() function with the select statement as follows:

1. Mysql> **SELECT** studentid, **name**, IF(pass, 'true', 'false') completed **FROM** student1;

It will give the following output:

studentid	name	completed
1	Peter	true
2	John	false
3	Miller	true

## MySQL Boolean Operators

MySQL also allows us to use operators with the Boolean data type. Execute the following query to get all the pass result of table student.

1. **SELECT** studentid, **name**, pass **FROM** student1 **WHERE** pass = **TRUE**;

This statement returns the following output:

1	Peter	1
NULL	NULL	NULL

The above statement only returns the pass result if the value is equal to 1. We can fix it by using the **IS** operator. This operator validates the value with the Boolean value. The following statement explains this:

1. **SELECT** studentid, **name**, pass **FROM** student1 **WHERE** pass **is TRUE**;

After executing this statement, you will get the following result:

studentid	name	pass
1	Peter	1
3	Miller	2
NULL	NULL	NULL

If you want to see the pending result, use **IS FALSE** or **IS NOT TRUE** operator as below:

1. **SELECT** studentid, **name**, pass **FROM** student1 **WHERE** pass **IS FALSE**;

2.

3. OR,

4.

5. **SELECT** studentid, **name**, pass **FROM** student1 **WHERE** pass **IS NOT TRUE**;

You will get the following output:

studentid	name	pass
2	John	0
NULL	NULL	NULL

# MySQL LIKE condition

In MySQL, LIKE condition is used to perform pattern matching to find the correct result. It is used in SELECT, INSERT, UPDATE and DELETE statement with the combination of WHERE clause.

## Syntax:

1. expression LIKE pattern [ **ESCAPE** 'escape\_character' ]

## Parameters

**expression:** It specifies a column or field.

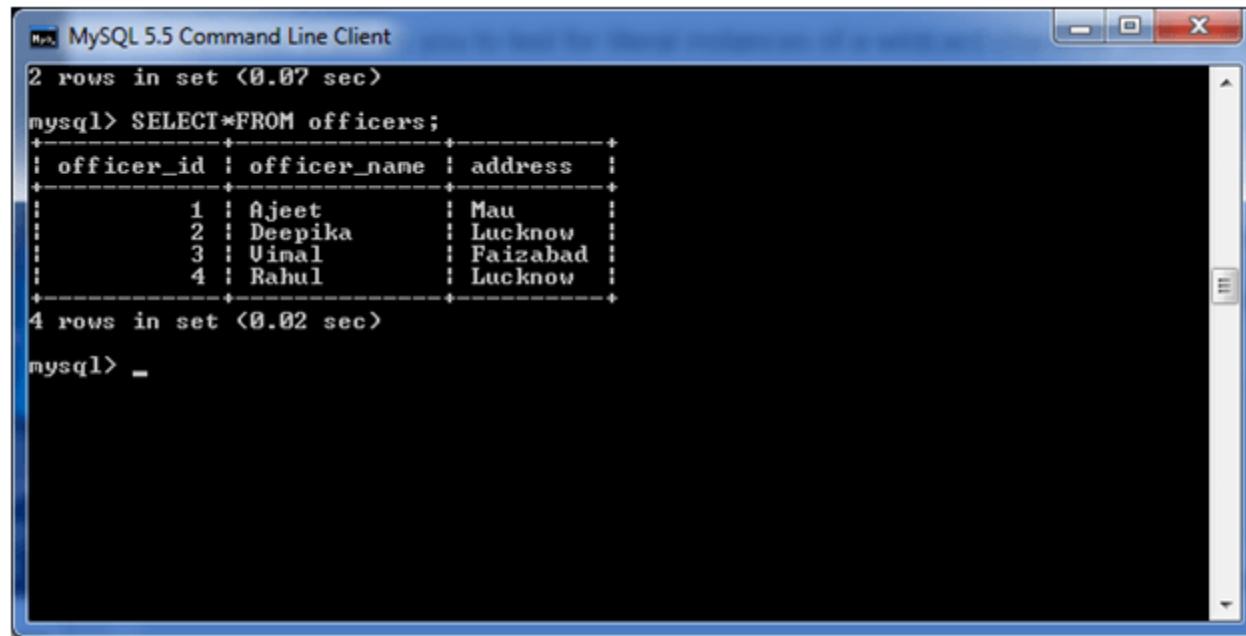
**pattern:** It is a character expression that contains pattern matching.

**escape\_character:** It is optional. It allows you to test for literal instances of a wildcard character such as % or \_. If you do not provide the escape\_character, MySQL assumes that "\\" is the escape\_character.

## MySQL LIKE Examples

### 1) Using % (percent) Wildcard:

Consider a table "officers" having the following data.



The screenshot shows the MySQL 5.5 Command Line Client window. The command `SELECT * FROM officers;` is run, resulting in the following output:

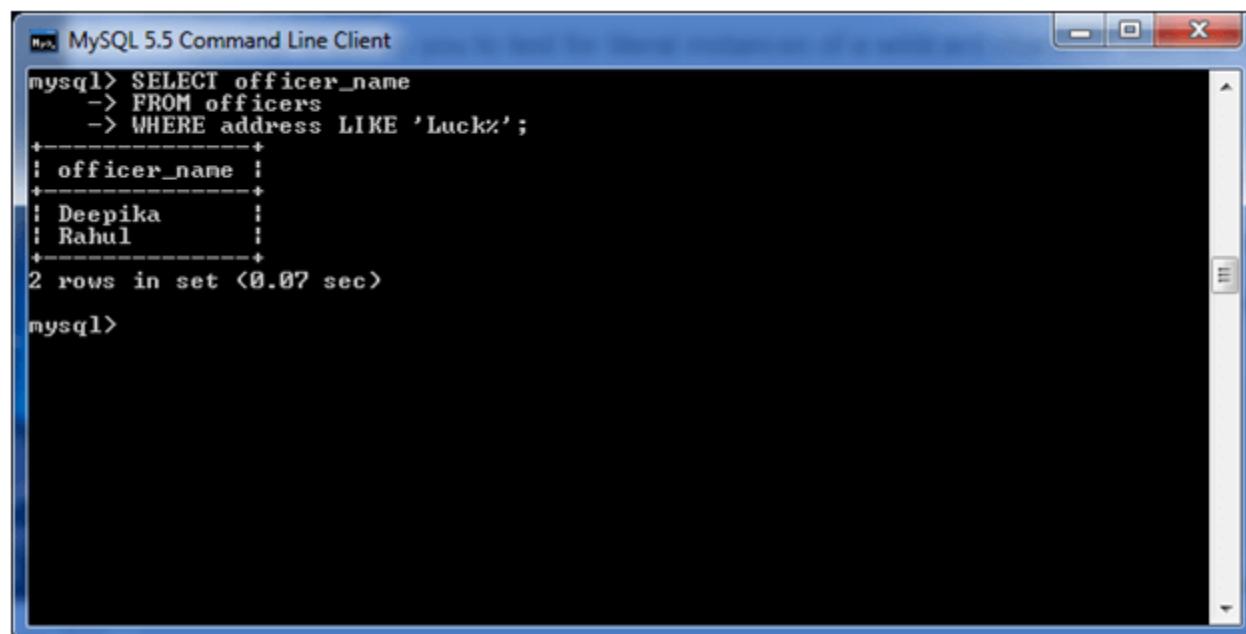
officer_id	officer_name	address
1	Ajeet	Mau
2	Deepika	Lucknow
3	Uinal	Faizabad
4	Rahul	Lucknow

4 rows in set (0.02 sec)

Execute the following query:

1. **SELECT** officer\_name
2. **FROM** officers
3. **WHERE** address LIKE 'Luck%';

Output:



The screenshot shows the MySQL 5.5 Command Line Client window. The query `SELECT officer_name FROM officers WHERE address LIKE 'Luck%';` is run, resulting in the following output:

officer_name
Deepika
Rahul

2 rows in set (0.07 sec)

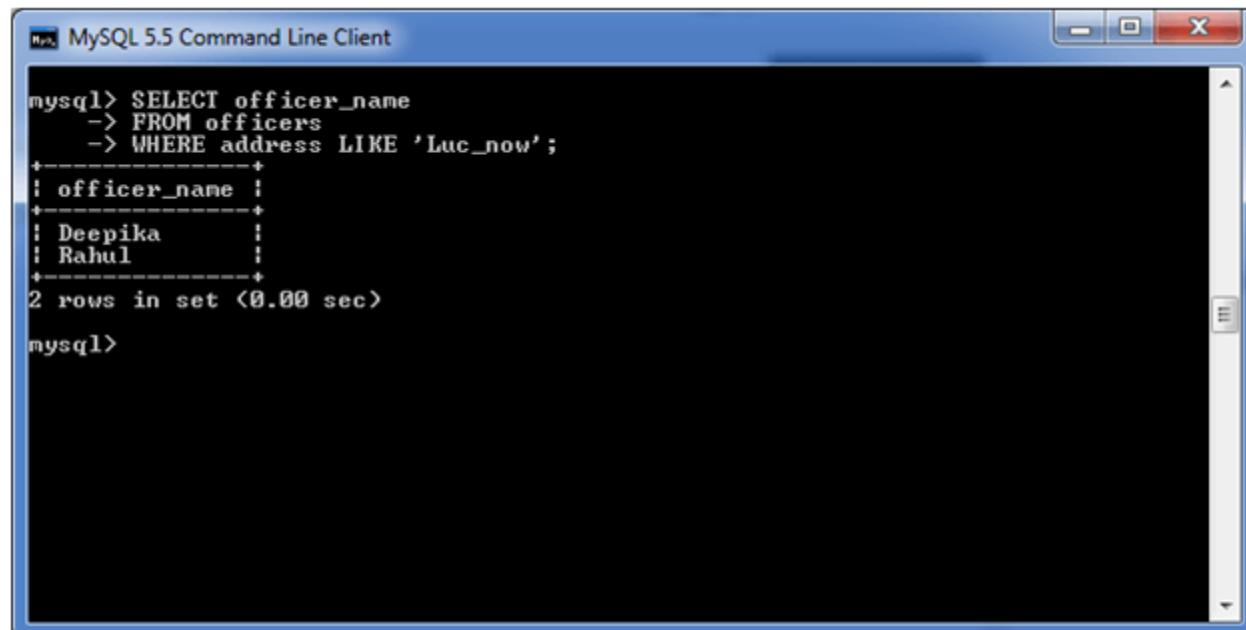
### 2) Using \_ (Underscore) Wildcard:

We are using the same table "officers" in this example too.

**Execute the following query:**

- 1.
2. **SELECT** officer\_name
3. **FROM** officers
4. **WHERE** address LIKE 'Luc\_now';

**Output:**



```
MySQL 5.5 Command Line Client

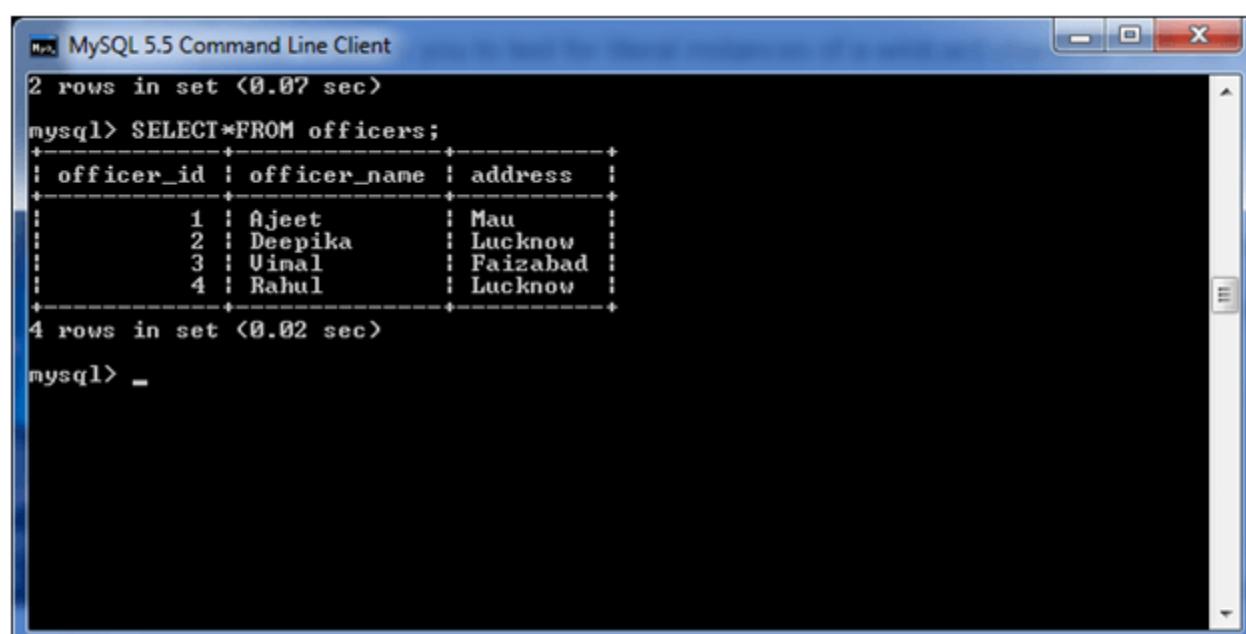
mysql> SELECT officer_name
-> FROM officers
-> WHERE address LIKE 'Luc_now';
+-----+
| officer_name |
+-----+
| Deepika    |
| Rahul      |
+-----+
2 rows in set (0.00 sec)

mysql>
```

### 3) Using NOT Operator:

You can also use NOT operator with MySQL LIKE condition. This example shows the use of % wildcard with the NOT Operator.

Consider a table "officers" having the following data.



```
MySQL 5.5 Command Line Client

2 rows in set (0.07 sec)

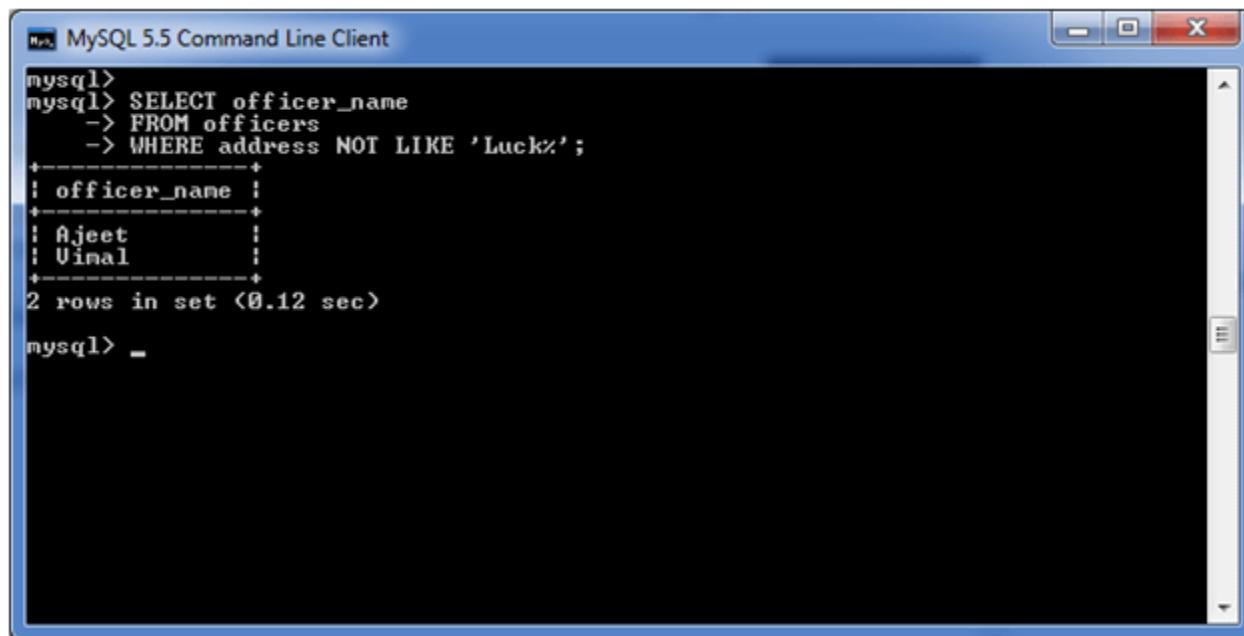
mysql> SELECT * FROM officers;
+-----+-----+-----+
| officer_id | officer_name | address |
+-----+-----+-----+
| 1          | Ajeeet       | Mau      |
| 2          | Deepika      | Lucknow  |
+-----+-----+-----+
4 rows in set (0.02 sec)

mysql> _
```

**Execute the following query:**

1. **SELECT** officer\_name
2. **FROM** officers
3. **WHERE** address NOT LIKE 'Luck%';

**Output:**



MySQL 5.5 Command Line Client

```
mysql> SELECT officer_name
-> FROM officers
-> WHERE address NOT LIKE 'Luck%';
+-----+
| officer_name |
+-----+
| Ajeet        |
| Vimal        |
+-----+
2 rows in set (0.12 sec)

mysql> _
```

**Note:** In the above example, you can see that the addresses NOT LIKE 'Luck%' are only shown.

## MySQL IN Condition

The MySQL IN condition is used to reduce the use of multiple OR conditions in a SELECT, INSERT, UPDATE and DELETE statement.

### Syntax:

1. expression IN (value1, value2, .... value\_n);

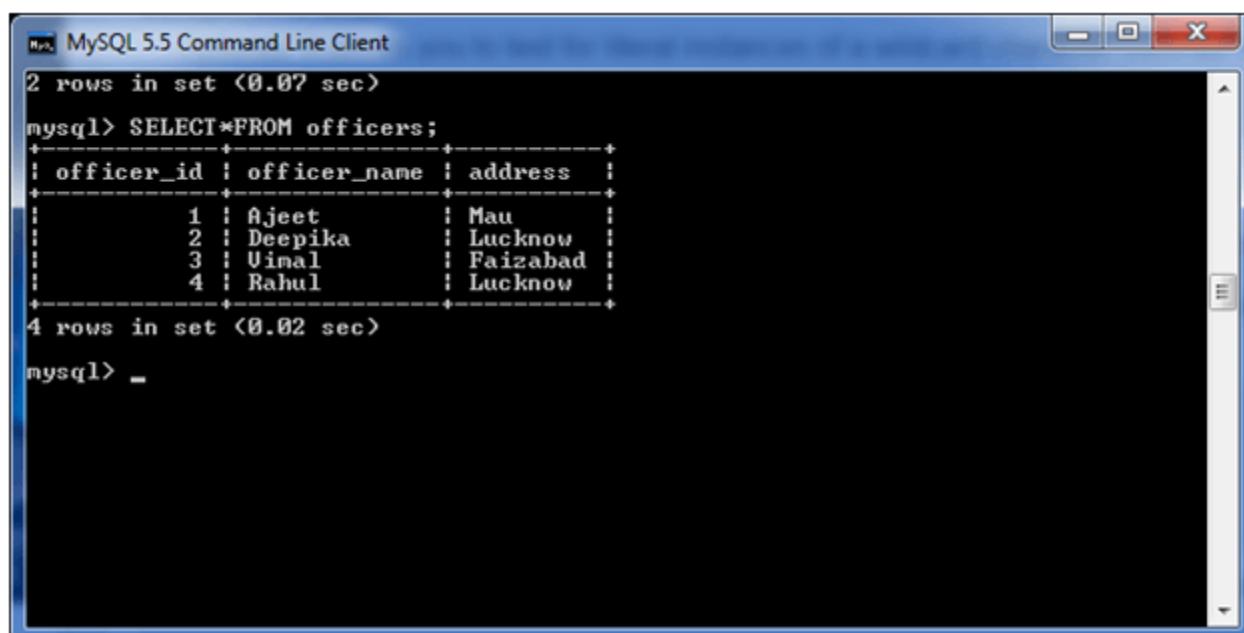
### Parameters

**expression:** It specifies a value to test.

**value1, value2, ... or value\_n:** These are the values to test against expression. If any of these values matches expression, then the IN condition will evaluate to true. This is a quick method to test if any one of the values matches expression.

## MySQL IN Example

Consider a table "officers", having the following data.



MySQL 5.5 Command Line Client

```
2 rows in set (0.07 sec)

mysql> SELECT * FROM officers;
+-----+-----+-----+
| officer_id | officer_name | address |
+-----+-----+-----+
| 1          | Ajeet       | Mau      |
| 2          | Deepika     | Lucknow  |
+-----+-----+-----+
4 rows in set (0.02 sec)

mysql> _
```

Execute the following query:

1. **SELECT \***
2. **FROM** officers
3. **WHERE** officer\_name IN ('Ajeet', 'Vimal', 'Deepika');

**Output:**

The screenshot shows the MySQL 5.5 Command Line Client window. The command entered is:

```
mysql> SELECT *  
    -> FROM officers  
    -> WHERE officer_name IN ('Ajeet', 'Vimal', 'Deepika');
```

The resulting table output is:

officer_id	officer_name	address
1	Ajeet	Mau
2	Deepika	Lucknow
3	Vimal	Faizabad

3 rows in set (0.07 sec)

mysql>

Let's see why it is preferred over OR condition:

**Execute the following query:**

1. **SELECT \***
2. **FROM officers**
3. **WHERE officer\_name = 'Ajeet'**
4. **OR officer\_name = 'Vimal'**
5. **OR officer\_name = 'Deepika';**

**Output:**

The screenshot shows the MySQL 5.5 Command Line Client window. The command entered is:

```
mysql> SELECT *  
    -> FROM officers  
    -> WHERE officer_name = 'Ajeet'  
    -> OR officer_name = 'Vimal'  
    -> OR officer_name = 'Deepika';
```

The resulting table output is:

officer_id	officer_name	address
1	Ajeet	Mau
2	Deepika	Lucknow
3	Vimal	Faizabad

3 rows in set (0.00 sec)

mysql>

It also produces the same result. So IN condition is preferred over OR condition because it has minimum number of codes.

## MySQL ANY

The ANY keyword is a MySQL operator that **returns the Boolean value TRUE** if the comparison is TRUE for ANY of the subquery condition. In other words, this keyword returns true if any of the subquery condition is fulfilled when the SQL query is executed. The ANY keyword must **follow the comparison operator**. It is noted that **ALL SQL** operator works related to ANY operator, but it returns true when all the subquery values are satisfied by the condition in MySQL.

The ANY operator works like comparing the value of a table to each value in the result set provided by the subquery condition. And then, if it finds any value that matches at least one value/row of the subquery, it returns the TRUE result.

### Syntax

The following is the syntax that illustrates the use of ANY operator in MySQL:

1. operand comparison\_operator ANY (subquery)

Where comparison operators can be one of the following:

1. = > < >= <= <> !=

This syntax can also be written as:

1. **SELECT** column\_lists **FROM** table\_name1 **WHERE** column\_name Operator ANY (**SELECT** column\_name **FROM** table\_name2 **WHERE** condition);

We can understand how ANY works in [MySQL](#) through the below statement:

1. **SELECT** colm1 **FROM** table1 **WHERE** colm1 > ANY (**SELECT** colm1 **FROM** table2);

Suppose **table1** has a row that contains a **number (10)**. In such a case, the above expression returns **true** if **table2** contains (20, 15, and 6). It is because there is a value 6 in table2, which is less than 10. This expression returns **false** if table2 contains (15, 20), or if table2 is **empty**. If all the table fields contain (NULL, NULL, NULL), this expression is **unknown**.

## Example

Let us create a two table named **table1** and **table2** and then insert some values into them using the below statements:

1. **CREATE TABLE** table1 (
2.   num\_value **INT**
3. );
4. **INSERT INTO** table1 (num\_value)
5.   **VALUES**(10), (20), (25);
- 6.
7. **CREATE TABLE** table2 (
8.   num\_val **int**
9. );
10. **INSERT INTO** table2 (num\_val)
11.   **VALUES**(20), (7), (10);

After successful execution of the above statement, we can verify it by using the [SELECT statement](#) as follows:

The screenshot shows the MySQL 8.0 Command Line Client window. It displays the following SQL commands and their results:

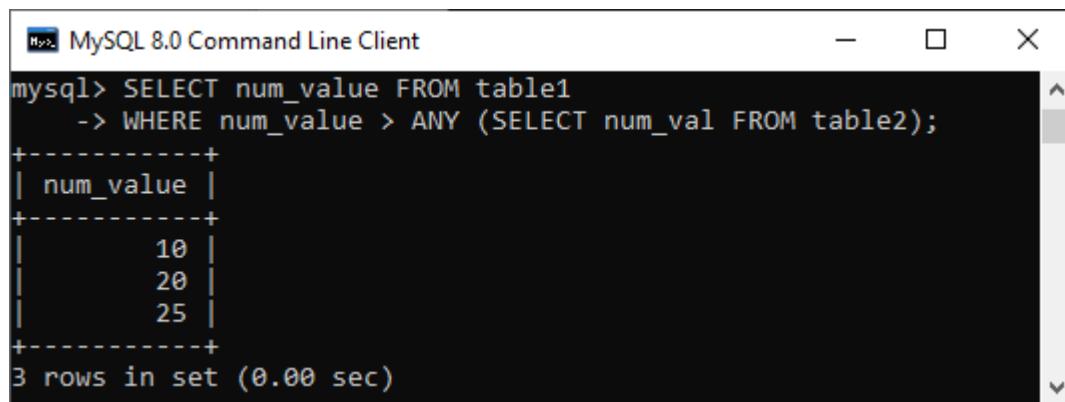
```
mysql> SELECT * FROM table1;
+-----+
| num_value |
+-----+
|      10   |
|      20   |
|      25   |
+-----+
3 rows in set (0.00 sec)

mysql> SELECT * FROM table2;
+-----+
| num_val |
+-----+
|      20   |
|       7   |
|      10   |
+-----+
3 rows in set (0.00 sec)
```

Now, we will execute the below statement to understand the use of the ANY operator:

1. **SELECT** num\_value **FROM** table1
2. **WHERE** num\_value > ANY (**SELECT** num\_val **FROM** table2);

This statement returns true and gives the below output because table2 contains (20, 10, and 7) and there is a value 7 in table2 which is less than 10, 20, and 25 of table1.



The screenshot shows a terminal window titled "MySQL 8.0 Command Line Client". The command entered is:

```
mysql> SELECT num_value FROM table1
   -> WHERE num_value > ANY (SELECT num_val FROM table2);
```

The output is:

num_value
10
20
25

3 rows in set (0.00 sec)

The **IN keyword** in MySQL is an **alias for = ANY** when used with a subquery in the statement. Hence, the below two statements are identical in MySQL:

1. **SELECT colm1 FROM table1 WHERE colm1 = ANY (SELECT colm1 FROM table2);**
2. **SELECT colm1 FROM table1 WHERE colm1 IN (SELECT colm1 FROM table2);**

But we cannot say that IN and = ANY were synonyms when we used it with an expression list. It is because IN can take a list of expressions, but = ANY cannot.

Also, **NOT IN** cannot be an alias for **<> ANY** operator, but it can be used for **<> ALL**.

The word **SOME** in MySQL can be an alias for ANY. Therefore, these two [SQL](#) statements are equivalent:

1. **SELECT colm1 FROM table1 WHERE colm1 <> ANY (SELECT colm1 FROM table2);**
2. **SELECT colm1 FROM table1 WHERE colm1 <> SOME (SELECT colm1 FROM table2);**

## Advantages of ANY operator in MySQL

- ANY is a logical operator that returns the Boolean value. It allows us to select any or some rows of the SELECT statement.
- Since comparison operators precede this operator, it always returns TRUE if any subqueries satisfy the specified condition.
- It provides the result, which is a unique column value from a table that matches any record in the second table.
- We can perform several comparisons using ANY operator with the SELECT and WHERE keywords.

In this article, we have learned how to use the ANY operator in MySQL. It filters the result set from SQL syntax only when any of the values satisfy the condition. Otherwise, it gives a false value.

## MySQL Exists

The EXISTS operator in MySQL is a type of Boolean operator which returns the **true or false** result. It is used in combination with a subquery and checks the existence of data in a subquery. It means if a subquery returns any record, this operator returns true. Otherwise, it will return false. The true value is always represented numeric value 1, and the false value represents 0. We can use it with SELECT, UPDATE, DELETE, INSERT statement.

### Syntax

The following are the syntax to use the EXISTS operator in [MySQL](#):

1. **SELECT col\_names**
2. **FROM tab\_name**
3. **WHERE [NOT] EXISTS (**
4. **SELECT col\_names**
5. **FROM tab\_name**
6. **WHERE condition**
7. **);**

The NOT operator is used to negate the EXISTS operator. It returns true when the subquery does not return any row. Otherwise, it returns false.

Generally, the EXISTS query begins with **SELECT \***, but it can start with the **SELECT column**, **SELECT a\_constant**, or anything in the subquery. It will give the same output because MySQL ignores the select list in the SUBQUERY.

This operator terminates immediately for further processing after the matching result found. This feature improves the performance of the query in MySQL.

## Parameter Explanation

The following are parameters used in the EXISTS operator:

Parameter Name	Descriptions
col_names	It is the name of column(s) that contains in the specified table.
tab_name	It is the name of the table from which we are going to perform the EXISTS operator.
condition	It specifies for searching the specific value from the table.
subquery	It is usually the SELECT statement that begins with SELECT *, but MySQL ignores it in a subquery.

## MySQL EXISTS Operator Example

Let us understand how the EXISTS operator works in MySQL. Here, we are going to first create two tables named "**customer**" and "**orders**" using the following statement:

1. **CREATE TABLE** customer(  
2.    **cust\_id int** NOT NULL,  
3.    **name varchar(35)**,  
4.    **occupation varchar(25)**,  
5.    **age int**  
6. );
  
1. **CREATE TABLE** orders (  
2.    **order\_id int** NOT NULL,  
3.    **cust\_id int**,  
4.    **prod\_name varchar(45)**,  
5.    **order\_date date**  
6. );

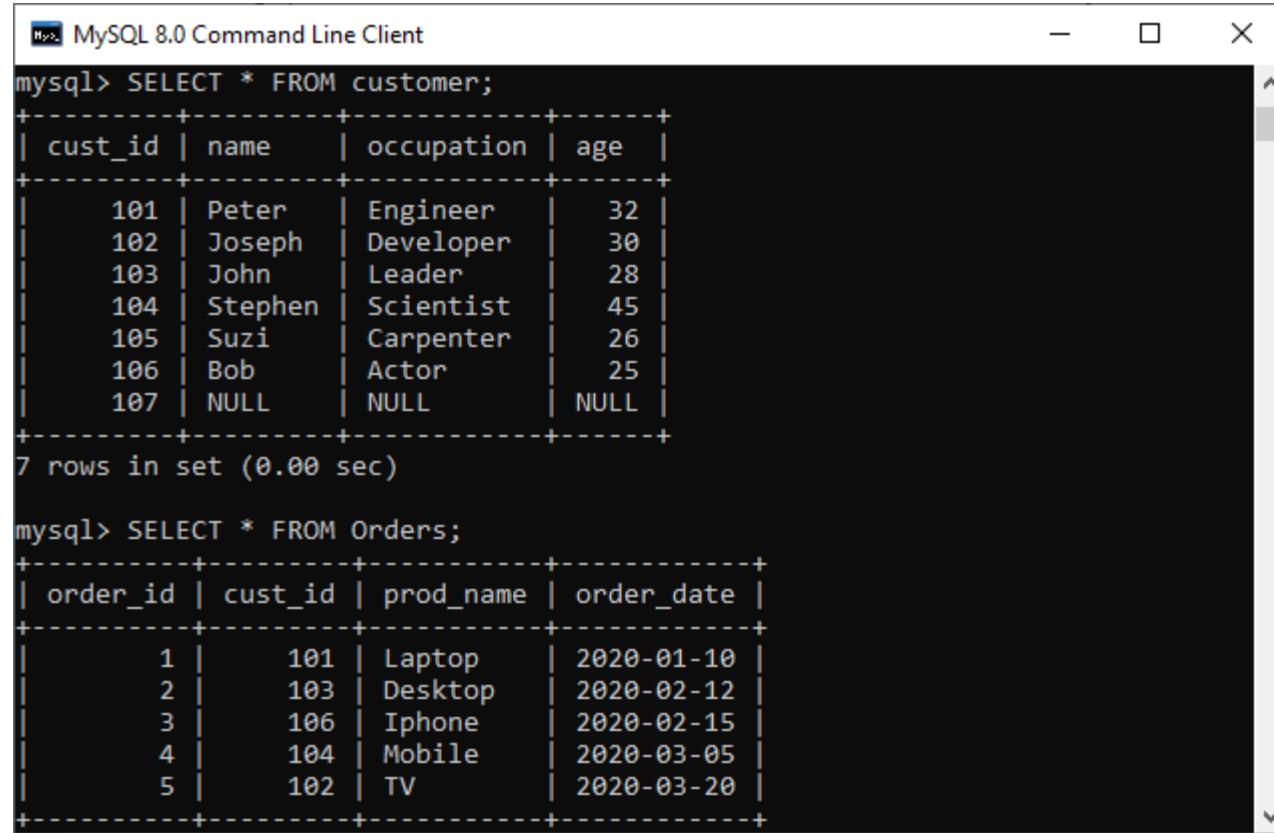
Next, we need to insert values into both tables. Execute the below statements:

1. **INSERT INTO** customer(**cust\_id**, **name**, **occupation**, **age**)  
2. **VALUES** (101, 'Peter', 'Engineer', 32),  
3. (102, 'Joseph', 'Developer', 30),  
4. (103, 'John', 'Leader', 28),  
5. (104, 'Stephen', 'Scientist', 45),  
6. (105, 'Suzi', 'Carpenter', 26),  
7. (106, 'Bob', 'Actor', 25),  
8. (107, NULL, NULL, NULL);
  
1. **INSERT INTO** orders (**order\_id**, **cust\_id**, **prod\_name**, **order\_date**)  
2. **VALUES** (1, '101', 'Laptop', '2020-01-10'),  
3. (2, '103', 'Desktop', '2020-02-12'),  
4. (3, '106', 'Iphone', '2020-02-15'),  
5. (4, '104', 'Mobile', '2020-03-05'),  
6. (5, '102', 'TV', '2020-03-20');

To verify the tables, run the **SELECT command** as below:

1. mysql> **SELECT \* FROM** customer;
2. AND,
3. mysql> **SELECT \* FROM** orders;

We will get the below output:



```
MySQL 8.0 Command Line Client
mysql> SELECT * FROM customer;
+-----+-----+-----+-----+
| cust_id | name   | occupation | age   |
+-----+-----+-----+-----+
| 101    | Peter   | Engineer   | 32    |
| 102    | Joseph  | Developer  | 30    |
| 103    | John    | Leader     | 28    |
| 104    | Stephen | Scientist  | 45    |
| 105    | Suzi    | Carpenter  | 26    |
| 106    | Bob     | Actor      | 25    |
| 107    | NULL    | NULL       | NULL  |
+-----+-----+-----+-----+
7 rows in set (0.00 sec)

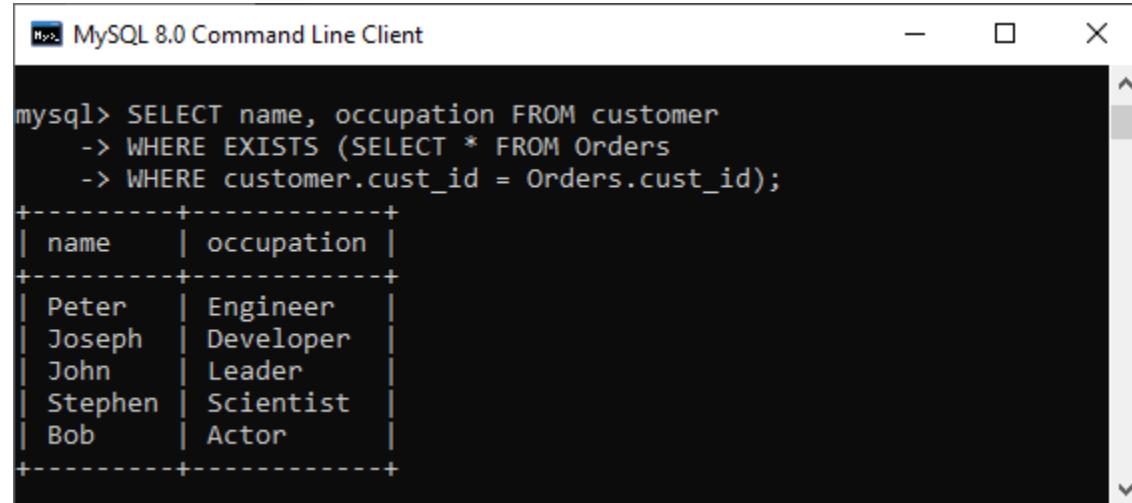
mysql> SELECT * FROM Orders;
+-----+-----+-----+-----+
| order_id | cust_id | prod_name | order_date |
+-----+-----+-----+-----+
| 1         | 101    | Laptop    | 2020-01-10 |
| 2         | 103    | Desktop   | 2020-02-12 |
| 3         | 106    | Iphone    | 2020-02-15 |
| 4         | 104    | Mobile    | 2020-03-05 |
| 5         | 102    | TV        | 2020-03-20 |
+-----+-----+-----+-----+
```

## MySQL SELECT EXISTS Example

In this example, we are going to use EXISTS operator to find the name and occupation of the customer who has placed at least one order:

1. mysql> **SELECT name, occupation FROM** customer
2. **WHERE EXISTS (SELECT \* FROM** Orders
3. **WHERE customer.cust\_id = Orders.cust\_id);**

The following output appears:



```
MySQL 8.0 Command Line Client
mysql> SELECT name, occupation FROM customer
   -> WHERE EXISTS (SELECT * FROM Orders
   -> WHERE customer.cust_id = Orders.cust_id);
+-----+-----+
| name   | occupation |
+-----+-----+
| Peter   | Engineer   |
| Joseph  | Developer  |
| John    | Leader     |
| Stephen | Scientist |
| Bob     | Actor      |
+-----+-----+
```

Again, if we want to get the name of the customer who has not placed an order, then use the NOT EXISTS operator:

1. mysql> **SELECT name, occupation FROM** customer
2. **WHERE NOT EXISTS (SELECT \* FROM** Orders
3. **WHERE customer.cust\_id = Orders.cust\_id);**

It will give the below output:

```

MySQL 8.0 Command Line Client
mysql> SELECT name FROM customer
-> WHERE NOT EXISTS (SELECT * FROM Orders
-> WHERE customer.cust_id = Orders.cust_id);
+-----+
| name |
+-----+
| Suzi |
| NULL |
+-----+
2 rows in set (0.00 sec)

```

## MySQL EXISTS With DELETE Statement Example

Suppose we want to delete a record from the Orders table whose order\_id = 3, execute the following query that deletes the record from Orders table permanently:

1. mysql> **DELETE FROM** Orders **WHERE** EXISTS (
2. **SELECT \* FROM** customer
3. **WHERE** order\_id=3);

To verify the output, run the below command:

1. mysql> **SELECT \* FROM** Orders;

In the output, we can see that the table record whose order\_id=3 is deleted successfully.

```

MySQL 8.0 Command Line Client
mysql> DELETE FROM Orders WHERE EXISTS (
-> SELECT * FROM customer
-> WHERE order_id=3);
Query OK, 1 row affected (0.18 sec)

mysql> SELECT * FROM Orders;
+-----+-----+-----+-----+
| order_id | cust_id | prod_name | order_date |
+-----+-----+-----+-----+
|      1   |    101  | Laptop     | 2020-01-10 |
|      2   |    103  | Desktop    | 2020-02-12 |
|      4   |    104  | Mobile     | 2020-03-05 |
|      5   |    102  | TV         | 2020-03-20 |
+-----+-----+-----+-----+

```

If we want to check whether a row exists in a table or not, use the following query:

1. mysql> **SELECT EXISTS**(**SELECT \* from** customer **WHERE** cust\_id=104) **AS** Result;

We will get the output 1 that means true. Hence, cust\_id=104 exists in the table.

```

MySQL 8.0 Command Line Client
mysql> SELECT EXISTS(SELECT * from customer WHERE cust_id=104) AS Result;
+-----+
| Result |
+-----+
|      1 |
+-----+
1 row in set (0.00 sec)

```

## Difference between EXISTS and IN operator

The main differences between the EXISTS and IN operator is given in a tabular form:

SN	IN	EXISTS
1.	It is used to minimize the multiple OR conditions in MySQL.	It is used to check the existence of data in a subquery.
2.	SELECT col_names FROM tab_name WHERE col_name IN (subquery);	SELECT col_names FROM tab_name WHERE [NOT] EXISTS (subquery);

<b>3.</b>	It compares all values inside the IN operator.	It stops for further execution as soon as it finds the first true occurrence.
<b>4.</b>	It can use for comparing NULL values.	It cannot use for comparing NULL values.
<b>5.</b>	It executes faster when the subquery result is less.	It executes faster when the subquery result is large.
<b>6.</b>	It performs a comparison between parent query and child query or subquery.	It does not perform a comparison between parent query and child query or subquery.

## MySQL NOT Condition

The MySQL NOT condition is opposite of MySQL IN condition. It is used to negate a condition in a SELECT, INSERT, UPDATE or DELETE statement.

### Syntax:

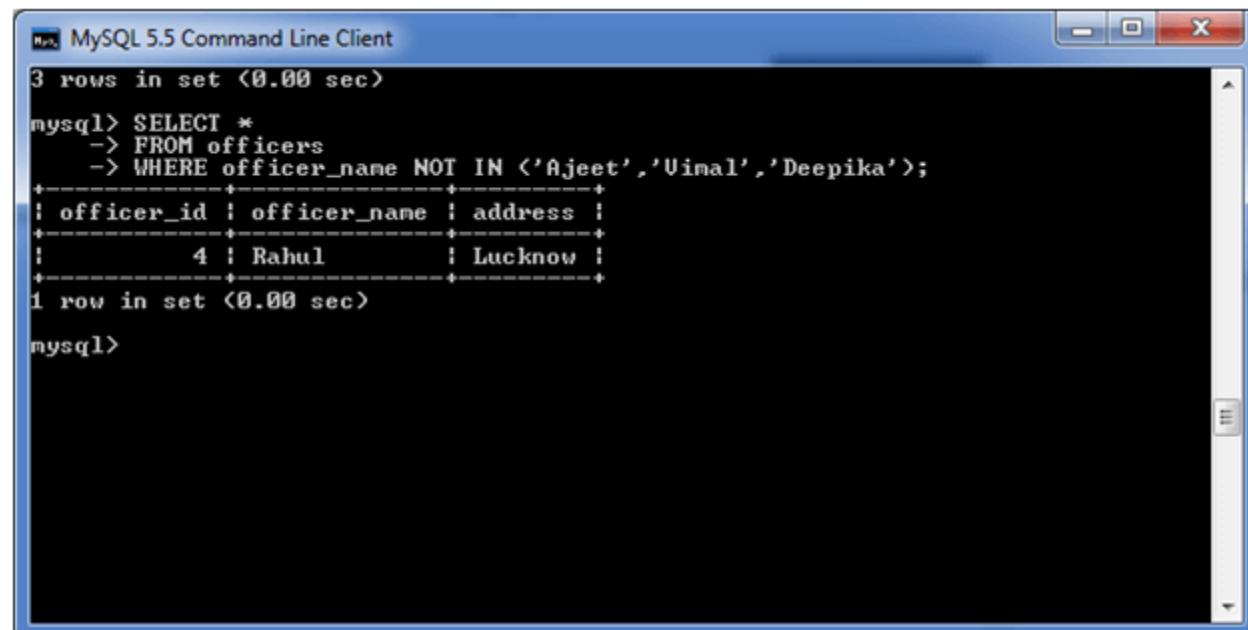
1. NOT condition

### Parameter

**condition:** It specifies the conditions that you want to negate.

### MySQL NOT Operator with IN condition

Consider a table "officers", having the following data.



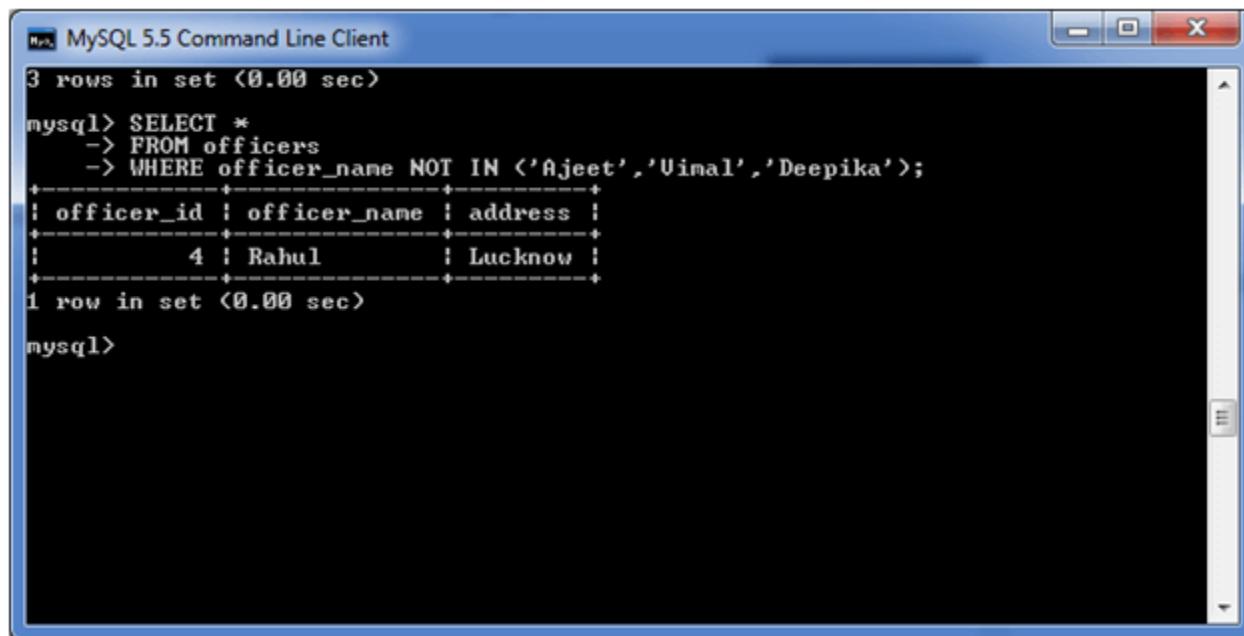
```
MySQL 5.5 Command Line Client
3 rows in set <0.00 sec>
mysql> SELECT *
-> FROM officers
-> WHERE officer_name NOT IN ('Ajeet','Vimal','Deepika');
+-----+-----+-----+
| officer_id | officer_name | address |
+-----+-----+-----+
|       4    | Rahul        | Lucknow  |
+-----+-----+-----+
1 row in set <0.00 sec>

mysql>
```

### Execute the following query:

1. **SELECT \***
2. **FROM** officers
3. **WHERE** officer\_name NOT IN ('Ajeet','Vimal','Deepika');

### Output:



MySQL 5.5 Command Line Client

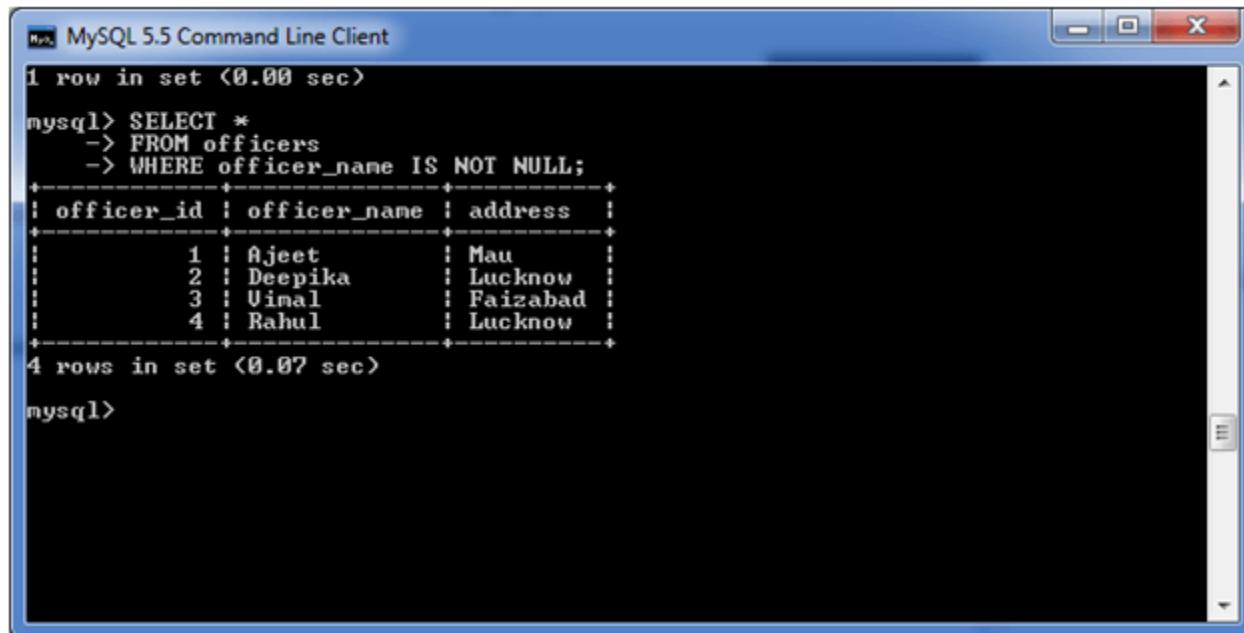
```
3 rows in set <0.00 sec>
mysql> SELECT *
    -> FROM officers
    -> WHERE officer_name NOT IN ('Ajeet','Uimal','Deepika');
+-----+-----+-----+
| officer_id | officer_name | address |
+-----+-----+-----+
|       4     |      Rahul   | Lucknow |
+-----+-----+-----+
1 row in set <0.00 sec>
mysql>
```

## MySQL NOT Operator with IS NULL condition:

Execute the following query:

1. **SELECT \***
2. **FROM officers**
3. **WHERE officer\_name IS NOT NULL;**

Output:



MySQL 5.5 Command Line Client

```
1 row in set <0.00 sec>
mysql> SELECT *
    -> FROM officers
    -> WHERE officer_name IS NOT NULL;
+-----+-----+-----+
| officer_id | officer_name | address |
+-----+-----+-----+
|       1     |      Ajeet    | Mau      |
|       2     |      Deepika  | Lucknow  |
|       3     |      Uimal    | Faizabad |
|       4     |      Rahul    | Lucknow  |
+-----+-----+-----+
4 rows in set <0.07 sec>
mysql>
```

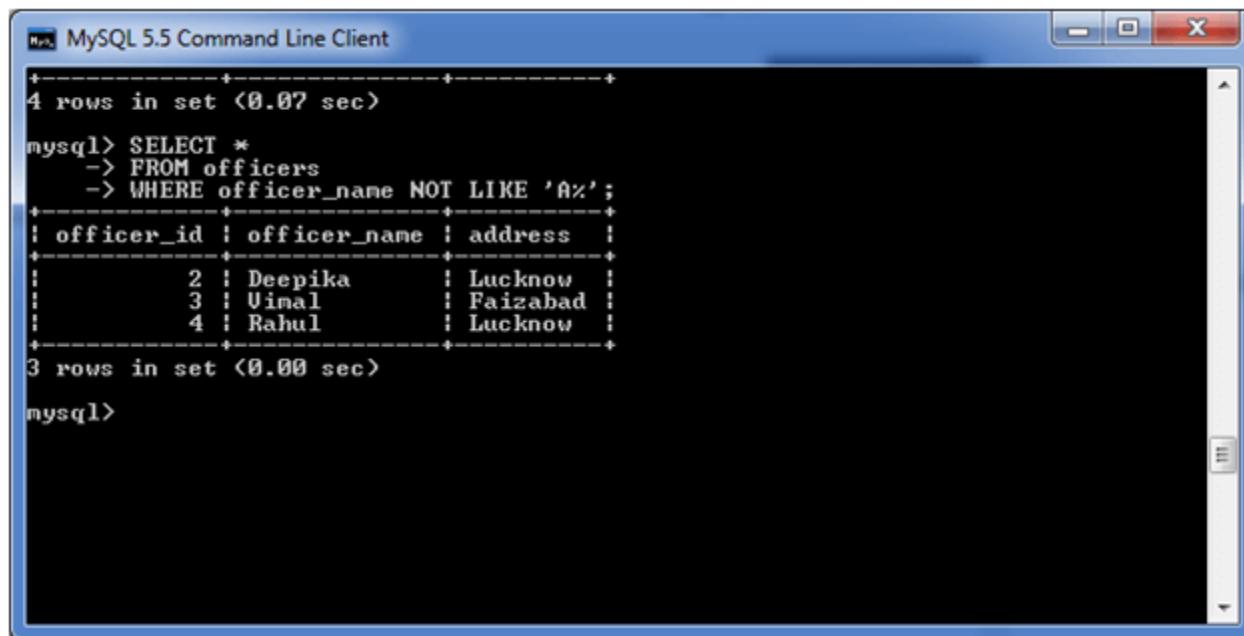
## MySQL NOT Operator with LIKE condition:

We are taking the same table "officer" for this operation also:

Execute the following query:

1. **SELECT \***
2. **FROM officers**
3. **WHERE officer\_name NOT LIKE 'A%';**

Output:



MySQL 5.5 Command Line Client

```
+-----+-----+
| officer_id | officer_name | address |
+-----+-----+
| 2 | Deepika | Lucknow |
| 3 | Vimal | Faizabad |
| 4 | Rahul | Lucknow |
+-----+
```

3 rows in set (0.00 sec)

mysql>

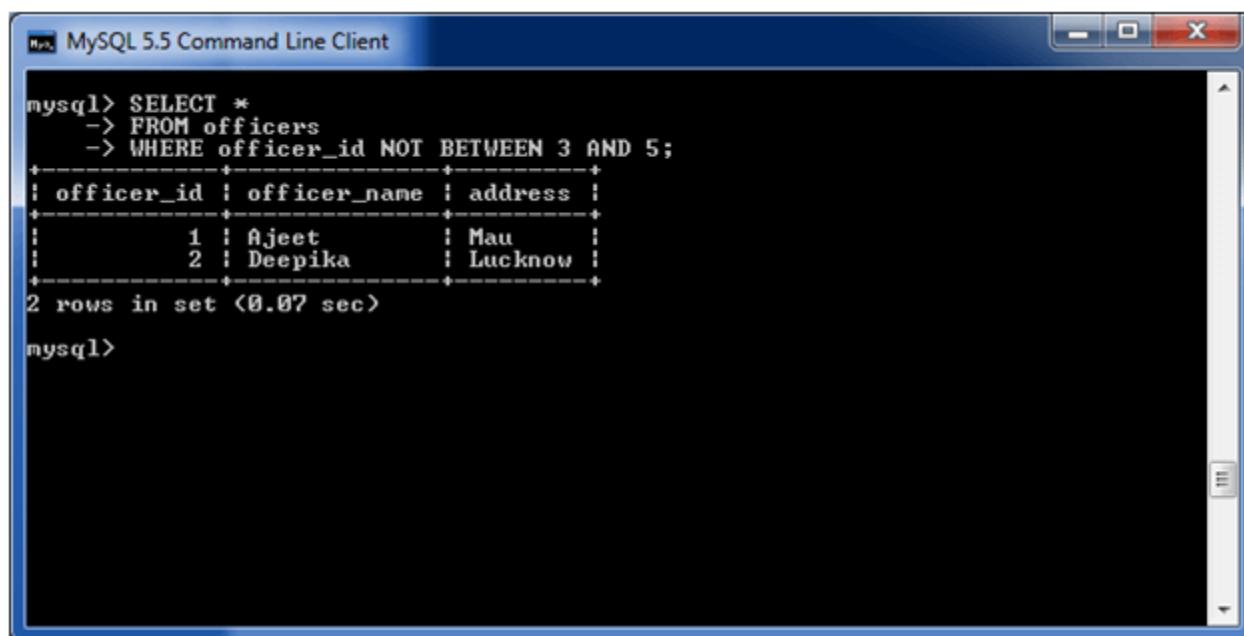
## MySQL NOT Operator with BETWEEN condition:

We are taking the same table "officer" for this operation also:

**Execute the following query:**

1. **SELECT \***
2. **FROM officers**
3. **WHERE officer\_id NOT BETWEEN 3 AND 5;**

**Output:**



MySQL 5.5 Command Line Client

```
+-----+-----+
| officer_id | officer_name | address |
+-----+-----+
| 1 | Ajeeet | Mau |
| 2 | Deepika | Lucknow |
+-----+
```

2 rows in set (0.07 sec)

mysql>

## MySQL Not Equal

MySQL Not Equal is an **inequality operator** that used for returning a set of rows after comparing two expressions that are not equal. The MySQL contains two types of Not Equal operator, which are (< >) and (! =).

### Difference Between (< >) and (! =) Operator

The Not Equal operators in MySQL works the same to perform an inequality test between two expressions. They always give the same result. However, they contain one difference that "< >" follows the **ISO standard** whereas "!=<" does not follow ISO standard.

#### Example 1

Let us create a table student to understand how Not Equal operator works in MySQL. Suppose the "students" table contains the following data:

**Table: students**

student_id	stud_fname	stud_lname	city
1	Devine	Putin	France
2	Michael	Clark	Australiya
3	Ethon	Miller	England
4	Mark	Boucher	South Africa
5	James	Anderson	England
6	Barack	Obama	US

If you want to get the student details who do not belong to **England**, then you need to execute the following statement:

1. **SELECT \* FROM** students **WHERE** city <> "England";

OR,

1. **SELECT \* FROM** students **WHERE** city != "England";

After successful execution of the above queries, we will get the same output as below:

student_id	stud_fname	stud_lname	city
1	Devine	Putin	France
2	Michael	Clark	Australiya
4	Mark	Boucher	South Africa
6	Barack	Obama	US

### Example 2

In this example, we are going to understand how Not Equal operator works with **Group By** clause. We can use the Group By clause for grouping rows that have the same data. If we want to get all customers who do not have **cellphone number** and **duplicate income** value, execute the following statement:

1. **SELECT \* FROM** customers
2. **JOIN** contacts **ON** customer\_id = contact\_id
3. **WHERE** cellphone <> "Null"
4. **GROUP BY** income;

We will get the following output:

customer_id	cust_name	occupation	income	qualification	contact_id	cellphone	homephone
1	John Miller	Developer	20000	Btech	1	6546645978	4565242557
2	Mark Robert	Enginneer	40000	Btech	2	8798634532	8652413954
3	Reyan Watson	Businessman	60000	MSc	3	8790744345	9874437396

### Example 3

In this example, we are going to understand how Not Equal operator works with a **JOIN** statement. Let us create a table "**contacts**" and "**customers**" in a database that contains the following data:

**Table: contacts**

contact_id	cellphone	homephone
1	6546645978	4565242557
2	8798634532	8652413954
3	8790744345	9874437396
4	7655654336	9934345363
5	Null	6786507067
6	Null	9086053684

**Table: customers**

customer_id	cust_name	occupation	income	qualification
1	John Miller	Developer	20000	Btech
2	Mark Robert	Enginneer	40000	Btech
3	Reyan Wa...	Scientists	60000	MSc
4	Shane Trump	Business...	10000	MBA
5	Adam Obama	Manager	80000	MBA
6	Ricky Pon...	Cricketer	200000	Btech
7	Adam Gilchrist	Cricketer	300000	PGDBA

If we want to get all the records from table customers and contacts where the cellphone is Null, execute the following statement that returns all customers who do not have a **cellphone** number:

1. **SELECT \* FROM** customers
2. **JOIN** contacts **ON** customer\_id = contact\_id

3. **WHERE** cellphone != "Null";

After successful execution, it will give the following output:

customer_id	cust_name	occupation	income	qualification	contact_id	cellphone	homephone
1	John Miller	Developer	20000	Btech	1	6546645978	4565242557
2	Mark Robert	Enginneer	40000	Btech	2	8798634532	8652413954
3	Reyan Watson	Scientists	60000	MSc	3	8790744345	9874437396
4	Shane Trump	Businessman	10000	MBA	4	7655654336	9934345363

#### Example 4

In this example, we are going to understand how the Not Equal operator works with multiple conditions in the WHERE clause. For example, we want to get the customer details where **income** is higher than **40000**, and **occupation** is not a **developer**. Execute the following statement to get the result:

1. **SELECT \* FROM** customers **Where** income>40000 and occupation<>"Developer";

After the successful execution of the above statement, we will get the following output.

customer_id	cust_name	occupation	income	qualification
3	Reyan Watson	Businessman	60000	MSc
5	Barack Obama	Leader	200000	PHD
6	Micheal Jordon	Footballer	500000	Btech

## MySQL IS NULL Condition

MySQL IS NULL condition is used to check if there is a NULL value in the expression. It is used with SELECT, INSERT, UPDATE and DELETE statement.

#### Syntax:

1. expression **IS** NULL

#### Parameter

**expression:** It specifies a value to test if it is NULL value.

Consider a table "officers" having the following data.

The screenshot shows the MySQL 5.5 Command Line Client window. The command entered is `SELECT * FROM officers;`. The resulting table has columns `officer_id`, `officer_name`, and `address`. The data consists of four rows:

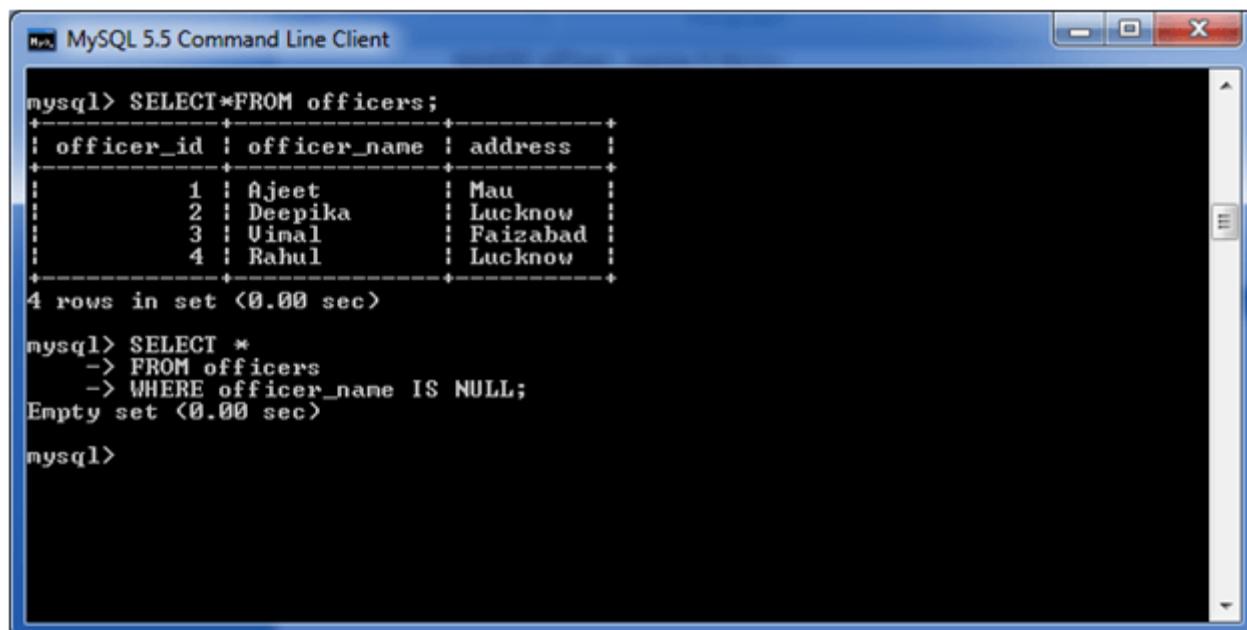
officer_id	officer_name	address
1	Ajeet	Mau
2	Deepika	Lucknow
3	Uimal	Faizabad
4	Rahul	Lucknow

At the bottom, it says `4 rows in set (0.03 sec)`.

**Execute the following query:**

1. **SELECT \***
2. **FROM** officers
3. **WHERE** officer\_name **IS** NULL;

**Output:**



```
MySQL 5.5 Command Line Client

mysql> SELECT * FROM officers;
+-----+-----+-----+
| officer_id | officer_name | address |
+-----+-----+-----+
| 1 | Ajeeet | Mau |
| 2 | Deepika | Lucknow |
| 3 | Uimal | Faizabad |
| 4 | Rahul | Lucknow |
+-----+-----+-----+
4 rows in set (0.00 sec)

mysql> SELECT *
-> FROM officers
-> WHERE officer_name IS NULL;
Empty set (0.00 sec)

mysql>
```

**Note:** Here, you are getting the empty result because there is no NULL value in officer\_name column.

## MySQL IS NOT NULL Condition

MySQL IS NOT NULL condition is used to check the NOT NULL value in the expression. It is used with SELECT, INSERT, UPDATE and DELETE statements.

### Syntax:

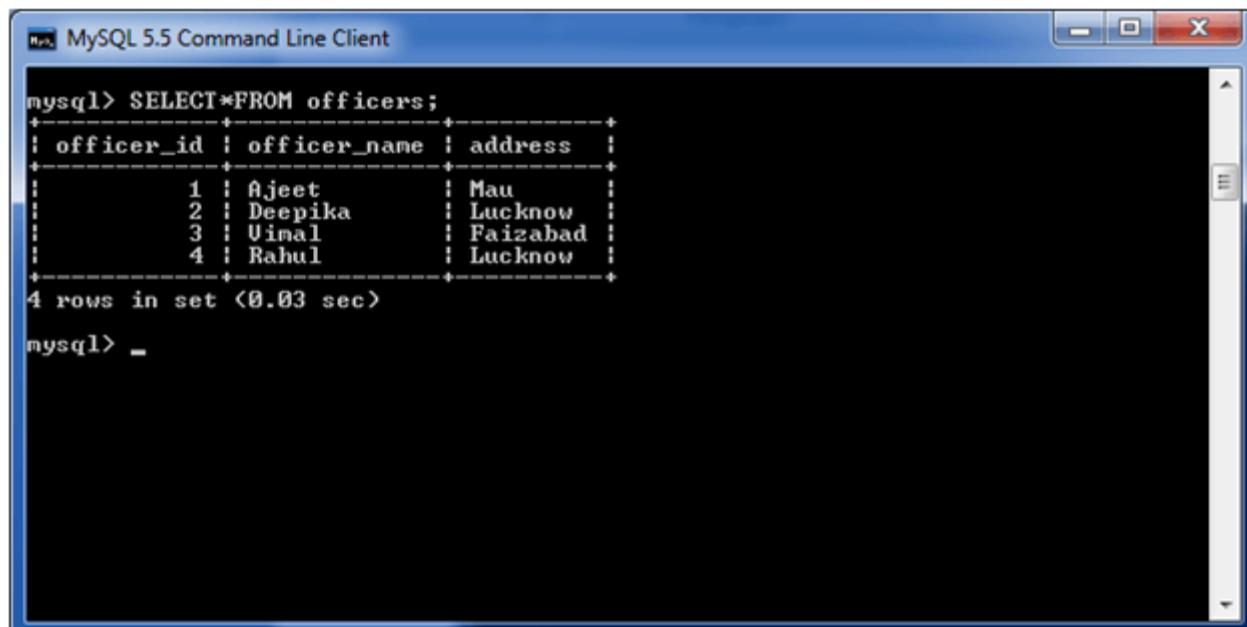
1. expression **IS NOT NULL**

### Parameter

**expression:** It specifies a value to test if it is not NULL value.

## MySQL IS NOT NULL Example

Consider a table "officers" having the following data.



```
MySQL 5.5 Command Line Client

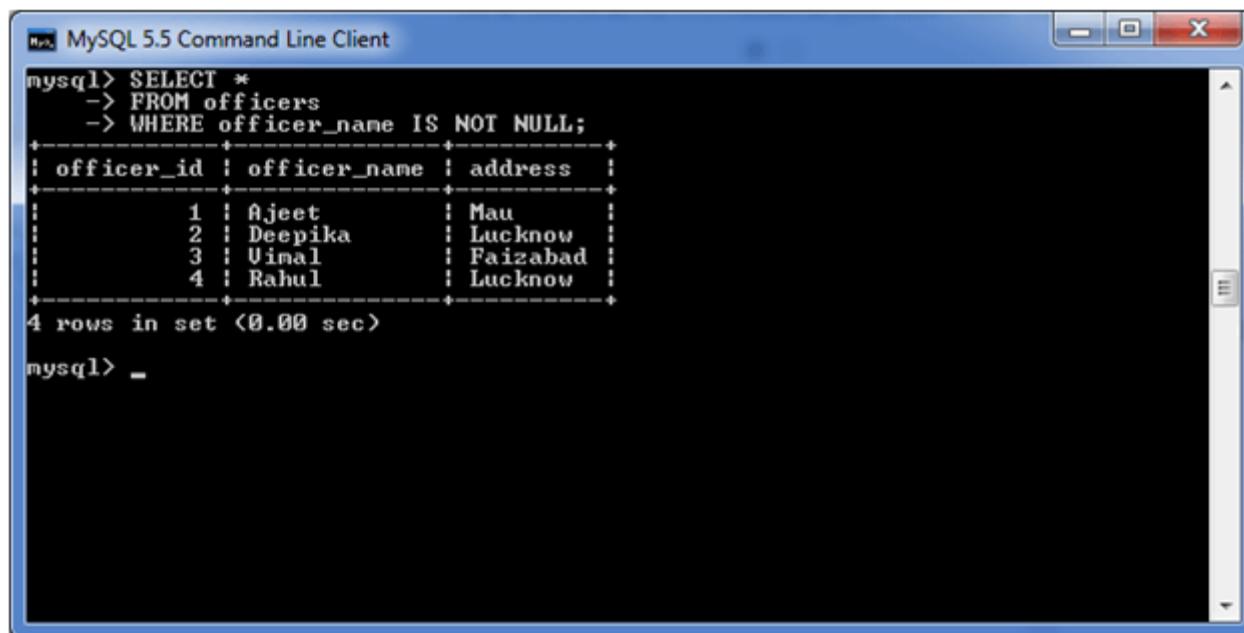
mysql> SELECT * FROM officers;
+-----+-----+-----+
| officer_id | officer_name | address |
+-----+-----+-----+
| 1 | Ajeeet | Mau |
| 2 | Deepika | Lucknow |
| 3 | Uimal | Faizabad |
| 4 | Rahul | Lucknow |
+-----+-----+-----+
4 rows in set (0.03 sec)

mysql> _
```

**Execute the following query:**

1. **SELECT \***
2. **FROM** officers
3. **WHERE** officer\_name **IS NOT NULL;**

### Output:



The screenshot shows the MySQL 5.5 Command Line Client window. The command entered is:

```
mysql> SELECT *  
    -> FROM officers  
    -> WHERE officer_name IS NOT NULL;
```

The resulting table is:

officer_id	officer_name	address
1	Ajeet	Mau
2	Deepika	Lucknow
3	Vimal	Faizabad
4	Rahul	Lucknow

4 rows in set (0.00 sec)

mysql> \_

**Note:** Here, you are getting the complete "officers" table as result because every value is NOT NULL in the table.

## MySQL BETWEEN Condition

The MYSQL BETWEEN condition specifies how to retrieve values from an expression within a specific range. It is used with SELECT, INSERT, UPDATE and DELETE statement.

### Syntax:

1. expression BETWEEN value1 AND value2;

### Parameters

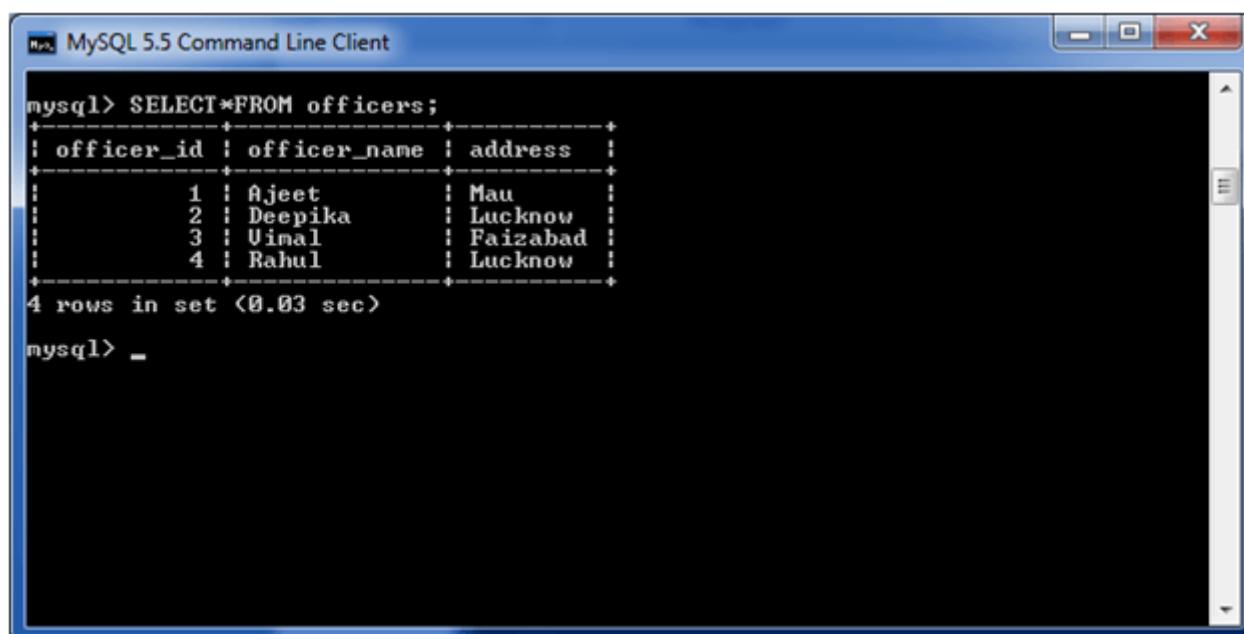
**expression:** It specifies a column.

**value1 and value2:** These values define an inclusive range that expression is compared to.

### Let's take some examples:

#### (i) MySQL BETWEEN condition with numeric value:

Consider a table "officers" having the following data.



The screenshot shows the MySQL 5.5 Command Line Client window. The command entered is:

```
mysql> SELECT * FROM officers;
```

The resulting table is:

officer_id	officer_name	address
1	Ajeet	Mau
2	Deepika	Lucknow
3	Vimal	Faizabad
4	Rahul	Lucknow

4 rows in set (0.03 sec)

mysql> \_

### Execute the following query:

1. **SELECT \***
2. **FROM officers**
3. **WHERE officer\_id BETWEEN 1 AND 3;**

### Output:

```
MySQL 5.5 Command Line Client

mysql> SELECT *  
    -> FROM officers  
    -> WHERE officer_id BETWEEN 1 AND 3;  
+-----+-----+-----+  
| officer_id | officer_name | address |  
+-----+-----+-----+  
| 1 | Ajeet | Mau |  
| 2 | Deepika | Lucknow |  
| 3 | Uimal | Faizabad |  
+-----+-----+-----+  
3 rows in set (0.06 sec)  
mysql>
```

**Note:** In the above example, you can see that only three rows are returned between 1 and 3.

## (ii) MySQL BETWEEN condition with date:

MySQL BETWEEN condition also facilitates you to retrieve records according to date.

**See this example:**

Consider a table "employees", having the following data.

```
MySQL 5.5 Command Line Client

mysql> SELECT * FROM employees;  
+-----+-----+-----+-----+  
| emp_id | emp_name | working_date | working_hours |  
+-----+-----+-----+-----+  
| 1 | Ajeet | 2015-01-24 | 12 |  
| 2 | Ayan | 2015-01-24 | 10 |  
| 3 | Milan | 2015-01-24 | 9 |  
| 4 | Ruchi | 2015-01-24 | 6 |  
| 1 | Ajeet | 2015-01-25 | 12 |  
| 2 | Ayan | 2015-01-25 | 10 |  
| 4 | Ruchi | 2015-01-25 | 6 |  
| 3 | Milan | 2015-01-25 | 9 |  
| 1 | Ajeet | 2015-01-26 | 12 |  
| 3 | Milan | 2015-01-26 | 9 |  
+-----+-----+-----+-----+  
10 rows in set (0.01 sec)  
mysql>
```

**Execute the following query:**

1. **SELECT \***
2. **FROM** employees
3. **WHERE** working\_date BETWEEN **CAST ('2015-01-24' AS DATE)** AND **CAST ('2015-01-25' AS DATE)**;

**Output:**

```
MySQL 5.5 Command Line Client

mysql> SELECT *  
    -> FROM employees  
    -> WHERE working_date BETWEEN CAST('2015-01-24' AS DATE) AND CAST('2015-01-25' AS DATE);  
+-----+-----+-----+-----+  
| emp_id | emp_name | working_date | working_hours |  
+-----+-----+-----+-----+  
| 1 | Ajeet | 2015-01-24 | 12 |  
| 2 | Ayan | 2015-01-24 | 10 |  
| 3 | Milan | 2015-01-24 | 9 |  
| 4 | Ruchi | 2015-01-24 | 6 |  
| 1 | Ajeet | 2015-01-25 | 12 |  
| 2 | Ayan | 2015-01-25 | 10 |  
| 4 | Ruchi | 2015-01-25 | 6 |  
| 3 | Milan | 2015-01-25 | 9 |  
+-----+-----+-----+-----+  
8 rows in set (0.14 sec)  
mysql>
```

**Note:** In the above example you can see that only data between specific dates are shown.

## MySQL Join

# MySQL JOINS

MySQL JOINS are used with SELECT statement. It is used to retrieve data from multiple tables. It is performed whenever you need to fetch records from two or more tables.

There are three types of MySQL joins:

- o MySQL INNER JOIN (or sometimes called simple join)
- o MySQL LEFT OUTER JOIN (or sometimes called LEFT JOIN)
- o MySQL RIGHT OUTER JOIN (or sometimes called RIGHT JOIN)

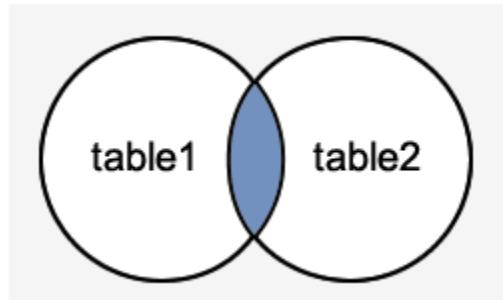
## MySQL Inner JOIN (Simple Join)

The MySQL INNER JOIN is used to return all rows from multiple tables where the join condition is satisfied. It is the most common type of join.

### Syntax:

1. **SELECT** columns
2. **FROM** table1
3. **INNER JOIN** table2
4. **ON** table1.**column** = table2.**column**;

### Image representation:



### Let's take an example:

Consider two tables "officers" and "students", having the following data.

```
MySQL 5.5 Command Line Client
+-----+
4 rows in set <0.00 sec>

mysql> SELECT*FROM officers;
+-----+
| officer_id | officer_name | address |
+-----+
| 1 | Ajeeet | Mau |
| 2 | Deepika | Lucknow |
| 3 | Uimal | Faizabad |
| 4 | Rahul | Lucknow |
+-----+
4 rows in set <0.00 sec>

mysql> SELECT*FROM students;
+-----+
| student_id | student_name | course_name |
+-----+
| 1 | Aryan | Java |
| 2 | Rohini | Hadoop |
| 3 | Lallu | MongoDB |
+-----+
3 rows in set <0.00 sec>

mysql>
```

### Execute the following query:

1. **SELECT** officers.officer\_name, officers.address, students.course\_name
2. **FROM** officers
3. **INNER JOIN** students
4. **ON** officers.officer\_id = students.student\_id;

### Output:

```

MySQL 5.5 Command Line Client
mysql> SELECT officers.officer_name, officers.address, students.course_name
-> FROM officers
-> INNER JOIN students
-> ON officers.officer_id = students.student_id;
+-----+-----+-----+
| officer_name | address | course_name |
+-----+-----+-----+
| Ajeet        | Mau     | Java       |
| Deepika      | Lucknow | Hadoop    |
| Uimal         | Faizabad| MongoDB   |
+-----+-----+-----+
3 rows in set (0.00 sec)

mysql> 

```

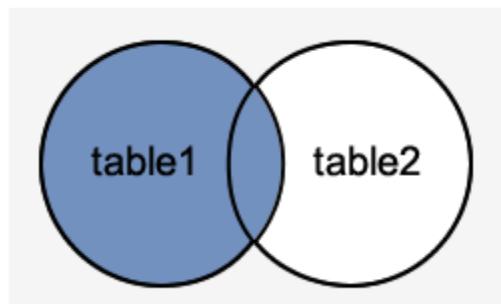
## MySQL Left Outer Join

The LEFT OUTER JOIN returns all rows from the left hand table specified in the ON condition and only those rows from the other table where the join condition is fulfilled.

### Syntax:

1. **SELECT** columns
2. **FROM** table1
3. **LEFT [OUTER] JOIN** table2
4. **ON** table1.**column** = table2.**column**;

### Image representation:



### Let's take an example:

Consider two tables "officers" and "students", having the following data.

```

MySQL 5.5 Command Line Client
+-----+
4 rows in set (0.00 sec)

mysql> SELECT * FROM officers;
+-----+-----+-----+
| officer_id | officer_name | address |
+-----+-----+-----+
| 1          | Ajeet        | Mau     |
| 2          | Deepika      | Lucknow |
| 3          | Uimal         | Faizabad|
| 4          | Rahul         | Lucknow |
+-----+-----+-----+
4 rows in set (0.00 sec)

mysql> SELECT * FROM students;
+-----+-----+-----+
| student_id | student_name | course_name |
+-----+-----+-----+
| 1          | Aryan        | Java       |
| 2          | Rohini       | Hadoop    |
| 3          | Lallu         | MongoDB   |
+-----+-----+-----+
3 rows in set (0.00 sec)

mysql> 

```

### Execute the following query:

1. **SELECT** officers.officer\_name, officers.address, students.course\_name
2. **FROM** officers
3. **LEFT JOIN** students
4. **ON** officers.officer\_id = students.student\_id;

### Output:

```

MySQL 5.5 Command Line Client
mysql> SELECT officers.officer_name, officers.address, students.course_name
-> FROM officers
-> LEFT JOIN students
-> ON officers.officer_id = students.student_id;
+-----+-----+-----+
| officer_name | address | course_name |
+-----+-----+-----+
| Ajeet        | Mau     | Java       |
| Deepika      | Lucknow | Hadoop     |
| Uimal         | Faizabad| MongoDB   |
| Rahul         | Lucknow | NULL       |
+-----+-----+-----+
4 rows in set (0.01 sec)

mysql>

```

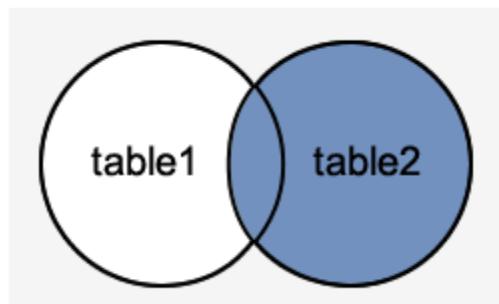
## MySQL Right Outer Join

The MySQL Right Outer Join returns all rows from the RIGHT-hand table specified in the ON condition and only those rows from the other table where he join condition is fulfilled.

### Syntax:

1. **SELECT** columns
2. **FROM** table1
3. **RIGHT [OUTER] JOIN** table2
4. **ON** table1.**column** = table2.**column**;

### Image representation:



### Let's take an example:

Consider two tables "officers" and "students", having the following data.

```

MySQL 5.5 Command Line Client
+-----+-----+
4 rows in set (0.00 sec)

mysql> SELECT * FROM officers;
+-----+-----+-----+
| officer_id | officer_name | address |
+-----+-----+-----+
| 1          | Ajeet        | Mau     |
| 2          | Deepika      | Lucknow |
| 3          | Uimal         | Faizabad|
| 4          | Rahul         | Lucknow |
+-----+-----+-----+
4 rows in set (0.00 sec)

mysql> SELECT * FROM students;
+-----+-----+-----+
| student_id | student_name | course_name |
+-----+-----+-----+
| 1          | Aryan        | Java       |
| 2          | Rohini       | Hadoop     |
| 3          | Lallu         | MongoDB   |
+-----+-----+-----+
3 rows in set (0.00 sec)

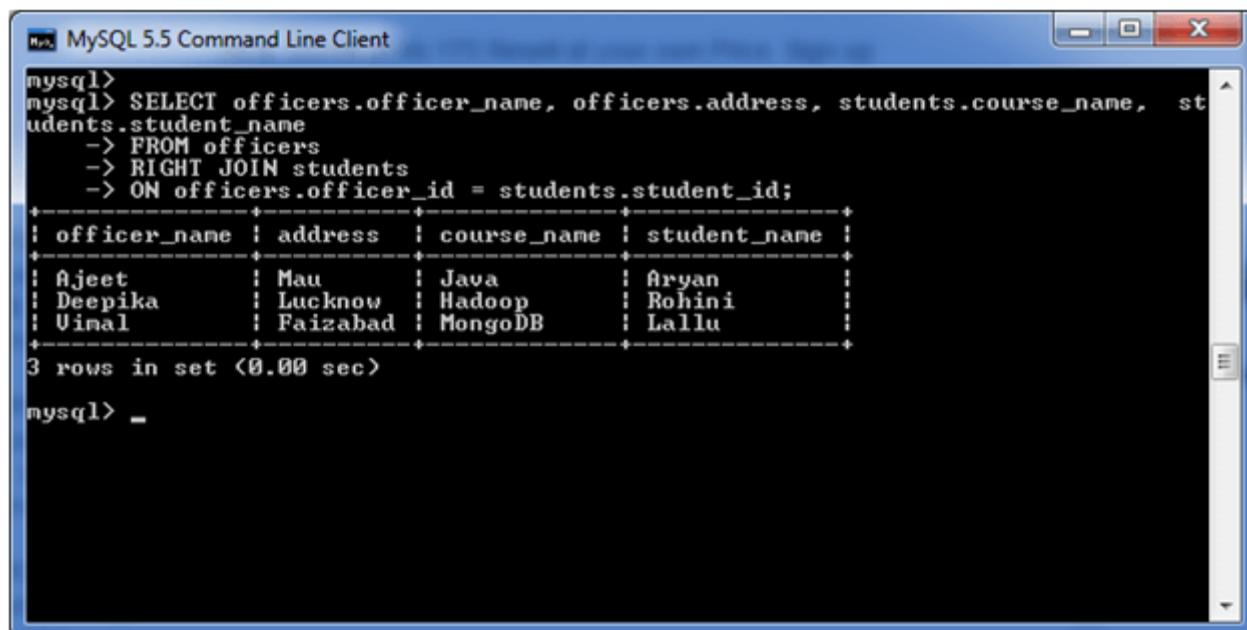
mysql>

```

### Execute the following query:

1. **SELECT** officers.officer\_name, officers.address, students.course\_name, students.student\_name
2. **FROM** officers
3. **RIGHT JOIN** students
4. **ON** officers.officer\_id = students.student\_id;

### Output:



The screenshot shows the MySQL 5.5 Command Line Client window. The command entered is:

```
mysql> SELECT officers.officer_name, officers.address, students.course_name, students.student_name
    FROM officers
    RIGHT JOIN students
    ON officers.officer_id = students.student_id;
```

The resulting table is:

officer_name	address	course_name	student_name
Ajeet	Mau	Java	Aryan
Deepika	Lucknow	Hadoop	Rohini
Uimal	Faizabad	MongoDB	Lallu

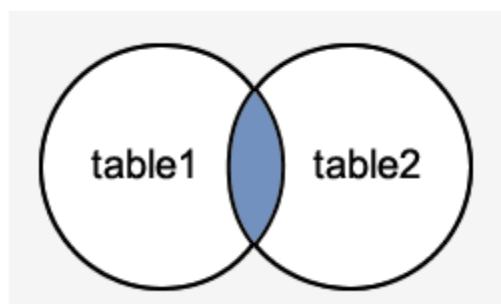
3 rows in set (0.00 sec)

mysql> \_

## MySQL Inner Join

The MySQL Inner Join is used to returns only those results from the tables that **match** the specified condition and hides other rows and columns. MySQL assumes it as a default Join, so it is optional to use the Inner Join keyword with the query.

We can understand it with the following visual representation where Inner Joins returns only the matching results from table1 and table2:



### MySQL Inner Join Syntax:

The Inner Join keyword is used with the [SELECT statement](#) and must be written after the FROM clause. The following syntax explains it more clearly:

1. **SELECT** columns
2. **FROM** table1
3. **INNER JOIN** table2 **ON** condition1
4. **INNER JOIN** table3 **ON** condition2
5. ...;

In this syntax, we first have to select the column list, then specify the table name that will be joined to the main table, appears in the Inner Join (table1, table2), and finally, provide the condition after the ON keyword. The Join condition returns the matching rows between the tables specifies in the Inner clause.

## MySQL Inner Join Example

Let us first create two tables "students" and "technologies" that contains the following data:

**Table: student**

student_id	stud_fname	stud_lname	city
1	Devine	Putin	France
2	Michael	Clark	Australiya
3	Ethon	Miller	England
4	Mark	Strauss	America

**Table: technologies**

student_id	tech_id	inst_name	technology
1	1	Java Training Inst	Java
2	2	Chroma Campus	Angular
3	3	CETPA Infotech	Big Data
4	4	Aptron	IOS

To select records from both tables, execute the following query:

1. **SELECT** students.stud\_fname, students.stud\_lname, students.city, technologies.technology
2. **FROM** students
3. **INNER JOIN** technologies
4. **ON** students.student\_id = technologies.tech\_id;

After successful execution of the query, it will give the following output:

stud_fname	stud_lname	city	technology
Devine	Putin	France	Java
Michael	Clark	Australiya	Angular
Ethon	Miller	England	Big Data
Mark	Strauss	America	IOS

## MySQL Inner Join with Group By Clause

The Inner Join can also be used with the GROUP BY clause. The following statement returns student id, technology name, city, and institute name using the Inner Join clause with the GROUP BY clause.

1. **SELECT** students.student\_id, technologies.inst\_name, students.city, technologies.technology
2. **FROM** students
3. **INNER JOIN** technologies
4. **ON** students.student\_id = technologies.tech\_id **GROUP BY** inst\_name;

The above statement will give the following output:

student_id	inst_name	city	technology
1	Java Training Inst.	France	Java
2	Croma Campus	Australiya	Angular
3	CETPA Infotech	England	Big Data
4	Aptron	America	IOS

## MySQL Inner Join with USING clause

Sometimes, the name of the columns is the same in both the tables. In that case, we can use a USING keyword to access the records. The following query explains it more clearly:

1. **SELECT** student\_id, inst\_name, city, technology
2. **FROM** students
3. **INNER JOIN** technologies
4. **USING** (student\_id);

It will give the following output:

student_id	inst_name	city	technology
1	Java Training Inst	France	Java
2	Chroma Campus	Australiya	Angular
3	CETPA Infotech	England	Big Data
4	Aptron	America	IOS

## Inner Join with WHERE Clause

The WHERE clause enables you to return the **filter** result. The following example illustrates this clause with Inner Join:

1. **SELECT** tech\_id, inst\_name, city, technology
2. **FROM** students
3. **INNER JOIN** technologies
4. **USING** (student\_id) **WHERE** technology = "Java";

This statement gives the below result:

tech_id	inst_name	city	technology
1	Java Training Inst	France	Java

## MySQL Inner Join Multiple Tables

We have already created two tables named **students** and **technologies**. Let us create one more table and name it as a contact.

college_id	cellphone	homephone
1	5465465645	4576787687
2	4987246464	5795645568
3	8907654334	8654784126
4	8973454904	9865321475

Execute the following statement to join the three table students, technologies, and contact:

1. **SELECT** student\_id, inst\_name, city, technology, cellphone
2. **FROM** students
3. **INNER JOIN** technologies USING (student\_id)
4. **INNER JOIN** contact **ORDER BY** student\_id;

After successful execution of the above query, it will give the following output:

student_id	inst_name	city	technology	cellphone
1	Java Training Inst	France	Java	4987246464
1	Java Training Inst	France	Java	8907654334
1	Java Training Inst	France	Java	8973454904
2	Chroma Campus	Australiya	Angular	5465465645
2	Chroma Campus	Australiya	Angular	4987246464
2	Chroma Campus	Australiya	Angular	8907654334
2	Chroma Campus	Australiya	Angular	8973454904
3	CETPA Infotech	England	Big Data	5465465645
3	CETPA Infotech	England	Big Data	4987246464
3	CETPA Infotech	England	Big Data	8907654334
3	CETPA Infotech	England	Big Data	8973454904
4	Aptron	America	IOS	5465465645
4	Aptron	America	IOS	4987246464
4	Aptron	America	IOS	8907654334

## MySQL Inner Join using Operators

[MySQL](#) allows many operators that can be used with Inner Join, such as greater than (>), less than (<), equal (=), not equal (=), etc. The following query returns the result whose income is in the range of 20000 to 80000:

1. **SELECT** emp\_id, designation, income, qualification
2. **FROM** employee
3. **INNER JOIN** customer
4. **WHERE** income>20000 and income<80000;

This will give the following output:

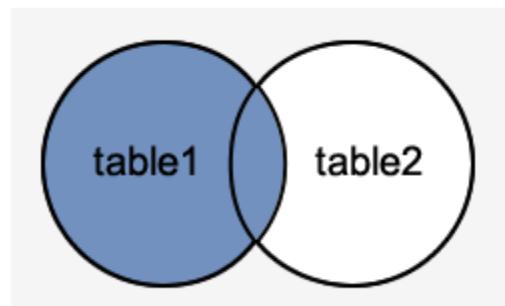
emp_id	designation	income	qualification
1	Engineer	60000	MSc
2	Manager	40000	Btech
2	Manager	60000	MSc
3	Scientist	40000	Btech
3	Scientist	60000	MSc
2	BDE	40000	Btech
2	BDE	60000	MSc
1	Developer	40000	Btech
1	Developer	60000	MSc
4	Receptionist	40000	Btech
4	Receptionist	60000	MSc
3	Engineer	40000	Btech
3	Engineer	60000	MSc
4	Clerk	40000	Btech
4	Clerk	60000	MSc

## MySQL LEFT JOIN

The Left Join in MySQL is used to query records from multiple tables. This clause is similar to the Inner Join clause that can be used with a SELECT statement immediately after the FROM keyword. When we use the Left Join clause, it will return all the records from the first (left-side) table, even no matching records found from the second (right side) table. If it will not find any matches record from the right side table, then returns null.

In other words, the Left Join clause returns all the rows from the left table and matched records from the right table or returns Null if no matching record found. This Join can also be called a **Left Outer Join** clause. So, Outer is the optional keyword to use with Left Join.

We can understand it with the following visual representation where Left Joins returns all records from the left-hand table and only the matching records from the right side table:



## MySQL LEFT JOIN Syntax

The following syntax explains the Left Join clause to join the two or more tables:

1. **SELECT** columns
2. **FROM** table1
3. **LEFT [OUTER] JOIN** table2
4. **ON** Join\_Condition;

In the above syntax, **table1** is the left-hand table, and **table2** is the right-hand table. This clause returns all records from table1 and matched records from table2 based on the specified **join condition**.

## MySQL LEFT JOIN Example

Let us take some examples to understand the working of Left Join or Left Outer Join clause:

### LEFT JOIN clause for joining two tables

Here, we are going to create two tables "**customers**" and "**orders**" that contains the following data:

**Table: customers**

customer_id	cust_name	occupation	income	qualification
1	John Miller	Developer	20000	Btech
2	Mark Robert	Enginneer	40000	Btech
3	Reyan Watson	Scientists	60000	MSc
4	Shane Trump	Businessman	10000	MBA
5	Adam Obama	Manager	80000	MBA
6	Ricky Ponting	Cricketer	200000	Btech

**Table: orders**

order_id	date	customer_id	price
1001	2020-03-20	2	3000
1002	2020-02-15	4	2500
1003	2020-01-31	5	5000
1004	2020-03-10	2	1500
1005	2020-02-20	1	4500

To select records from both tables, execute the following query:

1. **SELECT** customers.customer\_id, cust\_name, price, **date**
2. **FROM** customers
3. **LEFT JOIN** orders **ON** customers.customer\_id = orders.customer\_id;

After successful execution of the query, it will give the following output:

customer_id	cust_name	price	date
2	Mark Robert	3000	2020-03-20
4	Shane Trump	2500	2020-02-15
5	Adam Obama	5000	2020-01-31
2	Mark Robert	1500	2020-03-10
1	John Miller	4500	2020-02-20

## MySQL LEFT JOIN with USING Clause

The table customers and orders have the same column name, which is customer\_id. In that case, MySQL Left Join can also be used with the USING clause to access the records. The following statement returns customer id, customer name, occupation, price, and date using the Left Join clause with the USING keyword.

1. **SELECT** customer\_id, cust\_name, occupation, price, **date**
2. **FROM** customers
3. **LEFT JOIN** orders **USING**(customer\_id);

The above statement will give the following output:

customer_id	cust_name	occupation	price	date
2	Mark Robert	Enginneer	3000	2020-03-20
4	Shane Trump	Businessman	2500	2020-02-15
5	Adam Obama	Manager	5000	2020-01-31
2	Mark Robert	Enginneer	1500	2020-03-10
1	John Miller	Developer	4500	2020-02-20

### MySQL LEFT JOIN with Group By Clause

The Left Join can also be used with the GROUP BY clause. The following statement returns customer id, customer name, qualification, price, and date using the Left Join clause with the GROUP BY clause.

1. **SELECT** customers.customer\_id, cust\_name, qualification, price, **date**
2. **FROM** customers
3. **LEFT JOIN** orders **ON** customers.customer\_id = orders.customer\_id
4. **GROUP BY** price;

The above statement will give the following output:

customer_id	cust_name	qualification	price	date
2	Mark Robert	Btech	3000	2020-03-20
4	Shane Trump	MBA	2500	2020-02-15
5	Adam Obama	MBA	5000	2020-01-31
2	Mark Robert	Btech	1500	2020-03-10
1	John Miller	Btech	4500	2020-02-20

### LEFT JOIN with WHERE Clause

The WHERE clause is used to return the **filter** result from the table. The following example illustrates this with the Left Join clause:

1. **SELECT** customer\_id, cust\_name, occupation, price, **date**
2. **FROM** customers
3. **LEFT JOIN** orders
4. **USING**(customer\_id) **WHERE** price > 2500;

This statement gives the below result:

customer_id	cust_name	occupation	price	date
2	Mark Robert	Enginneer	3000	2020-03-20
5	Adam Obama	Manager	5000	2020-01-31
1	John Miller	Developer	4500	2020-02-20

### MySQL LEFT JOIN Multiple Tables

We have already created two tables named "**customers**" and "**orders**". Let us create one more table and name it as "**contacts**" that contains the following data:

contact_id	cellphone	homephone
1	6546645978	4565242557
2	8798634532	8652413954
3	8790744345	9874437396
4	7655654336	9934345363

Execute the following statement to join the three table customers, orders, and contacts:

1. **SELECT** customers.customer\_id, cust\_name, order\_id, price, cellphone
2. **FROM** customers
3. **LEFT JOIN** contacts **ON** customer\_id = contact\_id
4. **LEFT JOIN** orders **ON** customers.customer\_id = orders.customer\_id **ORDER BY** income;

After successful execution of the above query, it will give the following output:

customer_id	cust_name	order_id	price	cellphone
4	Shane Trump	1002	2500	7655654336
1	John Miller	1005	4500	6546645978
6	Ricky Ponting	NULL	NULL	NULL
2	Mark Robert	1001	3000	8798634532
2	Mark Robert	1004	1500	8798634532
3	Reyan Watson	NULL	NULL	8790744345
5	Adam Obama	1003	5000	NULL

## Use of LEFT JOIN clause to get unmatched records

The LEFT JOIN clause is also useful in such a case when we want to get records in the table that does not contain any matching rows of data from another table.

We can understand it with the following example that uses the LEFT JOIN clause to find a customer who has no cellphone number:

1. **SELECT** customer\_id, cust\_name, cellphone, homephone
2. **FROM** customers
3. **LEFT JOIN** contacts **ON** customer\_id = contact\_id
4. **WHERE** cellphone **IS NULL** ;

The above statement returns the following output:

customer_id	cust_name	cellphone	homephone
5	Adam Obama	NULL	6786507067
6	Ricky Ponting	NULL	9086053684

## Difference between WHERE and ON clause in MySQL LEFT JOIN

In the LEFT Join, the condition WHERE and ON gives a different result. We can see the following queries to understand their differences:

### WHERE Clause

1. **SELECT** cust\_name, occupation, order\_id, price, **date**
2. **FROM** customers
3. **LEFT JOIN** orders
4. **USING**(customer\_id) **WHERE** price=2500;

It will give the following output that returns:

cust_name	occupation	order_id	price	date
Shane Trump	Businessman	1002	2500	2020-02-15

### ON Clause

1. **SELECT** cust\_name, occupation, order\_id, price, **date**
2. **FROM** customers **LEFT JOIN** orders **ON** price=2500;

It will give the following output:

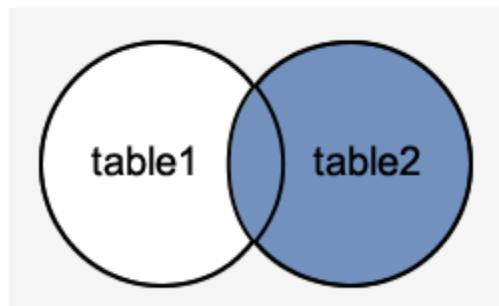
cust_name	occupation	order_id	price	date
John Miller	Developer	1002	2500	2020-02-15
Mark Robert	Engineer	1002	2500	2020-02-15
Reyan Watson	Scientists	1002	2500	2020-02-15
Shane Trump	Businessman	1002	2500	2020-02-15
Adam Obama	Manager	1002	2500	2020-02-15
Ricky Ponting	Cricketer	1002	2500	2020-02-15

**NOTE:** The WHERE and ON condition in the Inner Join clause always returns equivalent results.

## MySQL RIGHT JOIN

The Right Join is used to joins two or more tables and returns all rows from the right-hand table, and only those results from the other table that fulfilled the join condition. If it finds unmatched records from the left side table, it returns Null value. It is similar to the Left Join, except it gives the reverse result of the join tables. It is also known as Right Outer Join. So, Outer is the optional clause used with the Right Join.

We can understand it with the following visual representation where Right Outer Join returns all records from the left-hand table and only the matching records from the other table:



## RIGHT JOIN Syntax

The following are the syntax of Right Join that joins tables **Table1** and **Table2**:

1. **SELECT** column\_list
2. **FROM** Table1
3. **RIGHT** [OUTER] JOIN Table2
4. **ON** join\_condition;

**NOTE:** In the Right Join, if the tables contain the same column name, then **ON** and **USING** clause give the equivalent results.

Let us see how Right Join works.

This Join starts selecting the columns from the right-hand table and matches each record of this table from the left table. If both records fulfill the given join condition, it combines all columns in a new row set that will be returned as output. If the rows of the right-side table do not find any matching rows from the left table, it combines those rows from the right-side table with Null values. It means, the Right Join returns all data from the right-side table whether it matches the rows from the left table or not.

## MySQL RIGHT JOIN Examples

Let us take some examples to understand the working of Right Join clause:

### RIGHT JOIN clause for joining two tables

Here, we are going to create two tables "**customers**" and "**orders**" that contains the following data:

**Table: customers**

customer_id	cust_name	occupation	income	qualification
1	John Miller	Developer	20000	Btech
2	Mark Robert	Enginneer	40000	Btech
3	Reyan Watson	Scientists	60000	MSc
4	Shane Trump	Businessman	10000	MBA
5	Adam Obama	Manager	80000	MBA
6	Ricky Ponting	Cricketer	200000	Btech

**Table: orders**

order_id	date	customer_id	price
1001	2020-03-20	2	3000
1002	2020-02-15	4	2500
1003	2020-01-31	5	5000
1004	2020-03-10	2	1500
1005	2020-02-20	1	4500

To select records from both tables using RIGHT JOIN, execute the following query:

1. **SELECT** customers.customer\_id, cust\_name, price, **date**
2. **FROM** customers
3. **RIGHT JOIN** orders **ON** customers.customer\_id = orders.customer\_id
4. **ORDER BY** customer\_id;

OR,

1. **SELECT** customers.customer\_id, cust\_name, price, **date**
2. **FROM** customers
3. **RIGHT JOIN** orders **USING**(customer\_id)

4. **ORDER BY** customer\_id;

After successful execution of the above queries, it will give the equivalent output:

customer_id	cust_name	price	date
1	John Miller	4500	2020-02-20
2	Mark Robert	3000	2020-03-20
2	Mark Robert	1500	2020-03-10
4	Shane Trump	2500	2020-02-15
5	Adam Obama	5000	2020-01-31

### RIGHT JOIN with WHERE Clause

MySQL uses the **WHERE clause** to provide the **filter** result from the table. The following example illustrates this with the Right Join clause:

1. **SELECT \* FROM** customers
2. **RIGHT JOIN** orders **USING**(customer\_id)
3. **WHERE** price>2500 AND price<5000;

This statement gives the below result:

customer_id	order_id	date	price	cust_name	occupation	income	qualification
2	1001	2020-03-20	3000	Mark Robert	Enginneer	40000	Btech
1	1005	2020-02-20	4500	John Miller	Developer	20000	Btech

### MySQL RIGHT JOIN Multiple Tables

We have already created two tables, named "**customers**" and "**orders**". Let us create one more table and name it as a "**contacts**" that contains the following data:

contact_id	cellphone	homephone
1	6546645978	4565242557
2	8798634532	8652413954
3	8790744345	9874437396
4	7655654336	9934345363

Execute the following statement to join the three table customers, orders, and contacts:

1. **SELECT** customers.customer\_id, cust\_name, order\_id, price, cellphone
2. **FROM** customers
3. **RIGHT JOIN** contacts **ON** customer\_id = contact\_id
4. **RIGHT JOIN** orders **ON** customers.customer\_id = orders.customer\_id **ORDER BY** order\_id;

After successful execution of the above query, it will give the following output:

customer_id	cust_name	order_id	price	cellphone
2	Mark Robert	1001	3000	8798634532
4	Shane Trump	1002	2500	7655654336
5	Adam Obama	1003	5000	NULL
2	Mark Robert	1004	1500	8798634532
1	John Miller	1005	4500	6546645978

### Use of RIGHT JOIN clause to get unmatched records

The Right Join clause is also useful in such a case when we want to get records in the table that does not contain any matching rows of data from another table.

We can understand it with the following example that uses the RIGHT JOIN clause to find a customer who has no **cellphone** number:

1. **SELECT** customer\_id, cust\_name, cellphone, homephone
2. **FROM** customers
3. **RIGHT JOIN** contacts **ON** customer\_id = contact\_id
4. **WHERE** cellphone **IS** NULL
5. **ORDER BY** cellphone;

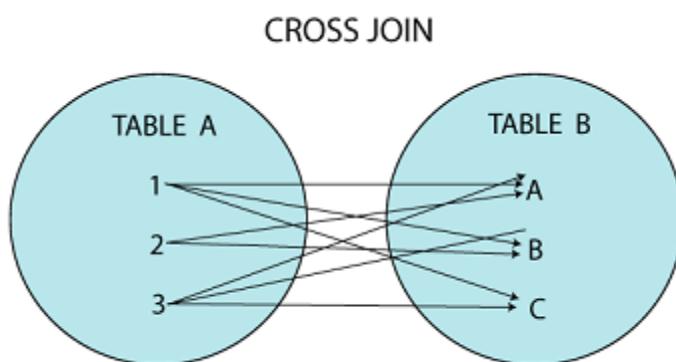
The above statement returns the following output:

customer_id	cust_name	cellphone	homephone
5	Adam Obama	NULL	6786507067
6	Rincky Ponting	NULL	9086053684

## MySQL CROSS JOIN

MySQL CROSS JOIN is used to combine all possibilities of the two or more tables and returns the result that contains every row from all contributing tables. The CROSS JOIN is also known as CARTESIAN JOIN, which provides the Cartesian product of all associated tables. The Cartesian product can be explained as all rows present in the first table multiplied by all rows present in the second table. It is similar to the Inner Join, where the join condition is not available with this clause.

We can understand it with the following visual representation where CROSS JOIN returns all the records from table1 and table2, and each row is the combination of rows of both tables.



## MySQL CROSS JOIN Syntax

The CROSS JOIN keyword is always used with the SELECT statement and must be written after the FROM clause. The following syntax fetches all records from both joining tables:

1. **SELECT column-lists**
2. **FROM table1**
3. **CROSS JOIN table2;**

In the above syntax, the column-lists is the name of the column or field that you want to return and table1 and table2 is the table name from which you fetch the records.

## MySQL CROSS JOIN Example

Let us take some examples to understand the working of Left Join or Left Outer Join clause:

### CROSS JOIN clause for joining two tables

Here, we are going to create two tables "**customers**" and "**contacts**" that contains the following data:

**Table: customers**

customer_id	cust_name	occupation	income	qualification
1	John Miller	Developer	20000	Btech
2	Mark Robert	Enginneer	40000	Btech
3	Reyan Watson	Scientists	60000	MSc
4	Shane Trump	Businessman	10000	MBA
5	Adam Obama	Manager	80000	MBA
6	Rincky Ponting	Cricketer	200000	Btech

**Table: contacts**

contact_id	cellphone	homephone
1	6546645978	4565242557
2	8798634532	8652413954
3	8790744345	9874437396
4	7655654336	9934345363

To fetch all records from both tables, execute the following query:

1. **SELECT \***
2. **FROM customers**
3. **CROSS JOIN contacts;**

After successful execution of the query, it will give the following output:

customer_id	cust_name	occupation	income	qualification	contact_id	cellphone	homephone
1	John Miller	Developer	20000	Btech	1	6546645978	4565242557
1	John Miller	Developer	20000	Btech	2	8798634532	8652413954
1	John Miller	Developer	20000	Btech	3	8790744345	9874437396
1	John Miller	Developer	20000	Btech	4	7655654336	9934345363
1	John Miller	Developer	20000	Btech	5	NULL	6786507067
1	John Miller	Developer	20000	Btech	6	NULL	9086053684
2	Mark Robert	Enginneer	40000	Btech	1	6546645978	4565242557
2	Mark Robert	Enginneer	40000	Btech	2	8798634532	8652413954
2	Mark Robert	Enginneer	40000	Btech	3	8790744345	9874437396
2	Mark Robert	Enginneer	40000	Btech	4	7655654336	9934345363
2	Mark Robert	Enginneer	40000	Btech	5	NULL	6786507067
2	Mark Robert	Enginneer	40000	Btech	6	NULL	9086053684
3	Reyan Watson	Scientists	60000	MSc	1	6546645978	4565242557
3	Reyan Watson	Scientists	60000	MSc	2	8798634532	8652413954
3	Reyan Watson	Scientists	60000	MSc	3	8790744345	9874437396

When the CROSS JOIN statement executed, you will observe that it displays 42 rows. It means seven rows from customers table multiplies by the six rows from the contacts table.

**NOTE:** To avoid the result of repeated columns twice, it is recommended to use individual column names instead of SELECT \* statement.

## Ambiguous Columns problem in MySQL CROSS JOIN

Sometimes, we need to fetch the selected column records from multiple tables. These tables can contain some column names similar. In that case, MySQL CROSS JOIN statement throws an error: the column name is ambiguous. It means the name of the column is present in both tables, and MySQL gets confused about which column you want to display. The following examples explain it more clearly:

1. **SELECT** customer\_id, cust\_name, income, order\_id, price
2. **FROM** customer
3. CROSS JOIN orders;

The above CROSS JOIN throws an error as given in the image below:

Action Output			Message	Duration / Fetch
#	Time	Action		
1	16:40:45	SELECT customer_id, cust_name, income, order_id, price FRO...	Error Code: 1052. Column 'customer_id' in field list is ambiguous	0.000 sec

This problem can be resolved by using the table name before the column name. The above query can be re-written as:

1. **SELECT** customer.customer\_id, customer.cust\_name, customer.income, orders.order\_id, orders.price
2. **FROM** customer
3. CROSS JOIN orders;

After executing the above query, we will get the following output:

customer_id	cust_name	income	order_id	price
1	John Miller	20000	1001	3000
1	John Miller	20000	1002	2500
1	John Miller	20000	1003	5000
1	John Miller	20000	1004	1500
1	John Miller	20000	1005	4500
2	Mark Robert	40000	1001	3000
2	Mark Robert	40000	1002	2500
2	Mark Robert	40000	1003	5000
2	Mark Robert	40000	1004	1500

## LEFT JOIN with WHERE Clause

The WHERE clause is used to return the **filter** result from the table. The following example illustrates this with the CROSS JOIN clause:

1. **SELECT** customers.customer\_id, customers.cust\_name, customers.income, orders.order\_id, orders.price
2. **FROM** customers
3. CROSS JOIN orders
4. **USING**(customer\_id) **WHERE** price > 1500 AND price < 5000;

This statement gives the below result:

customer_id	cust_name	income	order_id	price
2	Mark Robert	40000	1001	3000
4	Shane Trump	10000	1002	2500
1	John Miller	20000	1005	4500

## MySQL CROSS JOIN Multiple Tables

We have already created two tables named "**customers**" and "**orders**". Let us create one more table and name it as "**contacts**" that contains the following data:

contact_id	cellphone	homephone
1	6546645978	4565242557
2	8798634532	8652413954
3	8790744345	9874437396
4	7655654336	9934345363

Here, we are going to explain CROSS JOIN with LEFT JOIN using three tables. Execute the following statement to join the three table customers, orders, and contacts. In this statement, first CROSS JOIN completed between orders and contacts, and then LEFT JOIN executes according to the specified condition.

1. **SELECT \* FROM** customer
2. **LEFT JOIN**(orders **CROSS JOIN** contacts)
3. **ON** customer.customer\_id=contact\_id
4. **ORDER BY** income;

After successful execution of the above query, it will give the following output:

customer_id	cust_name	occupation	income	qualification	order_id	date	customer_id	price	contact_id	cellphone	homephone
4	Shane Trump	Business...	10000	MBA	1001	2020-03-20	2	3000	4	7655654336	9934345363
4	Shane Trump	Business...	10000	MBA	1002	2020-02-15	4	2500	4	7655654336	9934345363
4	Shane Trump	Business...	10000	MBA	1003	2020-01-31	5	5000	4	7655654336	9934345363
4	Shane Trump	Business...	10000	MBA	1004	2020-03-10	2	1500	4	7655654336	9934345363
4	Shane Trump	Business...	10000	MBA	1005	2020-02-20	1	4500	4	7655654336	9934345363
1	John Miller	Developer	20000	Btech	1001	2020-03-20	2	3000	1	6546645978	4565242557
1	John Miller	Developer	20000	Btech	1002	2020-02-15	4	2500	1	6546645978	4565242557
1	John Miller	Developer	20000	Btech	1003	2020-01-31	5	5000	1	6546645978	4565242557
1	John Miller	Developer	20000	Btech	1004	2020-03-10	2	1500	1	6546645978	4565242557
1	John Miller	Developer	20000	Btech	1005	2020-02-20	1	4500	1	6546645978	4565242557
6	Ricky Pon...	Cricketer	200000	Btech	1001	2020-03-20	2	3000	6	NULL	9086053684
6	Ricky Pon...	Cricketer	200000	Btech	1002	2020-02-15	4	2500	6	NULL	9086053684

## MySQL SELF JOIN

A SELF JOIN is a join that is used to join a table with **itself**. In the previous sections, we have learned about the joining of the table with the other tables using different JOINS, such as INNER, LEFT, RIGHT, and CROSS JOIN. However, there is a need to combine data with other data in the same table itself. In that case, we use Self Join.

We can perform Self Join using **table aliases**. The table aliases allow us not to use the same table name twice with a single statement. If we use the same table name more than one time in a single query without table aliases, it will throw an error.

The table aliases enable us to use the **temporary name** of the table that we are going to use in the query. Let us understand the table aliases with the following explanation.

Suppose we have a table named "**student**" that is going to use twice in the single query. To aliases the student table, we can write it as:

1. **Select ... FROM** student **AS** S1
2. **INNER JOIN** student **AS** S2;

## SELF JOIN Syntax

The syntax of self-join is the same as the syntax of joining two different tables. Here, we use aliases name for tables because both the table name are the same. The following are the syntax of a SELF JOIN in MySQL:

1. **SELECT** s1.col\_name, s2.col\_name...
2. **FROM** table1 s1, table1 s2
3. **WHERE** s1.common\_col\_name = s2.common\_col\_name;

**NOTE:** You can also use another condition instead of WHERE clause according to your requirements.

## SELF JOIN Example

Let us create a table "student" in a database that contains the following data:

student_id	name	course_id	duration
1	Adam	1	3
2	Peter	2	4
1	Adam	2	4
3	Brian	3	2
2	Shane	3	5

Now, we are going to get all the result (student\_id and name) from the table where **student\_id** is equal, and **course\_id** is not equal. Execute the following query to understand the working of self-join in MySQL:

1. **SELECT** s1.student\_id, s1.name
2. **FROM** student **AS** s1, student s2
3. **WHERE** s1.student\_id=s2.student\_id
4. **AND** s1.course\_id<>s2.course\_id;

After the successful execution, we will get the following output:

student_id	name
1	Adam
2	Shane
1	Adam
2	Peter

## SELF JOIN using INNER JOIN clause

The following example explains how we can use Inner Join with Self Join. This query returns the student id and name when the student\_id of both tables is equals, and course\_id are not equal.

1. **SELECT** s1.student\_id, s1.name
2. **FROM** student s1
3. **INNER JOIN** student s2
4. **ON** s1.student\_id=s2.student\_id
5. **AND** s1.course\_id<>s2.course\_id
6. **GROUP BY** student\_id;

After executing the above statement, we will get the following example:

student_id	name
1	Adam
2	Shane

## SELF JOIN using LEFT JOIN clause

The following example explains how we can use LEFT Join with Self Join. This query returns the student name as **monitor** and city when the student\_id of both tables are equals.

1. **SELECT** (CONCAT(s1.stud\_lname, ' ', s2.stud\_fname)) **AS** 'Monitor', s1.city
2. **FROM** students s1
3. **LEFT JOIN** students s2 **ON** s1.student\_id=s2.student\_id
4. **ORDER BY** s1.city **DESC**;

After executing the above statement, we will get the following example:

Monitor	city
Putin Devine	France
Miller Ethon	England
Clark Michael	Australiya
Strauss Mark	America

## MySQL DELETE JOIN

DELETE query is a sub-part of data manipulation language used for removing the rows from tables. How to delete join in MySQL is a very popular question during the interviews. It is not an easy process to use the delete join statements in MySQL. In this section, we are going to describe how you can delete records from multiple tables with the use of INNER JOIN or LEFT JOIN in the DELETE query.

## DELETE JOIN with INNER JOIN

The Inner Join query can be used with Delete query for removing rows from one table and the matching rows from the other table that fulfill the specified condition.

### Syntax

The following are the syntax that can be used for deleting rows from more than one table using Inner Join.

1. **DELETE** target **table**
2. **FROM** table1
3. **INNER JOIN** table2
4. **ON** table1.joining\_column= table2.joining\_column
5. **WHERE** condition

Here, the target is a table name from where we want to delete rows by matching the specified condition. Suppose you want to delete rows from table **T1** and **T2** where **student\_id = 2**, then it can be written as the following statement:

1. **DELETE** T1, T2
2. **FROM** T1
3. **INNER JOIN** T2
4. **ON** T1.student\_id=T2.student.id
5. **WHERE** T1.student\_id=2;

In the above syntax, the target table (T1 and T2) is written between DELETE and FROM keywords. If we omit any table name from there, then the **delete statement** only removes rows from a single table. The expression written with **ON** keyword is the condition that matches the rows in tables where you are going to delete.

### Example

Suppose we have two table **students** and **contacts** that contains the following data:

**Table: students**

student_id	stud_fname	stud_lname	city
1	Devine	Putin	France
2	Michael	Clark	Australiya
3	Ethon	Miller	England
4	Mark	Strauss	America

**Table: contacts**

college_id	cellphone	homephone
1	5465465645	4576787687
2	4987246464	5795645568
3	8907654334	8654784126
4	8973454904	9865321475

Execute the following query to understand the **Delete Join** with Inner Join. This statement deletes a row that has the same id in both tables.

1. **DELETE** students, contacts **FROM** students
2. **INNER JOIN** contacts **ON** students.student\_id=contacts.college\_id
3. **WHERE** students.student\_id = 4;

After successful execution, it will give the following message:

```
DELETE students, contact FROM students INNER JOIN cont... 2 row(s) affected 0.234 sec
```

Now, run the following query to verify the rows deleted successfully.

1. mysql> **SELECT \* FROM** students;
2. mysql> **SELECT \* FROM** contacts;

You can see that the rows where the **student\_id=4** is deleted.

student_id	stud_fname	stud_lname	city	college_id	cellphone	homephone
1	Devine	Putin	France	1	5465465645	4576787687
2	Michael	Clark	Australiya	2	4987246464	5795645568
3	Ethon	Miller	England	3	8907654334	8654784126

## DELETE JOIN with LEFT JOIN

We have already learned the LEFT JOIN clause with **SELECT statement** that returns all rows from the left(first) table and the matching or not matching rows from another table. Similarly, we can also use the LEFT JOIN clause with the **DELETE keyword** for deleting rows from the left(first) table that does not have matching rows from a right(second) table.

The following query explains it more clearly where **DELETE statement** use LEFT JOIN for deleting rows from **Table1** that does not have matching rows in the **Table2**:

1. **DELETE Table1 FROM Table1**
2. **LEFT JOIN Table2 ON Table1.key = Table2.key**
3. **WHERE Table2.key IS NULL;**

In the above query, notice that we will only use Table1 with the **DELETE keyword**, not both as did in the INNER JOIN statement.

### Example

Let us create a table "**contacts**" and "**customers**" in a database that contains the following data:

**Table: contacts**

contact_id	cellphone	homephone
1	6546645978	4565242557
2	8798634532	8652413954
3	8790744345	9874437396
4	7655654336	9934345363
5	NULL	6786507067
6	NULL	9086053684

**Table: customers**

customer_id	cust_name	occupation	income	qualification
1	John Miller	Developer	20000	Btech
2	Mark Robert	Enginneer	40000	Btech
3	Reyan Wa...	Scientists	60000	MSc
4	Shane Trump	Business...	10000	MBA
5	Adam Obama	Manager	80000	MBA
6	Rincky Pon...	Cricketer	200000	Btech
7	Adam Gilcrist	Cricketer	300000	PGDBA

Execute the following statement that removes the customer who does not have a **cellphone** number:

1. **DELETE customers FROM customers**
2. **LEFT JOIN contacts ON customers.customer\_id = contacts.contact\_id**
3. **WHERE cellphone IS NULL;**

After successful execution, it will give the following message:

```
DELETE customer FROM customer LEFT JOIN contacts ON c... 3 row(s) affected          0.172 sec
```

Now, run the following query to verify the rows deleted successfully.

1. mysql> **SELECT \* FROM customers;**

You can see that the rows where the customer does not have the cellphone number are deleted.

customer_id	cust_name	occupation	income	qualification
1	John Miller	Developer	20000	Btech
2	Mark Robert	Enginneer	40000	Btech
3	Reyan Wa...	Scientists	60000	MSc
4	Shane Trump	Business...	10000	MBA

# MySQL UPDATE JOIN

UPDATE query in MySQL is a DML statement used for modifying the data of a table. The UPDATE query must require the SET and WHERE clause. The SET clause is used to change the values of the column specified in the WHERE clause.

JOIN clause in MySQL is used in the statement to retrieve data by joining multiple tables within a single query.

**The UPDATE JOIN is a MySQL statement used to perform cross-table updates that means we can update one table using another table with the JOIN clause condition.** This query update and alter the data where more than one tables are joined based on **PRIMARY Key**

and **FOREIGN Key**

and a specified join condition. We can update single or multiple columns at a time using the [UPDATE query](#)

**NOTE:** The MySQL UPDATE JOIN statement is supported from version 4.0 or higher.

## Syntax

Following is a basic syntax of UPDATE JOIN statement to modify record into the MySQL table:

1. **UPDATE** Tab1, Tab2, [**INNER JOIN** | **LEFT JOIN**] Tab1 **ON** Tab1.C1 = Tab2.C1
2. **SET** Tab1.C2 = Tab2.C2, Tab2.C3 = expression
3. **WHERE** Condition;

In the above MySQL UPDATE JOIN syntax:

First, we have specified the two tables: the main table (Tab1) and another table (tab2) after the UPDATE clause. After the UPDATE clause, it is required to specify at least one table. Next, we have specified the types of [JOIN clauses](#)

, i.e., either [INNER JOIN](#)

or [LEFT JOIN](#)

, which appear right after the UPDATE clause and then a join predicate specified after the ON keyword. Then, we have to assign the new values to the columns in Tab1 and/or Tab2 for modification into the table. Finally, the [WHERE clause](#)

condition is used to limit rows for updation.

## How does UPDATE JOIN work in MySQL?

The UPDATE JOIN work process in [MySQL](#)

is the same as described in the above syntax. But sometimes, we would find that this query alone performed the cross-table update without involving any joins. **The following syntax is another way to update one table using another table:**

1. **UPDATE** Tab1, Tab2,
2. **SET** Tab1.C2 = Tab2.C2, Tab2.C3 = expression
3. **WHERE** Tab1.C1 = Tab2.C1 AND condition;

The above UPDATE statement produces the same result as the UPDATE JOIN with an INNER JOIN or LEFT JOIN clauses. It means we can re-write the above syntax as UPDATE JOIN syntax displayed above:

1. **UPDATE** Tab1,Tab2
2. **INNER JOIN** Tab2 **ON** Tab1.C1 = Tab2.C1
3. **SET** Tab1.C2 = Taba2.C2, Tab2.C3 = expression
4. **WHERE** condition

Let us take some examples to understand how the UPDATE JOIN statement works in MySQL table.

## UPDATE JOIN Examples

First, we will create two tables named **Performance** and **Employee**, and both tables are related through a foreign key. Here, the "Performance" is a **parent table**, and "Employees" is the **child table**. The following scripts create both tables along with their records.

### Table: Performance

1. **CREATE TABLE** Performance (
2.   performance **INT**(11) NOT NULL,
3.   percentage **FLOAT** NOT NULL,
4.   **PRIMARY KEY** (performance)
5. );

Next, fill the records in the table using the INSERT statement.

1. **INSERT INTO** Performance (performance, percentage)
2. **VALUES**(101,0),
3.   (102,0.01),
4.   (103,0.03),
5.   (104,0.05),
6.   (105,0.08);

Then, execute the SELECT query to verify the data as shown in the below image:

```
MySQL 8.0 Command Line Client
mysql> CREATE TABLE Performance (
    ->     performance INT(11) NOT NULL,
    ->     percentage FLOAT NOT NULL,
    ->     PRIMARY KEY (performance)
    -> );
Query OK, 0 rows affected, 1 warning (0.62 sec)

mysql> INSERT INTO Performance(performance,percentage)
    -> VALUES(101,0),
    ->       (102,0.01),
    ->       (103,0.03),
    ->       (104,0.05),
    ->       (105,0.08);
Query OK, 5 rows affected (0.13 sec)
Records: 5  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM Performance;
+-----+-----+
| performance | percentage |
+-----+-----+
|      101 |          0 |
|      102 |      0.01 |
|      103 |      0.03 |
|      104 |      0.05 |
|      105 |      0.08 |
+-----+-----+
5 rows in set (0.00 sec)
```

Table: Employees

1. **CREATE TABLE** Employees (
2. id **INT**(11) NOT NULL **AUTO\_INCREMENT PRIMARY KEY**,
3. name **VARCHAR**(255) NOT NULL,
4. performance **INT**(11) **DEFAULT** NULL,
5. salary **FLOAT DEFAULT** NULL,
6. **CONSTRAINT** fk\_performance **FOREIGN KEY** (performance) **REFERENCES** Performance (performance)
7. );

Next, fill the records in the table using the INSERT statement.

1. **INSERT INTO** Employees (**name**, performance, salary)
2. **VALUES**('Mary', 101, 55000),
3. ('John', 103, 65000),
4. ('Suzi', 104, 85000),
5. ('Gracia', 105, 110000),
6. ('Nancy Johnson', 103, 95000),
7. ('Joseph', 102, 45000),
8. ('Donald', 103, 50000);

Then, execute the SELECT query to verify the data as shown in the below image:

```

MySQL 8.0 Command Line Client

mysql> CREATE TABLE Employees (
->     id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
->     name VARCHAR(255) NOT NULL,
->     performance INT(11) DEFAULT NULL,
->     salary FLOAT DEFAULT NULL,
->     CONSTRAINT fk_performance FOREIGN KEY (performance) REFERENCES Performance (performance)
-> );
Query OK, 0 rows affected, 2 warnings (0.48 sec)

mysql> INSERT INTO Employees(name,performance,salary)
-> VALUES('Mary', 101, 55000),
->        ('John', 103, 65000),
->        ('Suzi', 104, 85000),
->        ('Gracia', 105, 110000),
->        ('Nancy Johnson', 103, 95000),
->        ('Joseph', 102, 45000),
->        ('Donald', 103, 50000);
Query OK, 7 rows affected (0.15 sec)
Records: 7  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM Employees;
+---+-----+-----+-----+
| id | name      | performance | salary |
+---+-----+-----+-----+
| 1  | Mary       |      101    | 55000  |
| 2  | John       |      103    | 65000  |
| 3  | Suzi       |      104    | 85000  |
| 4  | Gracia     |      105    | 110000 |
| 5  | Nancy Johnson | 103    | 95000  |
| 6  | Joseph     |      102    | 45000  |
| 7  | Donald     |      103    | 50000  |
+---+-----+-----+-----+

```

## UPDATE JOIN with INNER JOIN Example

Suppose we want to update the **employee's salary on the basis of their performance**. We can update an employee's salary in the Employees table using the UPDATE INNER JOIN statement because the performance **percentage** is stored in the performance table.

In the above tables, we have to use the **performance** field to join the Employees and Performance table. See the below query:

1. **UPDATE** Employees e
2. **INNER JOIN** Performance p
3. **ON** e.performance = p.performance
4. **SET** salary = salary + salary \* percentage;

After executing the above statement, we will get the below output, where we can see that the employee's salary column is updated successfully.

```

MySQL 8.0 Command Line Client

mysql> UPDATE Employees e
-> INNER JOIN Performance p
-> ON e.performance = p.performance
-> SET salary = salary + salary * percentage;
Query OK, 6 rows affected (0.58 sec)
Rows matched: 7  Changed: 6  Warnings: 0

mysql> SELECT * FROM Employees;
+---+-----+-----+-----+
| id | name      | performance | salary |
+---+-----+-----+-----+
| 1  | Mary       |      101    | 55000  |
| 2  | John       |      103    | 66950  |
| 3  | Suzi       |      104    | 89250  |
| 4  | Gracia     |      105    | 118800 |
| 5  | Nancy Johnson | 103    | 97850  |
| 6  | Joseph     |      102    | 45450  |
| 7  | Donald     |      103    | 51500  |
+---+-----+-----+-----+
7 rows in set (0.00 sec)

```

Let us understand how this query works in MySQL. In the query, we have specified only the Employees table after the UPDATE clause. It is because we want to change the record only in the Employees table, not in both tables.

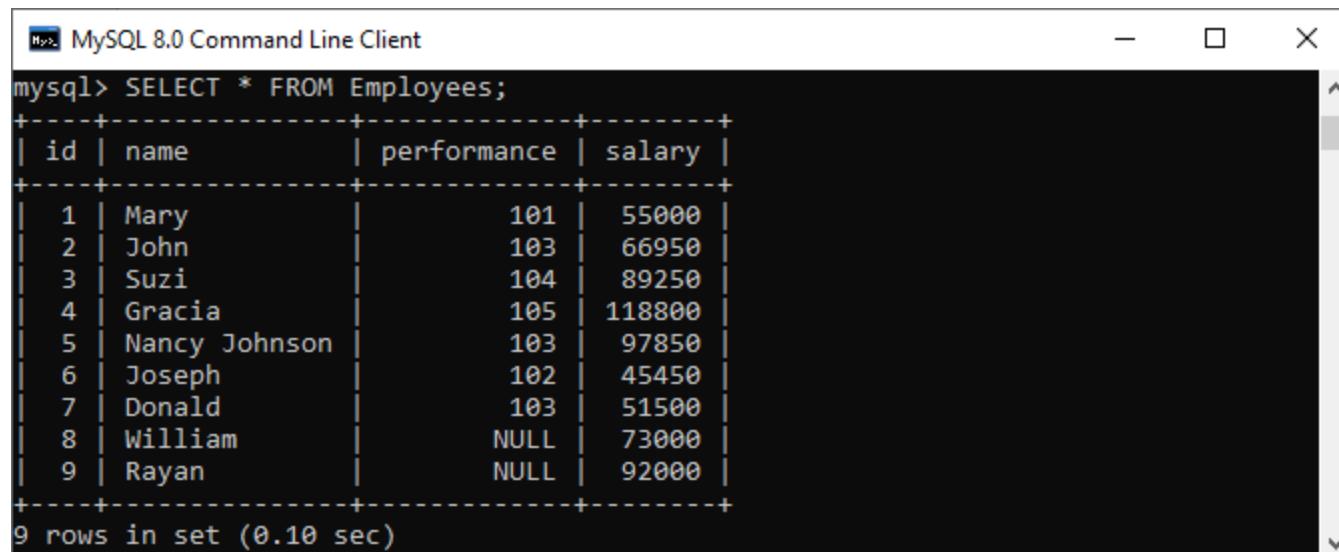
The query checks the performance column values for each row in the "Employees" table against the performance column of the "Performance" table. If it will get the matched performance column, then it takes the percentage in the Performance table and updates the Employees table's salary column. This query updates all records in the Employees table because we have not specified the WHERE clause in the UPDATE JOIN query.

## UPDATE JOIN with LEFT JOIN Example

To understand the UPDATE JOIN with LEFT JOIN, we first need to insert two new rows into the Employees table:

1. **INSERT INTO** Employees (**name**, performance, salary)
2. **VALUES**('William', NULL, 73000),
3. ('Rayan', NULL, 92000);

Since these employees are new hires, so their performance record is not available. See the below output:



The screenshot shows the MySQL 8.0 Command Line Client window. The command entered is `SELECT * FROM Employees;`. The resulting table has three columns: id, name, and salary. The salary values for the last two rows (William and Rayan) are 73000 and 92000 respectively, while the performance values are NULL. The table structure is as follows:

id	name	performance	salary
1	Mary	101	55000
2	John	103	66950
3	Suzi	104	89250
4	Gracia	105	118800
5	Nancy Johnson	103	97850
6	Joseph	102	45450
7	Donald	103	51500
8	William	NULL	73000
9	Rayan	NULL	92000

9 rows in set (0.10 sec)

If we want to update the salary for newly hired employees, we cannot use the UPDATE INNER JOIN query. It is due to the unavailability of their performance data in the Performance table. Thus, we will use the UPDATE LEFT JOIN statement to fulfill this need.

The UPDATE LEFT JOIN statement in MySQL is used to update a row in a table when there are no records found in another table's corresponding row.

For example, if we want to increase the salary for a newly hired employee by 2.5%, we can do this with the help of the following statement:

1. **UPDATE** Employees e
2. **LEFT JOIN** Performance p
3. **ON** e.performance = p.performance
4. **SET** salary = salary + salary \* 0.025
5. **WHERE** p.percentage IS NULL;

After executing the above query, we will get the output as below image where we can see that salary of the newly hired employees is successfully updated.

```

MySQL 8.0 Command Line Client
mysql> UPDATE Employees e
-> LEFT JOIN Performance p
-> ON e.performance = p.performance
-> SET salary = salary + salary * 0.025
-> WHERE p.percentage IS NULL;
Query OK, 2 rows affected (0.12 sec)
Rows matched: 2  Changed: 2  Warnings: 0

mysql> SELECT * FROM Employees;
+----+-----+-----+-----+
| id | name | performance | salary |
+----+-----+-----+-----+
| 1  | Mary  |        101 | 55000 |
| 2  | John  |        103 | 66950 |
| 3  | Suzi  |        104 | 89250 |
| 4  | Gracia |       105 | 118800 |
| 5  | Nancy Johnson | 103 | 97850 |
| 6  | Joseph |       102 | 45450 |
| 7  | Donald |       103 | 51500 |
| 8  | William |      NULL | 74825 |
| 9  | Rayan  |      NULL | 94300 |
+----+-----+-----+-----+

```

In this article, we have learned the MySQL Update Join statement that allows us to alter the existing data in one table with the new data from another table with the JOIN clause condition. This query is advantageous when we need to modify certain columns specified in the WHERE clause along with either using the INNER JOIN or LEFT JOIN clauses.

## MySQL EquiJoin

The process is called joining when we combine two or more tables based on some common columns and a join condition. **An equijoin is an operation that combines multiple tables based on equality or matching column values in the associated tables.**

We can use the **equal sign (=) comparison operator** to refer to equality in the **WHERE clause**. This joining operation returns the same result when we use the **JOIN keyword** with the **ON** clause and then specifying the column names and their associated tables.

Equijoin is a classified type of inner join that returns output by performing joining operations from two tables based on the common column that exists in them. This join returns only those data that are available in both tables based on the common primary field name. It does not display the null records or unmatchable data into the result set.

### Points to remember:

- There is no need to be the same column names.
- The resultant result can have repeated column names.
- We can also perform an equijoin operation on more than two tables.

### Syntax:

The following are the basic syntax that illustrates the equijoin operations:

1. **SELECT** column\_name (s)
2. **FROM** table\_name1, table\_name2, ...., table\_nameN
3. **WHERE** table\_name1.column\_name = table\_name2.column\_name;

OR

1. **SELECT** (column\_list | \*)
2. **FROM** table\_name1
3. **JOIN** table\_name2
4. **ON** table\_name1.column\_name = table\_name2.column\_name;

In this syntax, we need to specify the **column names** to be included in the result set after the **SELECT keyword**. If we want to select all columns from both tables, the **\*** operator will be used. Next, we will specify the **table names** for joining after the **FROM keyword**, and finally, write the **join condition** in the **WHERE** and **ON** clause.

### EquiJoin Example

Let us understand how equijoin works in MySQL through examples. Suppose we have already two tables named **customer** and **balance** that contains the following data:

```
MySQL 8.0 Command Line Client
mysql> select * from customer;
+----+-----+-----+
| id | customer_name | account | email
+----+-----+-----+
| 1  | Stephen       | 1030   | stephen@javatpoint.com
| 2  | Jenifer       | 2035   | jenifer@javatpoint.com
| 3  | Mathew        | 5564   | mathew@javatpoint.com
| 4  | Smith          | 4534   | smith@javatpoint.com
| 5  | david          | 7648   | david@javatpoint.com
+----+-----+-----+
5 rows in set (0.00 sec)

mysql> select * from balance;
+----+-----+
| id | account_num | balance
+----+-----+
| 1  | 1030         | 50000.00
| 2  | 2035         | 230000.00
| 3  | 5564         | 125000.00
| 4  | 4534         | 80000.00
| 5  | 7648         | 45000.00
+----+-----+
5 rows in set (0.00 sec)
```

Execute the below equijoin statement for joining tables:

1. mysql> **SELECT** cust.customer\_name, bal.balance
2. **FROM** customer **AS** cust, balance **AS** bal
3. **WHERE** cust.account = bal.account\_num;

We will get the following result:

```
MySQL 8.0 Command Line Client
mysql> SELECT cust.customer_name, bal.balance
    -> FROM customer AS cust, balance AS bal
    -> WHERE cust.account = bal.account_num;
+-----+-----+
| customer_name | balance
+-----+-----+
| Stephen       | 50000.00
| Jenifer       | 230000.00
| Mathew        | 125000.00
| Smith          | 80000.00
| david          | 45000.00
+-----+-----+
5 rows in set (0.00 sec)
```

We can also get the same result by using the below statement:

1. mysql> **SELECT** cust.customer\_name, bal.balance
2. **FROM** customer **AS** cust
3. **JOIN** balance **AS** bal
4. **WHERE** cust.account = bal.account\_num;

See the below output that is the same as the result returns from the previous query:

```
MySQL 8.0 Command Line Client
mysql> SELECT cust.customer_name, bal.balance
    -> FROM customer AS cust
    -> JOIN balance AS bal
    -> WHERE cust.account = bal.account_num;
+-----+-----+
| customer_name | balance
+-----+-----+
| Stephen       | 50000.00
| Jenifer       | 230000.00
| Mathew        | 125000.00
| Smith          | 80000.00
| david          | 45000.00
+-----+-----+
5 rows in set (0.00 sec)
```

## Equi Join Using Three Tables

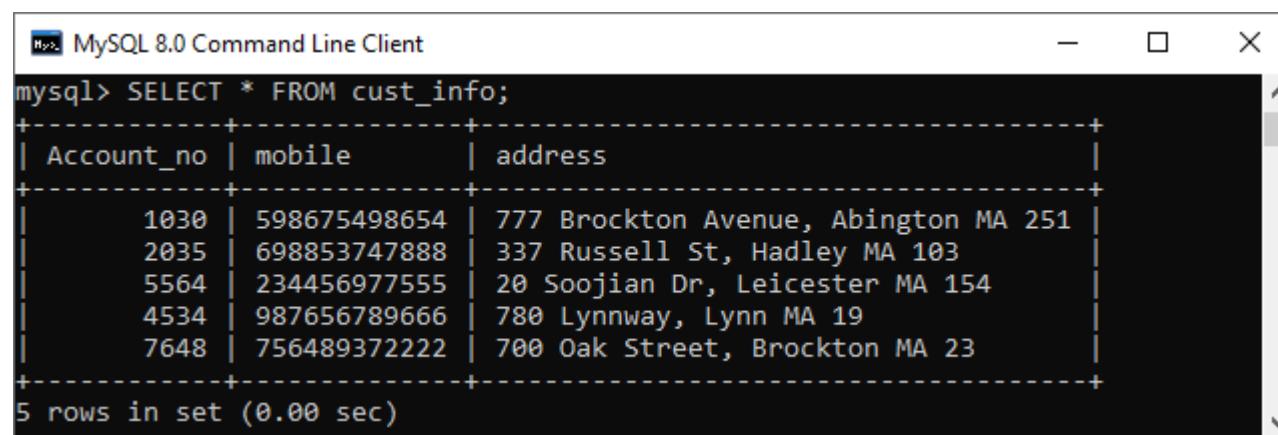
We know that equijoin can also perform a join operation on more than two tables. To understand this, let us create another table named **cust\_info** using the below statement:

```
1. CREATE TABLE cust_info (
2.   account_no int,
3.   mobile VARCHAR(15),
4.   address VARCHAR(65)
5. );
```

Then, we will fill records into this table:

```
1. INSERT INTO cust_info (account_no, mobile, address)
2. VALUES(1030, '598675498654', '777 Brockton Avenue, Abington MA 251'),
3. (2035, '698853747888', '337 Russell St, Hadley MA 103'),
4. (5564, '234456977555', '20 Soojian Dr, Leicester MA 154'),
5. (4534, '987656789666', '780 Lynnway, Lynn MA 19'),
6. (7648, '756489372222', '700 Oak Street, Brockton MA 23');
```

We can verify the data using the SELECT statement. See the below image:



The screenshot shows the MySQL 8.0 Command Line Client window. The command entered is `SELECT * FROM cust_info;`. The output is a table with three columns: Account\_no, mobile, and address. The data consists of five rows:

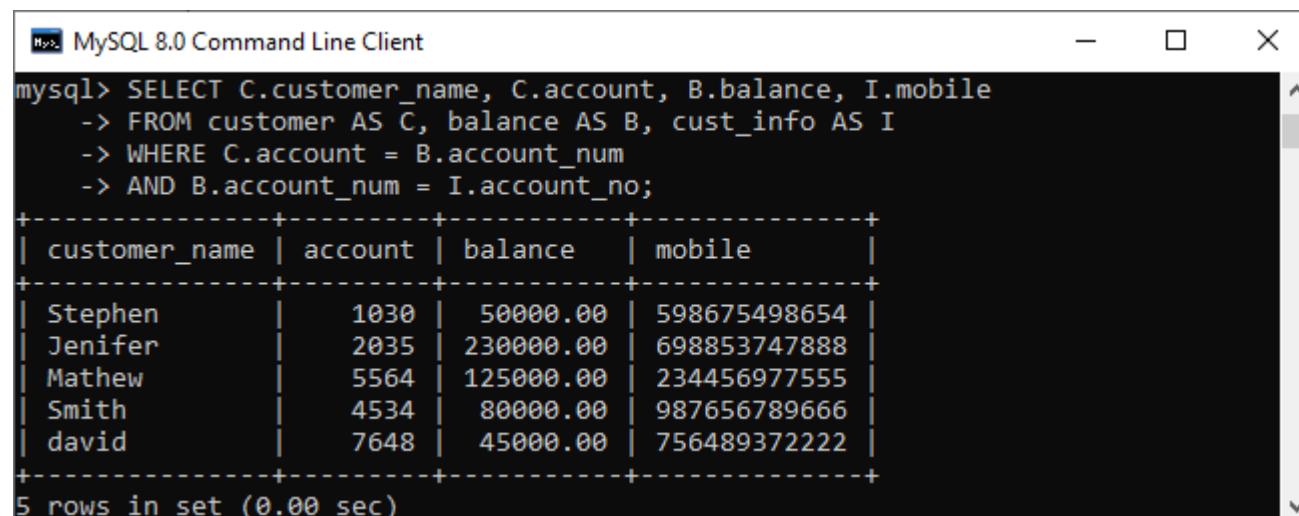
Account_no	mobile	address
1030	598675498654	777 Brockton Avenue, Abington MA 251
2035	698853747888	337 Russell St, Hadley MA 103
5564	234456977555	20 Soojian Dr, Leicester MA 154
4534	987656789666	780 Lynnway, Lynn MA 19
7648	756489372222	700 Oak Street, Brockton MA 23

5 rows in set (0.00 sec)

To join three tables using equijoin, we need to execute the statement as follows:

```
1. mysql> SELECT C.customer_name, C.account, B.balance, I.mobile
2. FROM customer AS C, balance AS B, cust_info AS I
3. WHERE C.account = B.account_num
4. AND B.account_num = I.account_no;
```

It will give the below result.



The screenshot shows the MySQL 8.0 Command Line Client window. The command entered is a complex SELECT statement involving three tables: customer (C), balance (B), and cust\_info (I). The output is a table with four columns: customer\_name, account, balance, and mobile. The data consists of five rows:

customer_name	account	balance	mobile
Stephen	1030	50000.00	598675498654
Jenifer	2035	230000.00	698853747888
Mathew	5564	125000.00	234456977555
Smith	4534	80000.00	987656789666
david	7648	45000.00	756489372222

5 rows in set (0.00 sec)

## Difference between Natural Join, Equi Join and Inner Join

Let us summaries the differences between natural, equi and inner join operation in the tabular form given below:

Natural Join	Equi Join	Inner Join
It joins the tables based on the same column names and their data types.	It joins the tables based on the equality or matching column values in the associated tables.	It joins the tables based on the column name specified in the ON clause explicitly. It returns only those rows that exist in both tables.

It always returns unique columns in the result set.	It can return all attributes of both tables along with duplicate columns that match the join condition.	It returns all the attributes of both tables along with duplicate columns that match the ON clause condition.
The syntax of a natural join is given below:  SELECT [column_names   *] FROM table_name1 NATURAL JOIN table_name2;	The syntax of equijoin is given below:  SELECT column_name (s)  FROM table_name1, table_name2, ..... table_nameN  WHERE table_name1.column_name = table_name2.column_name;	The syntax of inner join is given below:  SELECT [column_names   *]  FROM table_name1  INNER JOIN table_name2 ON table_name1.column_name = table_name2.column_name;

## MySQL Natural Join

When we combine rows of two or more tables based on a common column between them, this operation is called joining. **A natural join is a type of join operation that creates an implicit join by combining tables based on columns with the same name and data type.** It is similar to the **INNER** or **LEFT JOIN**, but we cannot use the **ON** or **USING** clause with natural join as we used in them.

### Points to remember:

- There is no need to specify the column names to join.
- The resultant table always contains unique columns.
- It is possible to perform a natural join on more than two tables.
- Do not use the **ON** clause.

### Syntax:

The following is a basic syntax to illustrate the natural join:

1. **SELECT** [column\_names | \*]
2. **FROM** table\_name1
3. **NATURAL JOIN** table\_name2;

In this syntax, we need to specify the **column names** to be included in the result set after the **SELECT** keyword. If we want to select all columns from both tables, the **\*** operator will be used. Next, we will specify the **table names** for joining after the **FROM** keyword and write the **NATURAL JOIN** clause between them.

### Natural Join Example

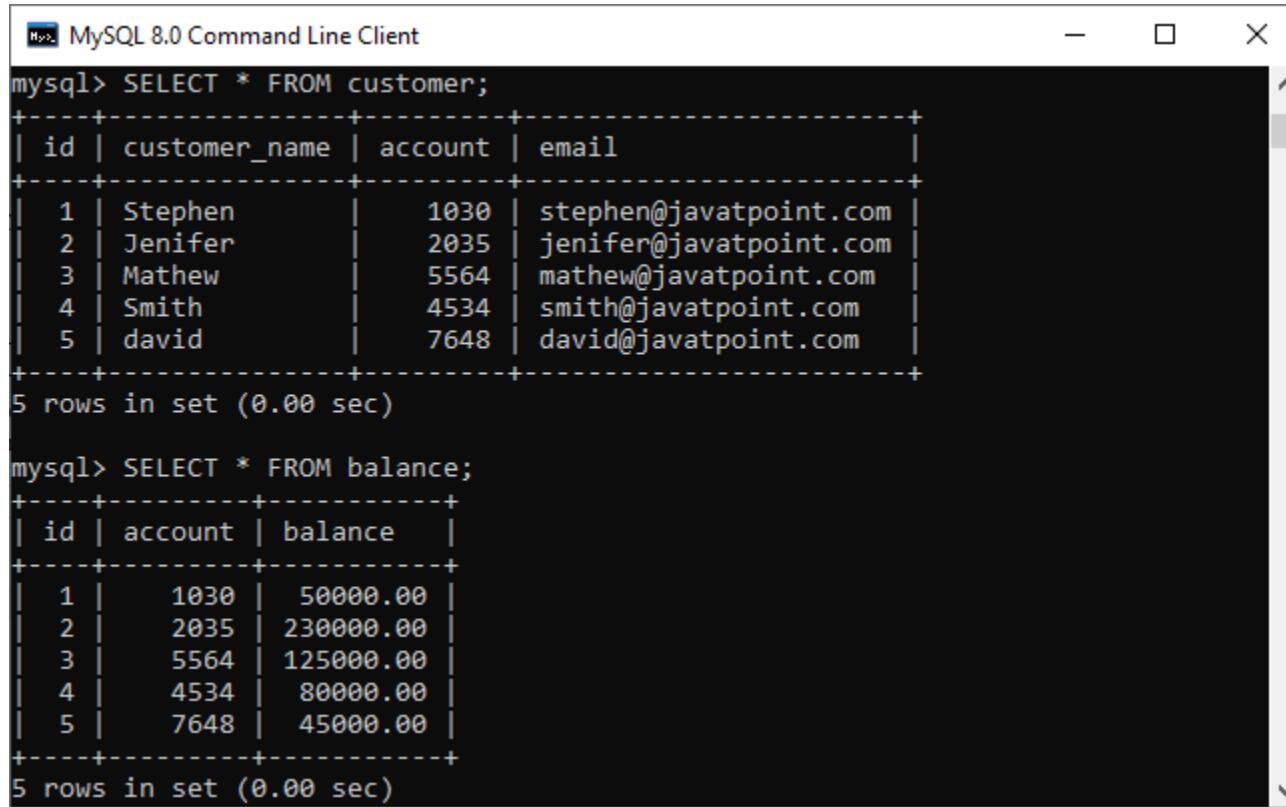
Let us understand how natural join works in MySQL through examples. First, we will create two tables named **customer** and **balance** using the below statements:

1. /\* -- Table name: customer \*/
2. **CREATE TABLE** customer (
3.   id **INT** AUTO\_INCREMENT **PRIMARY KEY**,
4.   customer\_name **VARCHAR**(55),
5.   account **int**,
6.   email **VARCHAR**(55)
7. );
- 8.
9. /\* -- Table name: balance \*/
10. **CREATE TABLE** balance (
11.   id **INT** AUTO\_INCREMENT **PRIMARY KEY**,
12.   account **int**,
13.   balance **FLOAT**(10, 2)
14. );

Next, we will fill some records into both tables using the below statements:

```
1. /* -- Data for customer table */
2. INSERT INTO customer(customer_name, account, email)
3. VALUES('Stephen', 1030, 'stephen@javatpoint.com'),
4. ('Jenifer', 2035, 'jenifer@javatpoint.com'),
5. ('Mathew', 5564, 'mathew@javatpoint.com'),
6. ('Smith', 4534, 'smith@javatpoint.com'),
7. ('David', 7648, 'david@javatpoint.com');
8.
9. /* -- Data for balance table */
10. INSERT INTO balance(account, balance)
11. VALUES(1030, 50000.00),
12. (2035, 230000.00),
13. (5564, 125000.00),
14. (4534, 80000.00),
15. (7648, 45000.00);
```

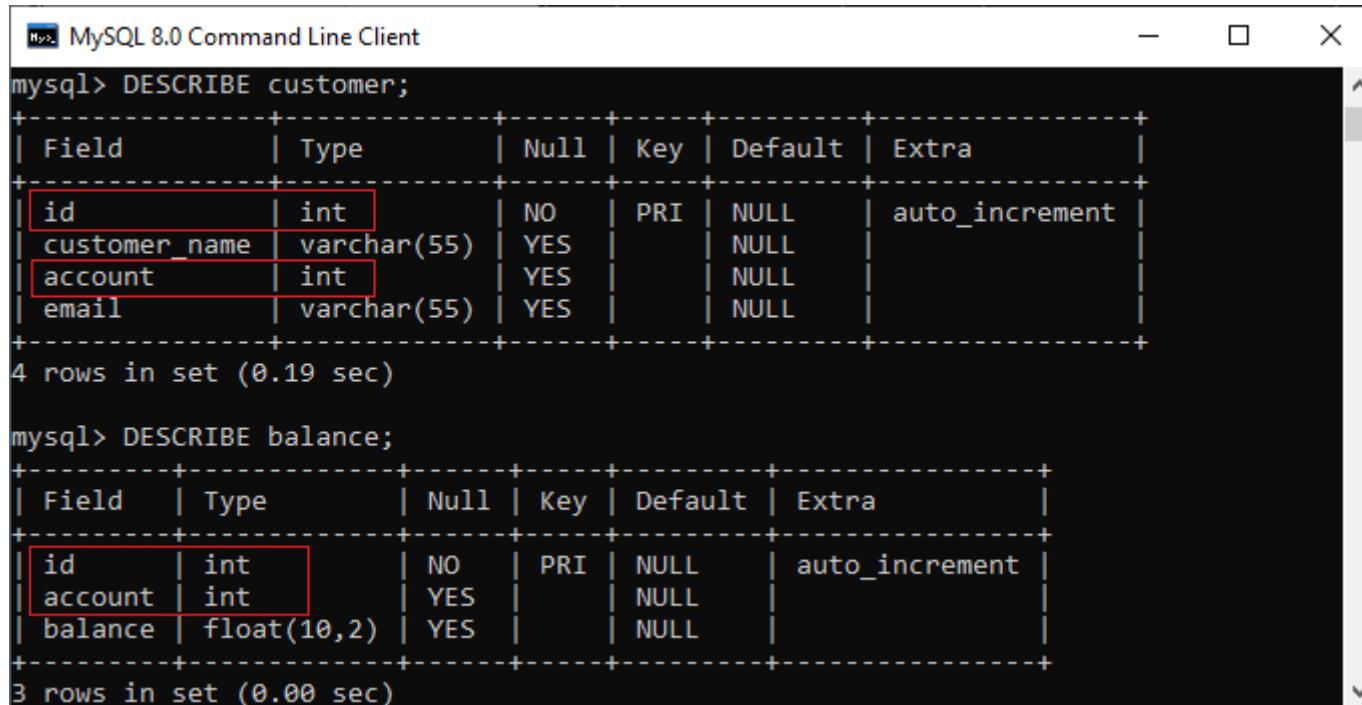
Next, we will execute the **SELECT statement** to verify the table data:



```
MySQL 8.0 Command Line Client
mysql> SELECT * FROM customer;
+----+-----+-----+-----+
| id | customer_name | account | email          |
+----+-----+-----+-----+
| 1  | Stephen      | 1030   | stephen@javatpoint.com
| 2  | Jenifer      | 2035   | jenifer@javatpoint.com
| 3  | Mathew       | 5564   | mathew@javatpoint.com
| 4  | Smith         | 4534   | smith@javatpoint.com
| 5  | david         | 7648   | david@javatpoint.com
+----+-----+-----+-----+
5 rows in set (0.00 sec)

mysql> SELECT * FROM balance;
+----+-----+-----+
| id | account | balance |
+----+-----+-----+
| 1  | 1030    | 50000.00
| 2  | 2035    | 230000.00
| 3  | 5564    | 125000.00
| 4  | 4534    | 80000.00
| 5  | 7648    | 45000.00
+----+-----+-----+
5 rows in set (0.00 sec)
```

Now, we will see the condition that fulfills the criteria for natural join. We can do this by examining the table structure using the **DESCRIBE** statement. See the below image:



```
MySQL 8.0 Command Line Client
mysql> DESCRIBE customer;
+-----+-----+-----+-----+
| Field   | Type    | Null | Key  | Default | Extra           |
+-----+-----+-----+-----+
| id      | int     | NO   | PRI   | NULL    | auto_increment
| customer_name | varchar(55) | YES  |       | NULL    |
| account  | int     | YES  |       | NULL    |
| email    | varchar(55) | YES  |       | NULL    |
+-----+-----+-----+-----+
4 rows in set (0.19 sec)

mysql> DESCRIBE balance;
+-----+-----+-----+-----+
| Field   | Type    | Null | Key  | Default | Extra           |
+-----+-----+-----+-----+
| id      | int     | NO   | PRI   | NULL    | auto_increment
| account | int     | YES  |       | NULL    |
| balance | float(10,2)| YES  |       | NULL    |
+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

In this image, we can see that **column names id and account and its data types** are the same that fulfill the natural join criteria. Hence we can use natural join on them.

Execute the below statement for joining tables using natural join:

1. mysql> **SELECT** cust.customer\_name, bal.balance
2. **FROM** customer **AS** cust
3. **NATURAL JOIN** balance **AS** bal;

We will get the following result:

```
MySQL 8.0 Command Line Client

mysql> SELECT cust.customer_name, bal.balance
-> FROM customer AS cust
-> NATURAL JOIN balance AS bal;
+-----+-----+
| customer_name | balance |
+-----+-----+
| Stephen       | 50000.00 |
| Jenifer       | 230000.00 |
| Mathew        | 125000.00 |
| Smith          | 80000.00  |
| david          | 45000.00  |
+-----+-----+
5 rows in set (0.00 sec)
```

We can do the same job with the help of **INNER JOIN using the ON clause**. Here is the query to explain this join:

1. mysql> **SELECT** cust.customer\_name, bal.balance
2. **FROM** customer **AS** cust
3. **INNER JOIN** balance **AS** bal
4. **ON** cust.id = bal.id;

After successful execution, we will get the same result as the natural join:

```
MySQL 8.0 Command Line Client

mysql> SELECT cust.customer_name, bal.balance
-> FROM customer AS cust
-> INNER JOIN balance AS bal
-> ON cust.id = bal.id;
+-----+-----+
| customer_name | balance |
+-----+-----+
| Stephen       | 50000.00 |
| Jenifer       | 230000.00 |
| Mathew        | 125000.00 |
| Smith          | 80000.00  |
| david          | 45000.00  |
+-----+-----+
5 rows in set (0.00 sec)
```

Now, we will use (\*) in the place of column names as follows:

1. mysql> **SELECT \* FROM** customer **NATURAL JOIN** balance;

Suppose we use the asterisk (\*) in the place of column names, then the natural join automatically searches the same column names and their data types and join them internally. Also, it does not display the repeated columns in the output. Hence, we should get the below output after executing the above statement:

```
MySQL 8.0 Command Line Client

mysql> SELECT * FROM customer NATURAL JOIN balance;
+-----+-----+-----+-----+-----+
| id  | account | customer_name | email           | balance |
+-----+-----+-----+-----+-----+
| 1   | 1030    | Stephen       | stephen@javatpoint.com | 50000.00 |
| 2   | 2035    | Jenifer       | jenifer@javatpoint.com | 230000.00 |
| 3   | 5564    | Mathew        | mathew@javatpoint.com | 125000.00 |
| 4   | 4534    | Smith          | smith@javatpoint.com | 80000.00  |
| 5   | 7648    | david          | david@javatpoint.com | 45000.00  |
+-----+-----+-----+-----+-----+
5 rows in set (0.00 sec)
```

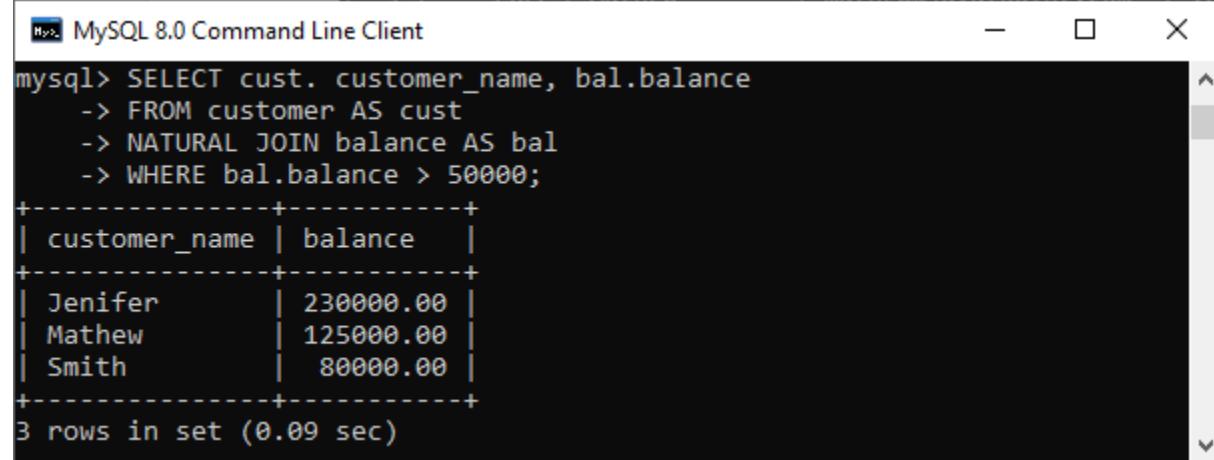
## Natural Join with WHERE Clause

The **WHERE clause** is used to return the filter result from the table. The following example illustrates this with the natural join clause:

1. mysql> **SELECT** cust.customer\_name, bal.balance

2. **FROM** customer **AS** cust
3. NATURAL JOIN balance **AS** bal
4. **WHERE** bal.balance > 50000;

We will get the following result where customer information is displayed whose **account balance is greater than 50000**.



```
MySQL 8.0 Command Line Client
mysql> SELECT cust.customer_name, bal.balance
-> FROM customer AS cust
-> NATURAL JOIN balance AS bal
-> WHERE bal.balance > 50000;
+-----+
| customer_name | balance |
+-----+
| Jenifer      | 230000.00 |
| Mathew        | 125000.00 |
| Smith         | 80000.00  |
+-----+
3 rows in set (0.09 sec)
```

## Natural Join Using Three Tables

We know that natural join can also perform a join operation on more than two tables. To understand this, we will use the syntax as follows:

1. **SELECT** [column\_names | \*]
2. **FROM** table\_name1
3. NATURAL JOIN table\_name2
4. NATURAL JOIN table\_name3;

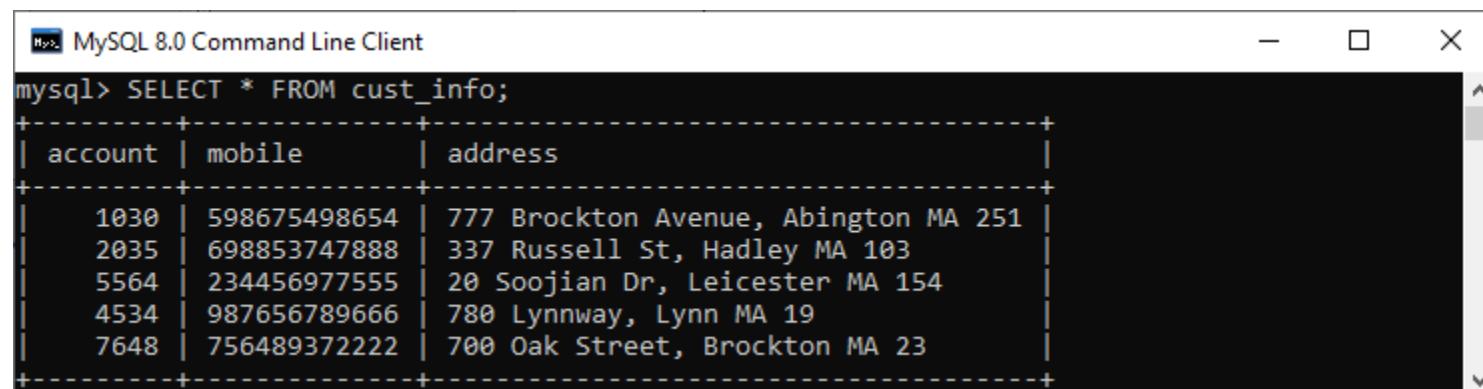
Let us create another table named **cust\_info** using the below statement:

1. **CREATE TABLE** cust\_info (
2. account **int**,
3. mobile **VARCHAR**(15),
4. address **VARCHAR**(65)
5. );

Then, we will fill records into this table:

1. **INSERT INTO** cust\_info(account, mobile, address)
2. **VALUES**(1030, '598675498654', '777 Brockton Avenue, Abington MA 251'),
3. (2035, '698853747888', '337 Russell St, Hadley MA 103'),
4. (5564, '234456977555', '20 Soojian Dr, Leicester MA 154'),
5. (4534, '987656789666', '780 Lynnway, Lynn MA 19'),
6. (7648, '756489372222', '700 Oak Street, Brockton MA 23');

We can verify the data using the SELECT statement. See the below image:

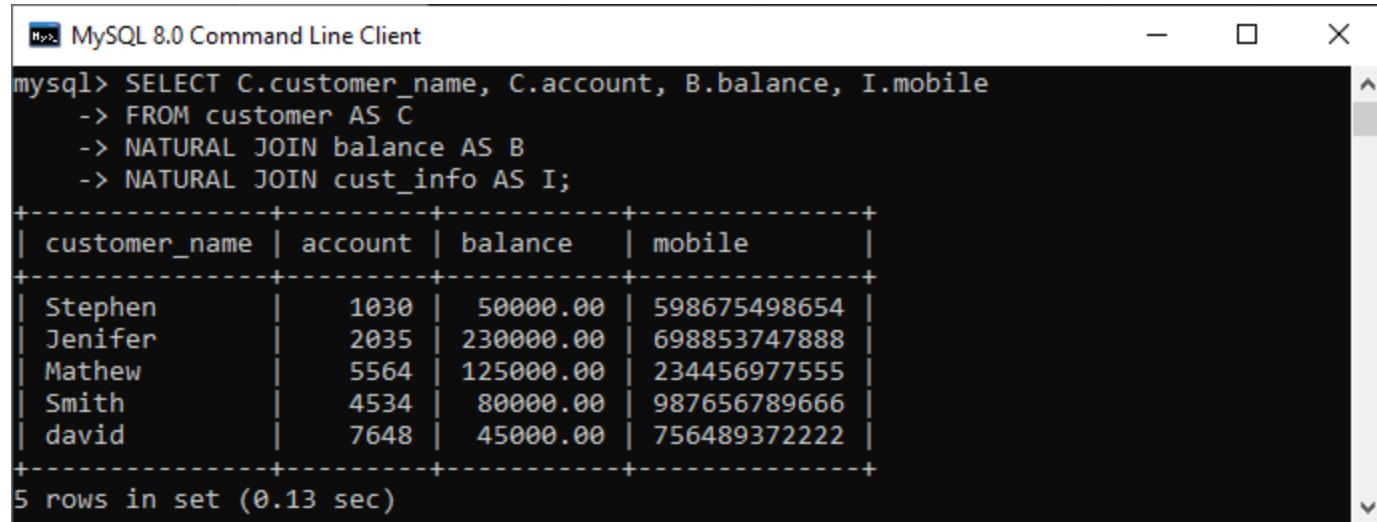


```
MySQL 8.0 Command Line Client
mysql> SELECT * FROM cust_info;
+-----+
| account | mobile      | address          |
+-----+
| 1030    | 598675498654 | 777 Brockton Avenue, Abington MA 251 |
| 2035    | 698853747888 | 337 Russell St, Hadley MA 103 |
| 5564    | 234456977555 | 20 Soojian Dr, Leicester MA 154 |
| 4534    | 987656789666 | 780 Lynnway, Lynn MA 19 |
| 7648    | 756489372222 | 700 Oak Street, Brockton MA 23 |
+-----+
```

To join three tables using natural join, we need to execute the statement as follows:

1. mysql> **SELECT** C.customer\_name, C.account, B.balance, I.mobile
2. **FROM** customer **AS** C
3. NATURAL JOIN balance **AS** B
4. NATURAL JOIN cust\_info **AS** I;

It will give the below result. Here we can see that the account number is present in all three columns but arrived only once in the output that fulfills the natural join criteria.



```
MySQL 8.0 Command Line Client
mysql> SELECT C.customer_name, C.account, B.balance, I.mobile
-> FROM customer AS C
-> NATURAL JOIN balance AS B
-> NATURAL JOIN cust_info AS I;
+-----+-----+-----+-----+
| customer_name | account | balance | mobile |
+-----+-----+-----+-----+
| Stephen       | 1030   | 50000.00 | 598675498654 |
| Jenifer       | 2035   | 230000.00 | 698853747888 |
| Mathew        | 5564   | 125000.00 | 234456977555 |
| Smith          | 4534   | 80000.00  | 987656789666 |
| david          | 7648   | 45000.00  | 756489372222 |
+-----+-----+-----+-----+
5 rows in set (0.13 sec)
```

## Difference between Natural Join and Inner Join

SN	Natural Join	Inner Join
1.	It joins the tables based on the same column names and their data types.	It joins the tables based on the column name specified in the ON clause explicitly.
2.	It always returns unique columns in the result set.	It returns all the attributes of both tables along with duplicate columns that match the ON clause condition.
3.	If we have not specified any condition in this join, it returns the records based on the common columns.	It returns only those rows that exist in both tables.
4.	The syntax of natural join is given below: SELECT [column_names   *] FROM table_name1 NATURAL JOIN table_name2;	The syntax of inner join is given below: SELECT [column_names   *] FROM table_name1 INNER JOIN table_name2 ON table_name1.column_name = table_name2.column_name;

## Difference between Left Join and Right Join

MySQL has mainly two kinds of joins named LEFT JOIN and RIGHT JOIN. The main difference between these joins is the **inclusion of non-matched rows**. The LEFT JOIN includes all records from the left side and matched rows from the right table, whereas RIGHT JOIN returns all rows from the right side and unmatched rows from the left table. In this section, we are going to know the popular differences between LEFT and RIGHT join. Before exploring the comparison, let us first understand JOIN, LEFT JOIN, and RIGHT JOIN clause in MySQL.

### What is JOIN Clause?

A **join**

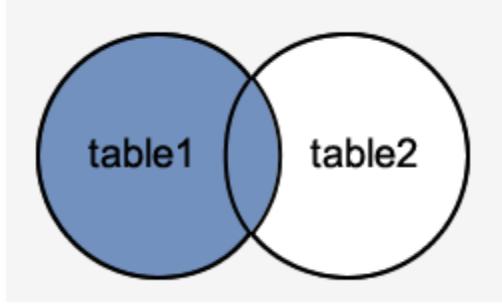
is used to query data from multiple tables and returns the combined result from two or more tables through a condition. The condition in the join clause indicates how columns are matched between the specified tables.

### What is the LEFT JOIN Clause?

The **Left Join clause**

joins two or more tables and returns all rows from the left table and matched records from the right table or returns null if it does not find any matching record. It is also known as **Left Outer Join**. So, Outer is the optional keyword to use with Left Join.

We can understand it with the following visual representation:



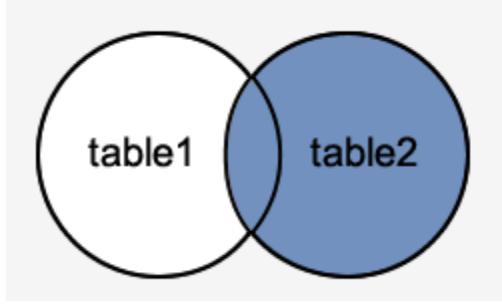
To read more information about the LEFT join, [click here](#)

## What is the RIGHT JOIN Clause?

The Right Join clause

joins two or more tables and returns all rows from the right-hand table, and only those results from the other table that fulfilled the specified join condition. If it finds unmatched records from the left side table, it returns Null value. It is also known as **Right Outer Join**. So, Outer is the optional clause to use with Right Join.

We can understand it with the following **visual representation**.



To read more information about RIGHT JOIN, [click here](#)

## Syntax of LEFT JOIN Clause

The following is the general syntax of LEFT JOIN:

1. **SELECT** column\_list **FROM** table\_name1
2. **LEFT JOIN** table\_name2
3. **ON** column\_name1 = column\_name2
4. **WHERE** join\_condition

The following is the general syntax of LEFT OUTER JOIN:

1. **SELECT** column\_list **FROM** table\_name1
2. **LEFT OUTER JOIN** table\_name2
3. **ON** column\_name1 = column\_name2
4. **WHERE** join\_condition

## Syntax of RIGHT JOIN Clause

The following is the general syntax of RIGHT JOIN:

1. **SELECT** column\_list **FROM** table\_name1

2. **RIGHT JOIN** table\_name2
3. **ON** column\_name1 = column\_name2
4. **WHERE** join\_condition

The following is the general syntax of RIGHT OUTER JOIN:

1. **SELECT** column\_list **FROM** table\_name1
2. **RIGHT OUTER JOIN** table\_name2
3. **ON** column\_name1 = column\_name2
4. **WHERE** join\_condition

## LEFT JOIN vs. RIGHT JOIN

The following comparison table explains their main differences in a quick manner:

<b>LEFT JOIN</b>	<b>RIGHT JOIN</b>
It joins two or more tables, returns all records from the left table, and matching rows from the right-hand table.	It is used to join two or more tables, returns all records from the right table, and matching rows from the left-hand table.
The result-set will contain null value if there is no matching row on the right side table.	The result-set will contain null value if there is no matching row on the left side table.
It is also known as LEFT OUTER JOIN.	It is also called as RIGHT OUTER JOIN.

## Example

Let us understand the differences between both joins through examples. Suppose we have a table named "**customer**" and "**orders**" that contains the following data:

**Table: customer**

cust_id	cust_name	city	occupation
1	Peter	London	Business
2	Joseph	Texas	Doctor
3	Mark	New Delhi	Engineer
4	Michael	New York	Scientist
5	Alexandar	Maxico	Student

**Table: orders**

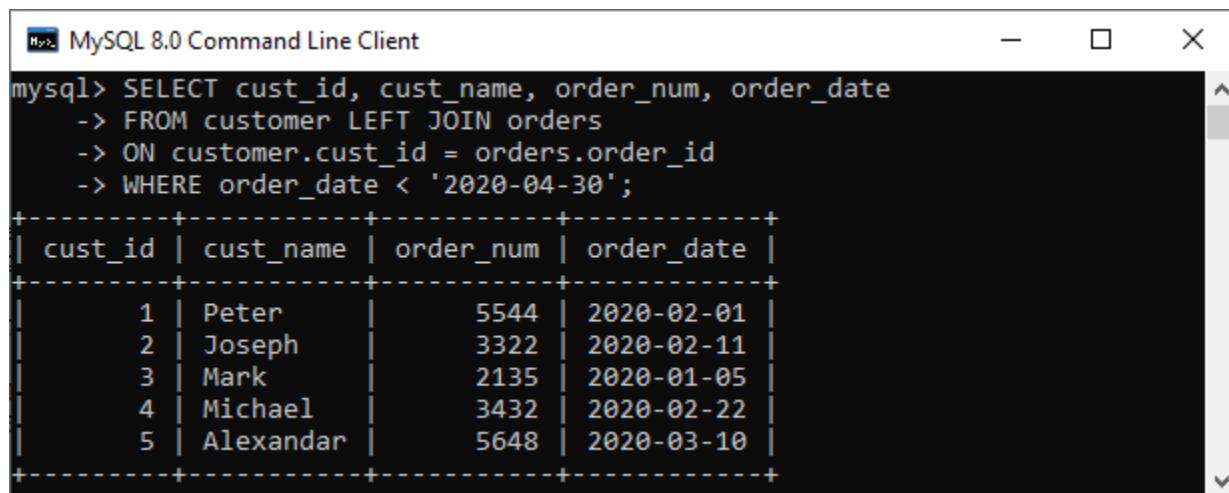
order_id	prod_name	order_num	order_date
1	Laptop	5544	2020-02-01
2	Mouse	3322	2020-02-11
3	Desktop	2135	2020-01-05
4	Mobile	3432	2020-02-22
5	Antivirus	5648	2020-03-10

### LEFT JOIN Example

Following SQL statement returns the matching records from both tables using the LEFT JOIN query:

1. **SELECT** cust\_id, cust\_name, order\_num, order\_date
2. **FROM** customer **LEFT JOIN** orders
3. **ON** customer.cust\_id = orders.order\_id
4. **WHERE** order\_date < '2020-04-30';

After successful execution of the query, we will get the output as follows:



MySQL 8.0 Command Line Client

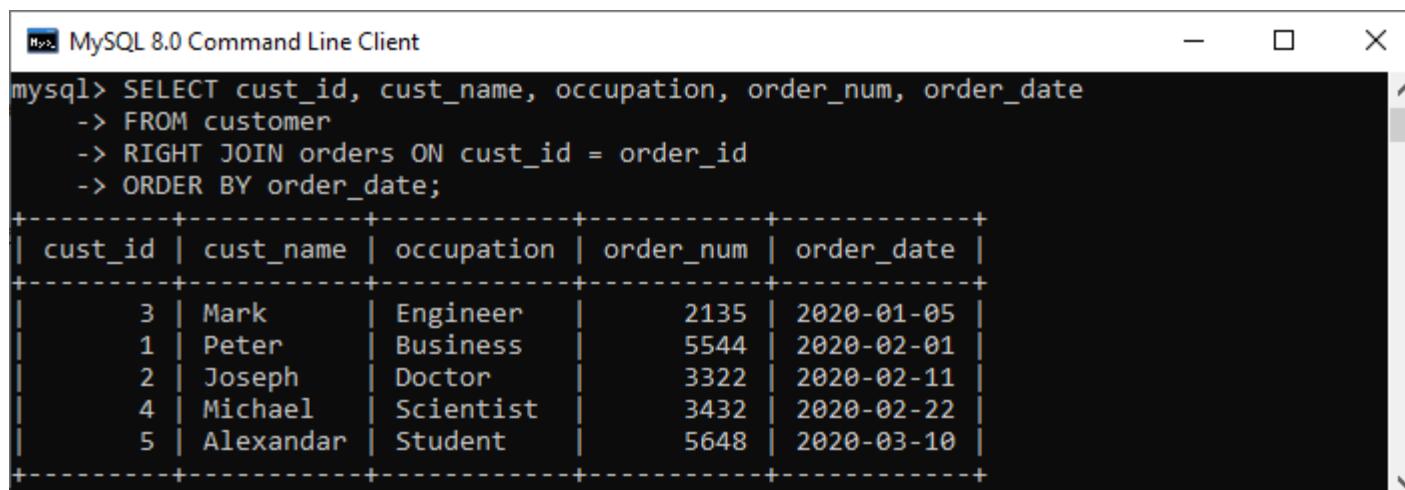
```
mysql> SELECT cust_id, cust_name, order_num, order_date
-> FROM customer LEFT JOIN orders
-> ON customer.cust_id = orders.order_id
-> WHERE order_date < '2020-04-30';
+-----+-----+-----+
| cust_id | cust_name | order_num | order_date |
+-----+-----+-----+
|      1 | Peter     |    5544 | 2020-02-01 |
|      2 | Joseph    |    3322 | 2020-02-11 |
|      3 | Mark      |    2135 | 2020-01-05 |
|      4 | Michael   |    3432 | 2020-02-22 |
|      5 | Alexandar |    5648 | 2020-03-10 |
+-----+-----+-----+
```

### RIGHT JOIN Example

Following SQL statement returns the matching records from both tables using the RIGHT JOIN query:

1. **SELECT** cust\_id, cust\_name, occupation, order\_num, order\_date
2. **FROM** customer
3. **RIGHT JOIN** orders **ON** cust\_id = order\_id
4. **ORDER BY** order\_date;

After successful execution of the query, we will get the output as follows:



MySQL 8.0 Command Line Client

```
mysql> SELECT cust_id, cust_name, occupation, order_num, order_date
-> FROM customer
-> RIGHT JOIN orders ON cust_id = order_id
-> ORDER BY order_date;
+-----+-----+-----+-----+
| cust_id | cust_name | occupation | order_num | order_date |
+-----+-----+-----+-----+
|      3 | Mark      | Engineer   |    2135 | 2020-01-05 |
|      1 | Peter     | Business   |    5544 | 2020-02-01 |
|      2 | Joseph    | Doctor     |    3322 | 2020-02-11 |
|      4 | Michael   | Scientist  |    3432 | 2020-02-22 |
|      5 | Alexandar | Student    |    5648 | 2020-03-10 |
+-----+-----+-----+-----+
```

## Difference between MySQL Union and Join

Union and Join are [SQL](#) clauses used to perform operations on more than one table in a relational database management system (RDBMS). They produce a result by **combining data** from two or more tables. But, the way of combining the data from two or more relations differ in both clauses. Before making a comparison, we are going to discuss in brief about these clauses.

### What is the Union clause?

[MySQL Union clause](#) allows us to combine two or more relations using multiple SELECT queries into a single result set. By default, it has a feature to remove the duplicate rows from the result set.

Union clause in [MySQL](#) must follow the rules given below:

- The order and number of the columns must be the same in all tables.
- The data type must be compatible with the corresponding positions of each select query.
- The column name in the SELECT queries should be in the same order.

### Syntax

1. **SELECT** column\_name(s) **FROM** table\_name1
2. **UNION**
3. **SELECT** column\_name(s) **FROM** table\_name2;

## What is the Join clause?

[Join in MySQL](#) is used with [SELECT statement](#) to **retrieve data** from multiple tables. It is performed whenever we need to fetch records from more than one tables. It returns only those records from the tables that match the specified conditions.

### Syntax

1. **SELECT** column\_name(s) **FROM** table\_name1
2. **JOIN** table\_name2 **ON** conditions;

## Union vs. Join

Let us discuss the essential differences between Union and [Join](#) using the following comparison chart.



SN	UNION	JOIN
1.	It is used to combine the result from multiple tables using SQL queries.	It is used to fetch the record from more than one table using SQL queries.
2.	It combines the records into new rows.	It combines the records into new columns.
3.	It allows us to connect the tables vertically.	It will enable us to join the tables vertically.
4.	It works as the conjunction of the more than one tables that sum ups all records.	It produces results in the intersection of the tables.
5.	In this, the order and number of the columns must be the same in all tables.	In this, the order and number of the columns do not need to be the same in all tables.
6.	It has a default feature to remove the duplicate rows from the result set.	It does not eliminate the duplicate rows from the result set.
7.	In this, the data type must be the same in all SELECT statements.	In this, there is no need to be the same data type. It can be different.
8.	The Union clause is applicable only when the number of columns and corresponding attributes has the same domain.	The Join clause is applicable only when the two tables that are going to be used have at least one column.
9.	The Union clause can have mainly two types that are given below: <ul style="list-style-type: none"><li>o Union</li><li>o Union All</li></ul>	The Join clause can have different types that are given below: <ul style="list-style-type: none"><li>o Inner Join (Sometimes Join)</li><li>o Left Join (Left Outer Join)</li><li>o Right Join (Right Outer Join)</li><li>o Full Join (Outer Join)</li></ul>

Now, we are going to understand it with the help of an example.

### Union Example

Suppose our database has the following tables: "**Student1**" and "**Student2**" that contains the data below:

stud_id	stud_name	subject	marks
1	Mark	English	68
2	Joseph	Physics	70
3	John	Maths	70
4	Barack	Maths	90
5	Rinky	Maths	85
6	Adam	Science	92
7	Andrew	Science	83
8	Brayan	Science	85

stud_id	stud_name	subject	marks
1	Donald	History	85
2	Joseph	Physics	70
3	Stephen	Geography	82
4	Abraham	Java	75
5	John	Maths	70

The following statement produces output that contains all student names and subjects by combining both tables.

1. **SELECT** stud\_name, subject **FROM** student1
2. **UNION**
3. **SELECT** stud\_name, subject **FROM** student2;

After the successful execution, we will get the output that contains all unique student names and subjects:

stud_name	subject
Mark	English
Joseph	Physics
John	Maths
Barack	Maths
Rinky	Maths
Adam	Science
Andrew	Science
Brayan	Science
Donald	History
Stephen	Geography
Abraham	Java

## Join Example

Suppose our database has the following tables: "**Students**" and "**Technologies**" that contains the data below:

student_id	stud_fname	stud_lname	city
1	Devine	Putin	France
2	Michael	Clark	Australiya
3	Ethon	Miller	England
4	Mark	Strauss	America

stud_fname	stud_lname	city	technology
Devine	Putin	France	Java
Michael	Clark	Australiya	Angular
Ethon	Miller	England	Big Data
Mark	Strauss	America	IOS

We can fetch records from both tables using the following query:

1. **SELECT** students.stud\_fname, students.stud\_lname, students.city, technologies.technology
2. **FROM** students
3. **JOIN** technologies
4. **ON** students.student\_id = technologies.tech\_id;

We will get the following output:

student_id	inst_name	city	technology
1	Java Training Inst.	France	Java
2	Croma Campus	Australiya	Angular
3	CETPA Infotech	England	Big Data
4	Aptron	America	IOS

## MySQL Key

## MySQL Unique Key

A unique key in MySQL is a single field or combination of fields that ensure all values going to store into the column will be unique. It means a column cannot stores **duplicate values**. For example, the email addresses and roll numbers of students in the "student\_info" table or contact number of employees in the "Employee" table should be unique.

MySQL allows us to use more than one column with UNIQUE constraint in a table. It can accept a **null** value, but MySQL allowed only one null value per column. It ensures the **integrity** of the column or group of columns to store different values into a table.

## Needs of Unique Key

- It is useful in preventing the two records from storing identical values into the column.
- It stores only distinct values that maintain the integrity and reliability of the database for accessing the information in an organized way.
- It also works with a foreign key in preserving the uniqueness of a table.
- It can contain null value into the table.

## Syntax

The following syntax is used to create a unique key in [MySQL](#).

If we want to create only one unique key column into a table, use the syntax as below:

1. **CREATE TABLE** table\_name(
2.    col1 datatype,
3.    col2 datatype **UNIQUE**,
4.    ...
5. );

If we want to create more than one unique key column into a table, use the syntax as below:

1. **CREATE TABLE** table\_name(
2.    col1 col\_definition,
3.    col2 col\_definition,
4.    ...
5.    [**CONSTRAINT** constraint\_name]
6.    **UNIQUE**(column\_name(s))
7. );

If we have not specified the name for a unique constraint, MySQL generates a name for this column automatically. So, it is recommended to use the constraint name while creating a table.

## Parameter Explanation

The following table explains the parameters in detail.

Parameter Name	Descriptions
table_name	It is the name of the table that we are going to create.
col1, col2	It is the column names that contain in the table.
constraint_name	It is the name of the unique key.
column_name(s)	It is the column name(s) that is going to be a unique key.

## Unique Key Example

The following example explains how a unique key used in MySQL.

This statement creates a table "**Student2**" with a UNIQUE constraint:

1. **CREATE TABLE** Student2 (
2.    Stud\_ID **int** NOT NULL **UNIQUE**,
3.    **Name varchar**(45),
4.    Email **varchar**(45),
5.    Age **int**,
6.    City **varchar**(25)
7. );

Next, execute the insert queries listed below to understand how it works:

1. mysql> **INSERT INTO** Student2 (Stud\_ID, **Name**, Email, Age, City)
2. **VALUES** (1, 'Peter', 'peter@javatpoint.com', 22, 'Texas'),
3. (2, 'Suzi', 'suzi@javatpoint.com', 24, 'California'),
4. (3, 'Joseph', 'joseph@javatpoint.com', 23, 'Alaska');
- 5.
6. mysql> **INSERT INTO** Student2 (Stud\_ID, **Name**, Email, Age, City)
7. **VALUES** (1, 'Stephen', 'stephen@javatpoint.com', 22, 'Texas');

## Output

In the below output, we can see that the first **INSERT query** executes correctly, but the second statement fails and gives an error that says: Duplicate entry '1' for key Stud\_ID.

```

MySQL 8.0 Command Line Client

mysql> CREATE TABLE Student2 (
-> Stud_ID int NOT NULL UNIQUE,
-> Name varchar(45),
-> Email varchar(45),
-> Age int,
-> City varchar(25)
-> );
Query OK, 0 rows affected (0.67 sec)

mysql> INSERT INTO Student2 (Stud_ID, Name, Email, Age, City)
-> VALUES (1, 'Peter', 'peter@javatpoint.com', 22, 'Texas'),
-> (2, 'Suzi', 'suzi@javatpoint.com', 24, 'California'),
-> (3, 'Joseph', 'joseph@javatpoint.com', 23, 'Alaska');
Query OK, 3 rows affected (0.13 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> INSERT INTO Student2 (Stud_ID, Name, Email, Age, City)
-> VALUES (1, 'Stephen', 'stephen@javatpoint.com', 22, 'Texas');
ERROR 1062 (23000): Duplicate entry '1' for key 'student2.Stud_ID'

```

If you want to define the unique key on **multiple columns**, use the query as below:

1. **CREATE TABLE** Student3 (
2. Stud\_ID **int**,
3. Roll\_No **int**,
4. **Name varchar**(45) NOT NULL,
5. Email **varchar**(45),
6. Age **int**,
7. City **varchar**(25),
8. **CONSTRAINT uc\_rollno\_email Unique**(Roll\_No, Email)
9. );

In the output, we can see that the unique key value contains two columns that are **Roll\_No** and **Email**.

```

MySQL 8.0 Command Line Client

mysql> CREATE TABLE Student3 (
-> Stud_ID int,
-> Roll_No int,
-> Name varchar(45) NOT NULL,
-> Email varchar(45),
-> Age int,
-> City varchar(25),
-> CONSTRAINT uc_rollno_email Unique(Roll_No, Email)
-> );
Query OK, 0 rows affected (1.24 sec)

```

To verify this, execute the following statement:

1. mysql> **SHOW INDEX FROM** Student3;

Here, we can see that the unique constraint has successfully added into the table:

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type
student3	0	uc_rollno_email	1	Roll_No	A	0	NULL	NULL	YES	BTREE
student3	0	uc_rollno_email	2	Email	A	0	NULL	NULL	YES	BTREE

## DROP Unique Key

The ALTER TABLE statement also allows us to drop the unique key from the table. The following syntax is used to drop the unique key:

1. **ALTER TABLE** table\_name **DROP INDEX** constraint\_name;

In the above syntax, the **table\_name** is the name of the table that we want to modify, and **constraint\_name** is the name of the unique key we are going to remove.

#### Example

This statement will remove the **uc\_rollno\_email** constraint from the table permanently.

1. mysql> **ALTER TABLE** Student3 **DROP INDEX** uc\_rollno\_email;

We can execute the SHOW INDEX statement to verify this.

### Unique Key Using ALTER TABLE Statement

This statement allows us to do the modification into the existing table. Sometimes we want to add a unique key to the column of an existing table; then, this statement is used to add the unique key for that column.

#### Syntax

Following are the syntax of the [ALTER TABLE statement](#) to add a unique key:

1. **ALTER TABLE** table\_name **ADD CONSTRAINT** constraint\_name **UNIQUE**(column\_list);

#### Example

This statement creates a table "**Students3**" that have no unique key column into the table definition.

1. **CREATE TABLE** Student3 (
2. Stud\_ID **int**,
3. Roll\_No **int**,
4. Name **varchar**(45) NOT NULL,
5. Email **varchar**(45),
6. Age **int**,
7. City **varchar**(25)
8. );

After creating a table, if we want to add a unique key to this table, we need to execute the ALTER TABLE statement as below:

1. mysql> **ALTER TABLE** Student3 **ADD CONSTRAINT** uc\_rollno\_email **UNIQUE**(Roll\_No, Email);

We can see the output where both statements executed successfully.

The screenshot shows the MySQL 8.0 Command Line Client interface. The command line displays two SQL statements. The first statement creates a table 'Student3' with columns Stud\_ID, Roll\_No, Name, Email, Age, and City. The second statement adds a unique constraint 'uc\_rollno\_email' to the 'Student3' table, indexed on the 'Roll\_No' and 'Email' columns. Both statements return 'Query OK' and show 0 rows affected.

```
MySQL 8.0 Command Line Client
mysql> CREATE TABLE Student3 (
-> Stud_ID int,
-> Roll_No int,
-> Name varchar(45) NOT NULL,
-> Email varchar(45),
-> Age int,
-> City varchar(25)
-> );
Query OK, 0 rows affected (1.06 sec)

mysql> ALTER TABLE Student3 ADD CONSTRAINT uc_rollno_email UNIQUE(Roll_No, Email);
Query OK, 0 rows affected (0.68 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

To verify this, execute the following statement:

1. mysql> **SHOW INDEX FROM** Student3;

Here, we can see that the unique constraint has successfully added into the table:

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type
student3	0	uc_rollno_email	1	Roll_No	A	0	NULL	NULL	YES	BTREE
student3	0	uc_rollno_email	2	Email	A	0	NULL	NULL	YES	BTREE

# MySQL Primary Key

MySQL primary key is a single or combination of the field, which is used to identify each record in a table **uniquely**. If the column contains primary key constraints, then it cannot be **null or empty**. A table may have duplicate columns, but it can contain only one primary key. It always contains unique value into a column.

When you insert a new row into the table, the primary key column can also use the **AUTO\_INCREMENT** attribute to generate a sequential number for that row automatically. MySQL automatically creates an index named "**Primary**" after defining a primary key into the table. Since it has an associated index, we can say that the primary key makes the query performance fast.

## Rules for Primary key

Following are the rules for the primary key:

1. The primary key column value must be unique.
2. Each table can contain only one primary key.
3. The primary key column cannot be null or empty.
4. MySQL does not allow us to insert a new row with the existing primary key.
5. It is recommended to use INT or BIGINT data type for the primary key column.

We can create a primary key in two ways:

- o CREATE TABLE Statement
- o ALTER TABLE Statement

Let us discuss each one in detail.

## Primary Key Using CREATE TABLE Statement

In this section, we are going to see how a primary key is created using the **CREATE TABLE** statement.

### Syntax

The following are the syntax used to create a primary key in MySQL.

If we want to create only one primary key column into the table, use the below syntax:

1. **CREATE TABLE** table\_name(
2. col1 datatype **PRIMARY KEY**,
3. col2 datatype,
4. ...
5. );

If we want to create more than one primary key column into the table, use the below syntax:

1. **CREATE TABLE** table\_name
2. (
3. col1 col\_definition,
4. col2 col\_definition,
5. ...
- 6.
7. **CONSTRAINT** [constraint\_name]
8. **PRIMARY KEY** (column\_name(s))
9. );

## Parameter Explanation

The following table explains the parameters in detail.

Parameter Name	Descriptions
Table_name	It is the name of the table that we are going to create.

Col1, col2	It is the column names that contain in the table.
Constraint_name	It is the name of the primary key.
Column_name(s)	It is the column name(s) that is going to be a primary key.

## Primary Key Example

The following example explains how a primary key used in MySQL.

This statement creates a table named "**Login**" whose "**login\_id**" column contains the primary key:

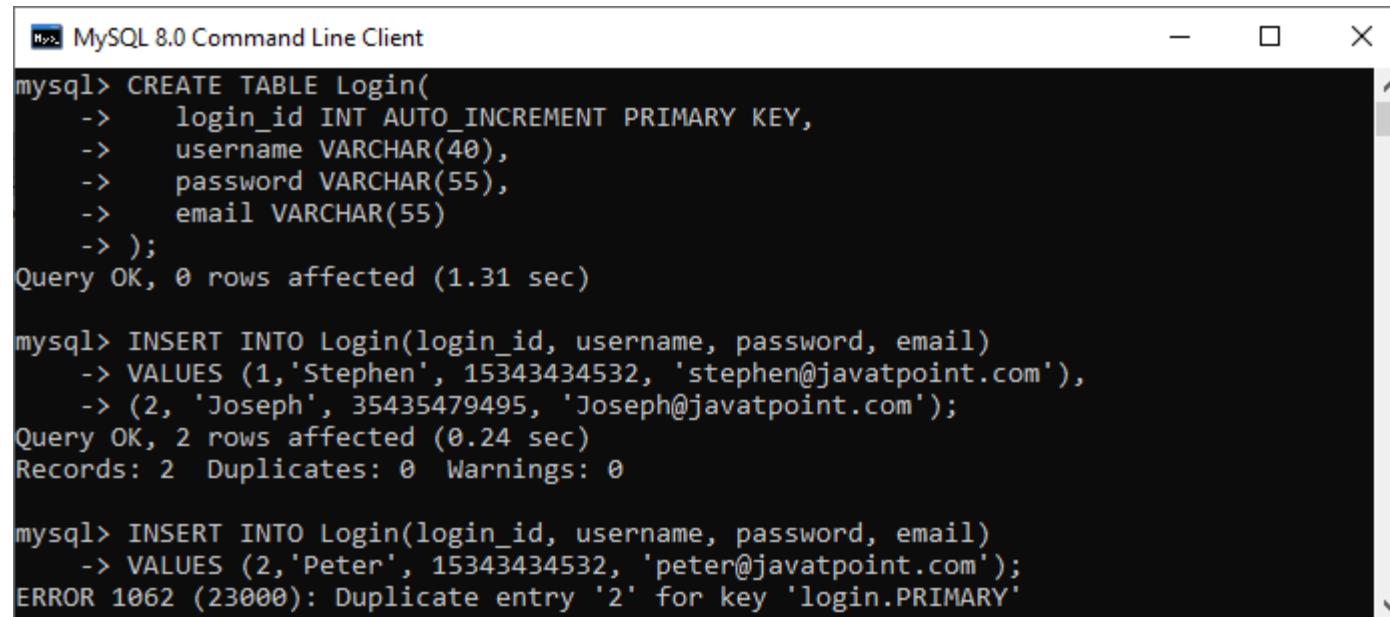
1. Mysql> **CREATE TABLE** Login(
2.   login\_id **INT AUTO\_INCREMENT PRIMARY KEY**,
3.   username **VARCHAR**(40),
4.   **password VARCHAR**(55),
5.   email **VARCHAR**(55)
6. );

Next, use the insert query to store data into a table:

1. mysql> **INSERT INTO** Login(login\_id, username, **password**, email)
2. **VALUES** (1,'Stephen', 15343434532, 'stephen@javatpoint.com'),
3. (2, 'Joseph', 35435479495, 'Joseph@javatpoint.com');
- 4.
5. mysql> **INSERT INTO** Login(login\_id, username, **password**, email)
6. **VALUES** (1,'Peter', 15343434532, 'peter@javatpoint.com');

## Output

In the below output, we can see that the first insert query executes successfully. While the second insert statement fails and gives an error that says: Duplicate entry for the primary key column.



```

MySQL 8.0 Command Line Client
mysql> CREATE TABLE Login(
->   login_id INT AUTO_INCREMENT PRIMARY KEY,
->   username VARCHAR(40),
->   password VARCHAR(55),
->   email VARCHAR(55)
-> );
Query OK, 0 rows affected (1.31 sec)

mysql> INSERT INTO Login(login_id, username, password, email)
-> VALUES (1,'Stephen', 15343434532, 'stephen@javatpoint.com'),
-> (2, 'Joseph', 35435479495, 'Joseph@javatpoint.com');
Query OK, 2 rows affected (0.24 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> INSERT INTO Login(login_id, username, password, email)
-> VALUES (2,'Peter', 15343434532, 'peter@javatpoint.com');
ERROR 1062 (23000): Duplicate entry '2' for key 'login.PRIMARY'

```

If you want to define the primary key on **multiple columns**, use the query as below:

1. mysql> **CREATE TABLE** Students (
2.   Student\_ID **int**,
3.   Roll\_No **int**,
4.   **Name varchar**(45) NOT NULL,
5.   Age **int**,
6.   City **varchar**(25),
7.   **Primary Key**(Student\_ID, Roll\_No)
8. );

In the output, we can see that the primary key value contains two columns that are **Student\_ID** and **Roll\_No**.

```
MySQL 8.0 Command Line Client
mysql> CREATE TABLE Students (
-> Student_ID int,
-> Roll_No int,
-> Name varchar(45) NOT NULL,
-> Age int,
-> City varchar(25),
-> Primary Key(Student_ID, Roll_No)
-> );
Query OK, 0 rows affected (1.48 sec)
```

## Primary Key Using ALTER TABLE Statement

This statement allows us to do the modification into the existing table. When the table does not have a primary key, this statement is used to add the primary key to the column of an existing table.

### Syntax

Following are the syntax of the ALTER TABLE statement to create a primary key in MySQL:

1. **ALTER TABLE** table\_name **ADD PRIMARY KEY**(column\_list);

### Example

The following statement creates a table "Persons" that have no primary key column into the table definition.

1. mysql> **CREATE TABLE** Persons (
2.     Person\_ID **int** NOT NULL,
3.     **Name varchar**(45),
4.     Age **int**,
5.     City **varchar**(25)
6. );

After creating a table, if we want to add a primary key to this table, we need to execute the ALTER TABLE statement as below:

1. mysql> **ALTER TABLE** Persons **ADD PRIMARY KEY**(Person\_ID);

We can see the output where both statements executed successfully.

```
MySQL 8.0 Command Line Client
mysql> CREATE TABLE Persons (
-> Person_ID int NOT NULL,
-> Name varchar(45),
-> Age int,
-> City varchar(25)
-> );
Query OK, 0 rows affected (0.94 sec)

mysql> ALTER TABLE Persons ADD PRIMARY KEY(Person_ID);
Query OK, 0 rows affected (1.84 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

If the table needs to add the primary key into a table that already has data into the column, then it must be sure to the column does not contains duplicates or null values.

## DROP Primary Key

The ALTER TABLE statement also allows us to drop the primary key from the table. The following syntax is used to drop the primary key:

1. **ALTER TABLE** table\_name **DROP PRIMARY KEY**;

### Example

1. mysql> **ALTER TABLE** Login **DROP PRIMARY KEY**;

## Primary Key vs. Unique Key

The following comparison chart explains some of the common differences between both of them:

SN	Primary Key	Unique Key
1.	It is a single or combination of the field, which is used to identify each record in a table uniquely.	It also determines each row of the table uniquely in the absence of a primary key.
2.	It does not allow to store a NULL value into the primary key column.	It can accept only one NULL value into the unique key column.
3.	A table can have only one primary key.	A table can have more than one unique key.
4.	It creates a clustered index.	It creates a non-clustered index.

## MySQL Foreign Key

The foreign key is used to link one or more than one table together. It is also known as the **referencing** key. A foreign key matches the primary key field of another table. It means a foreign key field in one table refers to the primary key field of the other table. It identifies each row of another table uniquely that maintains the **referential integrity** in MySQL.

A foreign key makes it possible to create a parent-child relationship with the tables. In this relationship, the parent table holds the initial column values, and column values of child table reference the parent column values. MySQL allows us to define a foreign key constraint on the child table.

[MySQL](#) defines the foreign key in two ways:

1. Using CREATE TABLE Statement
2. Using ALTER TABLE Statement

### Syntax

Following are the basic syntax used for defining a foreign key using CREATE TABLE OR ALTER TABLE statement in the MySQL:

1. [**CONSTRAINT** constraint\_name]
2. **FOREIGN KEY** [foreign\_key\_name] (col\_name, ...)
3. **REFERENCES** parent\_tbl\_name (col\_name,...)
4. **ON DELETE** referenceOption
5. **ON UPDATE** referenceOption

In the above syntax, we can see the following parameters:

**constraint\_name:** It specifies the name of the foreign key constraint. If we have not provided the constraint name, MySQL generates its name automatically.

**col\_name:** It is the names of the column that we are going to make foreign key.

**parent\_tbl\_name:** It specifies the name of a parent table followed by column names that reference the foreign key columns.

**Reference\_option:** It is used to ensure how foreign key maintains referential integrity using ON DELETE and ON UPDATE clause between parent and child table.

MySQL contains **five** different referential options, which are given below:

**CASCADE:** It is used when we delete or update any row from the parent table, the values of the matching rows in the child table will be deleted or updated automatically.

**SET NULL:** It is used when we delete or update any row from the parent table, the values of the foreign key columns in the child table are set to NULL.

**RESTRICT:** It is used when we delete or update any row from the parent table that has a matching row in the reference(child) table, MySQL does not allow to delete or update rows in the parent table.

**NO ACTION:** It is similar to RESTRICT. But it has one difference that it checks referential integrity after trying to modify the table.

**SET DEFAULT:** The MySQL parser recognizes this action. However, the InnoDB and NDB tables both rejected this action.

**NOTE:** MySQL mainly provides full support to CASCADE, RESTRICT, and SET NULL actions. If we have not specified the ON DELETE and ON UPDATE clause, MySQL takes default action RESTRICT.

## Foreign Key Example

Let us understand how foreign key works in MySQL. So first, we are going to create a database named "mysqltestdb" and start using it with the command below:

1. mysql> **CREATE DATABASE** mysqltestdb;
2. mysql> use mysqltestdb;

Next, we need to create two tables named "**customer**" and "**contact**" using the below statement:

### Table: customer

1. **CREATE TABLE** customer (
2. ID **INT** NOT NULL AUTO\_INCREMENT,
3. Name **varchar**(50) NOT NULL,
4. City **varchar**(50) NOT NULL,
5. **PRIMARY KEY** (ID)
6. );

### Table: contact

1. **CREATE TABLE** contact (
2. ID **INT**,
3. Customer\_Id **INT**,
4. Customer\_Info **varchar**(50) NOT NULL,
5. Type **varchar**(50) NOT NULL,
6. **INDEX** par\_ind (Customer\_Id),
7. **CONSTRAINT** fk\_customer **FOREIGN KEY** (Customer\_Id)
8. **REFERENCES** customer(ID)
9. **ON DELETE CASCADE**
10. **ON UPDATE CASCADE**
11. );

## Table Structure Verification

Here, we are going to see how our database structure looks like using the following queries:

1. mysql> SHOW TABLES;
2. mysql> DESCRIBE customer;
3. mysql> DESCRIBE contact;

We will get the structure as below:

```

MySQL 8.0 Command Line Client
mysql> SHOW TABLES;
+-----+
| Tables_in_mysqltestdb |
+-----+
| contact
| customer
+-----+
2 rows in set (0.00 sec)

mysql> DESCRIBE customer;
+-----+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra       |
+-----+-----+-----+-----+-----+-----+
| ID    | int    | NO   | PRI  | NULL    | auto_increment |
| Name  | varchar(50) | NO   |      | NULL    |              |
| City  | varchar(50) | NO   |      | NULL    |              |
+-----+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)

mysql> DESCRIBE contact;
+-----+-----+-----+-----+-----+-----+
| Field     | Type   | Null | Key | Default | Extra       |
+-----+-----+-----+-----+-----+-----+
| ID        | int    | YES  |      | NULL    |              |
| Customer_Id | int    | YES  | MUL  | NULL    |              |
| Customer_Info | varchar(50) | NO   |      | NULL    |              |
| Type      | varchar(50) | NO   |      | NULL    |              |
+-----+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)

```

In the above output, we can see that the **PRI** in the key column of the customer table tells that this field is the primary index value. Next, the **MUL** in the key column of the contact value tells that the **Customer\_Id** field can store multiple rows with the same value.

## Insert Data to the Table

Now, we have to insert the records into both tables. Execute this statement to insert data into table customer:

1. **INSERT INTO** customer(**Name**, City) **VALUES**
2. ('Joseph', 'California'),
3. ('Mary', 'NewYork'),
4. ('John', 'Alaska');

After insertion, execute the **SELECT TABLE** command to check the customer table data as below:

```

MySQL 8.0 Command Line Client
mysql> SELECT * FROM customer;
+----+-----+-----+
| ID | Name  | City   |
+----+-----+-----+
| 1  | Joseph | California |
| 2  | Mary   | NewYork |
| 3  | John   | Alaska  |
+----+-----+-----+
3 rows in set (0.00 sec)

```

Execute the below insert statement to add data into a table contact:

1. **INSERT INTO** contact (Customer\_Id, Customer\_Info, Type) **VALUES**
2. (1, 'Joseph@javatpoint.com', 'email'),
3. (1, '121-121-121', 'work'),
4. (1, '123-123-123', 'home'),
5. (2, 'Mary@javatpoint.com', 'email'),
6. (2, 'Mary@javatpoint.com', 'email'),
7. (2, '212-212-212', 'work'),
8. (3, 'John@javatpoint.com', 'email'),
9. (3, '313-313-313', 'home');

Our contact table looks like as below:

```

MySQL 8.0 Command Line Client
mysql> SELECT * FROM contact;
+----+-----+-----+-----+
| ID | Customer_Id | Customer_Info      | Type   |
+----+-----+-----+-----+
| NULL | 1 | Joseph@javatpoint.com | email  |
| NULL | 1 | 121-121-121          | work   |
| NULL | 1 | 123-123-123          | home   |
| NULL | 2 | Mary@javatpoint.com  | email  |
| NULL | 2 | Mary@javatpoint.com  | email  |
| NULL | 2 | 212-212-212          | work   |
| NULL | 3 | John@javatpoint.com  | email  |
| NULL | 3 | 313-313-313          | home   |
+----+-----+-----+-----+
8 rows in set (0.00 sec)

```

Now, let's see how foreign keys in MySQL preserve data integrity.

So here, we are going to delete the referential data that removes records from both tables. We have defined the foreign key in the contact table as:

1. **FOREIGN KEY** (Customer\_Id) **REFERENCES** customer(ID)
2. **ON DELETE CASCADE**
3. **ON UPDATE CASCADE**.

It means if we delete any customer record from the customer table, then the related records in the contact table should also be deleted. And the ON UPDATE CASCADE will update automatically on the parent table to referenced fields in the child table(Here, it is Customer\_Id).

Execute this statement that deletes a record from the table whose name is **JOHN**.

1. mysql> **DELETE FROM** customer **WHERE Name='John'**;

Again, if we look at our tables, we can see that both tables were changed. It means the fields with name JOHN will be removed entirely from both tables.

```

MySQL 8.0 Command Line Client
mysql> SELECT * FROM customer;
+----+-----+-----+
| ID | Name    | City   |
+----+-----+-----+
| 1  | Joseph  | California |
| 2  | Mary    | NewYork  |
+----+-----+-----+
2 rows in set (0.00 sec)

mysql> SELECT * FROM contact;
+----+-----+-----+-----+
| ID | Customer_Id | Customer_Info      | Type   |
+----+-----+-----+-----+
| NULL | 1 | Joseph@javatpoint.com | email  |
| NULL | 1 | 121-121-121          | work   |
| NULL | 1 | 123-123-123          | home   |
| NULL | 2 | Mary@javatpoint.com  | email  |
| NULL | 2 | Mary@javatpoint.com  | email  |
| NULL | 2 | 212-212-212          | work   |
+----+-----+-----+-----+
6 rows in set (0.00 sec)

```

Now, test the **ON UPDATE CASCADE**. Here, we are going to update the Customer\_Id of **Mary** in the contact table as:

1. mysql> **UPDATE** customer **SET id=3 WHERE Name='Mary'**;

Again, if we look at our tables, we can see that both tables were changed with Customer\_Id of Mary=3.

```

MySQL 8.0 Command Line Client
mysql> SELECT * FROM customer;
+----+-----+-----+
| ID | Name  | City   |
+----+-----+-----+
| 1  | Joseph | California |
| 3  | Mary   | NewYork  |
+----+-----+-----+
2 rows in set (0.00 sec)

mysql> SELECT * FROM contact;
+----+-----+-----+-----+
| ID | Customer_Id | Customer_Info | Type  |
+----+-----+-----+-----+
| NULL |          1 | Joseph@javatpoint.com | email |
| NULL |          1 | 121-121-121 | work  |
| NULL |          1 | 123-123-123 | home  |
| NULL |          3 | Mary@javatpoint.com | email |
| NULL |          3 | Mary@javatpoint.com | email |
| NULL |          3 | 212-212-212 | work  |
+----+-----+-----+-----+
6 rows in set (0.00 sec)

```

## Foreign Key example using SET NULL action

Here, we are going to understand how the SET NULL action works with a foreign key. First, we have to create two table named **Persons** and **Contacts**, as shown below:

**Table: Persons**

1. **CREATE TABLE** Persons (
2. ID **INT** NOT NULL AUTO\_INCREMENT,
3. **Name varchar**(50) NOT NULL,
4. City **varchar**(50) NOT NULL,
5. **PRIMARY KEY** (ID)
6. );

**Table: Customers**

1. **CREATE TABLE** Contacts (
2. ID **INT**,
3. Person\_Id **INT**,
4. Info **varchar**(50) NOT NULL,
5. Type **varchar**(50) NOT NULL,
6. **INDEX** par\_ind (Person\_Id),
7. **CONSTRAINT** fk\_person **FOREIGN KEY** (Person\_Id)
8. **REFERENCES** Persons(ID)
9. **ON DELETE SET** NULL
10. **ON UPDATE SET** NULL
11. );

Next, we need to insert the data into both tables using the following statement:

1. **INSERT INTO** Persons(**Name**, City) **VALUES**
2. ('Joseph', 'Texas'),
3. ('Mary', 'Arizona'),
4. ('Peter', 'Alaska');
  
1. **INSERT INTO** Contacts (Person\_Id, Info, Type) **VALUES**
2. (1, 'joseph@javatpoint.com', 'email'),
3. (1, '121-121-121', 'work' ),
4. (2, 'mary@javatpoint.com', 'email'),
5. (2, '212-212-212', 'work'),
6. (3, 'peter@javatpoint.com', 'email'),
7. (3, '313-313-313', 'home');

Now, update the ID of the "Persons" table:

1. mysql> **UPDATE** Persons **SET** ID=103 **WHERE** ID=3;

Finally, verify the update using the SELECT statement given below:

The screenshot shows the MySQL 8.0 Command Line Client interface. It displays two SQL queries:

```
mysql> SELECT * FROM Persons;
+----+-----+-----+
| ID | Name | City |
+----+-----+-----+
| 1  | Joseph | Texas |
| 2  | Mary   | Arizona |
| 103 | Peter  | Alaska |
+----+-----+-----+
3 rows in set (0.00 sec)

mysql> SELECT * FROM Contacts;
+----+-----+-----+-----+
| ID | Person_Id | Info          | Type  |
+----+-----+-----+-----+
| NULL |      1    | joseph@javatpoint.com | email |
| NULL |      1    | 121-121-121       | work   |
| NULL |      2    | mary@javatpoint.com | email  |
| NULL |      2    | 212-212-212       | work   |
| NULL |    NULL   | peter@javatpoint.com | email  |
| NULL |    NULL   | 313-313-313       | home   |
+----+-----+-----+-----+
6 rows in set (0.00 sec)
```

If we look at our tables, we can see that both tables were changed. The rows with a **Person\_Id=3** in the Contacts table automatically set to **NULL** due to the ON UPDATE SET NULL action.

## How to DROP Foreign Key

MySQL allows the ALTER TABLE statement to remove an existing foreign key from the table. The following syntax is used to drop a foreign key:

1. **ALTER TABLE** table\_name **DROP FOREIGN KEY** fk\_constraint\_name;

Here, the **table\_name** is the name of a table from where we are going to remove the foreign key. The **constraint\_name** is the name of the foreign key that was added during the creation of a table.

If we have not known the name of an existing foreign key into the table, execute the following command:

1. mysql> SHOW **CREATE TABLE** contact;

It will give the output as below where we can see that the table contact has one foreign key named fk\_customer shown in the red rectangle.

Table	Create Table
contact	CREATE TABLE `contact` (   `ID` int DEFAULT NULL,   `Customer_Id` int DEFAULT NULL,   `Customer_Info` varchar(50) NOT NULL,   `Type` varchar(50) NOT NULL,   KEY `par_ind` (`Customer_Id`),   CONSTRAINT `fk_customer` FOREIGN KEY (`Customer_Id`) REFERENCES `customer` (`ID`) ON   DELETE CASCADE ON UPDATE CASCADE ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci

Now, to delete this foreign key constraint from the contact table, execute the statement as below:

1. mysql> **ALTER TABLE** contact **DROP FOREIGN KEY** fk\_customer;

We can verify whether foreign key constraint removes or not, use the SHOW CREATE TABLE statement. It will give the output as below where we can see that the foreign key is no longer available in the table contact.

Table	Create Table
contact	CREATE TABLE `contact` (   `ID` int DEFAULT NULL,   `Customer_Id` int DEFAULT NULL,   `Customer_Info` varchar(50) NOT NULL,   `Type` varchar(50) NOT NULL,   KEY `par_ind` (`Customer_Id`) ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci

## Define Foreign Key Using ALTER TABLE Statement

This statement allows us to do the modification into the existing table. Sometimes there is a need to add a foreign key to the column of an existing table; then, this statement is used to add the foreign key for that column.

## Syntax

Following are the syntax of the ALTER TABLE statement to add a foreign key in the existing table:

1. **ALTER TABLE** table\_name
2. **ADD [CONSTRAINT [symbol]] FOREIGN KEY**
3. [index\_name] (column\_name, ...)
4. **REFERENCES** table\_name (column\_name,...)
5. **ON DELETE** referenceOption
6. **ON UPDATE** referenceOption

When we add a foreign key using the ALTER TABLE statement, it is recommended to first create an **index** on the column(s), which is referenced by the foreign key.

## Example

The following statement creates two tables, "**Person**" and "**Contact**", without having a foreign key column into the table definition.

### Table: Person

1. **CREATE TABLE** Person (
2. ID **INT** NOT NULL AUTO\_INCREMENT,
3. Name **varchar**(50) NOT NULL,
4. City **varchar**(50) NOT NULL,
5. **PRIMARY KEY** (ID)
6. );

### Table: Contact

1. **CREATE TABLE** Contact (
2. ID **INT**,
3. Person\_Id **INT**,
4. Info **varchar**(50) NOT NULL,
5. Type **varchar**(50) NOT NULL
6. );

After creating a table, if we want to add a foreign key to an existing table, we need to execute the ALTER TABLE statement as below:

1. **ALTER TABLE** Contact **ADD INDEX** par\_ind ( Person\_Id );
2. **ALTER TABLE** Contact **ADD CONSTRAINT** fk\_person
3. **FOREIGN KEY** ( Person\_Id ) **REFERENCES** Person ( ID ) **ON DELETE CASCADE ON UPDATE RESTRICT**;

## Foreign Key Checks

MySQL has a special variable **foreign\_key\_checks** to control the foreign key checking into the tables. By default, it is enabled to enforce the referential integrity during the normal operation on the tables. This variable is dynamic in nature so that it supports global and session scopes both.

Sometimes there is a need for disabling the foreign key checking, which is very useful when:

- o We drop a table that is a reference by the foreign key.
- o We import data from a CSV file into a table. It speeds up the import operation.
- o We use ALTER TABLE statement on that table which has a foreign key.
- o We can execute load data operation into a table in any order to avoid foreign key checking.

The following statement allows us to **disable** foreign key checks:

1. **SET** foreign\_key\_checks = 0;

The following statement allows us to **enable** foreign key checks:

1. **SET** foreign\_key\_checks = 1;

# MySQL Composite Key

A composite key in MySQL is a combination of two or more than two columns in a table that allows us to identify each row of the table uniquely. It is a type of **candidate key** which is formed by more than one column. MySQL guaranteed the uniqueness of the column only when they are combined. If they have taken individually, the uniqueness cannot maintain.

Any key such as primary key, super key, or candidate key can be called composite key when they have combined with more than one attribute. A composite key is useful when the table needs to identify each record with more than one attribute uniquely. A column used in the composite key can have different data types. Thus, it is not required to be the same data type for the columns to make a composite key in MySQL.

A composite key can be added in two ways:

1. Using CREATE Statement
2. Using ALTER Statement

Let us see both ways in detail.



## Composite Key Using CREATE Statement

Here, we are going to understand how composite key works in MySQL. Let us first create a table "**Product**", using the following statement:

1. **CREATE TABLE** Product (
2. Prod\_ID **int** NOT NULL,
3. **Name varchar(45)**,
4. Manufacturer **varchar(45)**,
5. **PRIMARY KEY(Name, Manufacturer)**
6. );

In the above statement, we have created a composite primary with the column names **Name** and **Manufacturer**.

We can verify the same using the command as below:

1. **DESCRIBE Product;**

After the successful execution, we can see that the Key column has two **PRI**. It means we have successfully added the composite primary key on Name and Manufacturer columns.

```
MySQL 8.0 Command Line Client
mysql> DESCRIBE Product;
+-----+-----+-----+-----+-----+
| Field | Type  | Null | Key  | Default | Extra |
+-----+-----+-----+-----+-----+
| Prod_ID | int   | NO   | PRI  | NULL    |       |
| Name    | varchar(45) | NO   | PRI  | NULL    |       |
| Manufacturer | varchar(45) | NO   | PRI  | NULL    |       |
+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

Next, we need to insert the values into this table as given below:

1. **INSERT INTO** Product (Prod\_ID, **Name**, Manufacturer)
2. **VALUES** (101, 'Soap', 'Hamam'),
3. (102, 'Shampoo', 'Teresme'),
4. (103, 'Oil', 'Daber Almond');

Next, execute the below command to show the table data:

1. **SELECT \* FROM** Product;

It will give the output below:

The screenshot shows the MySQL 8.0 Command Line Client window. The query `SELECT * FROM Product;` is run, resulting in the following table output:

Prod_ID	Name	Manufacturer
103	Oil	Daber Almond
102	Shampoo	Teresme
101	Soap	Hamam

3 rows in set (0.00 sec)

Again execute the below insert statement to understand composite key more clearly:

1. **INSERT INTO** Product (Prod\_ID, **Name**, Manufacturer)
2. **VALUES** (101, 'Soap', 'Hamam');
- 3.
4. **INSERT INTO** Product (Prod\_ID, **Name**, Manufacturer)
5. **VALUES** (101, 'Soap', 'LUX');

In the below output, we can see that if we try to add the combination of the same product name and manufacturer, then it will throw an error saying that: **Duplicate entry for product.primary**.

If we execute the second insert statement, it will be added successfully into the table. It is because we can insert any number of soap in the product column, but the manufacturer column should be different.

The screenshot shows the MySQL 8.0 Command Line Client window. Two insert statements are run:

```
mysql> INSERT INTO Product (Prod_ID, Name, Manufacturer)
-> VALUES (101, 'Soap', 'Hamam');
ERROR 1062 (23000): Duplicate entry 'Soap-Hamam' for key 'product.PRIMARY'
mysql> INSERT INTO Product (Prod_ID, Name, Manufacturer)
-> VALUES (101, 'Soap', 'LUX');
Query OK, 1 row affected (0.15 sec)
```

Hence, we can say that the composite key always enforces the uniqueness of the columns of that table, which has two keys.

## Composite Key Using ALTER TABLE Statement

ALTER statement always used to do the modification into the existing table. Sometimes it is required to add the composite key to uniquely identify each record of the table with more than one attribute. In that case, we use an **ALTER TABLE** statement.

Let us first create a table "Student" using the below statement:

1. **CREATE TABLE** Student(
2. stud\_id **int** NOT NULL,
3. stud\_code **varchar**(15),
4. stud\_name **varchar**(35),
5. subject **varchar**(25),
6. marks **int**
7. );

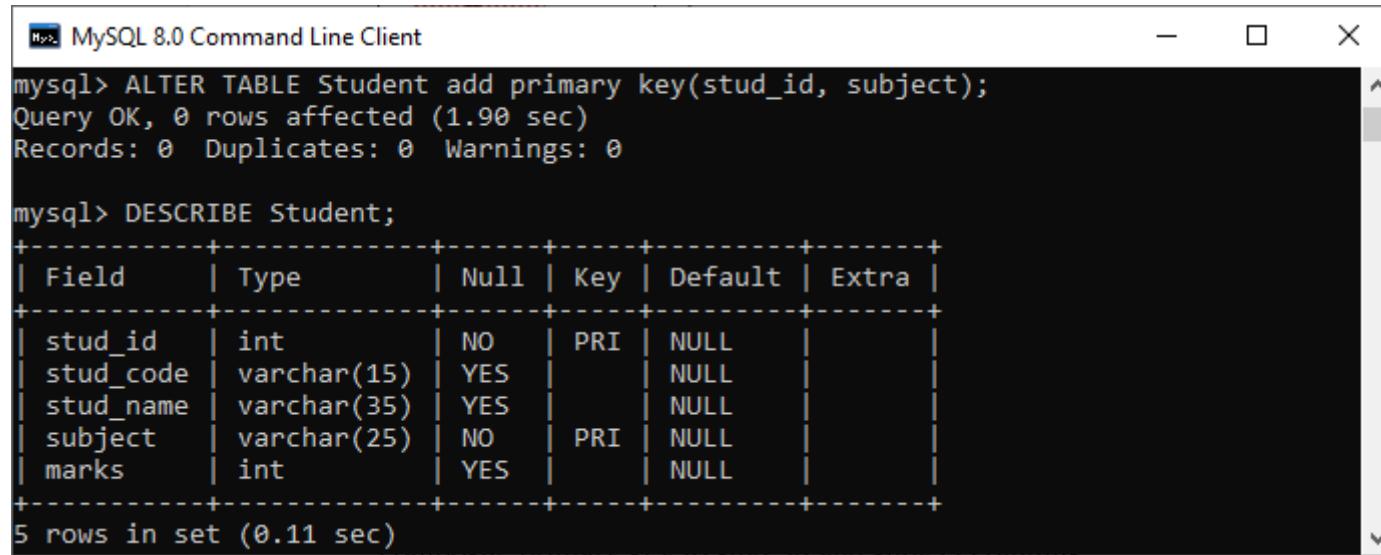
Now, execute the ALTER TABLE statement to add a composite primary key as follows:

1. **ALTER TABLE** Student **add primary key**(stud\_id, subject);

We can verify the composite primary key added into a table or not using the following command:

1. DESCRIBE Student;

In the output, we can see that the key column has PRI, which means we have successfully added the composite primary key to **stud\_id** and **subject** columns.



```
MySQL 8.0 Command Line Client
mysql> ALTER TABLE Student add primary key(stud_id, subject);
Query OK, 0 rows affected (1.90 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> DESCRIBE Student;
+-----+-----+-----+-----+-----+
| Field | Type  | Null | Key  | Default | Extra |
+-----+-----+-----+-----+-----+
| stud_id | int   | NO   | PRI   | NULL    |       |
| stud_code | varchar(15) | YES  |       | NULL    |       |
| stud_name | varchar(35) | YES  |       | NULL    |       |
| subject | varchar(25) | NO   | PRI   | NULL    |       |
| marks | int   | YES  |       | NULL    |       |
+-----+-----+-----+-----+-----+
5 rows in set (0.11 sec)
```

## MySQL Triggers

# MySQL Trigger

A trigger in MySQL is a set of SQL statements that reside in a system catalog. **It is a special type of stored procedure that is invoked automatically in response to an event.** Each trigger is associated with a table, which is activated on any DML statement such as **INSERT, UPDATE, or DELETE.**

A trigger is called a special procedure because it cannot be called directly like a stored procedure. The main difference between the trigger and procedure is that a trigger is called automatically when a data modification event is made against a table. In contrast, a stored procedure must be called explicitly.

Generally, **triggers are of two types** according to the [SQL](#) standard: row-level triggers and statement-level triggers.

**Row-Level Trigger:** It is a trigger, which is activated for each row by a triggering statement such as insert, update, or delete. For example, if a table has inserted, updated, or deleted multiple rows, the row trigger is fired automatically for each row affected by the [insert, update, or delete statement](#).

Keep Watching

**Statement-Level Trigger:** It is a trigger, which is fired once for each event that occurs on a table regardless of how many rows are inserted, updated, or deleted.

**NOTE:** We should know that MySQL doesn't support statement-level triggers. It provides supports for row-level triggers only.

## Why we need/use triggers in MySQL?

We need/use triggers in MySQL due to the following features:

- Triggers help us to enforce business rules.
- Triggers help us to validate data even before they are inserted or updated.
- Triggers help us to keep a log of records like maintaining audit trails in tables.
- SQL triggers provide an alternative way to check the integrity of data.
- Triggers provide an alternative way to run the scheduled task.
- Triggers increases the performance of SQL queries because it does not need to compile each time the query is executed.
- Triggers reduce the client-side code that saves time and effort.
- Triggers help us to scale our application across different platforms.
- Triggers are easy to maintain.

## Limitations of Using Triggers in MySQL

- MySQL triggers do not allow to use of all validations; they only provide extended validations. **For example**, we can use the NOT NULL, UNIQUE, CHECK and FOREIGN KEY constraints for simple validations.
- Triggers are invoked and executed invisibly from the client application. Therefore, it isn't easy to troubleshoot what happens in the database layer.
- Triggers may increase the overhead of the database server.

## Types of Triggers in MySQL?

We can define the maximum six types of actions or events in the form of triggers:

- Before Insert:** It is activated before the insertion of data into the table.
- After Insert:** It is activated after the insertion of data into the table.
- Before Update:** It is activated before the update of data in the table.
- After Update:** It is activated after the update of the data in the table.
- Before Delete:** It is activated before the data is removed from the table.
- After Delete:** It is activated after the deletion of data from the table.

When we use a statement that does not use INSERT, UPDATE or DELETE query to change the data in a table, the triggers associated with the trigger will not be invoked.

## Naming Conventions

Naming conventions are the set of rules that we follow to give appropriate unique names. It saves our time to keep the work organize and understandable. Therefore, **we must use a unique name for each trigger associated with a table**. However, it is a good practice to have the same trigger name defined for different tables.

The following naming convention should be used to name the trigger in [MySQL](#):

- (BEFOR | **AFTER**) table\_name (**INSERT** | **UPDATE** | **DELETE**)

Thus,

**Trigger Activation Time:** BEFORE | AFTER

**Trigger Event:** INSERT | UPDATE | DELETE

## How to create triggers in MySQL?

We can use the **CREATE TRIGGER** statement for creating a new trigger in MySQL. Below is the syntax of creating a trigger in MySQL:

- CREATE TRIGGER** trigger\_name
- (AFTER | BEFORE) (INSERT | UPDATE | DELETE)**
- ON** table\_name **FOR** EACH ROW
- BEGIN**
- variable declarations
- trigger code
- END;**

## MySQL Create Trigger

In this article, we are going to learn how to create the first trigger in MySQL. We can create a new trigger in MySQL by using the CREATE TRIGGER statement. It is to ensure that we have trigger privileges while using the CREATE TRIGGER command. The following is the basic syntax to create a trigger:

- CREATE TRIGGER** trigger\_name trigger\_time trigger\_event
- ON** table\_name **FOR** EACH ROW
- BEGIN**
- variable declarations
- trigger code
- END;**

## Parameter Explanation

The create trigger syntax contains the following parameters:

**trigger\_name:** It is the name of the trigger that we want to create. It must be written after the CREATE TRIGGER statement. It is to make sure that the trigger name should be unique within the schema.

**trigger\_time:** It is the trigger action time, which should be either BEFORE or AFTER. It is the required parameter while defining a trigger. It indicates that the trigger will be invoked before or after each row modification occurs on the table.

**trigger\_event:** It is the type of operation name that activates the trigger. It can be either INSERT, UPDATE, or DELETE operation. The trigger can invoke only one event at one time. If we want to define a trigger which is invoked by multiple events, it is required to define multiple triggers, and one for each event.

**table\_name:** It is the name of the table to which the trigger is associated. It must be written after the ON keyword. If we did not specify the table name, a trigger would not exist.

**BEGIN END Block:** Finally, we will specify the statement for execution when the trigger is activated. If we want to execute multiple statements, we will use the BEGIN END block that contains a set of queries to define the logic for the trigger.

The trigger body can access the column's values, which are affected by the DML statement. The **NEW** and **OLD** modifiers are used to distinguish the column values **BEFORE** and **AFTER** the execution of the DML statement. We can use the column name with NEW and OLD modifiers as **OLD.col\_name** and **NEW.col\_name**. The OLD.column\_name indicates the column of an existing row before the updation or deletion occurs. NEW.col\_name indicates the column of a new row that will be inserted or an existing row after it is updated.

**For example**, suppose we want to update the column name **message\_info** using the trigger. In the trigger body, we can access the column value before the update as **OLD.message\_info** and the new value **NEW.message\_info**.

We can understand the availability of OLD and NEW modifiers with the below table:

Trigger Event	OLD	NEW
INSERT	No	Yes
UPDATE	Yes	Yes
DELETE	Yes	No

## MySQL Trigger Example

Let us start creating a trigger in MySQL that makes modifications in the employee table. First, we will create a new table named **employee** by executing the below statement:

1. **CREATE TABLE** employee(
2.   **name varchar**(45) NOT NULL,
3.   **occupation varchar**(35) NOT NULL,
4.   **working\_date date**,
5.   **working\_hours varchar**(10)
6. );

Next, execute the below statement to **fill the records** into the employee table:

1. **INSERT INTO** employee **VALUES**
2. ('Robin', 'Scientist', '2020-10-04', 12),
3. ('Warner', 'Engineer', '2020-10-04', 10),
4. ('Peter', 'Actor', '2020-10-04', 13),
5. ('Marco', 'Doctor', '2020-10-04', 14),
6. ('Brayden', 'Teacher', '2020-10-04', 12),
7. ('Antonio', 'Business', '2020-10-04', 11);

Next, execute the **SELECT statement** to verify the inserted record:

```

MySQL 8.0 Command Line Client
mysql> SELECT * FROM employee;
+-----+-----+-----+-----+
| name | occupation | working_date | working_hours |
+-----+-----+-----+-----+
| Robin | Scientist | 2020-10-04 | 12
| Warner | Engineer | 2020-10-04 | 10
| Peter | Actor | 2020-10-04 | 13
| Marco | Doctor | 2020-10-04 | 14
| Brayden | Teacher | 2020-10-04 | 12
| Antonio | Business | 2020-10-04 | 11
+-----+-----+-----+-----+
6 rows in set (0.00 sec)

```

Next, we will create a **BEFORE INSERT trigger**. This trigger is invoked automatically insert the **working\_hours = 0** if someone tries to insert **working\_hours < 0**.

1. mysql> DELIMITER //
2. mysql> **Create Trigger** before\_insert\_empworkinghours
3. BEFORE **INSERT ON** employee **FOR EACH ROW**
4. **BEGIN**
5. IF NEW.working\_hours < 0 **THEN SET** NEW.working\_hours = 0;
6. **END IF;**
7. **END //**

If the trigger is created successfully, we will get the output as follows:

```

MySQL 8.0 Command Line Client
mysql> DELIMITER //
mysql> Create Trigger before_insert_empworkinghours
-> BEFORE INSERT ON employee FOR EACH ROW
-> BEGIN
-> IF NEW.working_hours < 0 THEN SET NEW.working_hours = 0;
-> END IF;
-> END //
Query OK, 0 rows affected (0.28 sec)

```

Now, we can use the following statements to invoke this trigger:

1. mysql> **INSERT INTO** employee **VALUES**
2. ('Markus', 'Former', '2020-10-08', 14);
- 3.
4. mysql> **INSERT INTO** employee **VALUES**
5. ('Alexander', 'Actor', '2020-10-012', -13);

After execution of the above statement, we will get the output as follows:

```

MySQL 8.0 Command Line Client
mysql> INSERT INTO employee VALUES
-> ('Markus', 'Former', '2020-10-08', 14);
Query OK, 1 row affected (0.18 sec)

mysql> INSERT INTO employee VALUES
-> ('Alexander', 'Actor', '2020-10-012', -13);
Query OK, 1 row affected (0.16 sec)

mysql> SELECT * FROM employee;
+-----+-----+-----+-----+
| name | occupation | working_date | working_hours |
+-----+-----+-----+-----+
| Robin | Scientist | 2020-10-04 | 12
| Warner | Engineer | 2020-10-04 | 10
| Peter | Actor | 2020-10-04 | 13
| Marco | Doctor | 2020-10-04 | 14
| Brayden | Teacher | 2020-10-04 | 12
| Antonio | Business | 2020-10-04 | 11
| Markus | Former | 2020-10-08 | 14
| Alexander | Actor | 2020-10-12 | 0
+-----+-----+-----+-----+
8 rows in set (0.00 sec)

```

In this output, we can see that on inserting the negative values into the **working\_hours** column of the table will automatically fill the zero value by a trigger.

# MySQL Show/List Triggers

The show or list trigger is much needed when we have many databases that contain various tables. Sometimes we have the same trigger names in many databases; this query plays an important role in that case. We can get the trigger information in the database server using the below statement. This statement returns all triggers in all databases:

1. mysql> SHOW TRIGGERS;

**The following steps are necessary to get the list of all triggers:**

**Step 1:** Open the [MySQL](#) Command prompt and logged into the database server using the password that you have created during [MySQL's installation](#). After a successful connection, we can execute all the [SQL](#) statements.

**Step 2:** Next, choose the specific database by using the command below:

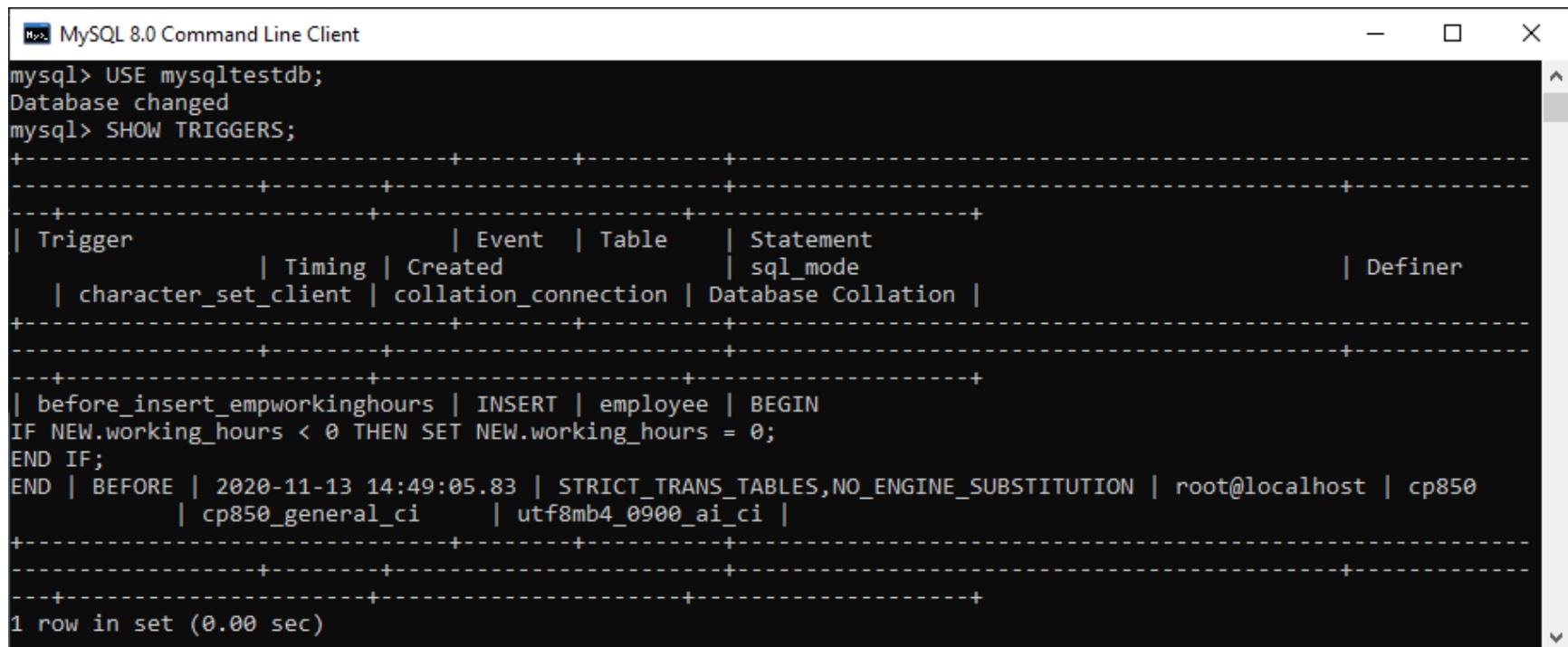
1. mysql> USE database\_name;

**Step 3:** Finally, execute the SHOW TRIGGERS command.

Let us understand it with the example given below. Suppose we have a database name "**mysqltestdb**" that contains many tables. Then execute the below statement to list the [triggers](#):

1. mysql> USE mysqltestdb;
2. mysql>SHOW TRIGGERS;

The following output explains it more clearly:



The screenshot shows the MySQL 8.0 Command Line Client window. The command entered is "SHOW TRIGGERS;" and the output is as follows:

```
mysql> USE mysqltestdb;
Database changed
mysql> SHOW TRIGGERS;
+-----+-----+-----+
| Trigger          | Event   | Table    | Statement
+-----+-----+-----+
| character_set_client | collation_connection | Database Collation | Definer
+-----+-----+-----+
| before_insert_empworkinghours | INSERT | employee | BEGIN
IF NEW.working_hours < 0 THEN SET NEW.working_hours = 0;
END IF;
END | BEFORE | 2020-11-13 14:49:05.83 | STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION | root@localhost | cp850
      | cp850_general_ci | utf8mb4_0900_ai_ci |
+-----+-----+-----+
1 row in set (0.00 sec)
```

If we want to show or list the trigger information in a specific database from the current database without switching, MySQL allows us to use the **FROM** or **IN** clause, followed by the database name. The following statement explains it more clearly:

1. mysql> SHOW TABLES IN database\_name;

The above statement can also be written as:

1. mysql> SHOW TABLES **FROM** database\_name;

When we execute the above statements, we will get the same result.

## Show Triggers Using Pattern Matching

MySQL also provides a **LIKE** clause option that enables us to filter the trigger name using different pattern matching. The following is the syntax to use pattern matching with show trigger command:

1. mysql> SHOW TRIGGERS LIKE pattern;
2. OR,
3. mysql> SHOW TRIGGERS **FROM** database\_name **LIKE** pattern;

If we want to list/show trigger names based on specific search condition, we can use the [WHERE clause](#) as follows:

1. mysql> SHOW TRIGGERS **WHERE** search\_condition;
2. OR,
3. mysql> SHOW TRIGGERS **FROM** database\_name **WHERE** search\_condition;

## Example

Suppose we want to show all triggers that belongs to the **employee** table, execute the statement as follows:

1. mysql> SHOW TRIGGERS **FROM** mysqltestdb **WHERE table** = 'employee';

We will get the output as follows:

Trigger	Event	Table	Statement	Timing	Created	sql_mode	Definer	character_set	collation_conn	Database	Collation
before_insert_empworkinghours	INSERT	employee	BEGIN IF NEW.working_hours < 0 THEN SET NEW.working_hours = 0; END IF; END	BEFORE	2020-11-13 14:49:05.83	STRICT_TRANS_TABLES,NO_ROOT@localhost	cp850	cp850_general_ci	utf8mb4_0900	cp850_general_ci	utf8mb4_0900

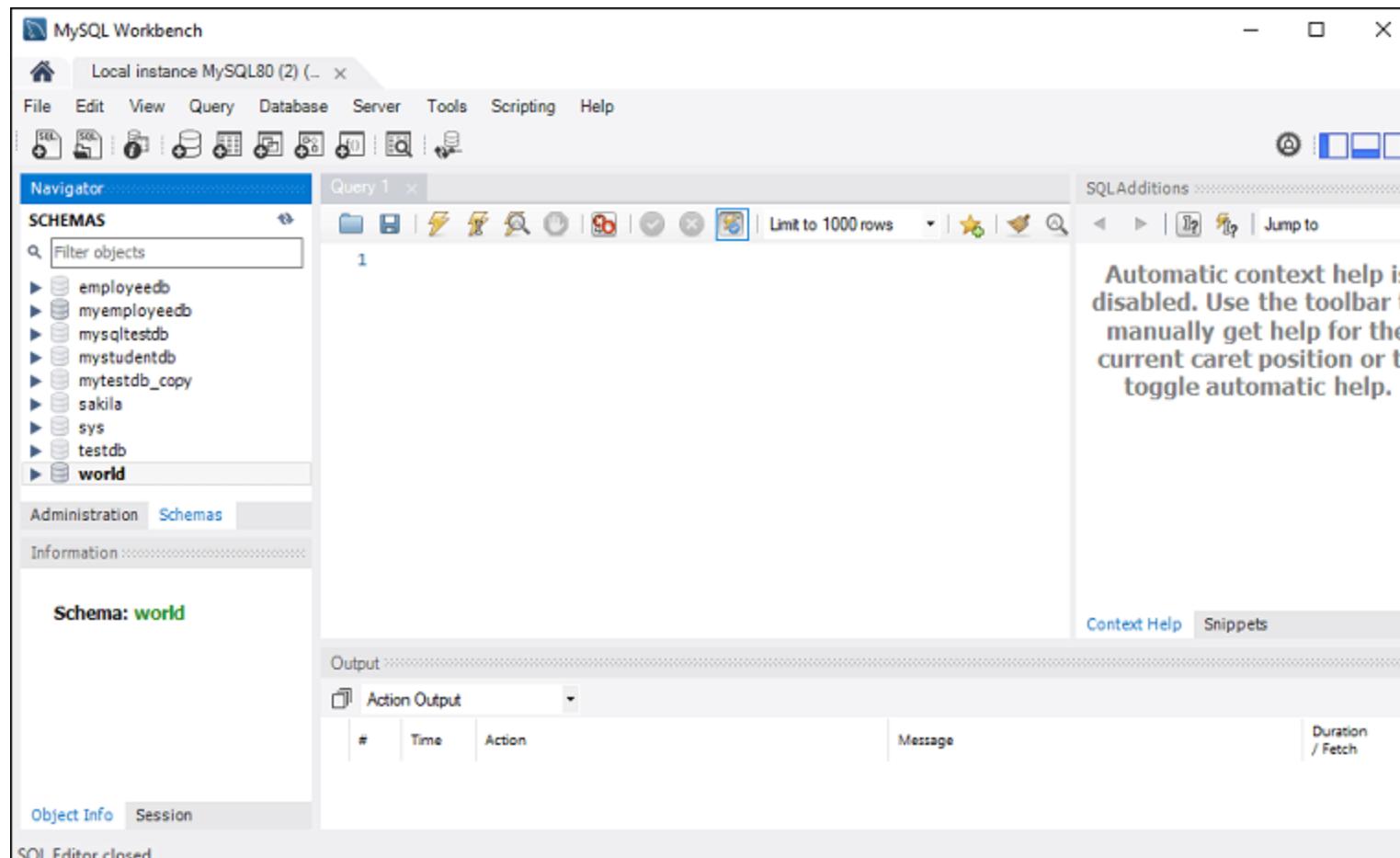
**NOTE: It is to note that we must have a SUPER privilege to execute the SHOW TRIGGERS statement.**

The show trigger statement contains several columns in the result set. Let us explain each column in detail.

- **Trigger:** It is the name of the trigger that we want to create and must be unique within the schema.
- **Event:** It is the type of operation name that invokes the trigger. It can be either INSERT, UPDATE, or DELETE operation.
- **Table:** It is the name of the table to which the trigger belongs.
- **Statement:** It is the body of the trigger that contains logic for the trigger when it activates.
- **Timing:** It is the activation time of the trigger, either BEFORE or AFTER. It indicates that the trigger will be invoked before or after each row of modifications occurs on the table.
- **Created:** It represents the time and date when the trigger is created.
- **sql\_mode:** It displays the SQL\_MODE when the trigger is executed.
- **Definer:** It is the name of a user account that created the trigger and should be in the 'user\_name'@'host\_name' format.
- **character\_set\_client:** It was the session value of the character\_set\_client system variable when the trigger was created.
- **collation\_connection:** It was the session value of the character\_set\_client system variable when the trigger was created.
- **Database Collation:** It determines the rules that compare and order the character string. It is the collation of the database with which the trigger belongs.

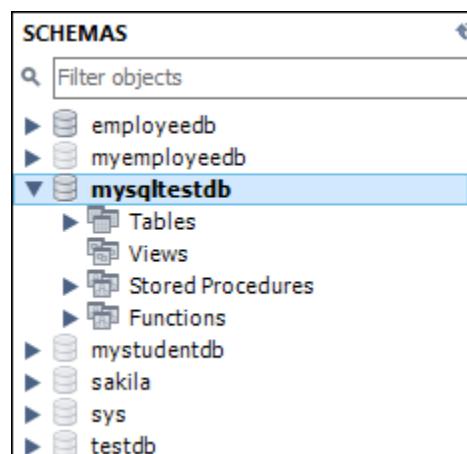
## How to show triggers in MySQL workbench?

It is a visual GUI tool used to create databases, tables, indexes, views, and stored procedures quickly and efficiently. To show a trigger using this tool, we first need to launch the MySQL Workbench and log in using the username and password that we have created earlier. We will get the screen as follows:

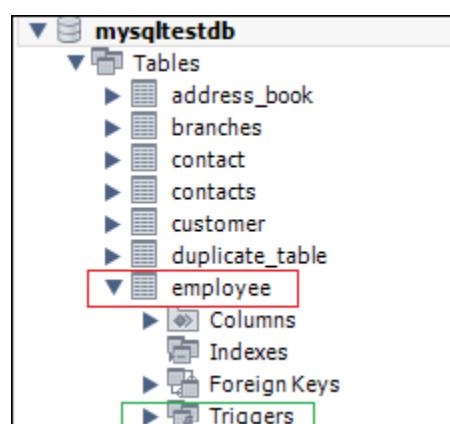


Now do the following steps to show triggers:

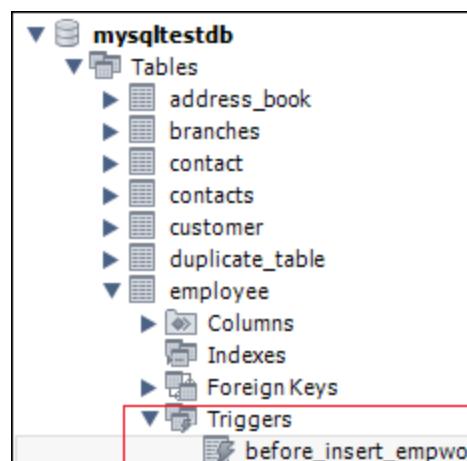
1. Go to the Navigation tab and click on the **Schema menu** that contains all the databases available in the MySQL server.
2. Select the database (for example, **mysqltestdb**), double click on it, and show the **sub-menu** containing Tables, Views, Functions, and Stored Procedures. See the below screen.



3. Click on the **Tables sub-menu** and select the table on which you have created a trigger. See the below image:



4. Clicking on the **Triggers sub-menu**, we can see all triggers associated with the selected table. See the below image.



## MySQL DROP Trigger

We can drop/delete/remove a trigger in MySQL using the **DROP TRIGGER** statement. You must be very careful while removing a trigger from the table. Because once we have deleted the trigger, it cannot be recovered. If a trigger is not found, the **DROP TRIGGER** statement throws an error.

MySQL allows us to drop/delete/remove a trigger mainly in two ways:

1. MySQL Command Line Client
2. MySQL Workbench

### MySQL Command Line Client

We can drop an existing trigger from the database by using the **DROP TRIGGER** statement with the below syntax:

1. **DROP TRIGGER** [IF EXISTS] [schema\_name.]trigger\_name;

#### Parameter Explanation

The parameters used in the drop trigger syntax are explained as follows:

Parameter	Descriptions
Trigger_name	It is the name of a trigger that we want to remove from the database server. It is a required parameter.
Schema_name	It is the database name to which the trigger belongs. If we skip this parameter, the statement will remove the trigger from the current database.
IF_EXISTS	It is an optional parameter that conditionally removes triggers only if they exist on the database server.

If we remove the trigger that does not exist, we will get an error. However, if we have specified the IF EXISTS clause, MySQL gives a **NOTE** instead of an error.

It is to note that we must have TRIGGER privileges before executing the DROP TRIGGER statement for the table associated with the trigger. Also, removing a table will automatically delete all triggers associated with the table.

## MySQL DROP Trigger Example

Let us see how we can drop the trigger associated with the table through an example. So first, we will **display all triggers** available in the selected database using the below statement:

1. mysql> SHOW TRIGGERS IN employeedb;

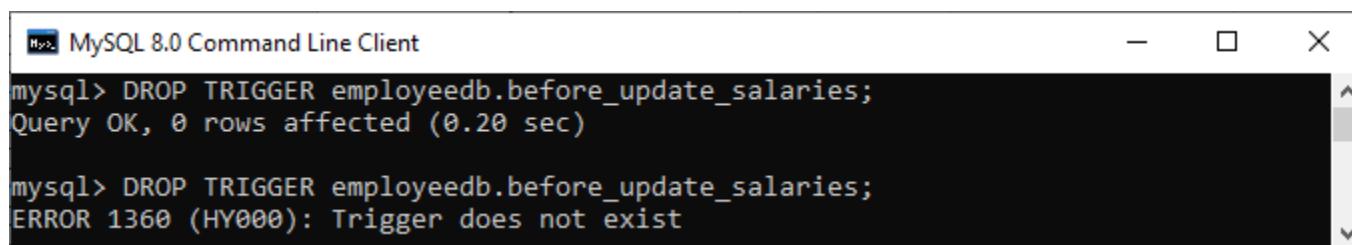
After executing the statement, we can see that there are two triggers named **before\_update\_salaries** and **sales\_info\_before\_update**. See the below image:

Trigger	Event	Table	Statement	Timing	Created	sql_mode	Definer
before_update_salaries	UPDATE	salary_account	BEGIN IF new.amount < old.a...	BEFORE	2020-11-2...	STRICT_TRA...	root@localhost
sales_info_BEFORE_UPDATE	UPDATE	sales_info	BEGIN DECLARE error_msg VARC...	BEFORE	2020-11-2...	STRICT_TRA...	root@localhost

If we want to remove the **before\_update\_salaries** trigger, execute the below statement:

1. mysql> **DROP TRIGGER** employeedb.before\_update\_salaries;

It will successfully delete a trigger from the database. If we execute the above statement again, it will return an error message. See the output:

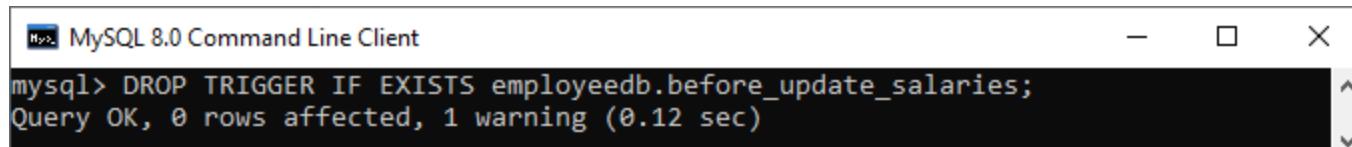


```
mysql> DROP TRIGGER employeedb.before_update_salaries;
Query OK, 0 rows affected (0.20 sec)

mysql> DROP TRIGGER employeedb.before_update_salaries;
ERROR 1360 (HY000): Trigger does not exist
```

If we execute the above statement again with an **IF EXISTS** clause, it will return the warning message instead of producing an error. See the output:

1. mysql> **DROP TRIGGER** IF EXISTS employeedb.before\_update\_salaries;



```
mysql> DROP TRIGGER IF EXISTS employeedb.before_update_salaries;
Query OK, 0 rows affected, 1 warning (0.12 sec)
```

We can execute the **SHOW WARNING** statement that generates a **NOTE** for a non-existent trigger when using IF EXISTS. See the output:

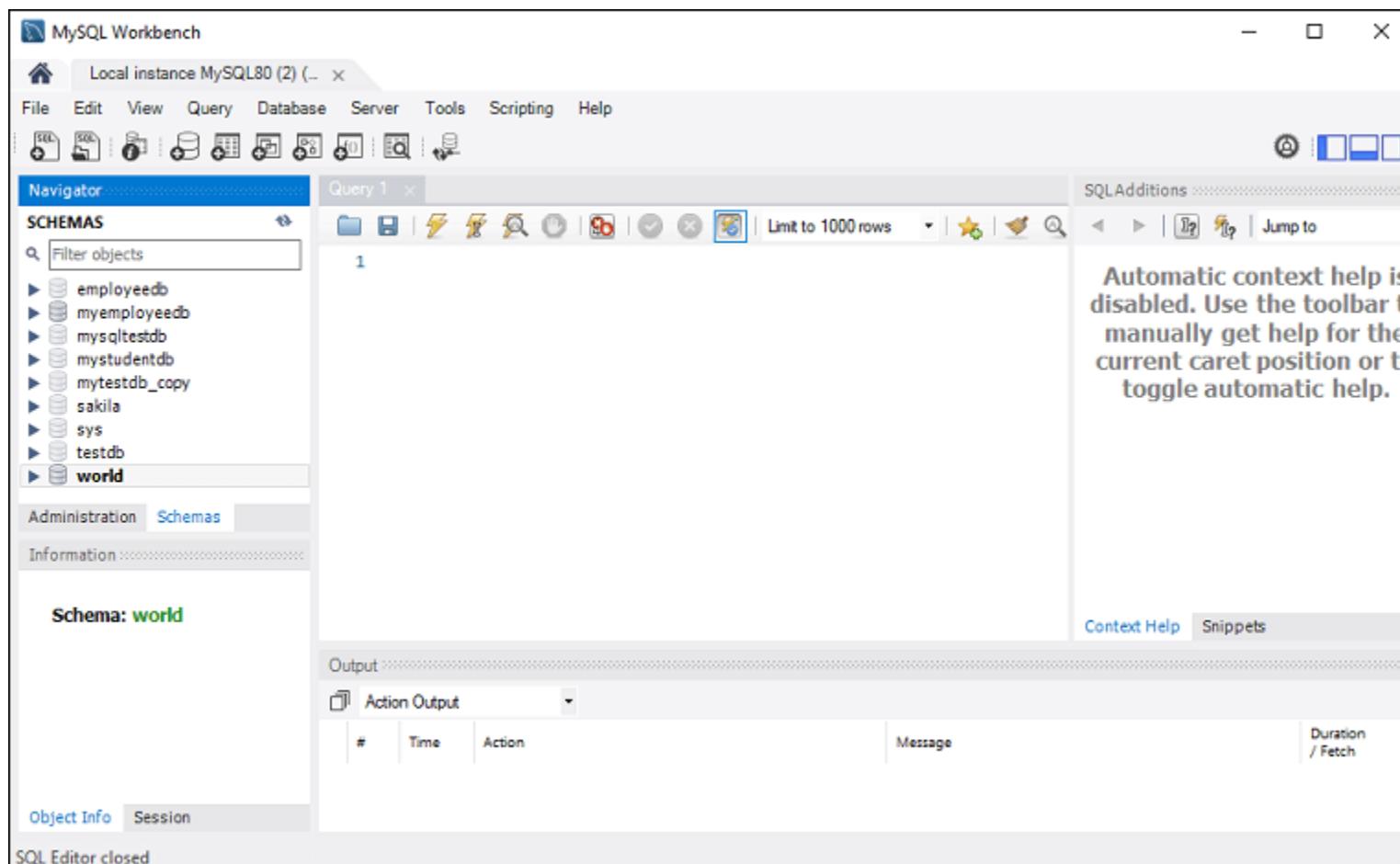
```

MySQL 8.0 Command Line Client
mysql> SHOW WARNINGS;
+-----+-----+-----+
| Level | Code | Message
+-----+-----+-----+
| Note  | 1360 | Trigger does not exist |
+-----+-----+-----+
1 row in set (0.00 sec)

```

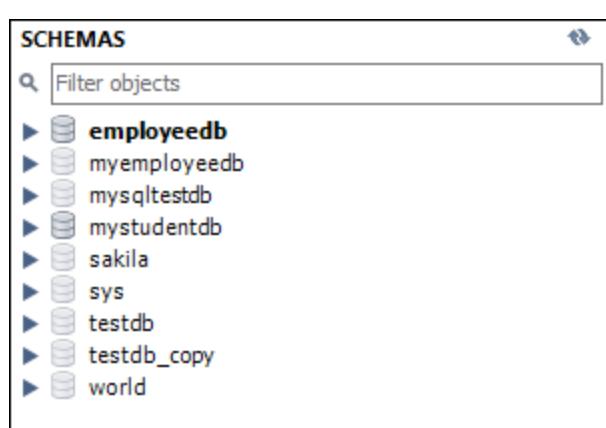
## How to Drop trigger in MySQL workbench?

To create an **AFTER UPDATE** trigger in workbench, we first **launch the MySQL Workbench** and log in using the username and password. We will get the UI as follows:

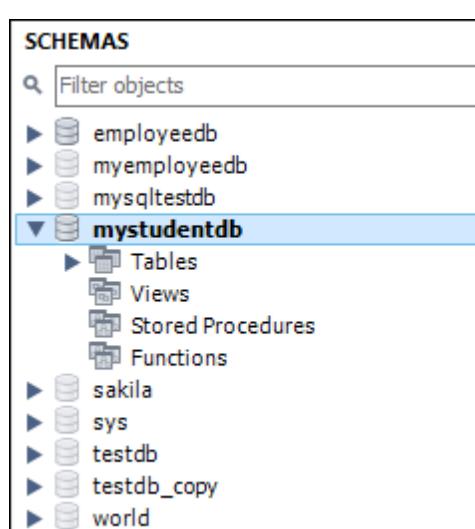


Now do the following steps to delete or destroy a trigger associated with the table:

1. Go to the **Navigation** tab and click on the **Schema** menu. It will display all databases available in the MySQL database server.



2. Select the database (for example, **mystudentdb**). Then, double click on the selected schema. It displays the **sub-menu** containing Tables, Views, Functions, and Stored Procedures. See the below screen.



3. Expand the **Tables** sub-menu and select a table on which a trigger is associated. Again expand the selected **Table -> Triggers**; we will get the below image:

The screenshot shows the MySQL Workbench schema browser. The left pane lists schemas and tables. Under the 'mystudentdb' schema, the 'Tables' node is expanded, showing tables like 'customer', 'duplicate\_table', 'employees', etc. The 'students' table is selected, and its sub-nodes 'Columns', 'Indexes', 'Foreign Keys', and 'Triggers' are visible. The 'Triggers' node is expanded, and the specific trigger 'student\_update\_trigger' is highlighted with a blue selection bar.

4. Now, right-click on the selected table and choose the **Alter Table** option that gives the screen as below:

The screenshot shows the 'Alter Table' dialog for the 'students' table. The top section displays basic table information: Table Name: 'students', Schema: 'mystudentdb', Charset/Collation: 'utf8mb4', Engine: 'InnoDB'. Below this, the 'Columns' tab is selected, showing a single column 'id' with type 'INT'. The 'Triggers' tab is highlighted with a red box and is currently active, showing the trigger definition:

```

CREATE DEFINER='root'@'localhost' TRIGGER `student_update_trigger` AFTER
    INSERT
    ON `students` FOR EACH ROW
BEGIN
    INSERT into `students_log` VALUES (user(),
        CONCAT('Update Student Record ', OLD.name, ' Previous Class: ',
        OLD.class, ' Present Class: ', NEW.class));
END
  
```

At the bottom of the dialog are 'Apply' and 'Revert' buttons.

5. Now, click on the **Trigger** tab shown in the previous section's **red rectangular box**. You will notice that there is a **(+)** and **(-)** icon button to add or delete a trigger:

The screenshot shows the 'Alter Table' dialog for the 'students' table. The 'Triggers' tab is selected and highlighted with a red box. The trigger definition is displayed:

```

CREATE DEFINER='root'@'localhost' TRIGGER `student_update_trigger` AFTER
    INSERT
    ON `students` FOR EACH ROW
BEGIN
    INSERT into `students_log` VALUES (user(),
        CONCAT('Update Student Record ', OLD.name, ' Previous Class: ',
        OLD.class, ' Present Class: ', NEW.class));
END
  
```

A red box highlights the '-' icon button located next to the trigger definition. At the bottom of the dialog are 'Apply' and 'Revert' buttons.

6. Now, clicking on the **(-) button** will permanently remove the trigger associated with the table.

## MySQL BEFORE INSERT TRIGGER

Before Insert Trigger in MySQL is invoked automatically whenever an insert operation is executed. In this article, we are going to learn how to create a before insert trigger with its syntax and example.

### Syntax

The following is the syntax to create a BEFORE INSERT [trigger in MySQL](#):

1. **CREATE TRIGGER** trigger\_name
2. BEFORE **INSERT**
3. **ON** table\_name **FOR EACH ROW**
4. Trigger\_body ;

The BEFORE INSERT trigger syntax parameter can be explained as below:

- First, we will specify the **name of the trigger** that we want to create. It should be unique within the schema.
- Second, we will specify the **trigger action time**, which should be BEFORE INSERT. This trigger will be invoked before each row modifications occur on the table.
- Third, we will specify the **name of a table** to which the trigger is associated. It must be written after the ON keyword. If we did not specify the table name, a trigger would not exist.
- Finally, we will specify the statement for execution when the trigger is activated.

If we want to execute multiple statements, we will use the **BEGIN END** block that contains a set of queries to define the logic for the trigger. See the below syntax:

1. DELIMITER \$\$
2. **CREATE TRIGGER** trigger\_name BEFORE **INSERT**
3. **ON** table\_name **FOR EACH ROW**
4. **BEGIN**
5. variable declarations
6. **trigger** code
7. **END\$\$**
8. DELIMITER ;

## Restrictions

- We can access and change the **NEW** values only in a BEFORE INSERT trigger.
- We cannot access the **OLD** If we try to access the OLD values, we will get an error because OLD values do not exist.
- We cannot create a BEFORE INSERT trigger on a **VIEW**.

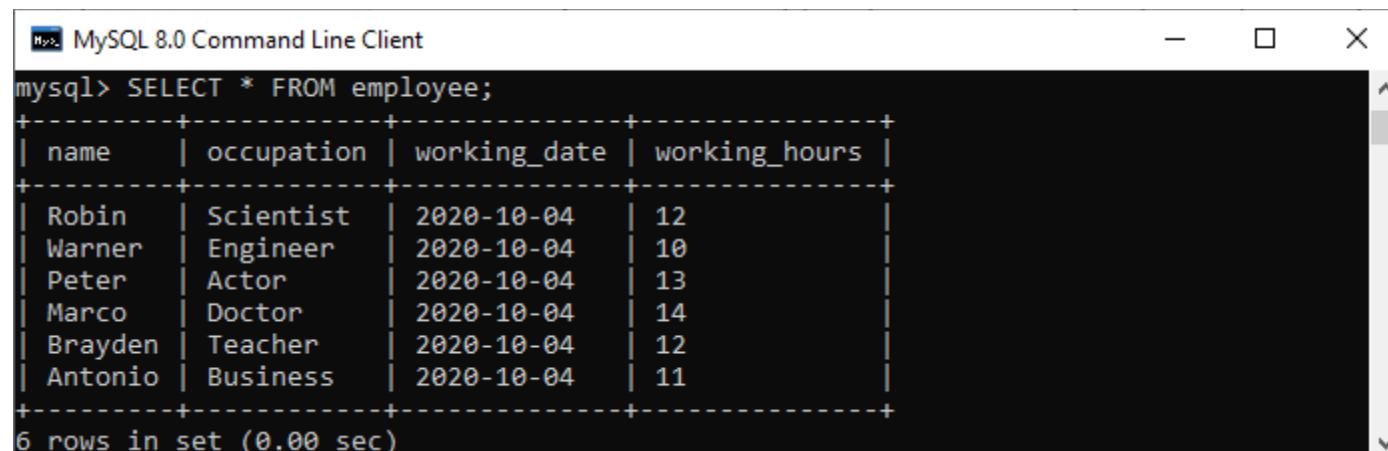
## BEFORE INSERT Trigger Example

Let us understand how to create a BEFORE INSERT trigger using the [CREATE TRIGGER statement](#) in [MySQL](#) with an example.

Suppose we have created a table named **employee** as follows:

1. **CREATE TABLE** employee(
2. **name** **varchar**(45) NOT NULL,
3. **occupation** **varchar**(35) NOT NULL,
4. **working\_date** **date**,
5. **working\_hours** **varchar**(10)
6. );

Next, we will insert some records into the employee table and then execute the [SELECT statement](#) to see the table data as follows:



```
MySQL 8.0 Command Line Client
mysql> SELECT * FROM employee;
+-----+-----+-----+-----+
| name | occupation | working_date | working_hours |
+-----+-----+-----+-----+
| Robin | Scientist | 2020-10-04 | 12
| Warner | Engineer | 2020-10-04 | 10
| Peter | Actor | 2020-10-04 | 13
| Marco | Doctor | 2020-10-04 | 14
| Brayden | Teacher | 2020-10-04 | 12
| Antonio | Business | 2020-10-04 | 11
+-----+-----+-----+-----+
6 rows in set (0.00 sec)
```

Next, we will use a **CREATE TRIGGER** statement to create a BEFORE INSERT trigger. This trigger is invoked automatically that inserts the **occupation = 'Leader'** if someone tries to insert the **occupation = 'Scientist'**.

1. mysql> DELIMITER //
2. mysql> **Create Trigger** before\_insert\_occupation
3. BEFORE **INSERT ON** employee **FOR EACH ROW**
4. **BEGIN**
5. IF NEW.occupation = 'Scientist' **THEN SET** NEW.occupation = 'Doctor';
6. **END IF;**
7. **END //**

If the trigger is created successfully, we will get the output as follows:

```
MySQL 8.0 Command Line Client
mysql> DELIMITER //
mysql> CREATE TRIGGER before_insert_occupation
    -> BEFORE INSERT ON employee FOR EACH ROW
    -> BEGIN
    -> IF NEW.occupation = 'Scientist' THEN SET NEW.occupation = 'Doctor';
    -> END IF;
    -> END //
Query OK, 0 rows affected (0.18 sec)
```

## How to call the BEFORE INSERT trigger?

We can use the following statements to invoke the above-created trigger:

1. mysql> **INSERT INTO** employee **VALUES**
2. ('Markus', 'Scientist', '2020-10-08', 14);
- 3.
4. mysql> **INSERT INTO** employee **VALUES**
5. ('Alexander', 'Actor', '2020-10-012', 13);

After execution of the above statement, we will get the output as follows:

```
MySQL 8.0 Command Line Client
mysql> INSERT INTO employee VALUES
    -> ('Markus', 'Scientist', '2020-10-08', 14);
Query OK, 1 row affected (0.13 sec)

mysql> INSERT INTO employee VALUES
    -> ('Alexander', 'Actor', '2020-10-012', 13);
Query OK, 1 row affected (0.70 sec)
```

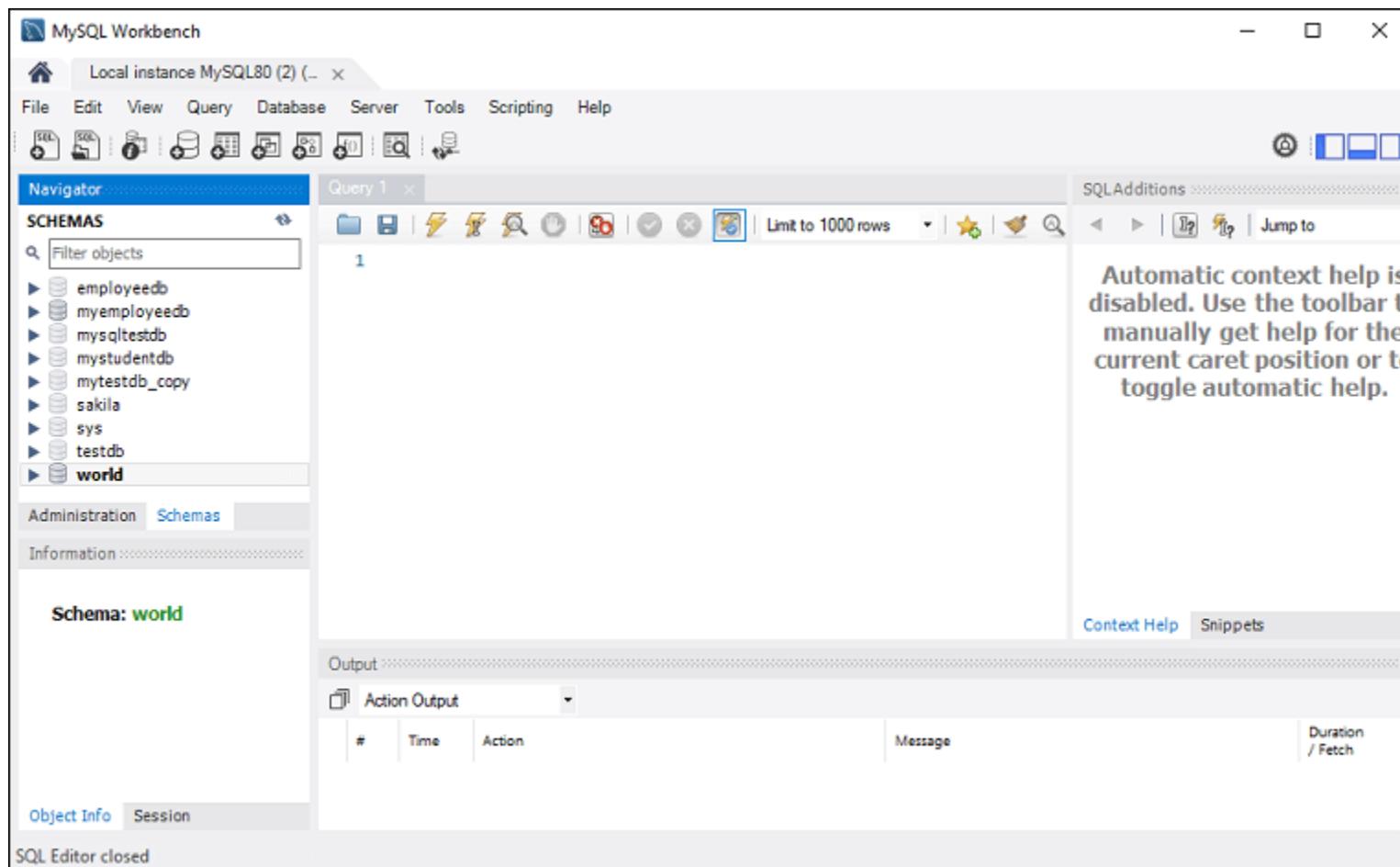
Execute the SELECT statement to verify the output:

```
MySQL 8.0 Command Line Client
mysql> SELECT * FROM employee;
+-----+-----+-----+-----+
| name | occupation | working_date | working_hours |
+-----+-----+-----+-----+
| Robin | Scientist | 2020-10-04 | 12
| Warner | Engineer | 2020-10-04 | 10
| Peter | Actor | 2020-10-04 | 13
| Marco | Doctor | 2020-10-04 | 14
| Brayden | Teacher | 2020-10-04 | 12
| Antonio | Business | 2020-10-04 | 11
| Markus | Doctor | 2020-10-08 | 14
| Alexander | Actor | 2020-10-12 | 13
+-----+-----+-----+-----+
8 rows in set (0.00 sec)
```

In this output, we can see that on inserting the occupation column values as 'Scientist', the table will automatically fill the 'Doctor' value by invoking a trigger.

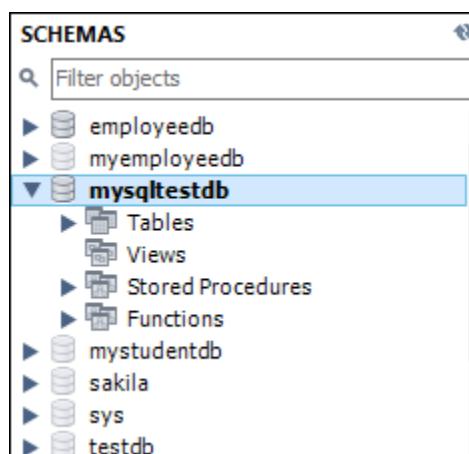
## How to create BEFORE INSERT Trigger in MySQL workbench?

To create a before insert trigger using this tool, we first need to launch the [MySQL Workbench](#) and log in using the username and password we created earlier. We will get the screen as follows:

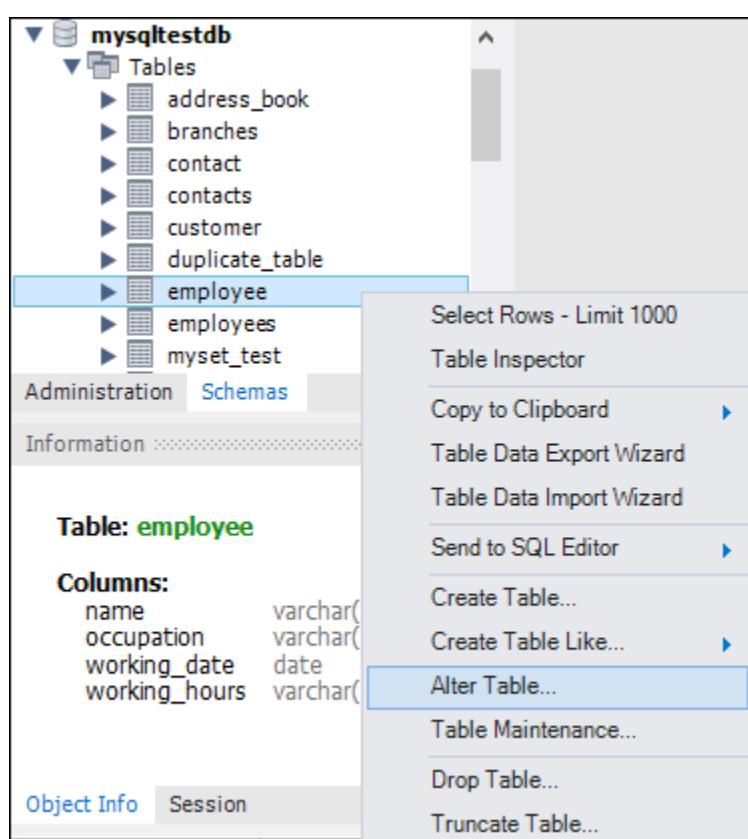


Now do the following steps for creating BEFORE INSERT trigger:

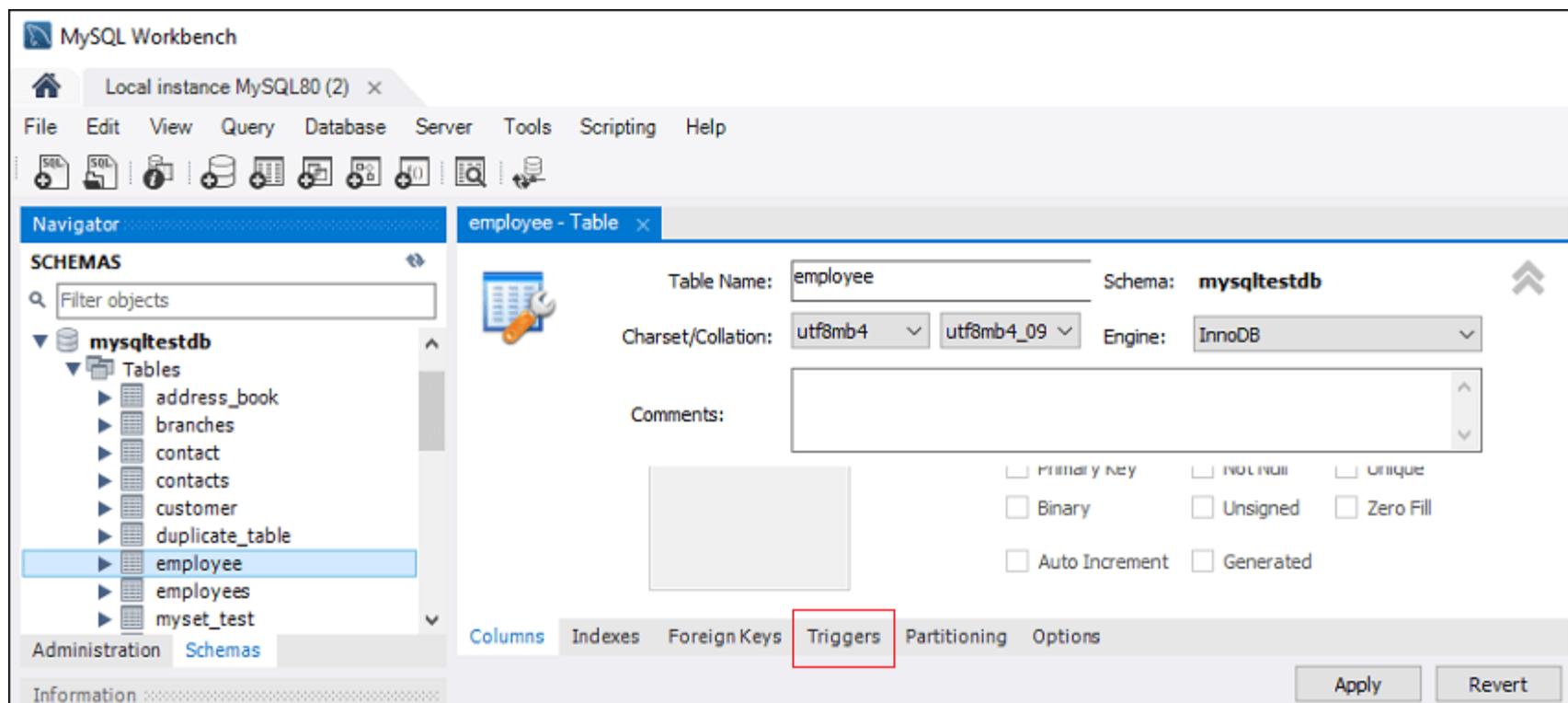
1. Go to the Navigation tab and click on the **Schema menu** that contains all the databases available in the MySQL server.
2. Select the database (for example, **mysqltestdb**), double click on it. It will show the **sub-menu** containing Tables, Views, Functions, and Stored Procedures. See the below screen.



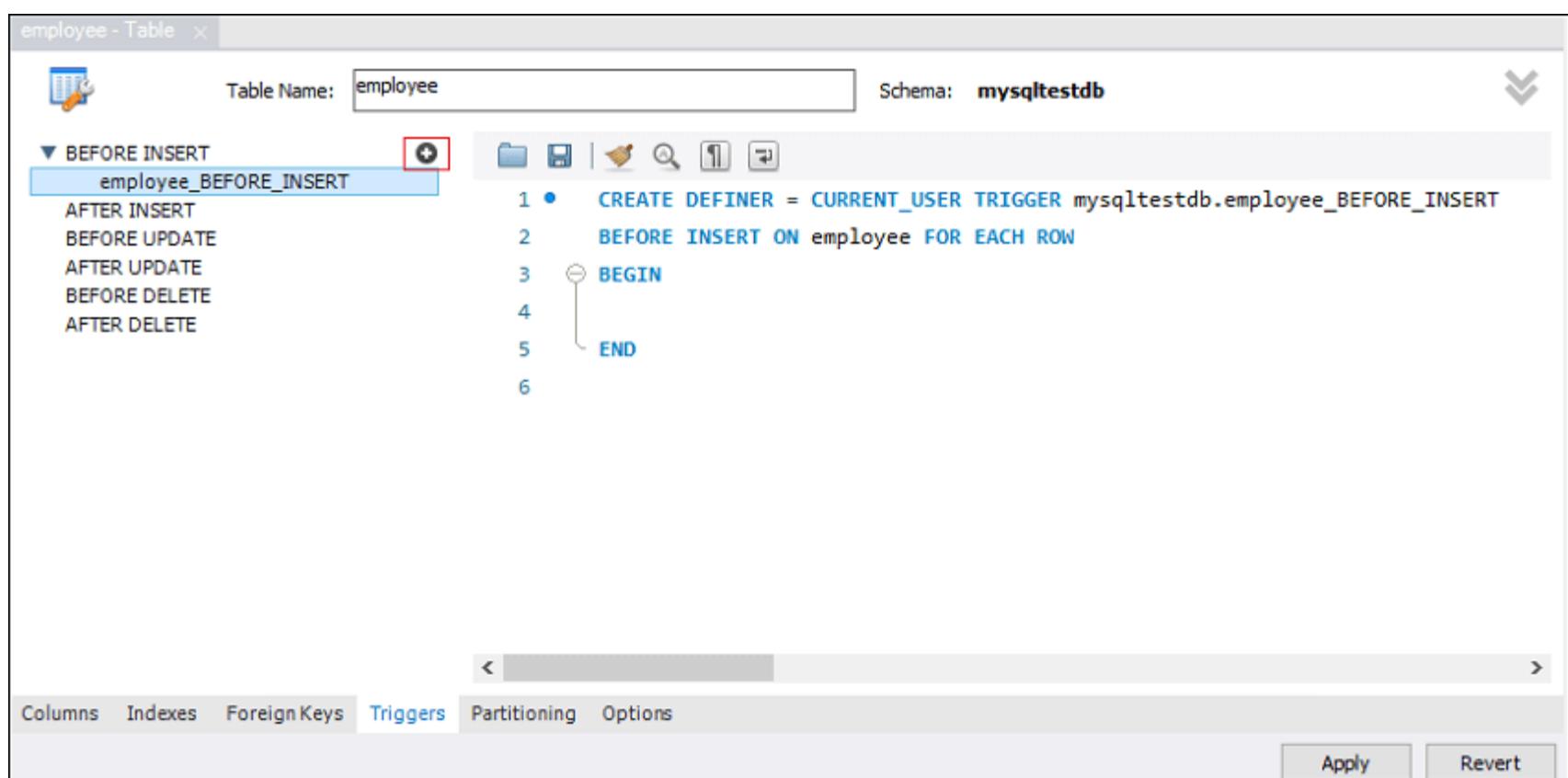
3. Expand the **Tables sub-menu** and select the table on which you want to create a trigger. After selecting a table, right-click on the selected table (for example, **employee**), and then click on the **Alter Table** option. See the below image:



4. Clicking on the Alter Table option gives the screen as below:



5. Now, click on the **Trigger tab** shown in the previous section's red rectangular box, then select the Timing/Event BEFORE INSERT. We will notice that there is a (+) icon button to add a trigger. Clicking on that button, we will get a default code on the trigger based on choosing Timing/Event:



6. Now, complete the trigger code, review them once again, and no error found, click on the **Apply button**.

Apply SQL Script to Database

Review SQL Script

Apply SQL Script

Review the SQL Script to be Applied on the Database

Online DDL

Algorithm: Default Lock Type: Default

```

1 USE `mysqltestdb`;
2
3 DELIMITER $$

5 USE `mysqltestdb`$$
6 DROP TRIGGER IF EXISTS `mysqltestdb`.`before_insert_occupation` $$

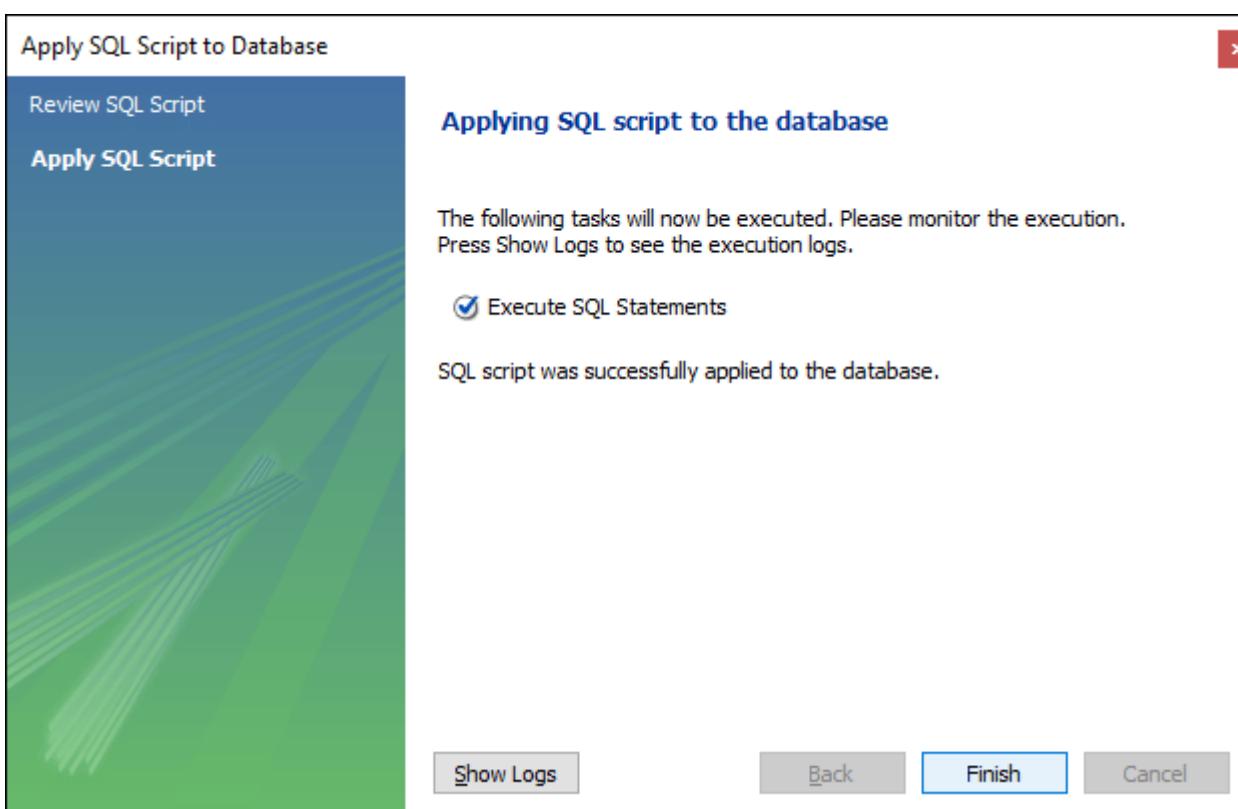
10 DELIMITER $$

11 USE `mysqltestdb`$$
12 CREATE DEFINER = CURRENT_USER TRIGGER mysqltestdb.employee_BEFORE_INSERT
    BEFORE INSERT ON employee FOR EACH ROW
14 BEGIN
15 IF NEW.occupation = 'Scientist' THEN SET NEW.occupation = 'Doctor';
16 END IF;
17 END$$
18 DELIMITER ;

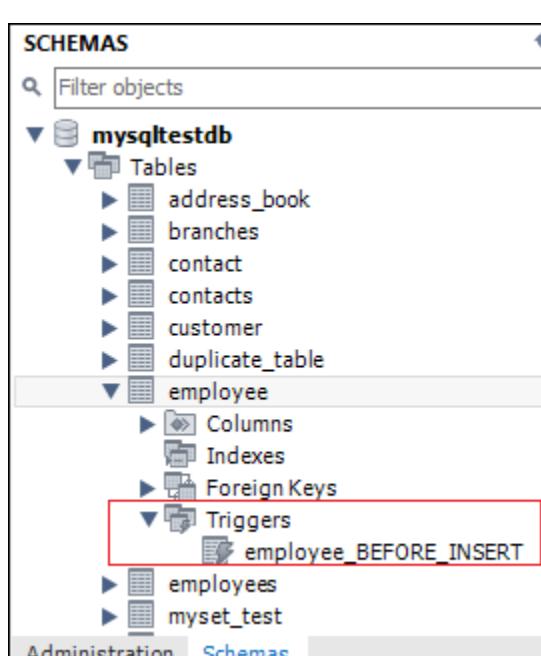
```

Back Apply Cancel

7. After clicking on the Apply button, click on the **Finish button** for completion.



8. If we take at the schema menu, we will see **employee\_BEFORE\_INSERT** trigger under the employee table as follows:



# MySQL AFTER INSERT Trigger

After Insert Trigger in MySQL is invoked automatically whenever an insert event occurs on the table. In this article, we are going to learn how to create an after insert trigger with its syntax and example.

## Syntax

The following is the syntax to create an **AFTER INSERT** trigger in MySQL:

1. **CREATE TRIGGER** trigger\_name
2. **AFTER INSERT**
3. **ON** table\_name **FOR EACH ROW**
4. trigger\_body ;

The AFTER INSERT trigger syntax parameter can be explained as below:

- o First, we will specify the **name of the trigger** that we want to create. It should be unique within the schema.
- o Second, we will specify the **trigger action time**, which should be AFTER INSERT clause to invoke the trigger.
- o Third, we will specify the **name of a table** to which the trigger is associated. It must be written after the ON keyword. If we did not specify the table name, a trigger would not exist.
- o Finally, we will specify the **trigger body** that contains one or more statements for execution when the trigger is activated.

If we want to execute multiple statements, we will use the **BEGIN END** block that contains a set of SQL queries to define the logic for the trigger. See the below syntax:

1. DELIMITER \$\$
2. **CREATE TRIGGER** trigger\_name **AFTER INSERT**
3. **ON** table\_name **FOR EACH ROW**
4. **BEGIN**
5. variable declarations
6. **trigger** code
7. **END\$\$**
8. DELIMITER ;

## Restrictions

- o We can access the **NEW** values but **cannot change them** in an AFTER INSERT trigger.
- o We cannot access the **OLD** If we try to access the OLD values, we will get an error because there is no OLD on the INSERT trigger.
- o We cannot create the AFTER INSERT trigger on a **VIEW**.

## AFTER INSERT Trigger Example

Let us understand how to create an AFTER INSERT trigger using the **CREATE TRIGGER** statement in MySQL with an example.

Suppose we have created a table named "**student\_info**" as follows:

1. **CREATE TABLE** student\_info (
2. stud\_id **int** NOT NULL,
3. stud\_code **varchar**(15) **DEFAULT** NULL,
4. stud\_name **varchar**(35) **DEFAULT** NULL,
5. subject **varchar**(25) **DEFAULT** NULL,
6. marks **int** **DEFAULT** NULL,
7. phone **varchar**(15) **DEFAULT** NULL,
8. **PRIMARY KEY** (stud\_id)
9. )

Next, we will insert some records into this table and then execute the **SELECT statement** to see the table data as follows:

The screenshot shows the MySQL 8.0 Command Line Client interface. The command entered is `SELECT * FROM student_info;`. The resulting table has columns: stud\_id, stud\_code, stud\_name, subject, marks, and phone. The data contains 8 rows of student information.

stud_id	stud_code	stud_name	subject	marks	phone
1	101	Mark	English	68	34545693537
2	102	Joseph	Physics	70	98765435659
3	103	John	Maths	70	97653269756
4	104	Barack	Maths	90	87698753256
5	105	Rinky	Maths	85	67531579757
6	106	Adam	Science	92	79642256864
7	107	Andrew	Science	83	56742437579
8	108	Brayan	Science	85	75234165670

8 rows in set (0.00 sec)

Again, we will create a new table named "**student\_detail**" as follows:

1. **CREATE TABLE** student\_detail (
2. stud\_id **int** NOT NULL,
3. stud\_code **varchar**(15) **DEFAULT** NULL,
4. stud\_name **varchar**(35) **DEFAULT** NULL,
5. subject **varchar**(25) **DEFAULT** NULL,
6. marks **int** **DEFAULT** NULL,
7. phone **varchar**(15) **DEFAULT** NULL,
8. Lasinserted **Time**,
9. **PRIMARY KEY** (stud\_id)
- 10.);

Next, we will use a CREATE TRIGGER statement to create an **after\_insert\_details** trigger on the **student\_info** table. This trigger will be fired after an insert operation is performed on the table.

1. mysql> DELIMITER //
2. mysql> **Create Trigger** after\_insert\_details
3. **AFTER INSERT ON** student\_info **FOR EACH ROW**
4. **BEGIN**
5. **INSERT INTO** student\_detail **VALUES** (new.stud\_id, new.stud\_code,
6. new.stud\_name, new.subject, new.marks, new.phone, CURTIME());
7. **END //**

If the trigger is created successfully, we will get the output as follows:

The screenshot shows the MySQL 8.0 Command Line Client interface. The command entered is `Create Trigger after_insert_details -> AFTER INSERT ON student_info FOR EACH ROW -> BEGIN -> INSERT INTO student_detail VALUES (new.stud_id, new.stud_code, new.stud_name, new.subject, new.marks, new.phone, CURTIME()); -> END //`. The output shows "Query OK, 0 rows affected (0.12 sec)".

## How to call the AFTER INSERT trigger?

We can use the following statements to invoke the above-created trigger:

1. mysql> **INSERT INTO** student\_info **VALUES**
2. (10, 110, 'Alexandar', 'Biology', 67, '2347346438');

The table that has been modified after the update query executes is **student\_detail**. We can verify it by using the **SELECT** statement as follows:

1. mysql> **SELECT \* FROM** student\_detail;

```

MySQL 8.0 Command Line Client

mysql> INSERT INTO student_info VALUES
-> (10, 110, 'Alexandar', 'Biology', 67, '2347346438');
Query OK, 1 row affected (0.09 sec)

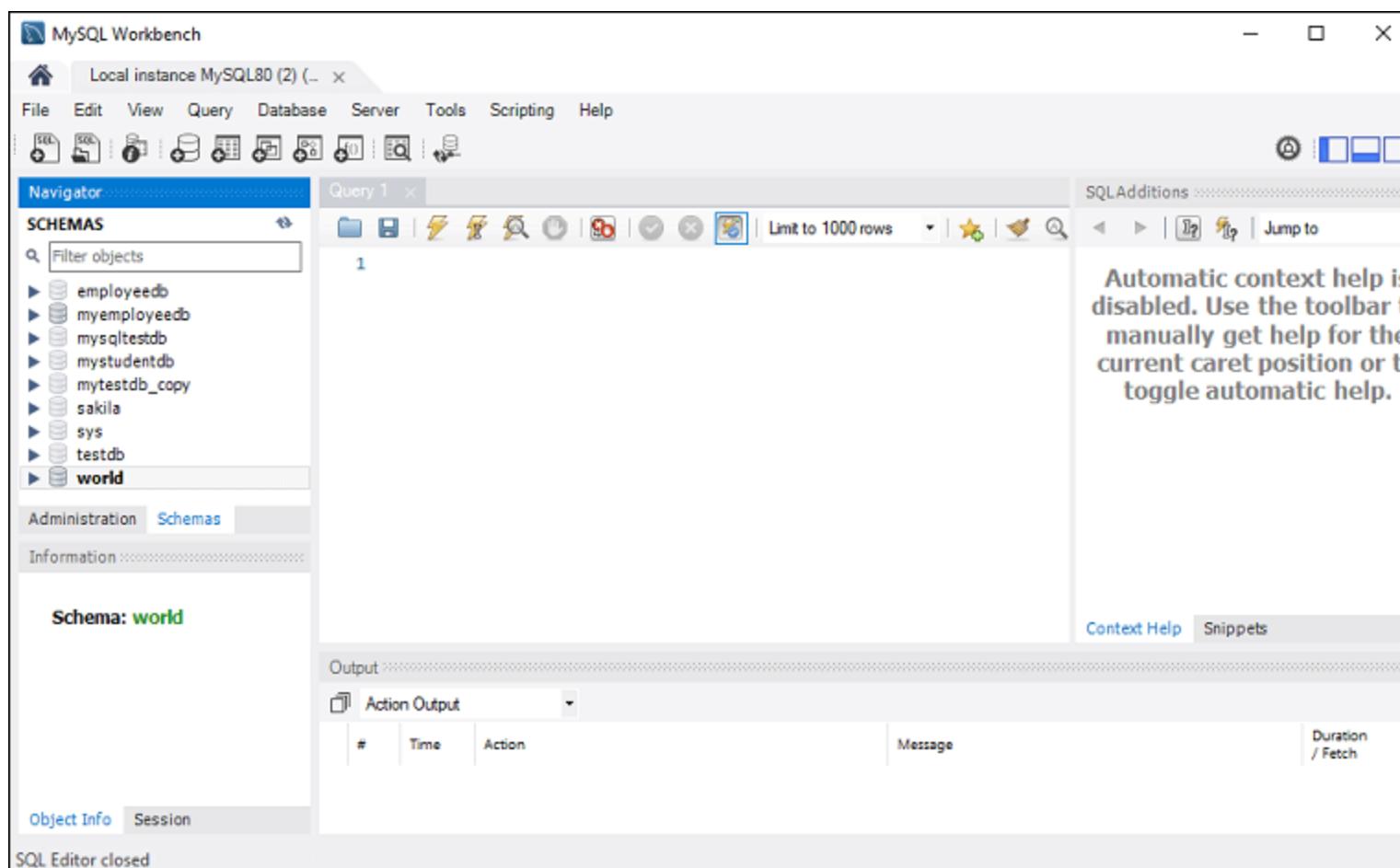
mysql> SELECT * FROM student_detail;
+-----+-----+-----+-----+-----+-----+
| stud_id | stud_code | stud_name | subject | marks | phone      | Lasinserted |
+-----+-----+-----+-----+-----+-----+
|    10   |    110    | Alexandar | Biology  |    67  | 2347346438 | 14:41:35   |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

```

In this output, we can see that on inserting values into the student\_info table, the student\_detail table will automatically fill the records by invoking a trigger.

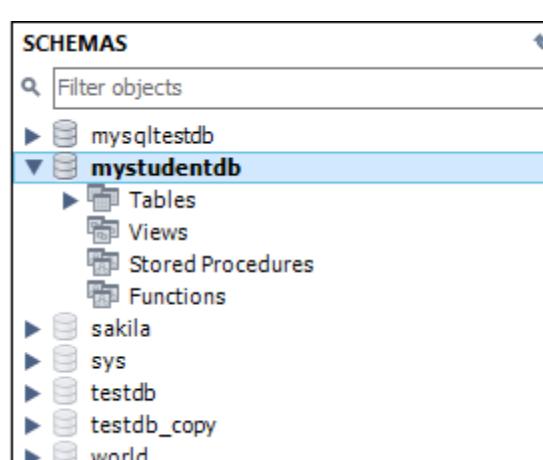
## How to create AFTER INSERT Trigger in MySQL workbench?

To create an after insert trigger using this tool, we first need to [launch the MySQL Workbench](#) and log in using the username and password that we have created earlier. We will get the screen as follows:

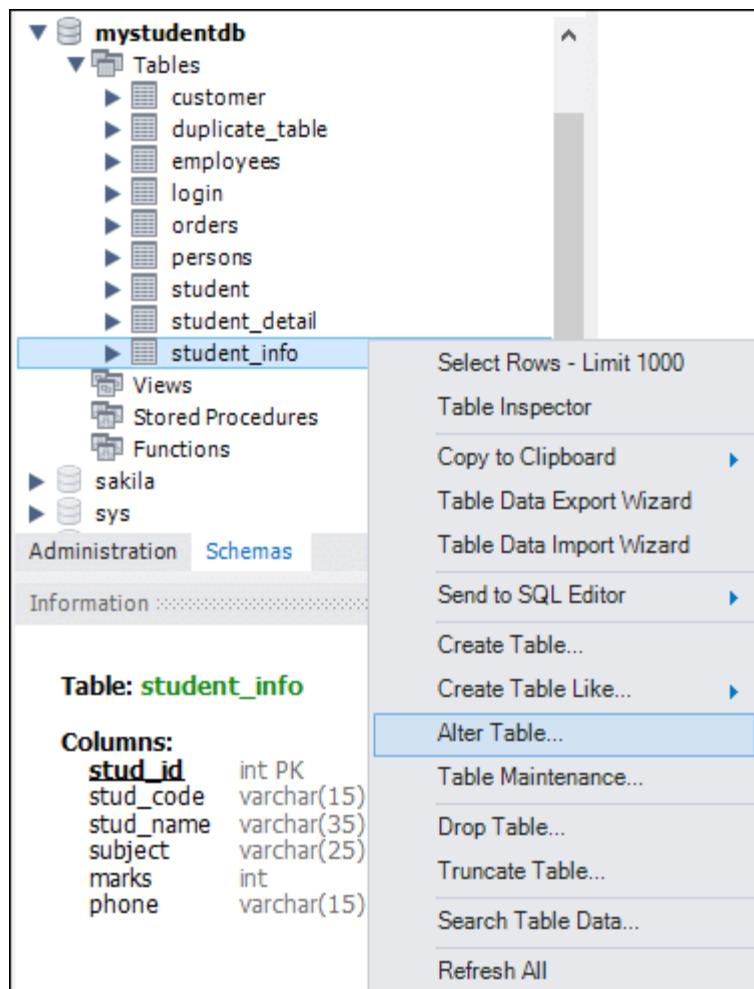


Now do the following steps for creating an AFTER INSERT trigger:

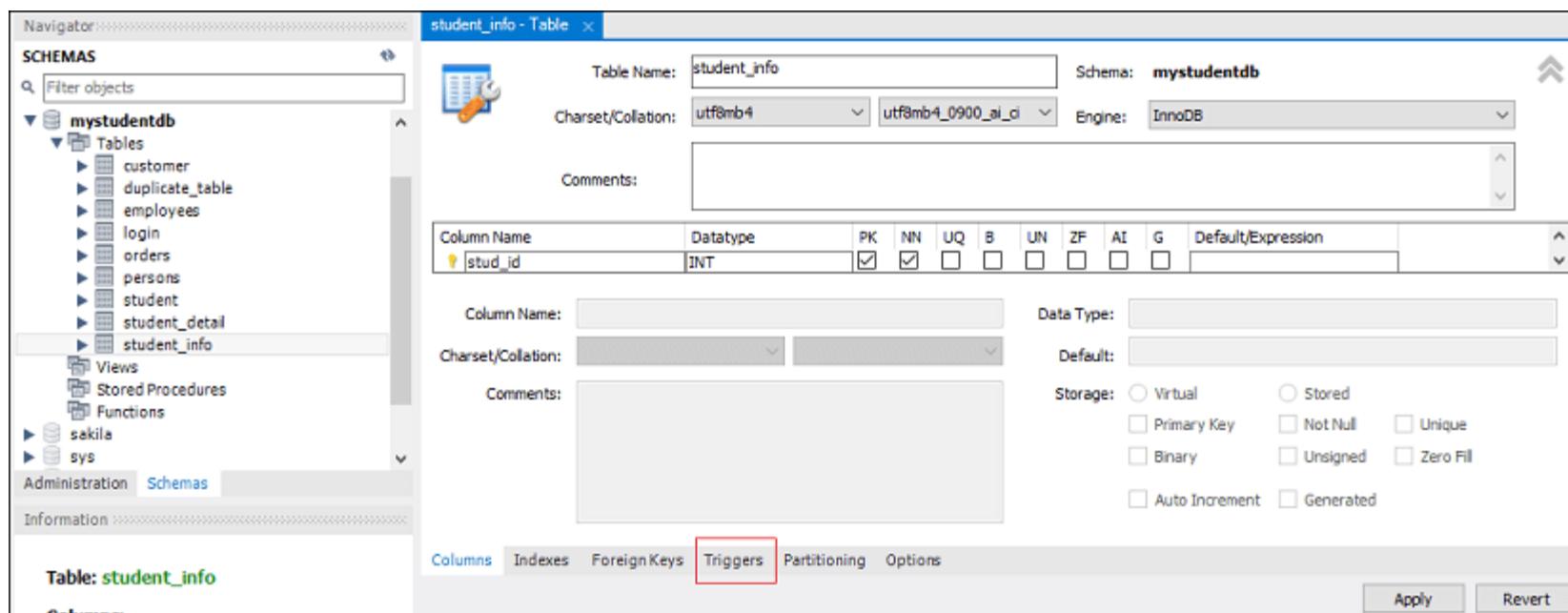
1. Go to the Navigation tab and click on the **Schema menu** that contains all the databases available in the MySQL server.
2. Select the database (for example, mystudentdb), double click on it that shows the **sub-menu** containing Tables, Views, Functions, and Stored Procedures. See the below screen.



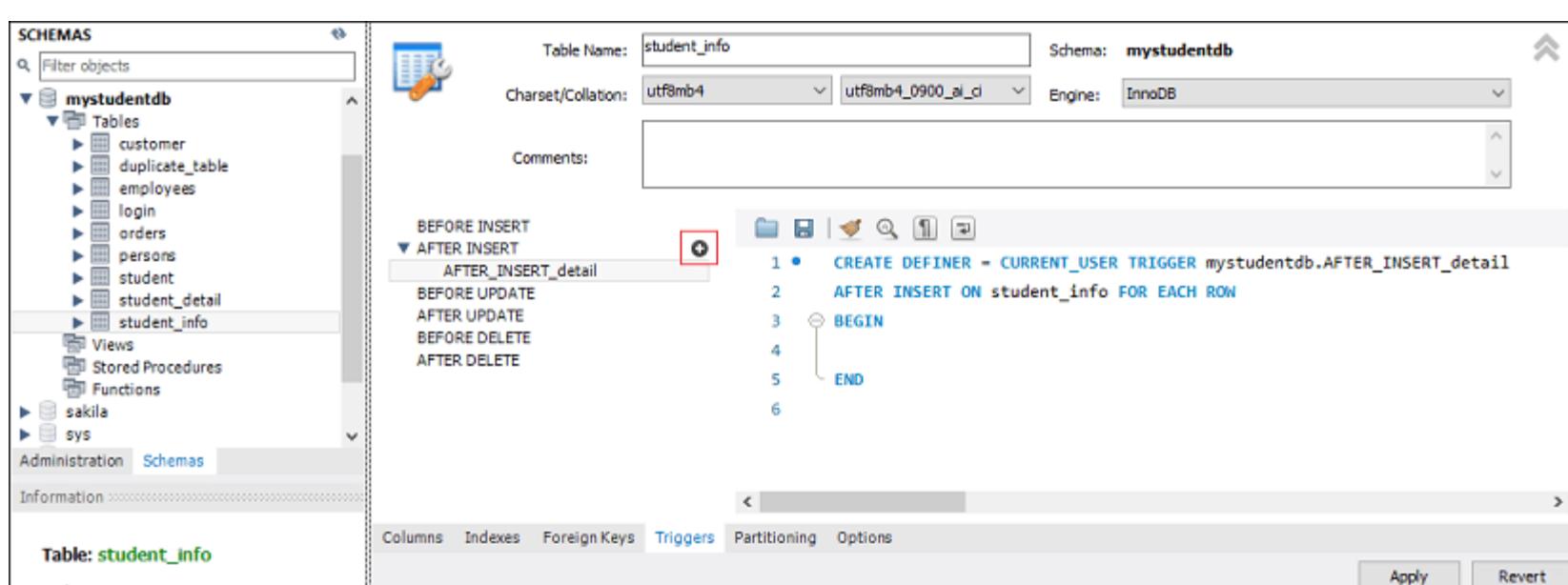
3. Expand the **Tables sub-menu** and select the table on which you want to create a trigger. After selecting a table, right-click on the selected table (**for example**, mystudentdb), and then click on the Alter Table option. See the below image:



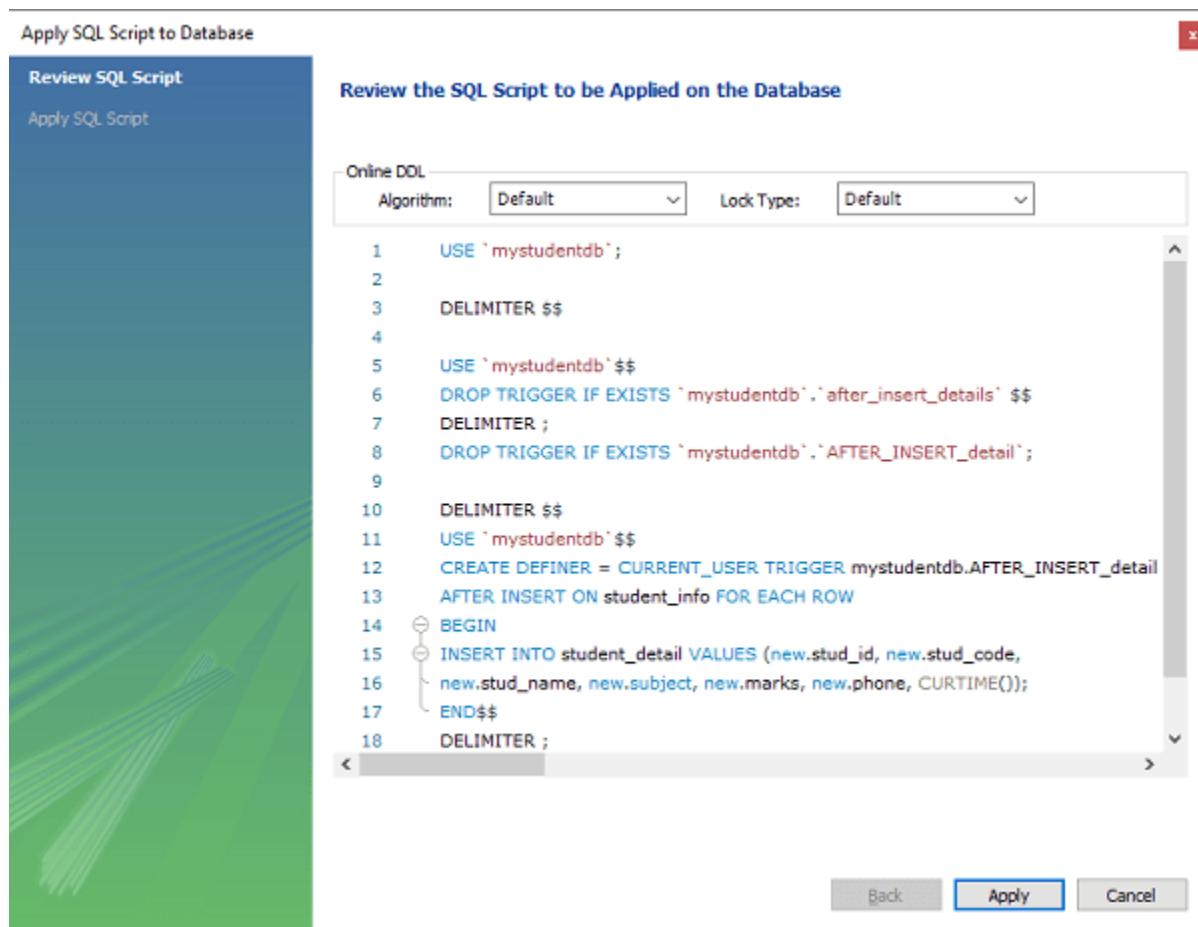
4. Clicking on the **Alter Table** option gives the screen as below:



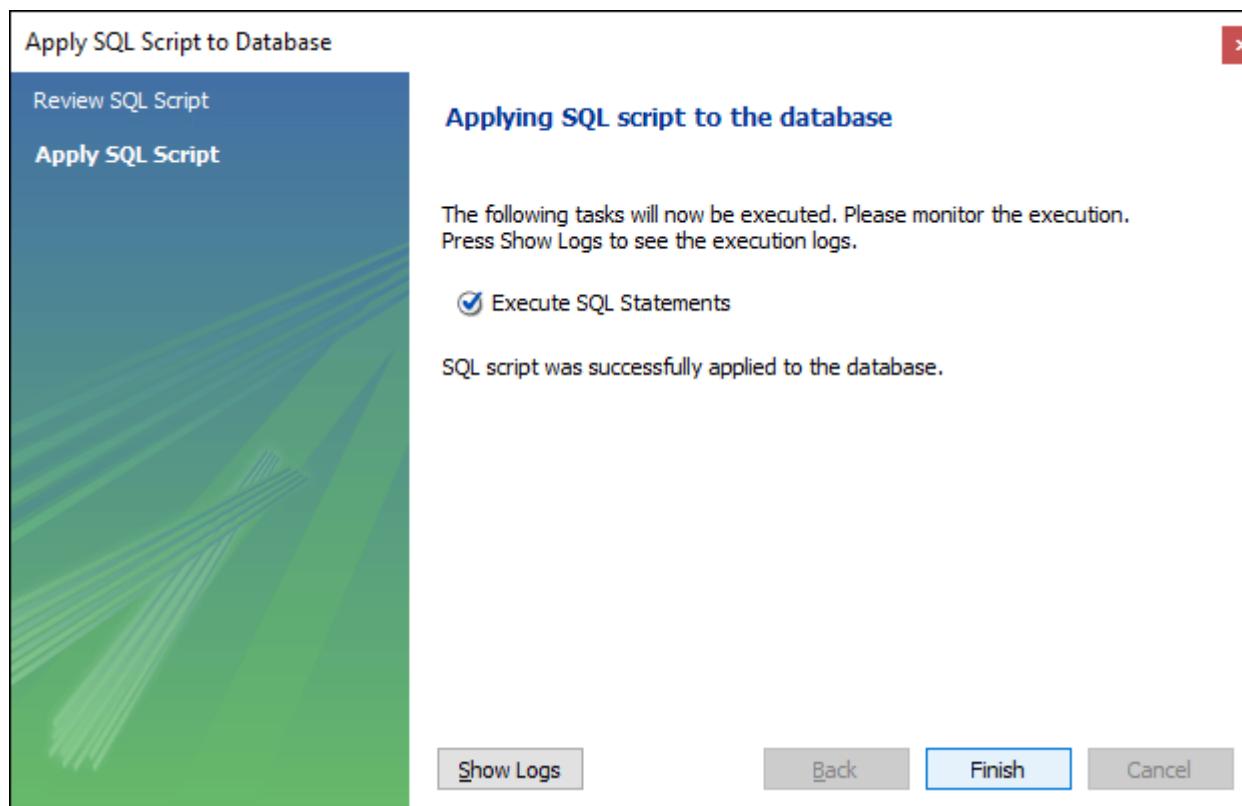
5. Now, click on the **Trigger tab** shown in the red rectangular box of the previous section, then select the Timing/Event **AFTER INSERT**. We will notice that there is a (+) icon button to add a trigger. Clicking on that button, we will get a default code on the trigger based on choosing Timing/Event:



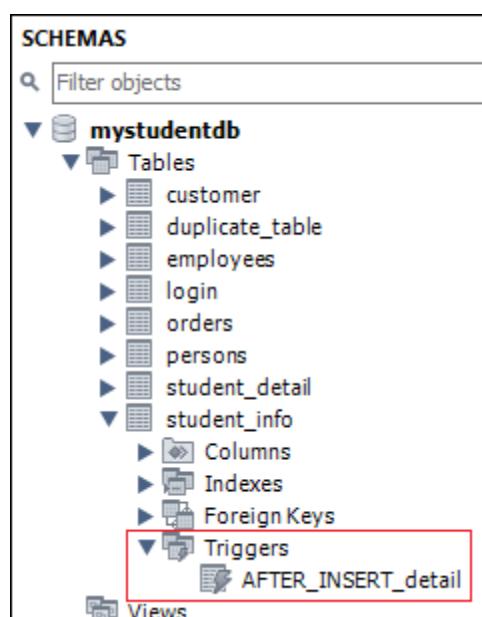
6. Now, complete the trigger code, review them once again, and if no error is found, click on the Apply button.



7. After clicking on the Apply button, click on the **Finish button** for completion.



8. If we look at the schema menu, we can see **AFTER\_INSERT\_detail** trigger under the **student\_info** table as follows:



## MySQL BEFORE UPDATE Trigger

BEFORE UPDATE Trigger in MySQL is invoked automatically whenever an update operation is fired on the table associated with the trigger. In this article, we are going to learn how to create a before update trigger with its syntax and example.

## Syntax

The following is the syntax to create a BEFORE UPDATE trigger in MySQL:

1. **CREATE TRIGGER** trigger\_name
2. **BEFORE UPDATE**
3. **ON** table\_name **FOR EACH ROW**
4. trigger\_body ;

The BEFORE UPDATE trigger syntax parameters are explained as below:

- o First, we will specify the **trigger name** that we want to create. It should be unique within the schema.
- o Second, we will specify the **trigger action time**, which should be BEFORE UPDATE. This trigger will be invoked before each row of alterations occurs on the table.
- o Third, we will specify the name of a table to which the trigger is associated. It must be written after the **ON keyword**. If we did not specify the table name, a trigger would not exist.
- o Finally, we will specify the **trigger body** that contains a statement for execution when the trigger is activated.

If we want to execute multiple statements, we will use the **BEGIN END** block that contains a set of queries to define the logic for the trigger. See the below syntax:

1. DELIMITER \$\$
2. **CREATE TRIGGER** trigger\_name **BEFORE UPDATE**
3. **ON** table\_name **FOR EACH ROW**
4. **BEGIN**
5. variable declarations
6. trigger code
7. **END\$\$**
8. DELIMITER ;

## Restrictions

- o We cannot update the OLD values in a BEFORE UPDATE trigger.
- o We can change the NEW values.
- o We cannot create a BEFORE UPDATE trigger on a VIEW.

## BEFORE UPDATE Trigger Example

Let us understand how to create a BEFORE UPDATE trigger using the [CREATE TRIGGER statement in MySQL](#) with an example.

Suppose we have created a table named **sales\_info** as follows:

1. **CREATE TABLE** sales\_info (
2. id **INT** AUTO\_INCREMENT,
3. product **VARCHAR**(100) NOT NULL,
4. quantity **INT** NOT NULL **DEFAULT** 0,
5. fiscalYear **SMALLINT** NOT NULL,
6. **CHECK**(fiscalYear BETWEEN 2000 and 2050),
7. **CHECK** (quantity >=0),
8. **UNIQUE**(product, fiscalYear),
9. **PRIMARY KEY**(id)
10. );

Next, we will insert some records into the sales\_info table as follows:

1. **INSERT INTO** sales\_info(product, quantity, fiscalYear)
2. **VALUES**
3. ('2003 Maruti Suzuki', 110, 2020),
4. ('2015 Avenger', 120, 2020),
5. ('2018 Honda Shine', 150, 2020),

6. ('2014 Apache', 150,2020);

Then, execute the **SELECT statement** to see the table data as follows:

The screenshot shows the MySQL 8.0 Command Line Client interface. The command entered is `SELECT * FROM sales_info;`. The resulting table has three columns: id, product, quantity, and fiscalYear. The data consists of four rows:

id	product	quantity	fiscalYear
1	2003 Maruti Suzuki	110	2020
2	2015 Avenger	120	2020
3	2018 Honda Shine	150	2020
4	2014 Apache	150	2020

4 rows in set (0.00 sec)

Next, we will use a **CREATE TRIGGER** statement to create a BEFORE UPDATE trigger. This trigger is invoked automatically before an update event occurs in the table.

1. DELIMITER \$\$
- 2.
3. **CREATE TRIGGER** before\_update\_salesInfo
4. BEFORE **UPDATE**
5. **ON** sales\_info **FOR** EACH ROW
6. **BEGIN**
7. **DECLARE** error\_msg **VARCHAR**(255);
8. **SET** error\_msg = ('The new quantity cannot be greater than 2 times the current quantity');
9. IF new.quantity > old.quantity \* 2 **THEN**
10. SIGNAL SQLSTATE '45000'
11. **SET** MESSAGE\_TEXT = error\_msg;
12. **END IF;**
13. **END \$\$**
- 14.
15. DELIMITER ;

If the trigger is created successfully, we will get the output as follows:

The screenshot shows the MySQL 8.0 Command Line Client interface. The commands entered are:

```
mysql>
mysql> DELIMITER $$
```

```
mysql> CREATE TRIGGER before_update_salesInfo
-> BEFORE UPDATE
-> ON sales_info FOR EACH ROW
-> BEGIN
->     DECLARE error_msg VARCHAR(255);
->     SET error_msg = ('The new quantity cannot be greater than 2 times the current quantity');
->     IF new.quantity > old.quantity * 2 THEN
->         SIGNAL SQLSTATE '45000'
->     SET MESSAGE_TEXT = error_msg;
->     END IF;
-> END $$
```

Query OK, 0 rows affected (0.26 sec)

```
mysql> DELIMITER ;
```

The trigger produces an error message and stops the updation if we update the value in the quantity column to a new value two times greater than the current value.

Let us understand the created trigger in details:

First, we have specified the trigger name as `before_update_salesInfo` in the **CREATE TRIGGER** clause. Second, specify the triggering event and then the table name on which the trigger is associated. Third, we have declared a variable and set its value. Finally, we have specified the trigger body that checks if the new value is two times greater than the old value and then raises an error.

## How to call the BEFORE UPDATE trigger?

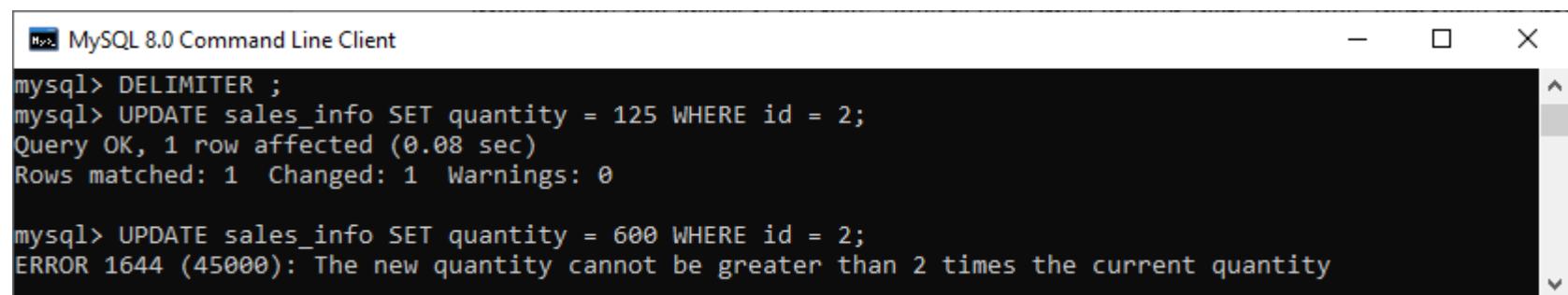
First, we can use the following statements that update the quantity of the row whose id = 2:

1. mysql> **UPDATE** sales\_info **SET** quantity = 125 **WHERE** id = 2;

This statement works well because it does not violate the rule. Next, we will execute the below statements that update the quantity of the row as 600 whose id = 2

1. mysql> **UPDATE** sales\_info **SET** quantity = 600 **WHERE** id = 2;

It will give the error as follows because it violates the rule. See the below output.

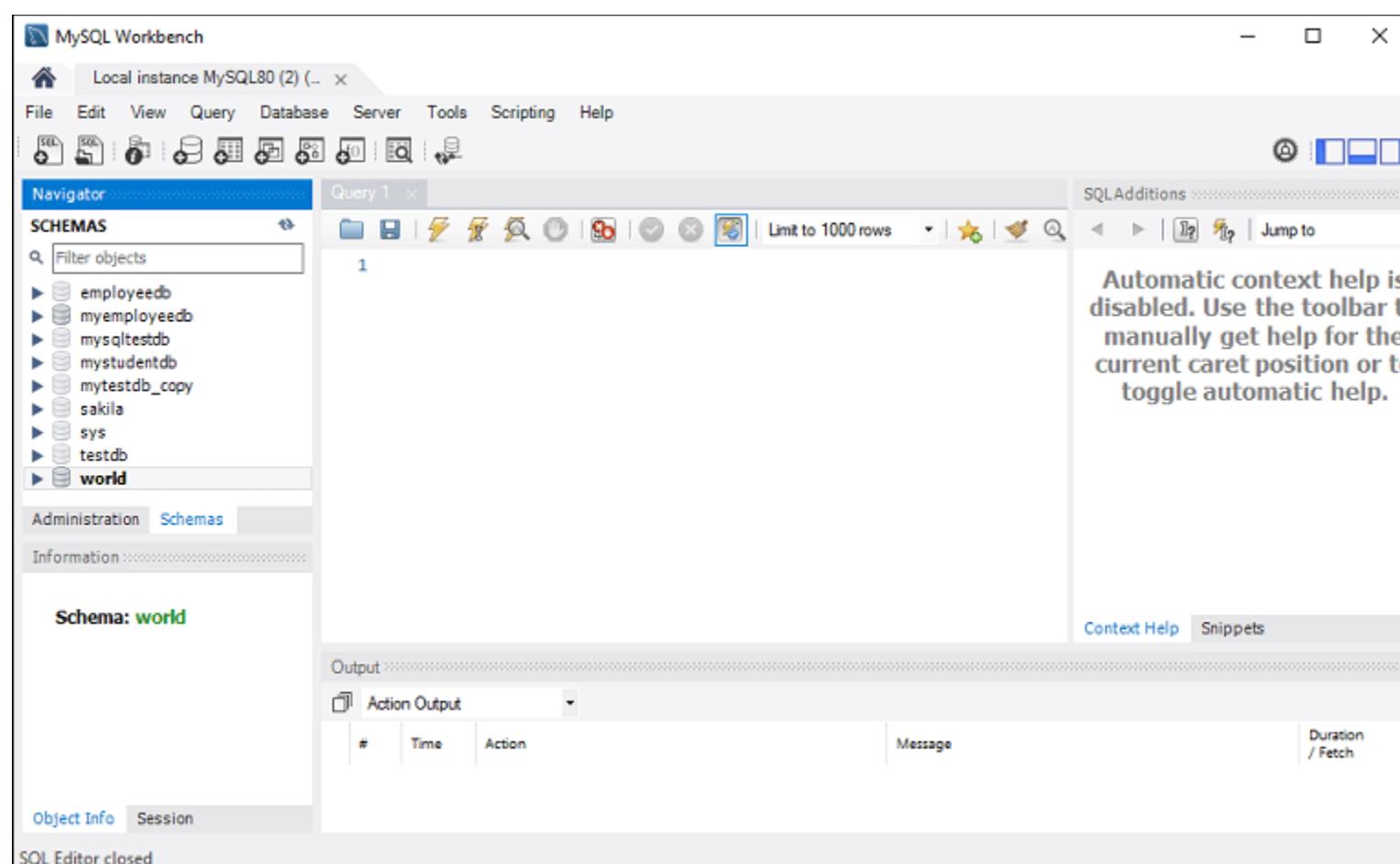


```
MySQL 8.0 Command Line Client
mysql> DELIMITER ;
mysql> UPDATE sales_info SET quantity = 125 WHERE id = 2;
Query OK, 1 row affected (0.08 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> UPDATE sales_info SET quantity = 600 WHERE id = 2;
ERROR 1644 (45000): The new quantity cannot be greater than 2 times the current quantity
```

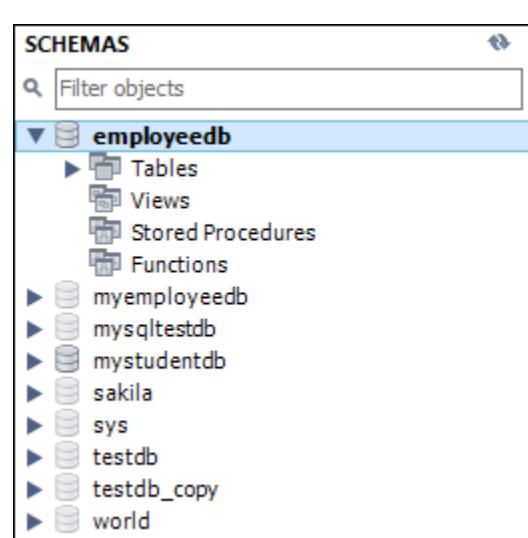
## How to create BEFORE UPDATE Trigger in MySQL workbench?

To create a BEFORE UPDATE trigger using [MySQL workbench](#), we first need to launch it and then log in using the username and password we created earlier. We will get the screen as follows:



Now do the following steps for creating BEFORE UPDATE trigger:

1. Go to the Navigation tab and click on the **Schema menu** that contains all the databases available in the [MySQL](#) server.
2. Select the database (for example, **employeedb**), double click on it, and display the **sub-menu** containing Tables, Views, Functions, and Stored Procedures. See the below screen.



3. Expand the **Tables sub-menu** and select the table on which you want to create a trigger. After selecting a table, right-click on the selected table (for example, sales\_info), and then click on the **Alter Table** option. See the below image:

The screenshot shows the MySQL Workbench interface under the 'Schemas' tab. The 'sales\_info' table is selected in the list. A context menu is open, with the 'Alter Table...' option highlighted. Other options visible in the menu include 'Select Rows - Limit 1000', 'Table Inspector', 'Copy to Clipboard', 'Table Data Export Wizard', 'Table Data Import Wizard', 'Send to SQL Editor', 'Create Table...', 'Create Table Like...', 'Table Maintenance...', 'Drop Table...', 'Truncate Table...', and 'Search Table Data...'. Below the table list, there is information about the 'sales\_info' table, including its columns: id (int AI PK), product (varchar(100)), quantity (int), and fiscalYear (smallint).

4. Clicking on the Alter Table option gives the screen as below:

This screenshot shows the 'Alter Table' dialog for the 'sales\_info' table. The 'Table Name' is set to 'sales\_info', 'Schema' to 'employeedb', 'Charset/Collation' to 'utf8mb4 utf8mb4\_0900\_ai\_ci', and 'Engine' to 'InnoDB'. The 'Comments' field is empty. The 'Columns' tab is selected, showing one column: 'id' (INT, PK, NN, UQ, AI). The 'Triggers' tab is also selected, showing a list of triggers: 'sales\_info\_BEFORE\_UPDATE' (selected), 'sales\_info\_AFTER\_UPDATE', 'sales\_info\_BEFORE\_INSERT', 'sales\_info\_AFTER\_INSERT', 'sales\_info\_BEFORE\_DELETE', and 'sales\_info\_AFTER\_DELETE'. The 'Default/Expression' section for the 'id' column is visible, along with options for 'Data Type', 'Default', 'Storage', and 'Comments'. At the bottom, there are 'Apply' and 'Revert' buttons.

5. Now, click on the **Trigger tab** shown in the previous section's red rectangular box, then select the Timing/Event BEFORE UPDATE. We will notice that there is a (+) icon button to add a trigger. Clicking on that button, we will get a default code on trigger based on choosing Timing/Event:

This screenshot shows the 'Alter Table' dialog for the 'sales\_info' table. The 'Table Name' is set to 'sales\_info', 'Schema' to 'employeedb', 'Charset/Collation' to 'utf8mb4 utf8mb4\_0900\_ai\_ci', and 'Engine' to 'InnoDB'. The 'Comments' field is empty. The 'Triggers' tab is selected, showing a list of triggers: 'sales\_info\_BEFORE\_UPDATE' (selected), 'sales\_info\_AFTER\_UPDATE', 'sales\_info\_BEFORE\_INSERT', 'sales\_info\_AFTER\_INSERT', 'sales\_info\_BEFORE\_DELETE', and 'sales\_info\_AFTER\_DELETE'. A red box highlights the '+' icon next to 'sales\_info\_BEFORE\_UPDATE'. The trigger code is displayed in the main area:

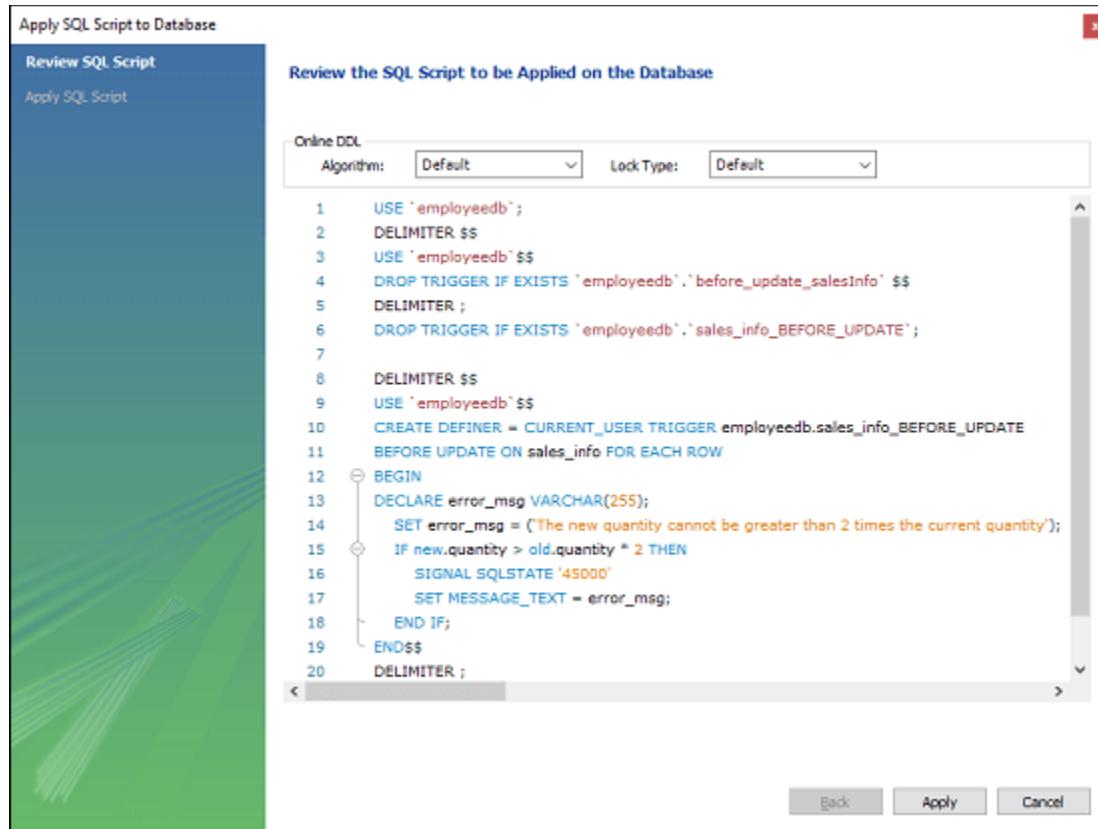
```

1 *  CREATE DEFINER = CURRENT_USER TRIGGER employeedb.sales_info_BEFORE_UPDATE
2   BEFORE UPDATE ON sales_info FOR EACH ROW
3     BEGIN
4
5   END

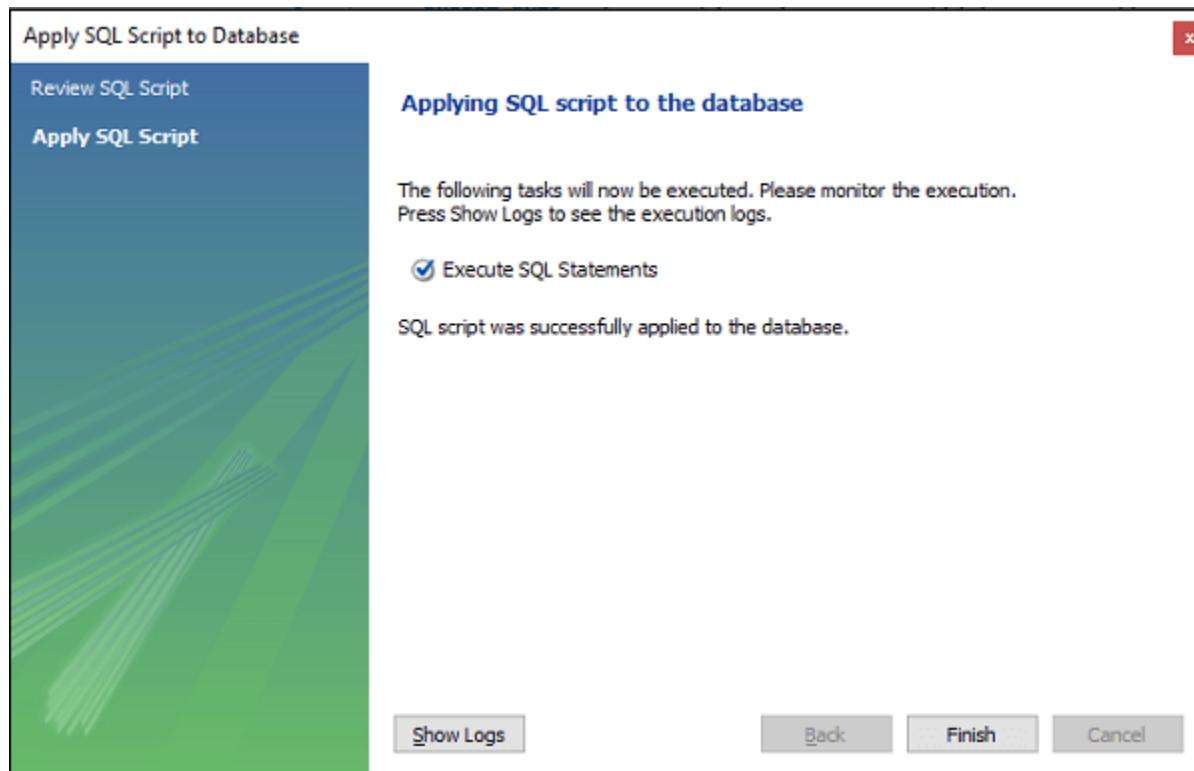
```

At the bottom, there are 'Columns', 'Indexes', 'Foreign Keys', 'Triggers' (selected), 'Partitioning', and 'Options' tabs, along with 'Apply' and 'Revert' buttons.

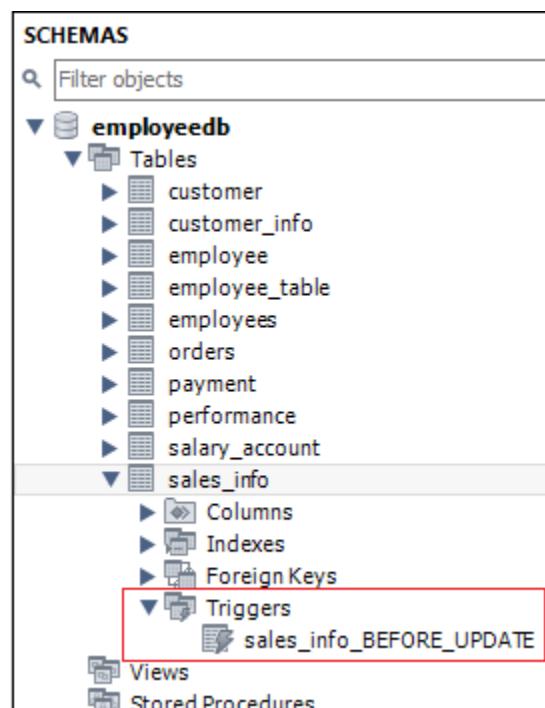
6. Now, complete the trigger code, review them once again, and if no error is found, click on the **Apply** button.



7. After clicking on the Apply button, click on the **Finish** button for completion.



8. If we take at the schema menu, we will see the [trigger sales\\_info\\_before\\_update](#) under the sales\_info table as follows:



## MySQL AFTER UPDATE TRIGGER

The AFTER UPDATE trigger in MySQL is invoked automatically whenever an UPDATE event is fired on the table associated with the triggers. In this article, we are going to learn how to create an AFTER UPDATE trigger with its syntax and example.

## Syntax

The following is the syntax to create an **AFTER UPDATE** trigger in MySQL:

1. **CREATE TRIGGER** trigger\_name
2. **AFTER UPDATE**
3. **ON** table\_name **FOR EACH ROW**
4. trigger\_body;

We can explain the parameters of AFTER UPDATE trigger syntax as below:

- o First, we will specify the **trigger name** that we want to create. It should be unique within the schema.
- o Second, we will specify the **trigger action time**, which should be AFTER UPDATE. This trigger will be invoked after each row of alterations occurs on the table.
- o Third, we will specify the **table name** to which the trigger is associated. It must be written after the **ON**. If we did not specify the table name, a trigger would not exist.
- o Finally, we will specify the **trigger body** that contains a statement for execution when the trigger is activated.

If we want to execute more than one statement, we will use the **BEGIN END** block that contains a set of SQL queries to define the logic for the **trigger**. See the below syntax:

1. DELIMITER \$\$
2. **CREATE TRIGGER** trigger\_name **AFTER UPDATE**
3. **ON** table\_name **FOR EACH ROW**
4. **BEGIN**
5. variable declarations
6. **trigger** code
7. **END\$\$**
8. DELIMITER ;

## Restrictions

- o We can access the OLD rows but cannot update them.
- o We can access the NEW rows but cannot update them.
- o We cannot create an AFTER UPDATE trigger on a **VIEW**.

## AFTER UPDATE Trigger Example

Let us understand how to create an AFTER UPDATE trigger using the [CREATE TRIGGER statement in MySQL](#) with an example.

Suppose we have created a table named **students** to store the student's information as follows:

1. mysql> **CREATE TABLE** students(
2.   id **int** NOT NULL AUTO\_INCREMENT,
3.   **name varchar**(45) NOT NULL,
4.   class **int** NOT NULL,
5.   email\_id **varchar**(65) NOT NULL,
6.   **PRIMARY KEY** (id)
7. );

Next, we will insert some records into this table using the below statement:

1. **INSERT INTO** students (**name**, class, email\_id)
2. **VALUES** ('Stephen', 6, 'stephen@javatpoint.com'),
3. ('Bob', 7, 'bob@javatpoint.com'),
4. ('Steven', 8, 'steven@javatpoint.com'),
5. ('Alexandar', 7, 'alexandar@javatpoint.com');

Execute the **SELECT** query to see the table data.

```

MySQL 8.0 Command Line Client
mysql> SELECT * FROM students;
+----+-----+-----+-----+
| id | name | class | email_id |
+----+-----+-----+-----+
| 1  | Stephen | 6 | stephen@javatpoint.com |
| 2  | Bob | 7 | bob@javatpoint.com |
| 3  | Steven | 8 | steven@javatpoint.com |
| 4  | Alexandar | 7 | alexandar@javatpoint.com |
+----+-----+-----+-----+
4 rows in set (0.00 sec)

```

Third, we will create another table named **students\_log** that keeps the updated information in the selected user.

1. mysql> **CREATE TABLE** students\_log(
2.   **user** **varchar**(45) NOT NULL,
3.   **descreptions** **varchar**(65) NOT NULL
4. );

We will then create an AFTER UPDATE **trigger that promotes all students in the next class**, i.e., 6 will be 7, 7 will be 8, and so on. Whenever an updation is performed on a single row in the "students" table, a new row will be inserted in the "students\_log" table. This table keeps the **current user id** and a **description** regarding the current update. See the below trigger code.

1. DELIMITER \$\$
- 2.
3. **CREATE TRIGGER** after\_update\_studentsInfo
4. **AFTER UPDATE**
5. **ON** students **FOR EACH ROW**
6. **BEGIN**
7.   **INSERT into** students\_log **VALUES** (**user**()),
8.   **CONCAT**('Update Student Record ', OLD.**name**, ' Previous Class :',
9.    OLD.class, ' Present Class ', NEW.class));
10. **END** \$\$
- 11.
12. DELIMITER ;

```

MySQL 8.0 Command Line Client
mysql> DELIMITER $$

mysql> CREATE TRIGGER after_update_studentsInfo
-> AFTER UPDATE
-> ON students FOR EACH ROW
-> BEGIN
->   INSERT into students_log VALUES (user()),
->   CONCAT('Update Student Record ', OLD.name, ' Previous Class :',
->   OLD.class, ' Present Class ', NEW.class));
-> END $$

Query OK, 0 rows affected (0.23 sec)

mysql> DELIMITER ;

```

In this trigger, we have first specified the trigger name **after\_update\_studentsInfo**. Then, specify the triggering event. Third, we have specified the table name on which the trigger is associated. Finally, we have written the trigger logic inside the trigger body that performs updation in the "students" table and keeps the log information in the "students\_log" table.

## How to call the AFTER UPDATE trigger?

First, we will update the "students" table using the following statements that invoke the above-created trigger:

1. mysql> **UPDATE** students **SET** class = class + 1;

Next, we will query data from the **students** and **students\_log table**. We can see that table has been updated after the execution of the query. See the below output:

```

MySQL 8.0 Command Line Client
mysql> UPDATE students SET class = class + 1;
Query OK, 4 rows affected (0.15 sec)
Rows matched: 4  Changed: 4  Warnings: 0

mysql> SELECT * FROM students;
+----+-----+-----+
| id | name | class |
+----+-----+-----+
| 1  | Stephen | 7    |
| 2  | Bob     | 8    |
| 3  | Steven   | 9    |
| 4  | Alexandar | 8    |
+----+-----+-----+
4 rows in set (0.09 sec)

```

Again, we will query data from the students\_log table that keeps the current user id and a description regarding the current update. See the below output:

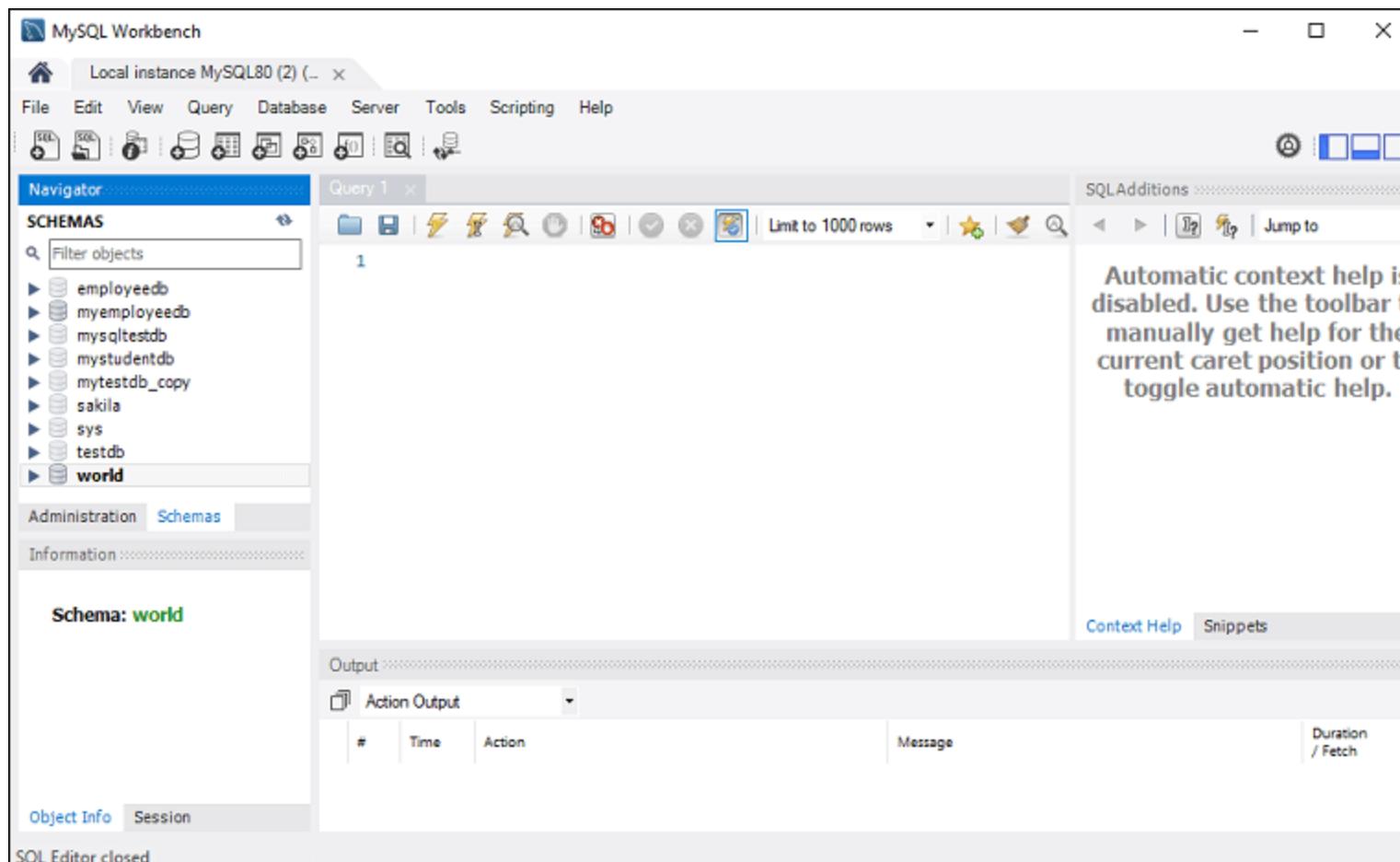
```

MySQL 8.0 Command Line Client
mysql> SELECT * FROM students_log;
+-----+-----+
| user      | descreptions          |
+-----+-----+
| root@localhost | Update Student Record Stephen Previous Class :6 Present Class 7 |
| root@localhost | Update Student Record Bob Previous Class :7 Present Class 8 |
| root@localhost | Update Student Record Steven Previous Class :8 Present Class 9 |
| root@localhost | Update Student Record Alexandar Previous Class :7 Present Class 8 |
+-----+-----+
4 rows in set (0.00 sec)

```

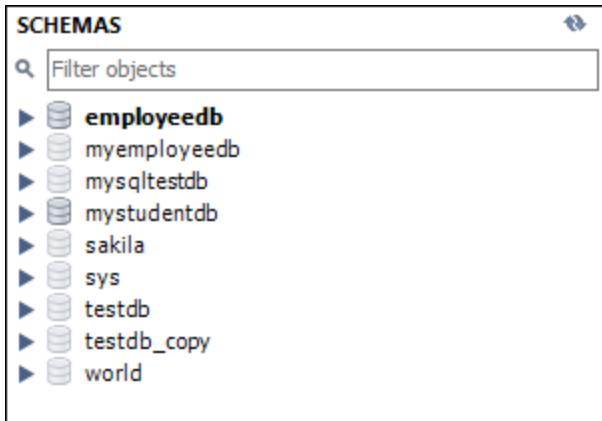
## How to create AFTER UPDATE trigger in MySQL workbench?

To create an AFTER UPDATE trigger in workbench, we first [launch the MySQL Workbench](#) and log in using the username and password. We will get the UI as follows:

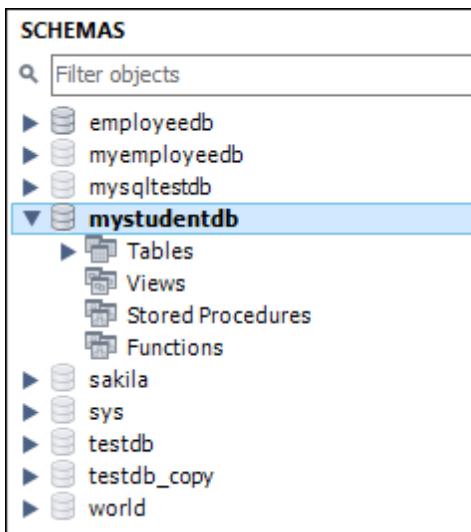


Now do the following steps to create an AFTER UPDATE trigger:

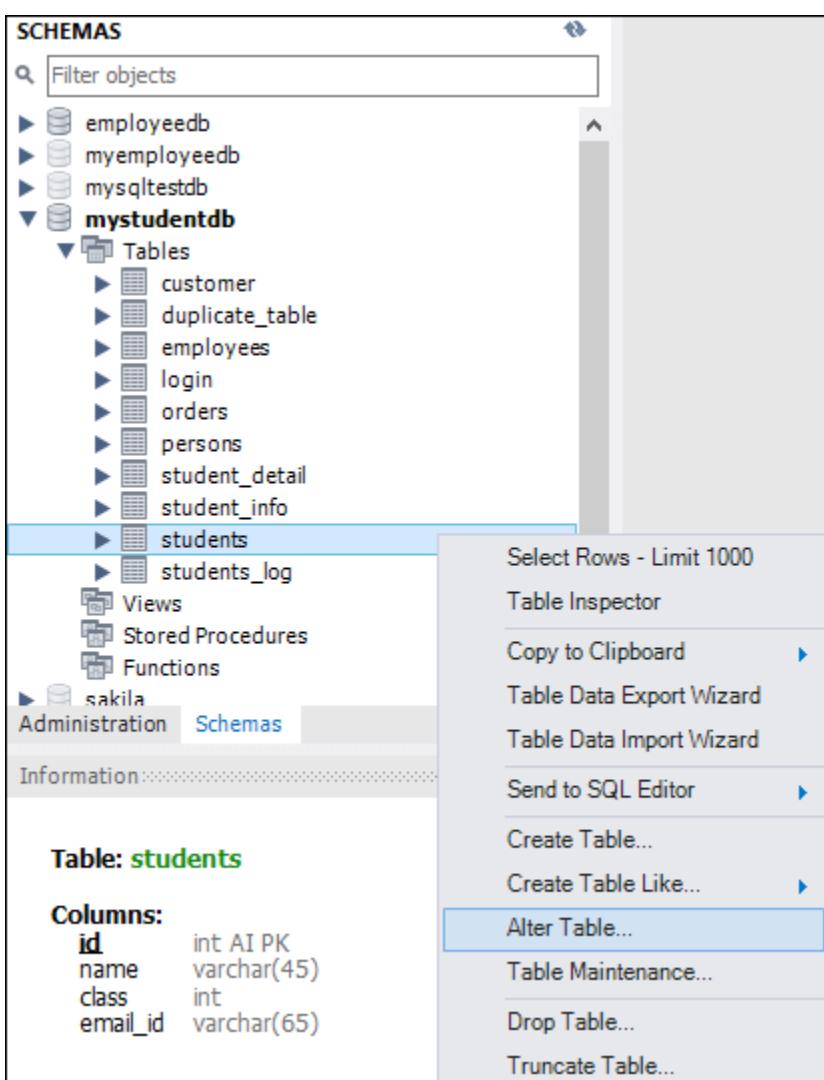
1. Go to the **Navigation** tab and click on the **Schema** menu. It will display all databases available in the [MySQL](#) database server.



2. Select the database (for example, **mystudentdb**). Then, double click on the selected schema. It displays the **sub-menu** containing Tables, Views, Functions, and Stored Procedures. See the below screen.



3. Expand the **Tables** sub-menu and select a table on which you want to create a trigger. Then, right-click on the selected table (**for example, students**), and click on the **Alter Table** option. See the below image:



4. Clicking on the **Alter Table** option gives the screen as below:

Table Name: students Schema: mystudentdb

Charset/Collation: utf8mb4 utf8mb4\_0900\_ai\_ci Engine: InnoDB

Comments:

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
id	INT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>					

Column Name: Data Type: Default: Storage:

Charset/Collation: Comments:

Virtual      Stored  
 Primary Key       Not Null       Unique  
 Binary       Unsigned       Zero Fill  
 Auto Increment       Generated

Columns Indexes Foreign Keys Triggers Partitioning Options Apply Revert

5. Now, click on the **Trigger** tab shown in the previous section's **red rectangular box**, then select the **Timing/Event AFTER UPDATE**. We will notice that there is a **(+)** icon button to add a trigger. Clicking on that button, we will get a default code on the trigger based on choosing Timing/Event:

Table Name: students Schema: mystudentdb

Charset/Collation: utf8mb4 utf8mb4\_0900\_ai\_ci Engine: InnoDB

Comments:

BEFORE INSERT  
AFTER INSERT  
BEFORE UPDATE  
**AFTER UPDATE**  
student\_update\_trigger  
BEFORE DELETE  
AFTER DELETE

```

CREATE DEFINER = CURRENT_USER TRIGGER mystudentdb.student_update_trigger
AFTER UPDATE ON students FOR EACH ROW
BEGIN
    INSERT into students_log VALUES (user(),
        CONCAT('Update Student Record ', OLD.name, ' Previous Class: ',
        OLD.class, ' Present Class: ', NEW.class));
END

```

Columns Indexes Foreign Keys Triggers Partitioning Options Apply Revert

6. Now, complete the trigger code, review them once again, and if no error is found, click on the **Apply** button.

Apply SQL Script to Database

Review SQL Script

Review the SQL Script to be Applied on the Database

Online DDL

Algorithm: Default Lock Type: Default

```

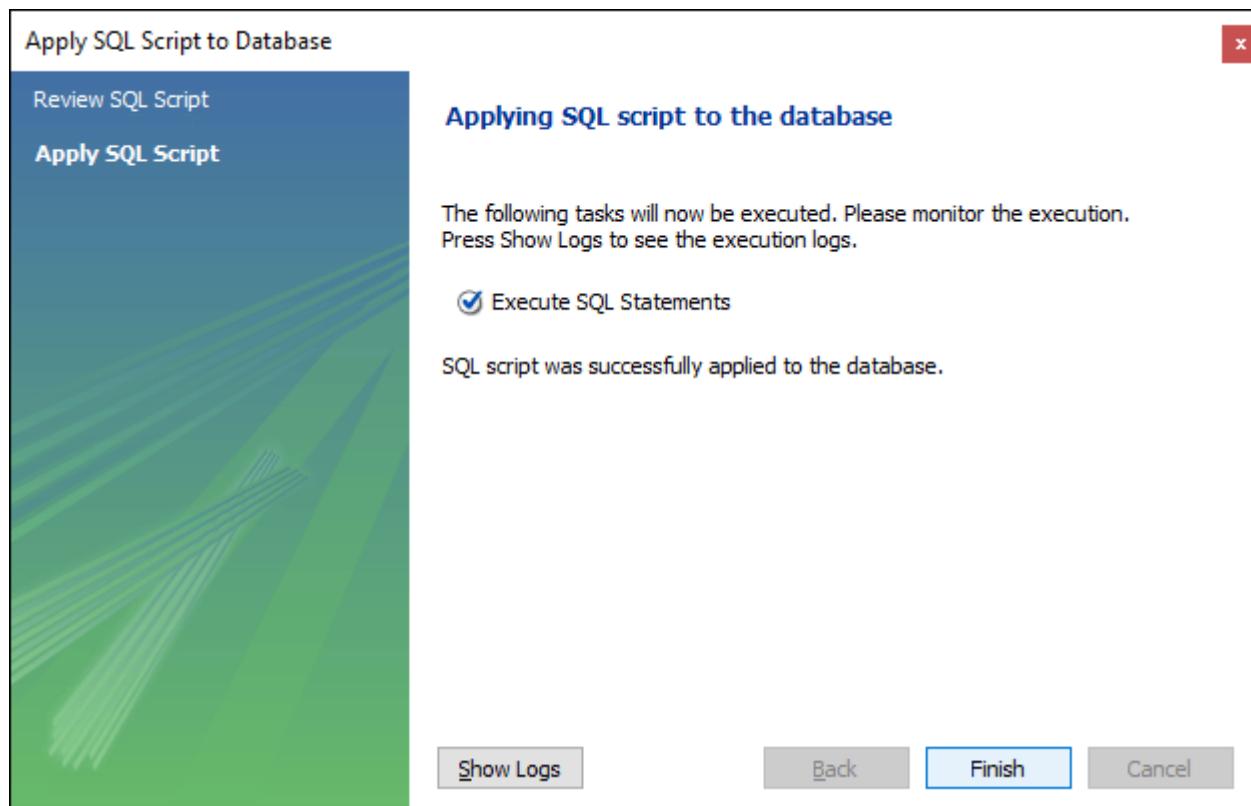
DROP TRIGGER IF EXISTS `mystudentdb`.`student_update_trigger`;

DELIMITER $$
USE `mystudentdb`$$
CREATE DEFINER = CURRENT_USER TRIGGER mystudentdb.student_update_trigger
AFTER UPDATE ON students FOR EACH ROW
BEGIN
    INSERT into students_log VALUES (user(),
        CONCAT('Update Student Record ', OLD.name, ' Previous Class: ',
        OLD.class, ' Present Class: ', NEW.class));
END$$
DELIMITER ;

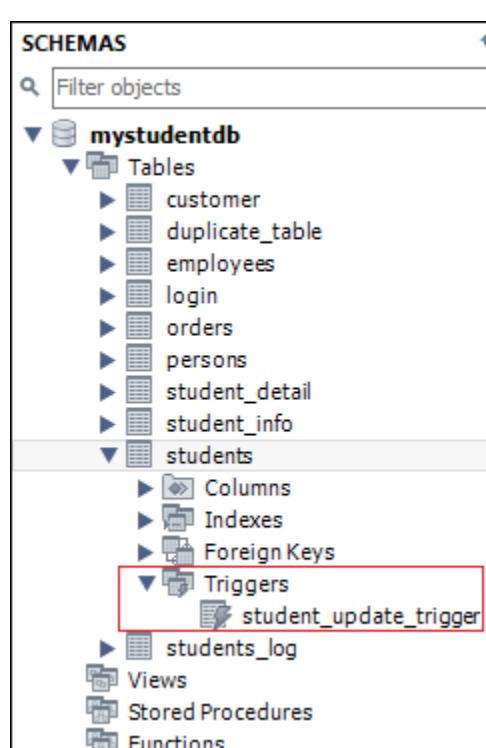
```

Back Apply Cancel

7. After clicking on the **Apply** button, click on the **Finish button** to complete the process.



8. If we look at the schema menu, we can see **student\_update\_trigger** under the "**students**" table as follows:



## MySQL BEFORE DELETE Trigger

BEFORE DELETE Trigger in MySQL is invoked automatically whenever a delete operation is fired on the table. In this article, we are going to learn how to create a before delete trigger with its syntax and example.

### Syntax

The following is the syntax to create a BEFORE DELETE trigger in MySQL:

1. **CREATE TRIGGER** trigger\_name
2. **BEFORE DELETE**
3. **ON** table\_name **FOR EACH ROW**
4. **Trigger\_body ;**

The BEFORE DELETE trigger syntax parameter can be explained as below:

- First, we will specify the name of the trigger that we want to create. It should be unique within the schema.
- Second, we will specify the trigger action time, which should be BEFORE DELETE. This trigger will be invoked before each row of alterations occurs on the table.
- Third, we will specify the name of a table to which the trigger is associated. It must be written after the ON keyword. If we did not specify the table name, a trigger would not exist.
- Finally, we will specify the statement for execution when the trigger is activated.

If we want to execute multiple statements, we will use the BEGIN END block that contains a set of queries to define the logic for the trigger. See the below syntax:

1. DELIMITER \$\$
2. **CREATE TRIGGER** trigger\_name BEFORE **DELETE**
3. **ON** table\_name **FOR EACH ROW**
4. **BEGIN**
5. variable declarations
6. **trigger** code
7. **END\$\$**
8. DELIMITER ;

## Restrictions

- o We can access the OLD rows but cannot update them in a BEFORE DELETE trigger.
- o We cannot access the NEW rows. It is because there are no new row exists.
- o We cannot create a BEFORE DELETE trigger on a VIEW.

## BEFORE DELETE Trigger Example

Let us understand how to create a BEFORE DELETE trigger using the [CREATE TRIGGER statement in MySQL](#) with an example.

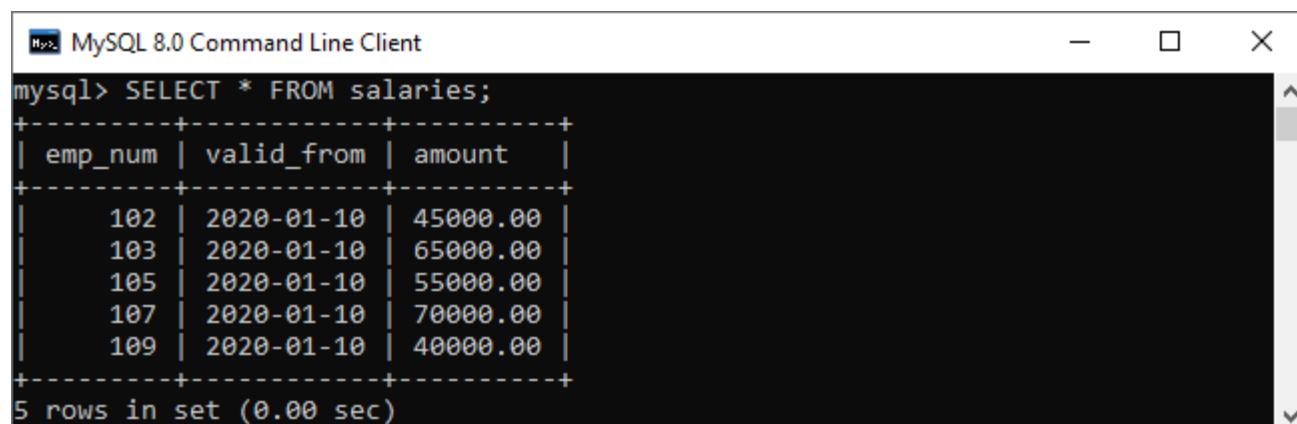
Suppose we have created a table named salaries to store the salary information of an employee as follows:

1. **CREATE TABLE** salaries (
2.   emp\_num **INT PRIMARY KEY**,
3.   valid\_from **DATE** NOT NULL,
4.   amount **DEC(8 , 2 )** NOT NULL **DEFAULT** 0
5. );

Next, we will insert some records into this table using the below statement:

1. **INSERT INTO** salaries (emp\_num, valid\_from, amount)
2. **VALUES**
3.   (102, '2020-01-10', 45000),
4.   (103, '2020-01-10', 65000),
5.   (105, '2020-01-10', 55000),
6.   (107, '2020-01-10', 70000),
7.   (109, '2020-01-10', 40000);

Execute the SELECT query to see the table data.



The screenshot shows the MySQL 8.0 Command Line Client window. The command entered is "SELECT \* FROM salaries;". The resulting table has three columns: emp\_num, valid\_from, and amount. The data consists of five rows with the following values:

emp_num	valid_from	amount
102	2020-01-10	45000.00
103	2020-01-10	65000.00
105	2020-01-10	55000.00
107	2020-01-10	70000.00
109	2020-01-10	40000.00

5 rows in set (0.00 sec)

Third, we will create another table named salary\_archives that keeps the information of deleted salary.

1. **CREATE TABLE** salary\_archives (
2.   id **INT PRIMARY KEY** AUTO\_INCREMENT,
3.   emp\_num **INT**,
4.   valid\_from **DATE** NOT NULL,
5.   amount **DEC(18 , 2 )** NOT NULL **DEFAULT** 0,

```
6. deleted_time TIMESTAMP DEFAULT NOW()
7. );
```

We will then create a BEFORE DELETE trigger that inserts a new record into the salary\_archives table before a row is deleted from the salaries table.

```
1. DELIMITER $$  
2.  
3. CREATE TRIGGER before_delete_salaries  
4. BEFORE DELETE  
5. ON salaries FOR EACH ROW  
6. BEGIN  
7.   INSERT INTO salary_archives (emp_num, valid_from, amount)  
8.     VALUES(OLD.emp_num, OLD.valid_from, OLD.amount);  
9.   END$$  
10.  
11. DELIMITER ;
```

```
MySQL 8.0 Command Line Client  
mysql> DELIMITER $$  
mysql> CREATE TRIGGER before_delete_salaries  
-> BEFORE DELETE  
-> ON salaries FOR EACH ROW  
-> BEGIN  
->   INSERT INTO salary_archives (emp_num, valid_from, amount)  
->     VALUES(OLD.emp_num, OLD.valid_from, OLD.amount);  
-> END$$  
Query OK, 0 rows affected (0.14 sec)  
mysql> DELIMITER ;
```

In this trigger, we have first specified the trigger name before\_delete\_salaries. Then, specify the triggering event. Third, we have specified the table name on which the trigger is associated. Finally, we have written the trigger logic inside the trigger body that insert the deleted row into the salary\_archives table.

## How to call the BEFORE DELETE trigger?

Let us test the above created BEFORE DELETE trigger and how we can call them. So first, we will remove a row from the salaries table:

```
1. mysql> DELETE FROM salaries WHERE emp_num = 105;
```

Second, we will query data from the salary\_archives table to verify the above-created trigger is invoked or not by using the select statement:

```
1. mysql> SELECT * FROM salary_archives;
```

After executing a statement, we can see that the trigger was invoked successfully and inserted a new record into the salary\_archives table.

```
MySQL 8.0 Command Line Client  
mysql> DELETE FROM salaries WHERE emp_num = 105;  
Query OK, 1 row affected (0.13 sec)  
  
mysql> SELECT * FROM salary_archives;  
+----+-----+-----+-----+  
| id | emp_num | valid_from | amount | deleted_time |  
+----+-----+-----+-----+  
| 1  |    105 | 2020-01-10 | 55000.00 | 2020-11-20 13:16:29 |  
+----+-----+-----+-----+  
1 row in set (0.00 sec)
```

Third, we will remove all rows from the salaries table:

```
1. mysql> DELETE FROM salaries;
```

Finally, we will query data from the salary\_archives table again. The trigger was called four times because the DELETE statement removed four records from the salaries table. See the below output:

```

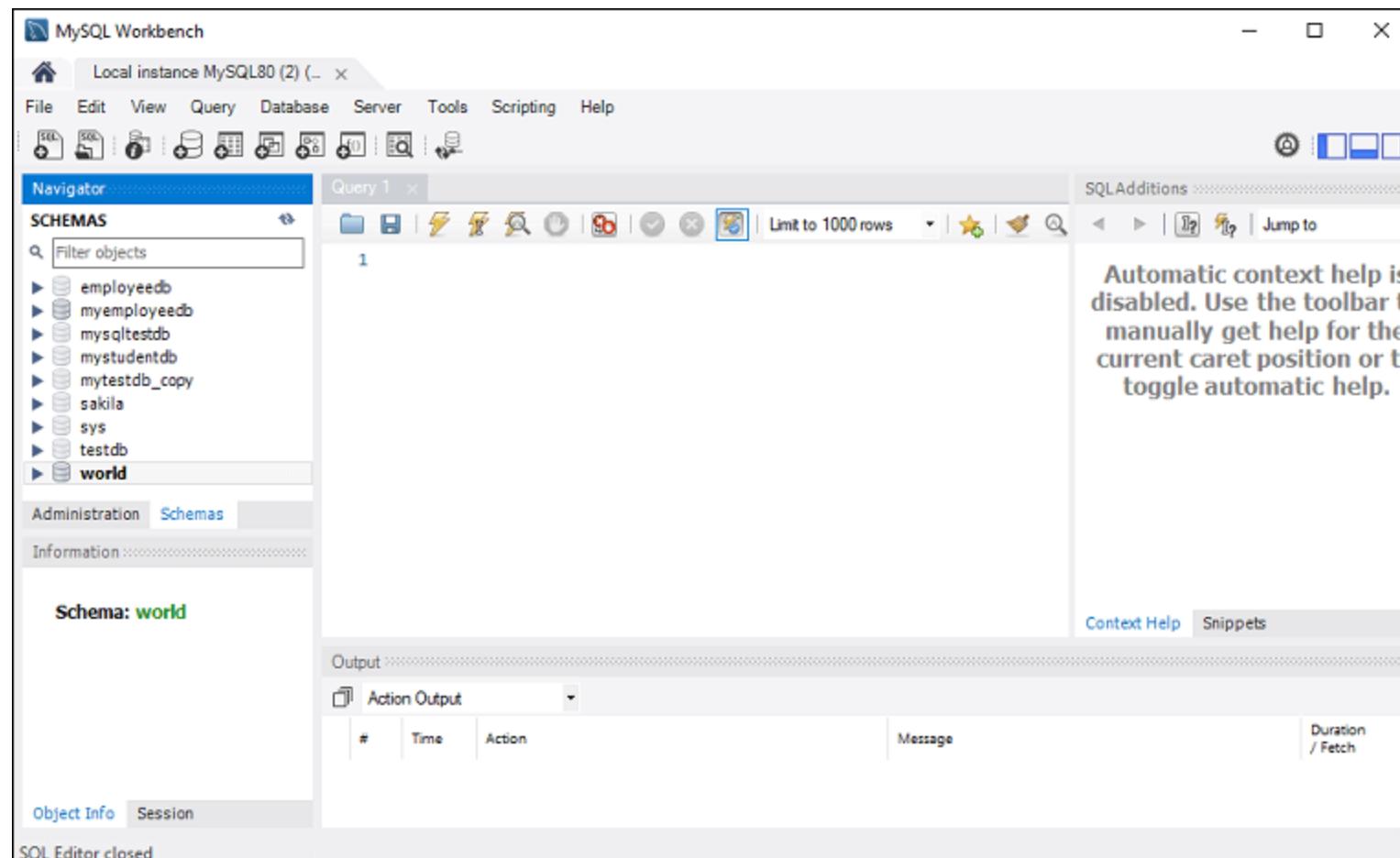
MySQL 8.0 Command Line Client
mysql> DELETE FROM salaries;
Query OK, 4 rows affected (0.18 sec)

mysql> SELECT * FROM salary_archives;
+----+-----+-----+-----+-----+
| id | emp_num | valid_from | amount | deleted_time |
+----+-----+-----+-----+-----+
| 1  |    105  | 2020-01-10 | 55000.00 | 2020-11-20 13:16:29 |
| 2  |    102  | 2020-01-10 | 45000.00 | 2020-11-20 13:17:59 |
| 3  |    103  | 2020-01-10 | 65000.00 | 2020-11-20 13:17:59 |
| 4  |    107  | 2020-01-10 | 70000.00 | 2020-11-20 13:17:59 |
| 5  |    109  | 2020-01-10 | 40000.00 | 2020-11-20 13:17:59 |
+----+-----+-----+-----+-----+
5 rows in set (0.00 sec)

```

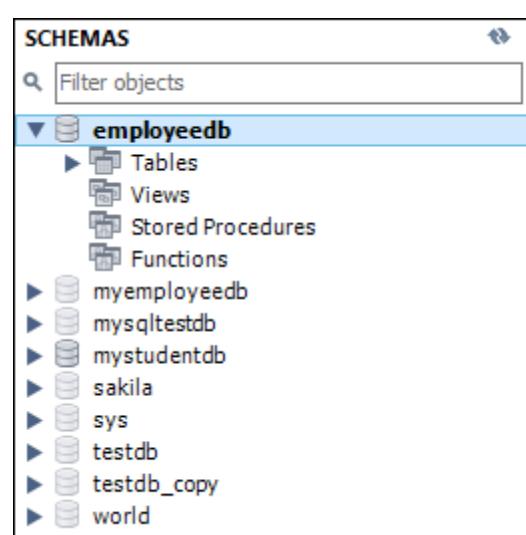
## How to create BEFORE DELETE Trigger in MySQL workbench?

To create a BEFORE DELETE trigger using [MySQL workbench](#), we first need to launch it and then log in using the username and password we created earlier. We will get the screen as follows:



Now do the following steps for creating BEFORE DELETE trigger:

1. Go to the Navigation tab and click on the **Schema menu** that contains all the databases available in the [MySQL](#) server.
2. Select the database (for example, employeedb), double click on it. It will show the **sub-menu** that contains Tables, Views, Functions, and Stored Procedures. See the below screen.



3. Expand the **Tables sub-menu** and select the table on which you want to create a [trigger](#). After selecting a table, right-click on the selected table (for example, salaries), and then click on the [Alter Table](#) option. See the below image:

The screenshot shows the MySQL Workbench interface under the 'Schemas' tab. The 'employeedb' schema is selected. In the 'Tables' section, 'salaries' is highlighted. A context menu is open, with 'Alter Table...' being the selected option.

4. Clicking on the Alter Table option gives the screen as below:

The screenshot shows the 'Alter Table' dialog for the 'salaries' table in the 'employeedb' schema. The 'Triggers' tab is selected. The dialog includes fields for Table Name (salaries), Schema (employeedb), Charset/Collation (utf8mb4), Engine (InnoDB), and a column definition for 'emp\_num'. At the bottom, the 'Triggers' tab is highlighted with a red box.

5. Now, click on the **Trigger tab** shown in the red rectangular box of the previous section, then select the Timing/Event BEFORE DELETE. We will notice that there is a (+) icon button to add a trigger. Clicking on that button, we will get a default code on trigger based on choosing Timing/Event:

The screenshot shows the 'Alter Table' dialog for the 'salaries' table. The 'Triggers' tab is selected. A trigger named 'salaries\_before\_delete' is defined with the following code:

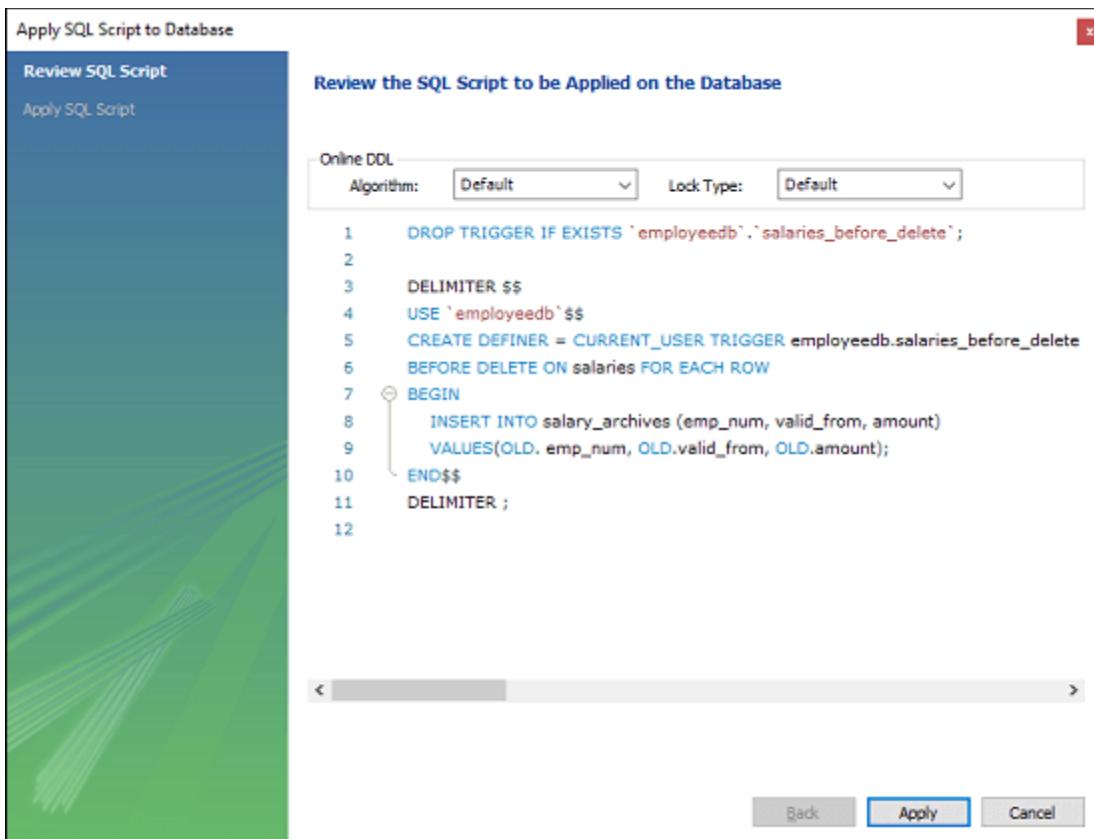
```

1 • CREATE DEFINER = CURRENT_USER TRIGGER employeedb.salaries_before_delete
2 BEFORE DELETE ON salaries FOR EACH ROW
3 BEGIN
4
5 END

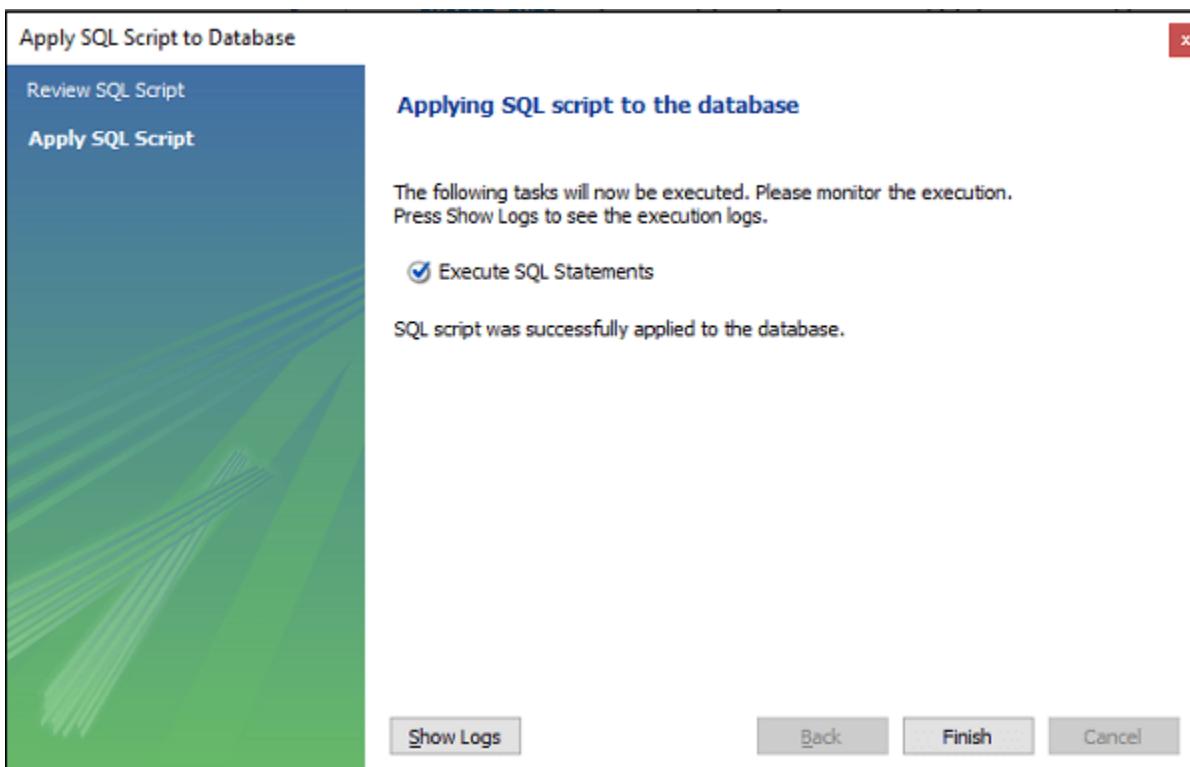
```

The code area is highlighted with a red box.

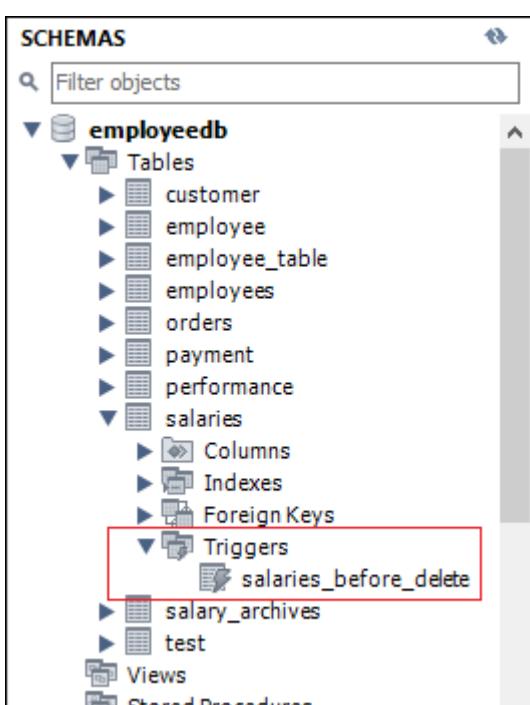
6. Now, complete the trigger code, review them once again, and if no error is found, click on the **Apply** button.



7. After clicking on the Apply button, click on the **Finish** button for completion.



8. If we take at the schema menu, we will see the trigger **salaries\_before\_trigger** under the salaries table as follows:



## MySQL AFTER DELETE Trigger

The AFTER DELETE Trigger in MySQL is invoked automatically whenever a delete event is fired on the table. In this article, we are going to learn how to create an AFTER DELETE trigger with its syntax and example.

## Syntax

The following is the syntax to create an **AFTER DELETE** trigger in MySQL:

1. **CREATE TRIGGER** trigger\_name
2. **AFTER DELETE**
3. **ON** table\_name **FOR EACH ROW**
4. Trigger\_body ;

The AFTER DELETE trigger syntax parameter can be explained as below:

- o First, we will specify the **name of the trigger** that we want to create. It should be unique within the schema.
- o Second, we will specify the **trigger action time**, which should be AFTER DELETE. This trigger will be invoked after each row of alterations occurs on the table.
- o Third, we will specify the **name of a table** to which the trigger is associated. It must be written after the ON keyword. If we did not specify the table name, a trigger would not exist.
- o Finally, we will specify the **trigger body** that contains a statement for execution when the trigger is activated.

If we want to execute multiple statements, we will use the **BEGIN END** block that contains a set of SQL queries to define the logic for the trigger. See the below syntax:

1. DELIMITER \$\$
2. **CREATE TRIGGER** trigger\_name **AFTER DELETE**
3. **ON** table\_name **FOR EACH ROW**
4. **BEGIN**
5. variable declarations
6. **trigger** code
7. **END\$\$**
8. DELIMITER ;

## Restrictions

- o We can access the OLD rows but cannot update them in the AFTER DELETE trigger.
- o We cannot access the NEW rows. It is because there are no NEW row exists.
- o We cannot create an AFTER DELETE trigger on a VIEW.

## AFTER DELETE Trigger Example

Let us understand how to create an AFTER DELETE trigger using the [CREATE TRIGGER statement in MySQL](#) with an example.

Suppose we have created a table named **salaries** to store the salary information of an employee as follows:

1. **CREATE TABLE** salaries (
2.   emp\_num **INT PRIMARY KEY**,
3.   valid\_from **DATE** NOT NULL,
4.   amount **DEC(8 , 2 )** NOT NULL **DEFAULT** 0
5. );

Next, we will insert some records into this table using the below statement:

1. **INSERT INTO** salaries (emp\_num, valid\_from, amount)
2. **VALUES**
3.   (102, '2020-01-10', 45000),
4.   (103, '2020-01-10', 65000),
5.   (105, '2020-01-10', 55000),
6.   (107, '2020-01-10', 70000),
7.   (109, '2020-01-10', 40000);

Execute the SELECT query to see the table data.

```

MySQL 8.0 Command Line Client
mysql> SELECT * FROM salaries;
+-----+-----+-----+
| emp_num | valid_from | amount |
+-----+-----+-----+
|    102 | 2020-01-10 | 45000.00 |
|    103 | 2020-01-10 | 65000.00 |
|    105 | 2020-01-10 | 55000.00 |
|    107 | 2020-01-10 | 70000.00 |
|    109 | 2020-01-10 | 40000.00 |
+-----+-----+-----+
5 rows in set (0.00 sec)

```

Third, we will create another table named **total\_salary\_budget** that keeps the salary information from the salaries table.

1. **CREATE TABLE** total\_salary\_budget(
2.    **total\_budget DECIMAL(10,2) NOT NULL**
3. );

Fourth, we will use the **SUM()** function that returns the total salary from the salaries table and keep this information in the total\_salary\_budget table:

1. mysql> **INSERT INTO** total\_salary\_budget (**total\_budget**)
2. **SELECT SUM(amount) FROM** salaries;

Execute the SELECT statement to verify the table:

```

MySQL 8.0 Command Line Client
mysql> INSERT INTO total_salary_budget (total_budget)
-> SELECT SUM(amount) FROM salaries;
Query OK, 1 row affected (0.09 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM total_salary_budget;
+-----+
| total_budget |
+-----+
| 275000.00 |
+-----+
1 row in set (0.00 sec)

```

We will then create an AFTER DELETE trigger that updates the total salary into the total\_salary\_budget table after a row is deleted from the salaries table.

1. **DELIMITER \$\$**
- 2.
3. **CREATE TRIGGER** after\_delete\_salaries
4. **AFTER DELETE**
5. **ON** salaries **FOR EACH ROW**
6. **BEGIN**
7.    **UPDATE** total\_salary\_budget **SET** total\_budget = total\_budget - old.amount;
8. **END\$\$**
- 9.
10. **DELIMITER ;**

```

MySQL 8.0 Command Line Client
mysql> DELIMITER $$ 
mysql> CREATE TRIGGER after_delete_salaries
-> AFTER DELETE
-> ON salaries FOR EACH ROW
-> BEGIN
->     UPDATE total_salary_budget SET total_budget = total_budget - old.amount;
-> END$$
Query OK, 0 rows affected (0.19 sec)

mysql> DELIMITER ;

```

In this trigger, we have first specified the trigger name **after\_delete\_salaries**. Then, specify the triggering event. Third, we have specified the table name on which the trigger is associated. Finally, we have written the trigger logic inside the trigger body that updates the total salary into the total\_salary\_budget table after a row is deleted from the salaries table.

## How to call the AFTER DELETE trigger?

First, we will delete a salary from the salaries table using the following statements to invoke the above-created trigger:

1. mysql> **DELETE FROM** salaries **WHERE** emp\_num = 105;

Next, we will query data from the total\_salary\_budget table. We can see that table has been modified after the execution of the query. See the below output:

1. mysql> **SELECT \* FROM** total\_salary\_budget;

```
MySQL 8.0 Command Line Client

mysql> DELETE FROM salaries WHERE emp_num = 105;
Query OK, 1 row affected (0.15 sec)

mysql> SELECT * FROM total_salary_budget;
+-----+
| total_budget |
+-----+
| 220000.00 |
+-----+
1 row in set (0.00 sec)
```

In the output, we can see that the deleted salary reduces the total\_budget.

Third, we will remove all data from the salaries table:

1. mysql> **DELETE FROM** salaries;

Again, we will query data from the total\_salary\_budget table. We can see that the trigger updated the table to zero after the execution of the query. See the below output:

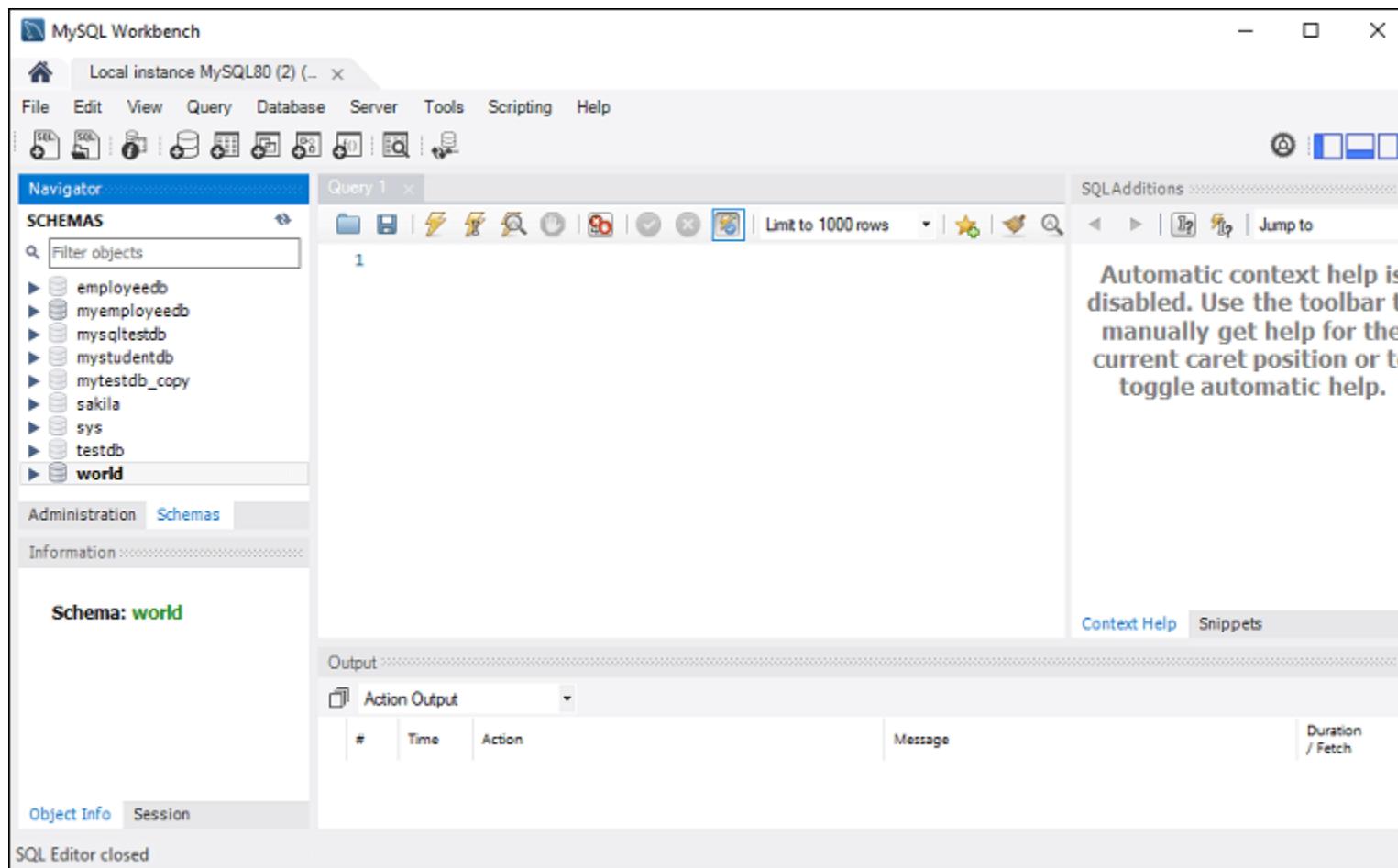
```
MySQL 8.0 Command Line Client

mysql> DELETE FROM salaries;
Query OK, 4 rows affected (0.13 sec)

mysql> SELECT * FROM total_salary_budget;
+-----+
| total_budget |
+-----+
| 0.00 |
+-----+
1 row in set (0.00 sec)
```

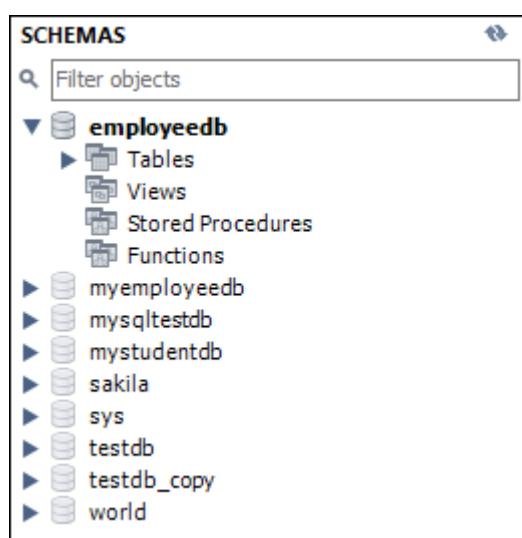
## How to create AFTER DELETE Trigger in MySQL workbench?

To create an after insert trigger using this tool, we first need to [launch the MySQL Workbench](#) and log in using the username and password we have created earlier. We will get the screen as follows:



Now do the following steps for creating an AFTER DELETE trigger:

1. Go to the Navigation tab and click on the **Schema menu** that contains all the databases available in the [MySQL](#) server.
2. Select the database (for example, **employeedb**), double click on it that shows the **sub-menu** containing Tables, Views, Functions, and Stored Procedures. See the below screen.



3. Expand the **Tables sub-menu** and select the table on which you want to create a trigger. After selecting a table, right-click on the selected table (for example, salaries), and then click on the [Alter Table](#) option. See the below image:

The screenshot shows the MySQL Workbench interface. The left pane displays the 'SCHEMAS' tree, with 'employeesdb' expanded to show tables like customer, employee, employee\_table, employees, orders, payment, performance, salaries, and total\_salary\_budget. The 'Views', 'Stored Procedures', and 'Functions' nodes are also visible under the schema. The 'Information' node is selected. The right pane shows details for the 'salaries' table, including its columns: emp\_num (int PK), valid\_from (date), and amount (decimal(8,2)). A context menu is open over the 'salaries' table, with 'Alter Table...' highlighted.

4. Clicking on the **Alter Table** option gives the screen as below:

This screenshot shows the 'Alter Table' dialog for the 'salaries' table in the 'employeesdb' schema. The 'Triggers' tab is active. The 'Trigger' section lists several triggers: BEFORE INSERT, AFTER INSERT, BEFORE UPDATE, AFTER UPDATE, BEFORE DELETE, AFTER DELETE, and salaries\_AFTER\_DELETE. The 'salaries\_AFTER\_DELETE' trigger is selected. The dialog includes fields for Table Name (salaries), Schema (employeesdb), Charset/Collation (utf8mb4), Engine (InnoDB), and various column definition options. At the bottom, there are tabs for Columns, Indexes, Foreign Keys, Triggers, Partitioning, Options, and buttons for Apply and Revert.

5. Now, click on the **Trigger** tab shown in the previous section's red rectangular box, then select the Timing/Event **AFTER DELETE**. We will notice that there is a (+) icon button to add a trigger. Clicking on that button, we will get a default code on the trigger based on choosing Timing/Event:

This screenshot shows the 'Alter Table' dialog for the 'salaries' table in the 'employeesdb' schema. The 'Triggers' tab is active, and the 'salaries\_AFTER\_DELETE' trigger is selected. The trigger code in the editor area is as follows:

```

1 * CREATE DEFINER = CURRENT_USER TRIGGER employeesdb.salaries_AFTER_DELETE
2 AFTER DELETE ON salaries FOR EACH ROW
3 BEGIN
4
5 END

```

The left pane shows the 'SCHEMAS' tree with the 'employeesdb' schema selected. The 'Tables' node is expanded, showing tables like customer, employee, employee\_table, employees, orders, payment, performance, salaries, and total\_salary\_budget. The 'Views', 'Stored Procedures', and 'Functions' nodes are also visible under the schema. The 'Information' node is selected.

6. Now, complete the trigger code, review them once again, and if no error is found, click on the **Apply** button.

Apply SQL Script to Database

Review SQL Script

Review the SQL Script to be Applied on the Database

Online DDL

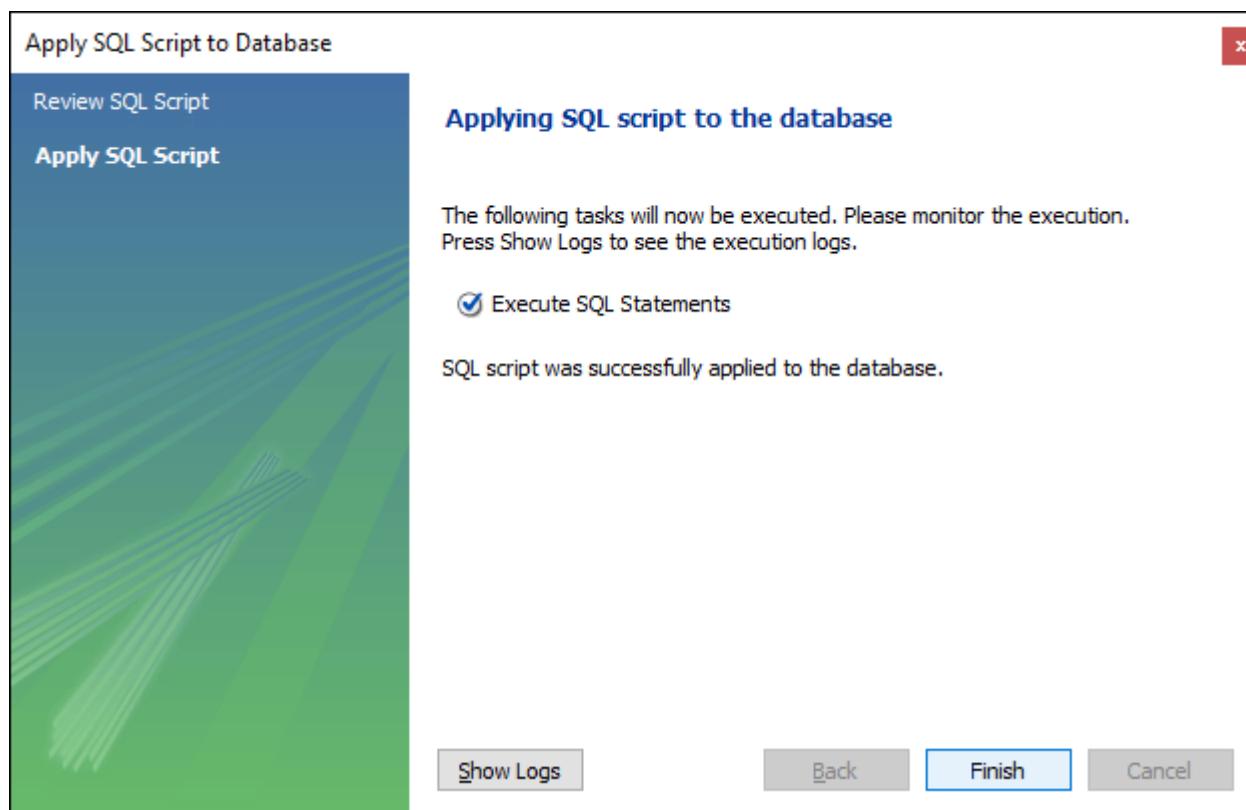
```

1  DROP TRIGGER IF EXISTS `employeedb`.`salaries_AFTER_DELETE`;
2
3  DELIMITER $$;
4  USE `employeedb`$$;
5  CREATE DEFINER = CURRENT_USER TRIGGER employeedb.salaries_AFTER_DELETE
6  AFTER DELETE ON salaries FOR EACH ROW
7  BEGIN
8      UPDATE total_salary_budget SET total_budget = total_budget - old.amount;
9  END$$;
10 DELIMITER ;
11

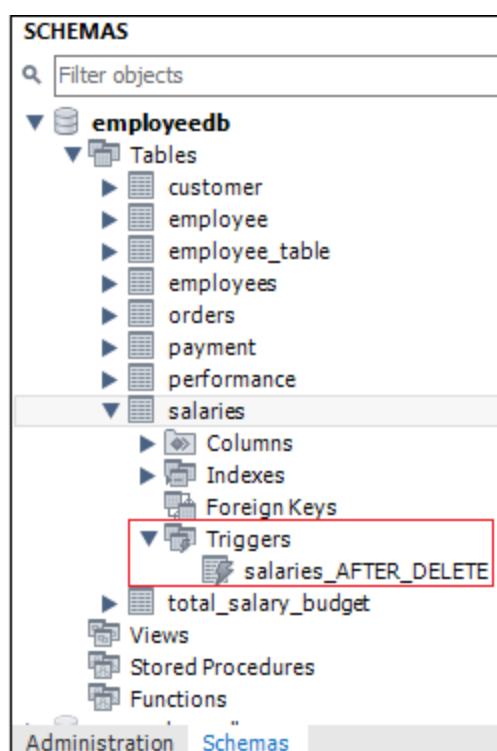
```

Back Apply Cancel

7. After clicking on the Apply button, click on the **Finish button** for completion.



8. If we look at the schema menu, we can see **salaries\_AFTER\_DELETE** trigger under the **salaries** table as follows:



## Aggregate Functions

# MySQL Aggregate Functions

MySQL's aggregate function is used to **perform calculations on multiple values and return the result in a single value like the average of all values**, the sum of all values, and maximum & minimum value among certain groups of values. We mostly use the aggregate functions with [SELECT statements](#) in the data query languages.

## Syntax:

The following are the syntax to use aggregate functions in MySQL:

1. function\_name (**DISTINCT** | ALL expression)

In the above syntax, we had used the following parameters:

- o First, we need to specify the name of the aggregate function.
- o Second, we use the **DISTINCT** modifier when we want to calculate the result based on distinct values or **ALL** modifiers when we calculate all values, including duplicates. The default is ALL.
- o Third, we need to specify the expression that involves columns and arithmetic operators.

There are various aggregate functions available in [MySQL](#). Some of the most commonly used aggregate functions are summarised in the below table:

Aggregate Function	Descriptions
<a href="#">count()</a>	It returns the number of rows, including rows with NULL values in a group.
<a href="#">sum()</a>	It returns the total summed values (Non-NULL) in a set.
<a href="#">average()</a>	It returns the average value of an expression.
<a href="#">min()</a>	It returns the minimum (lowest) value in a set.
<a href="#">max()</a>	It returns the maximum (highest) value in a set.
<a href="#">group_concat()</a>	It returns a concatenated string.
<a href="#">first()</a>	It returns the first value of an expression.
<a href="#">last()</a>	It returns the last value of an expression.

## Why we use aggregate functions?

We mainly use the aggregate functions in databases, spreadsheets and many other data manipulation software packages. In the context of business, different organization levels need different information such as top level managers interested in knowing whole figures and not the individual details. These functions produce the summarised data from our database. Thus they are extensively used in economics and finance to represent the economic health or stock and sector performance.

Let us take an example of myflix (video streaming website which has huge collections of the movie) database, where management may require the following details:

- o Most rented movies.
- o Least rented movies.
- o Average number that each movie is rented out in a month.

We can easily produce these details with the help of aggregate functions.

Let us discuss the most commonly used aggregate functions in detail. First, we will create a new table for the demonstration of all aggregate functions.

Execute the below statement to create an **employee** table:

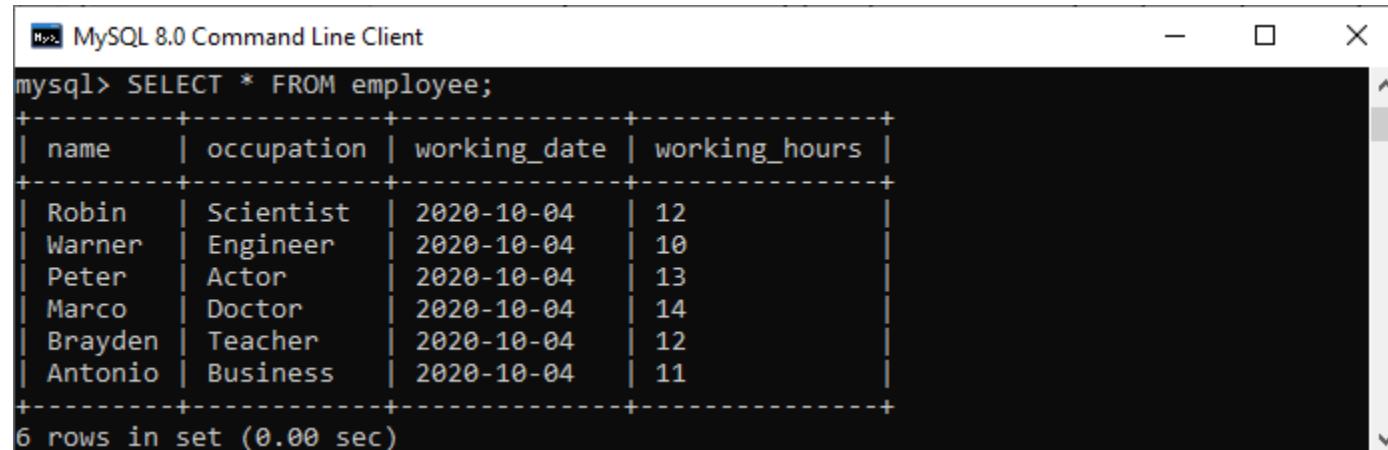
1. **CREATE TABLE** employee(
2.   **name varchar**(45) NOT NULL,
3.   **occupation varchar**(35) NOT NULL,
4.   **working\_date date**,

```
5.    working_hours varchar(10)
6. );
```

Execute the below statement to **insert the records** into the employee table:

```
1. INSERT INTO employee VALUES
2. ('Robin', 'Scientist', '2020-10-04', 12),
3. ('Warner', 'Engineer', '2020-10-04', 10),
4. ('Peter', 'Actor', '2020-10-04', 13),
5. ('Marco', 'Doctor', '2020-10-04', 14),
6. ('Brayden', 'Teacher', '2020-10-04', 12),
7. ('Antonio', 'Business', '2020-10-04', 11);
```

Now, execute the **SELECT statement** to show the record:



The screenshot shows the MySQL 8.0 Command Line Client window. The command entered is `SELECT * FROM employee;`. The output is a table with six rows, each representing an employee with columns: name, occupation, working\_date, and working\_hours. The data is as follows:

name	occupation	working_date	working_hours
Robin	Scientist	2020-10-04	12
Warner	Engineer	2020-10-04	10
Peter	Actor	2020-10-04	13
Marco	Doctor	2020-10-04	14
Brayden	Teacher	2020-10-04	12
Antonio	Business	2020-10-04	11

6 rows in set (0.00 sec)

## Count() Function

MySQL count() function **returns the total number of values** in the expression. This function produces all rows or only some rows of the table based on a specified condition, and its return type is **BIGINT**. It returns zero if it does not find any matching rows. It can work with both numeric and non-numeric data types.

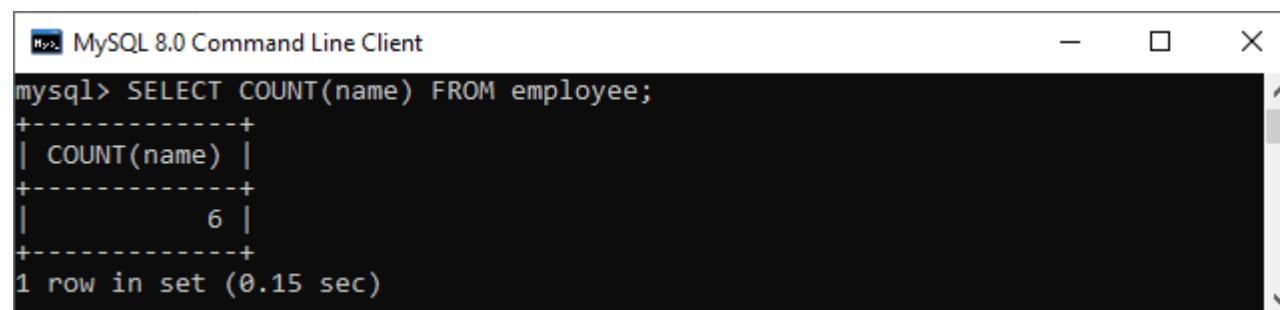
### Example

Suppose we want to get the total number of employees in the employee table, we need to use the count() function as shown in the following query:

```
1. mysql> SELECT COUNT(name) FROM employee;
```

### Output:

After execution, we can see that this table has six employees.



The screenshot shows the MySQL 8.0 Command Line Client window. The command entered is `SELECT COUNT(name) FROM employee;`. The output is a table with one row, showing the count of employees as 6.

COUNT(name)
6

1 row in set (0.15 sec)

To read more information, [click here](#).

## Sum() Function

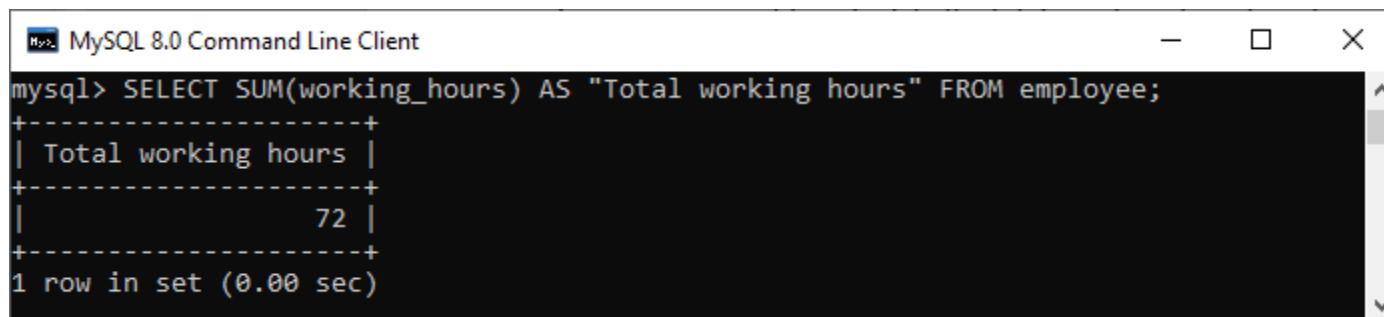
The MySQL sum() function **returns the total summed (non-NULL) value** of an expression. It returns NULL if the result set does not have any rows. It works with numeric data type only.

Suppose we want to calculate the total number of working hours of all employees in the table, we need to use the sum() function as shown in the following query:

```
1. mysql> SELECT SUM(working_hours) AS "Total working hours" FROM employee;
```

### Output:

After execution, we can see the total working hours of all employees in the table.



```
MySQL 8.0 Command Line Client
mysql> SELECT SUM(working_hours) AS "Total working hours" FROM employee;
+-----+
| Total working hours |
+-----+
| 72 |
+-----+
1 row in set (0.00 sec)
```

To read more information, [click here](#).

## AVG() Function

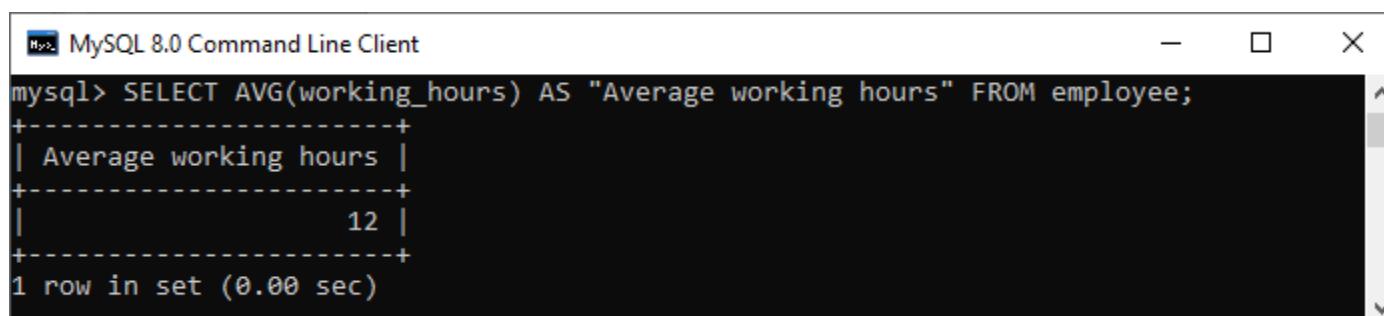
MySQL AVG() function **calculates the average of the values** specified in the column. Similar to the SUM() function, it also works with numeric data type only.

Suppose we want to get the average working hours of all employees in the table, we need to use the AVG() function as shown in the following query:

1. mysql> **SELECT AVG(working\_hours) AS "Average working hours" FROM employee;**

### Output:

After execution, we can see that the average working hours of all employees in the organization:



```
MySQL 8.0 Command Line Client
mysql> SELECT AVG(working_hours) AS "Average working hours" FROM employee;
+-----+
| Average working hours |
+-----+
| 12 |
+-----+
1 row in set (0.00 sec)
```

To read more information, [click here](#).

## MIN() Function

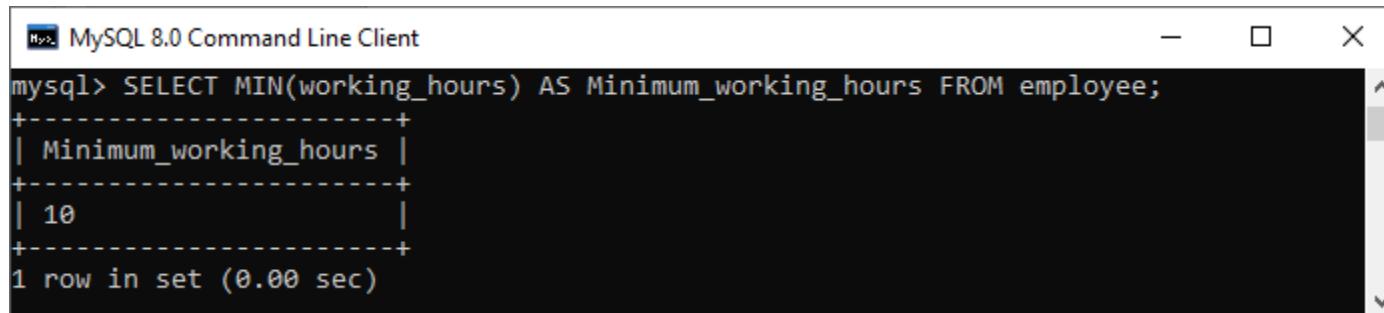
MySQL MIN() function **returns the minimum (lowest) value** of the specified column. It also works with numeric data type only.

Suppose we want to get minimum working hours of an employee available in the table, we need to use the MIN() function as shown in the following query:

1. mysql> **SELECT MIN(working\_hours) AS Minimum\_working\_hours FROM employee;**

### Output:

After execution, we can see that the minimum working hours of an employee available in the table:



```
MySQL 8.0 Command Line Client
mysql> SELECT MIN(working_hours) AS Minimum_working_hours FROM employee;
+-----+
| Minimum_working_hours |
+-----+
| 10 |
+-----+
1 row in set (0.00 sec)
```

To read more information, [click here](#).

## MAX() Function

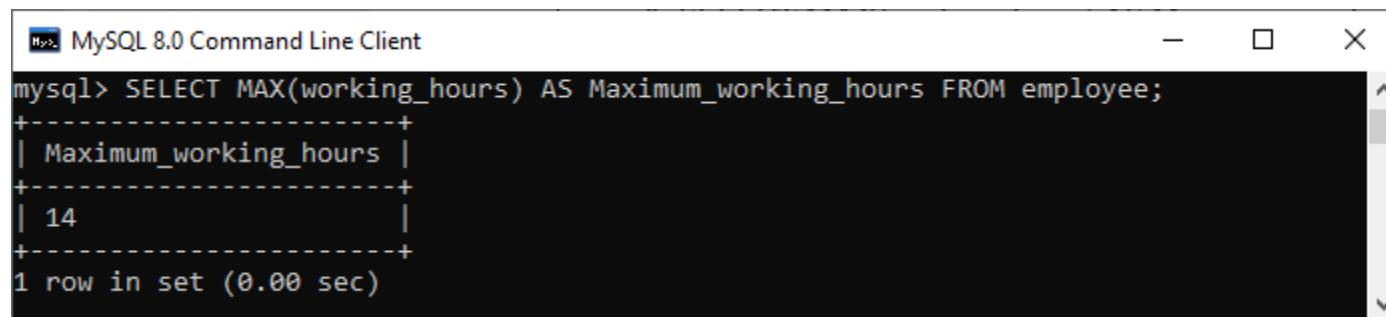
MySQL MAX() function **returns the maximum (highest) value** of the specified column. It also works with numeric data type only.

Suppose we want to get maximum working hours of an employee available in the table, we need to use the MAX() function as shown in the following query:

1. mysql> **SELECT MAX(working\_hours) AS Maximum\_working\_hours FROM employee;**

## Output:

After execution, we can see that the maximum working hours of an employee available in the table:



```
MySQL 8.0 Command Line Client
mysql> SELECT MAX(working_hours) AS Maximum_working_hours FROM employee;
+-----+
| Maximum_working_hours |
+-----+
| 14 |
+-----+
1 row in set (0.00 sec)
```

To read more information, [click here](#).

## FIRST() Function

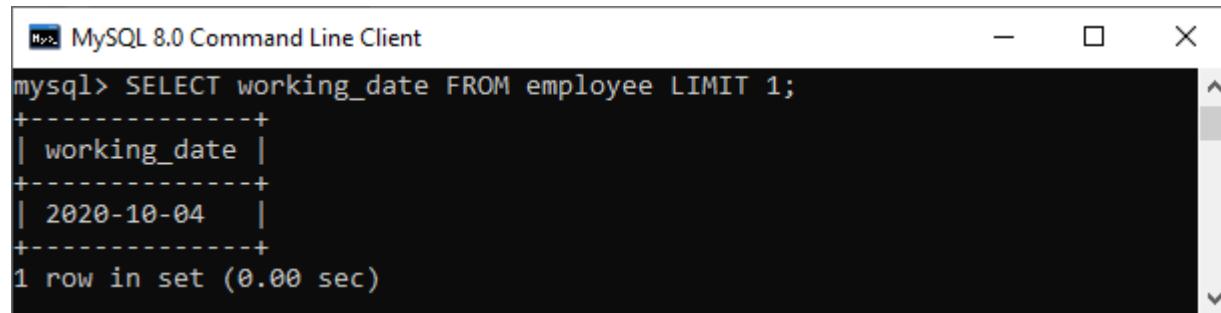
This function **returns the first value** of the specified column. To get the first value of the column, we must have to use the **LIMIT** clause. It is because FIRST() function only supports in MS Access.

Suppose we want to get the first working date of an employee available in the table, we need to use the following query:

1. mysql> **SELECT** working\_date **FROM** employee **LIMIT** 1;

## Output:

After execution, we can see that the first working date of an employee available in the table:



```
MySQL 8.0 Command Line Client
mysql> SELECT working_date FROM employee LIMIT 1;
+-----+
| working_date |
+-----+
| 2020-10-04 |
+-----+
1 row in set (0.00 sec)
```

To read more information, [click here](#).

## LAST() Function

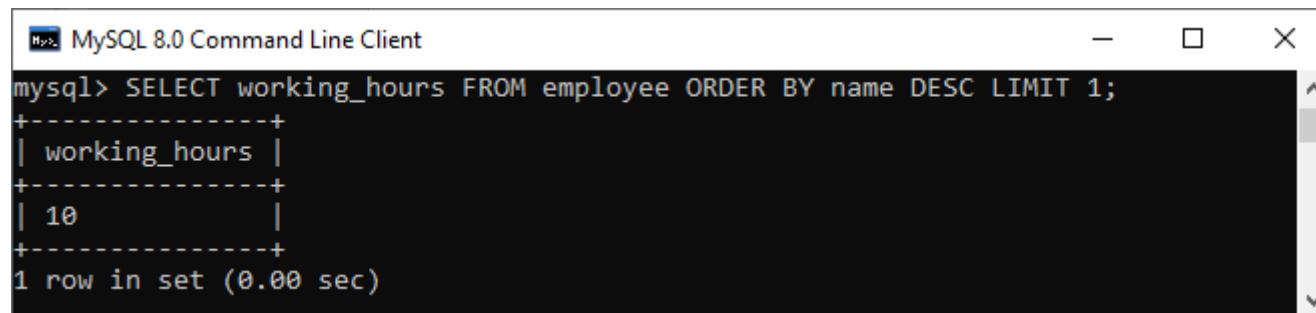
This function **returns the last value** of the specified column. To get the last value of the column, we must have to use the **ORDER BY** and **LIMIT** clause. It is because the LAST() function only supports in MS Access.

Suppose we want to get the last working hour of an employee available in the table, we need to use the following query:

1. mysql> **SELECT** working\_hours **FROM** employee **ORDER BY** name **DESC** **LIMIT** 1;

## Output:

After execution, we can see that the last working hour of an employee available in the table:



```
MySQL 8.0 Command Line Client
mysql> SELECT working_hours FROM employee ORDER BY name DESC LIMIT 1;
+-----+
| working_hours |
+-----+
| 10 |
+-----+
1 row in set (0.00 sec)
```

To read more information, [click here](#).

## GROUP\_CONCAT() Function

The GROUP\_CONCAT() function **returns the concatenated string from multiple rows into a single string**. If the group contains at least one non-null value, it always returns a string value. Otherwise, we will get a null value.

Suppose we have another employee table as below:

emp_id	emp_fname	emp_lname	dept_id	designation
1	David	Miller	2	Engineer
2	Peter	Watson	3	Manager
3	Mark	Boucher	1	Scientist
2	Peter	Watson	3	BDE
1	David	Miller	2	Developer
4	Adam	Warner	4	Receptionist
3	Mark	Boucher	1	Engineer
4	Adam	Warner	4	Clerk

If we want to concatenate the designation of the same dept\_id on the employee table, we need to use the following query:

1. mysql> **SELECT** emp\_id, emp\_fname, emp\_lname, dept\_id,
2. GROUP\_CONCAT(designation) **as "designation"** **FROM** employee **group by** emp\_id;

#### Output:

After execution, we can see that the designation of the same dept\_id concatenated successfully:

emp_id	emp_fname	emp_lname	dept_id	designation
1	David	Miller	2	Engineer,Developer
2	Peter	Watson	3	Manager,BDE
3	Mark	Boucher	1	Scientist,Engineer
4	Adam	Warner	4	Receptionist,Clerk

To read more information, [click here](#).

## MySQL Count() Function

MySQL count() function is used to returns the count of an expression. It allows us to count all rows or only some rows of the table that matches a specified condition. It is a type of aggregate function whose return type is BIGINT. This function returns 0 if it does not find any matching rows.

We can use the count function in three forms, which are explained below:

- o Count (\*)
- o Count (expression)
- o Count (distinct)

Let us discuss each in detail.

**COUNT(\*) Function:** This function uses the **SELECT statement** to returns the count of rows in a result set. The result set contains all Non-Null, Null, and duplicates rows.

**COUNT(expression) Function:** This function returns the result set without containing Null rows as the result of an expression.

**COUNT(distinct expression) Function:** This function returns the count of distinct rows without containing NULL values as the result of the expression.

## Syntax

The following are the syntax of the COUNT() function:

1. **SELECT COUNT** (aggregate\_expression)
2. **FROM** table\_name
3. **[WHERE conditions];**

## Parameter explanation

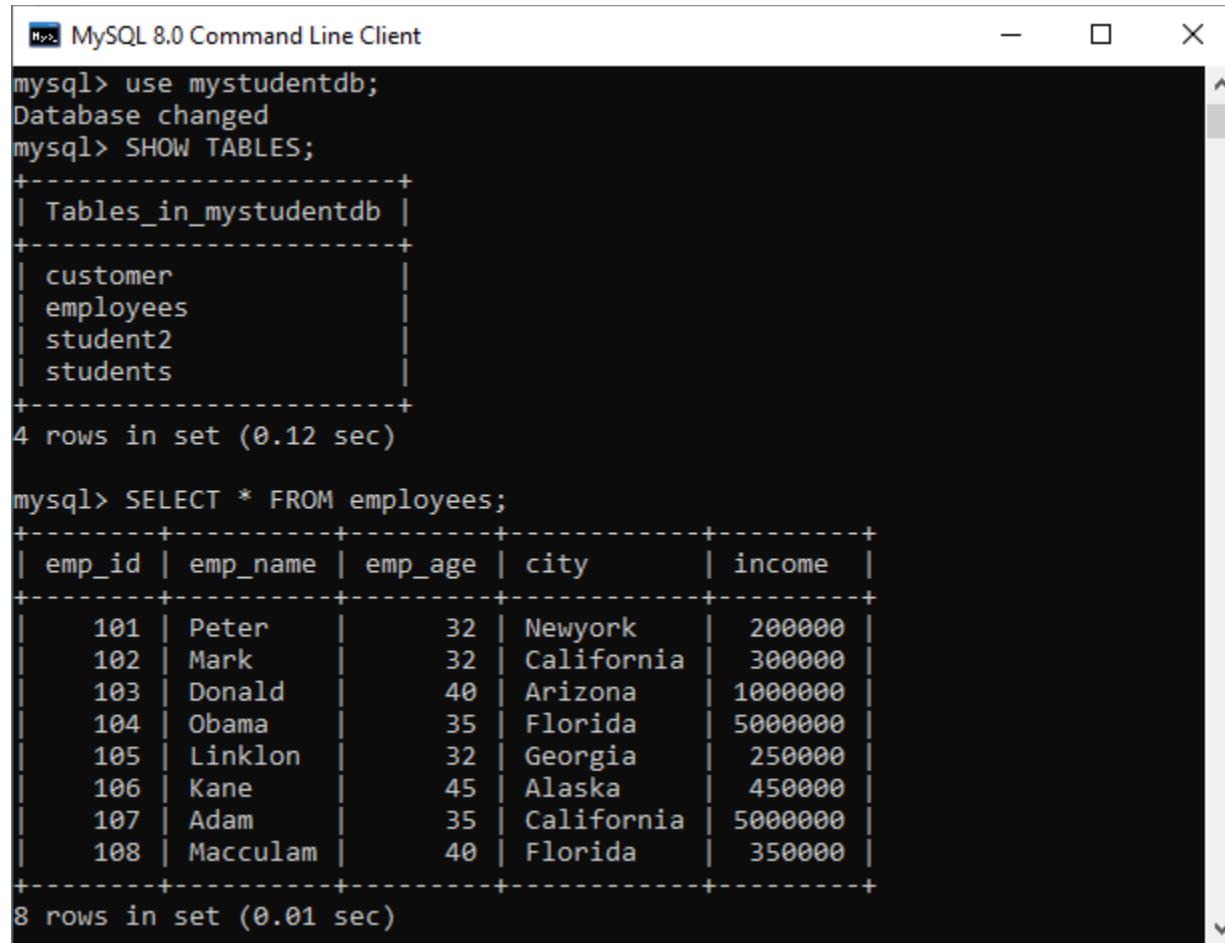
**aggregate\_expression:** It specifies the column or expression whose NON-NUL values will be counted.

**table\_name:** It specifies the tables from where you want to retrieve records. There must be at least one table listed in the **FROM clause**.

**WHERE conditions:** It is optional. It specifies the conditions that must be fulfilled for the records to be selected.

## MySQL count() function example

Consider a table named "employees" that contains the following data.



MySQL 8.0 Command Line Client

```
mysql> use mystudentdb;
Database changed
mysql> SHOW TABLES;
+-----+
| Tables_in_mystudentdb |
+-----+
| customer              |
| employees              |
| student2               |
| students                |
+-----+
4 rows in set (0.12 sec)

mysql> SELECT * FROM employees;
+-----+-----+-----+-----+-----+
| emp_id | emp_name | emp_age | city      | income   |
+-----+-----+-----+-----+-----+
| 101    | Peter     | 32       | Newyork  | 200000  |
| 102    | Mark      | 32       | California | 300000  |
| 103    | Donald    | 40       | Arizona   | 1000000 |
| 104    | Obama     | 35       | Florida   | 5000000 |
| 105    | Linklon   | 32       | Georgia   | 250000  |
| 106    | Kane      | 45       | Alaska    | 450000  |
| 107    | Adam      | 35       | California | 5000000 |
| 108    | Macculam  | 40       | Florida   | 350000  |
+-----+-----+-----+-----+-----+
8 rows in set (0.01 sec)
```

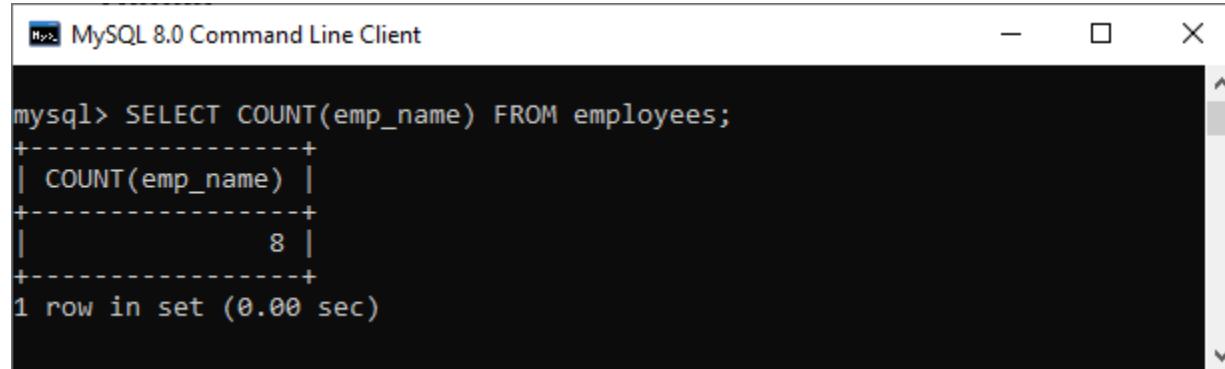
Let us understand how count() functions work in MySQL.

### Example1

Execute the following query that uses the COUNT(expression) function to calculates the total number of employees name available in the table:

1. mysql> **SELECT COUNT(emp\_name) FROM employees;**

### Output:



MySQL 8.0 Command Line Client

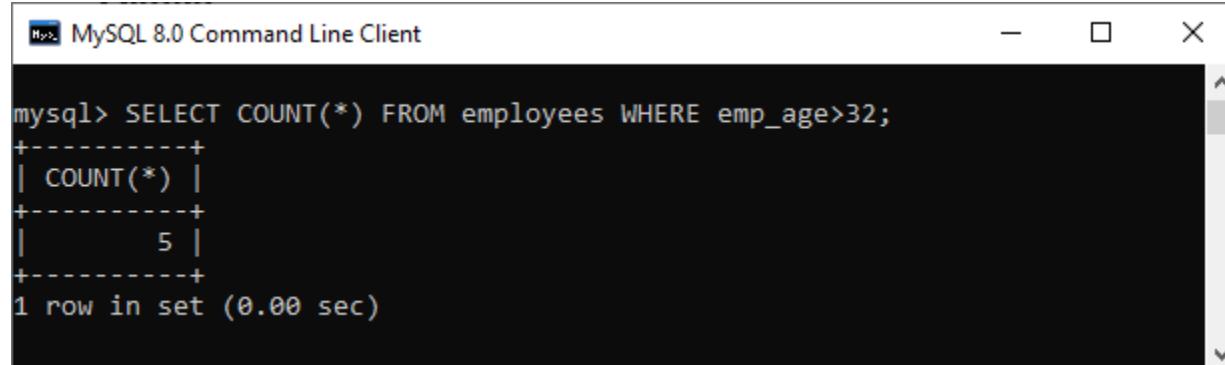
```
mysql> SELECT COUNT(emp_name) FROM employees;
+-----+
| COUNT(emp_name) |
+-----+
| 8 |
+-----+
1 row in set (0.00 sec)
```

### Example2

Execute the following statement that returns all rows from the employee table and WHERE clause specifies the rows whose value in the column emp\_age is greater than 32:

1. mysql> **SELECT COUNT(\*) FROM employees WHERE emp\_age>32;**

### Output:



MySQL 8.0 Command Line Client

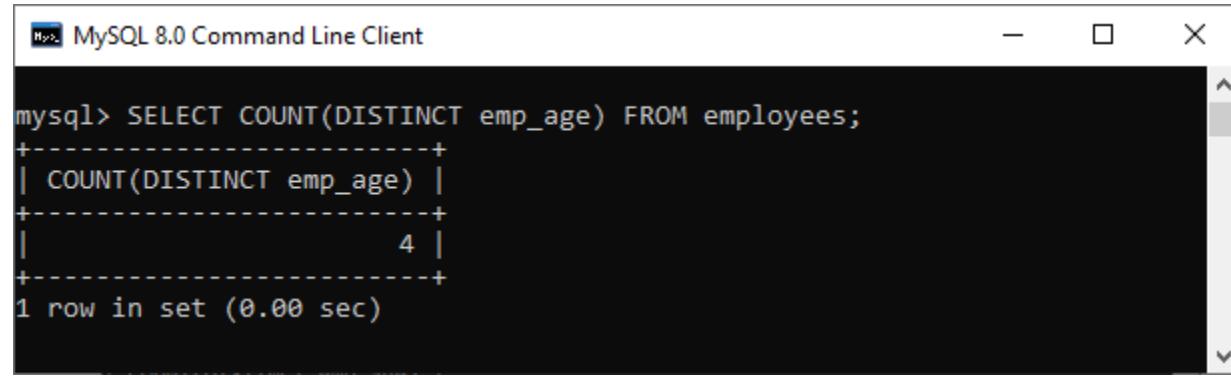
```
mysql> SELECT COUNT(*) FROM employees WHERE emp_age>32;
+-----+
| COUNT(*) |
+-----+
| 5 |
+-----+
1 row in set (0.00 sec)
```

### Example3

This statement uses the COUNT(distinct expression) function that counts the Non-Null and distinct rows in the column emp\_age:

1. mysql> **SELECT COUNT(DISTINCT emp\_age) FROM employees;**

**Output:**



```
MySQL 8.0 Command Line Client

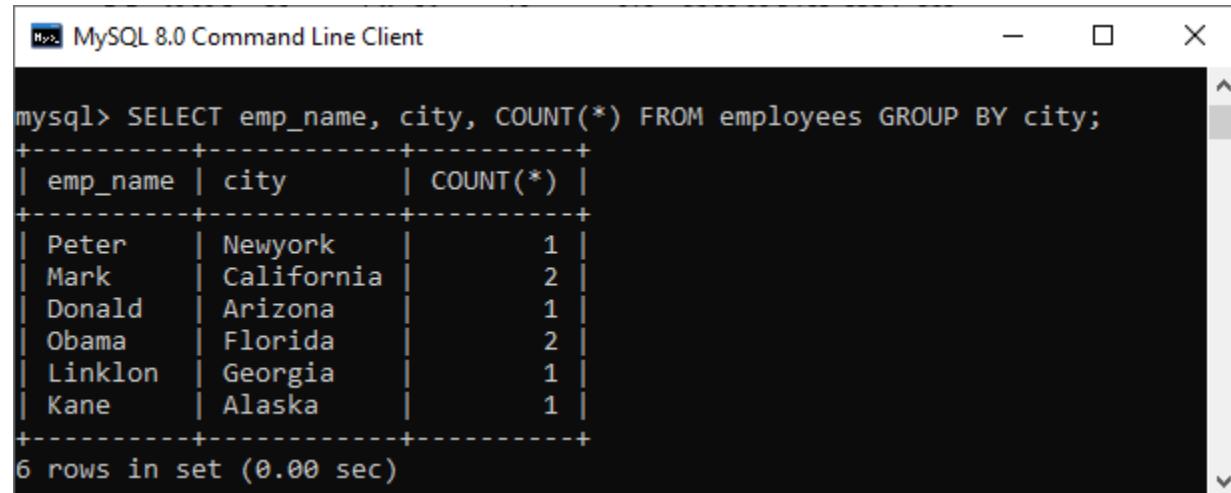
mysql> SELECT COUNT(DISTINCT emp_age) FROM employees;
+-----+
| COUNT(DISTINCT emp_age) |
+-----+
|                      4 |
+-----+
1 row in set (0.00 sec)
```

## MySQL Count() Function with GROUP BY Clause

We can also use the count() function with the GROUP BY clause that returns the count of the element in each group. For example, the following statement returns the number of employee in each city:

1. mysql> **SELECT emp\_name, city, COUNT(\*) FROM employees GROUP BY city;**

After the successful execution, we will get the result as below:



```
MySQL 8.0 Command Line Client

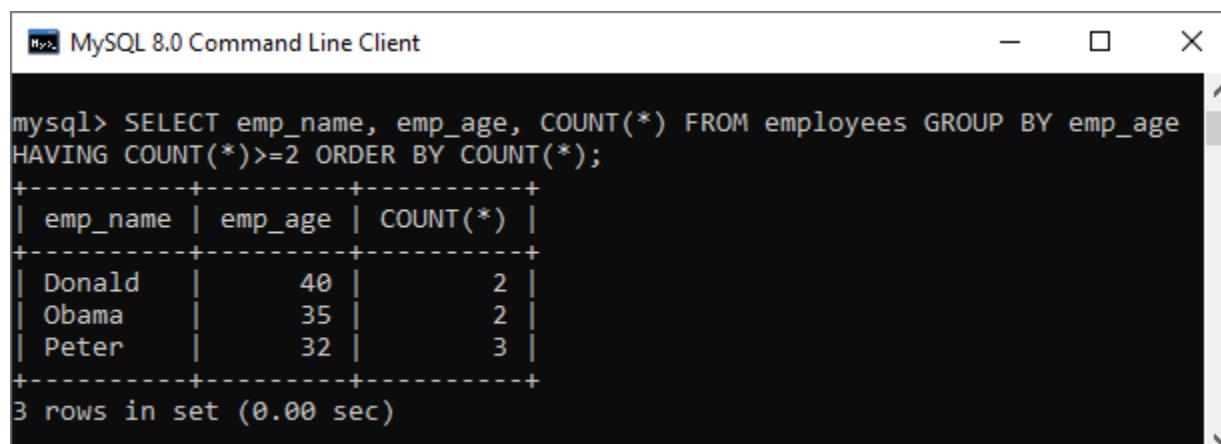
mysql> SELECT emp_name, city, COUNT(*) FROM employees GROUP BY city;
+-----+-----+-----+
| emp_name | city    | COUNT(*) |
+-----+-----+-----+
| Peter    | Newyork |          1 |
| Mark    | California |        2 |
| Donald  | Arizona  |          1 |
| Obama   | Florida  |          2 |
| Linklon | Georgia  |          1 |
| Kane    | Alaska   |          1 |
+-----+-----+-----+
6 rows in set (0.00 sec)
```

## MySQL Count() Function with HAVING and ORDER BY Clause

Let us see another clause that uses **ORDER BY** and **HAVING clause** with the count() function. Execute the following statement that gives the employee name who has at least two age same and sorts them based on the count result:

1. mysql> **SELECT emp\_name, emp\_age, COUNT(\*) FROM employees**
2. **GROUP BY** emp\_age
3. **HAVING COUNT(\*)>=2**
4. **ORDER BY COUNT(\*)**;

This statement will give the output as below:



```
MySQL 8.0 Command Line Client

mysql> SELECT emp_name, emp_age, COUNT(*) FROM employees GROUP BY emp_age HAVING COUNT(*)>=2 ORDER BY COUNT(*);
+-----+-----+-----+
| emp_name | emp_age | COUNT(*) |
+-----+-----+-----+
| Donald  |      40 |          2 |
| Obama   |      35 |          2 |
| Peter   |      32 |          3 |
+-----+-----+-----+
3 rows in set (0.00 sec)
```

## MySQL sum() function

The MySQL sum() function is used to return the total summed value of an expression. It returns **NULL** if the result set does not have any rows. It is one of the kinds of aggregate functions in MySQL.

### Syntax

Following are the syntax of sum() function in MySQL:

1. **SELECT** **SUM**(aggregate\_expression)
2. **FROM** tables
3. [**WHERE** conditions];

## Parameter Explanation

**aggregate\_expression:** It specifies the column or expression that we are going to calculate the sum.

**table\_name:** It specifies the tables from where we want to retrieve records. There must be at least one table listed in the FROM clause.

**WHERE conditions:** It is optional. It specifies the conditions that must be fulfilled for the records to be selected.

## MySQL sum() function example

Consider our database has a table named **employees**, having the following data. Now, we are going to understand this function with various examples:

The screenshot shows the MySQL 8.0 Command Line Client interface. A query is run: `SELECT * FROM employees;`. The output displays 10 rows of data from the employees table, which includes columns: emp\_id, emp\_name, occupation, working\_date, and working\_hours. The data shows various employees (Joseph, Stephen, Mark, Peter) with different occupations and working hours ranging from 9 to 15. The command prompt shows the result set size as 10 rows in 0.00 seconds.

emp_id	emp_name	occupation	working_date	working_hours
1	Joseph	Business	2020-04-10	10
2	Stephen	Doctor	2020-04-10	15
3	Mark	Engineer	2020-04-10	12
4	Peter	Teacher	2020-04-10	9
1	Joseph	Business	2020-04-12	10
2	Stephen	Doctor	2020-04-12	15
4	Peter	Teacher	2020-04-12	9
3	Mark	Engineer	2020-04-12	12
1	Joseph	Business	2020-04-14	10
4	Peter	Teacher	2020-04-14	9

10 rows in set (0.00 sec)

### 1. Basic Example

Execute the following query that calculates the total number of working hours of all employees in the table:

1. mysql> **SELECT** **SUM**(working\_hours) **AS** "Total working hours" **FROM** employees;

#### Output:

We will get the result as below:

The screenshot shows the MySQL 8.0 Command Line Client interface. A query is run: `SELECT SUM(working_hours) AS "Total working hours" FROM employees;`. The output displays a single row with the column name "Total working hours" and the value 111. The command prompt shows the result set size as 1 row in 0.00 seconds.

Total working hours
111

1 row in set (0.00 sec)

### 2. MySQL sum() function with WHERE clause

This example is used to return the result based on the condition specified in the WHERE clause. Execute the following query to calculate the total working hours of employees whose **working\_hours >= 12**.

1. mysql> **SELECT** **SUM**(working\_hours) **AS** "Total working hours" **FROM** employees **WHERE** working\_hours>=12;

#### Output:

This statement will give the output as below:

MySQL 8.0 Command Line Client

```
mysql> SELECT SUM(working_hours) AS "Total working hours" FROM employees WHERE working_hours>=12;
+-----+
| Total working hours |
+-----+
|      54 |
+-----+
1 row in set (0.00 sec)
```

### 3. MySQL sum() function with GROUP BY clause

We can also use the SUM() function with the GROUP BY clause to return the total summed value for each group. For example, this statement calculates the total working hours of each employee by using the SUM() function with the GROUP BY clause, as shown in the following query:

1. mysql> **SELECT** emp\_id, emp\_name, occupation, **SUM**(working\_hours) **AS** "Total working hours" **FROM** employees **GROUP BY** occupation;

#### Output:

Here, we can see that the total working hours of each employee calculates by grouping them based on their occupation.

MySQL 8.0 Command Line Client

```
mysql> SELECT emp_id, emp_name, occupation, SUM(working_hours) AS "Total working hours" FROM employees GROUP BY occupation;
+-----+-----+-----+
| emp_id | emp_name | occupation | Total working hours |
+-----+-----+-----+
|     1 | Joseph   | Business   |          30 |
|     2 | Stephen  | Doctor    |          30 |
|     3 | Mark     | Engineer  |          24 |
|     4 | Peter    | Teacher   |          27 |
+-----+-----+-----+
4 rows in set (0.00 sec)
```

### 4. MySQL sum() function with HAVING clause

The HAVING clause is used to **filter** the group with the sum() function in MySQL. Execute the following statement that calculates the working hours of all employees, grouping them based on their occupation and returns the result whose Total\_working\_hours>24.

1. mysql> **SELECT** emp\_id, emp\_name, occupation,
2. **SUM**(working\_hours) **Total\_working\_hours**
3. **FROM** employees
4. **GROUP BY** occupation
5. **HAVING** **SUM**(working\_hours)>24;

#### Output:

MySQL 8.0 Command Line Client

```
mysql> SELECT emp_id, emp_name, occupation, SUM(working_hours) Total_working_hours FROM employees GROUP BY occupation HAVING SUM(working_hours) > 24;
+-----+-----+-----+
| emp_id | emp_name | occupation | Total_working_hours |
+-----+-----+-----+
|     1 | Joseph   | Business   |          30 |
|     2 | Stephen  | Doctor    |          30 |
|     4 | Peter    | Teacher   |          27 |
+-----+-----+-----+
3 rows in set (0.00 sec)
```

### 5. MySQL sum() function with DISTINCT clause

MySQL uses the DISTINCT keyword to remove the **duplicate** rows from the column name. This clause can also be used with sum() function to return the total summed value of a Unique number of records present in the table.

Execute the following query that removes the duplicate records in the working\_hours column of the employee table and then calculates the sum:

1. mysql> **SELECT** emp\_name, occupation,
2. **SUM(DISTINCT** working\_hours) Total\_working\_hours
3. **FROM** employees
4. **GROUP BY** occupation;

**Output:**

```
MySQL 8.0 Command Line Client

mysql> SELECT emp_name, occupation, SUM(DISTINCT working_hours) Total_working_hours FROM employees GROUP BY occupation;
+-----+-----+-----+
| emp_name | occupation | Total_working_hours |
+-----+-----+-----+
| Joseph   | Business    |          10 |
| Stephen  | Doctor      |          15 |
| Mark     | Engineer    |          12 |
| Peter    | Teacher     |           9 |
+-----+-----+-----+
4 rows in set (0.10 sec)
```

## MySQL avg() function

The MySQL avg() is an aggregate function used to return the average value of an expression in various records.

### Syntax

The following are the basic syntax an avg() function in MySQL:

1. **SELECT AVG(aggregate\_expression)**
2. **FROM** tables
3. **[WHERE conditions];**

### Parameter explanation

**aggregate\_expression:** It specifies the column or expression that we are going to find the average result.

**table\_name:** It specifies the tables from where we want to retrieve records. There must be at least one table listed in the FROM clause.

**WHERE conditions:** It is optional. It specifies the conditions that must be fulfilled for the records to be selected.

## MySQL avg() function example

Consider our database has a table named **employees**, having the following data. Now, we are going to understand this function with various examples:

```
MySQL 8.0 Command Line Client

mysql> SELECT * FROM employees;
+-----+-----+-----+-----+-----+
| emp_id | emp_name | occupation | working_date | working_hours |
+-----+-----+-----+-----+-----+
| 1      | Joseph   | Business    | 2020-04-10   |          10 |
| 2      | Stephen  | Doctor      | 2020-04-10   |          15 |
| 3      | Mark     | Engineer    | 2020-04-10   |          12 |
| 4      | Peter    | Teacher     | 2020-04-10   |           9 |
| 1      | Joseph   | Business    | 2020-04-12   |          10 |
| 2      | Stephen  | Doctor      | 2020-04-12   |          15 |
| 4      | Peter    | Teacher     | 2020-04-12   |           9 |
| 3      | Mark     | Engineer    | 2020-04-12   |          12 |
| 1      | Joseph   | Business    | 2020-04-14   |          10 |
| 4      | Peter    | Teacher     | 2020-04-14   |           9 |
+-----+-----+-----+-----+-----+
10 rows in set (0.00 sec)
```

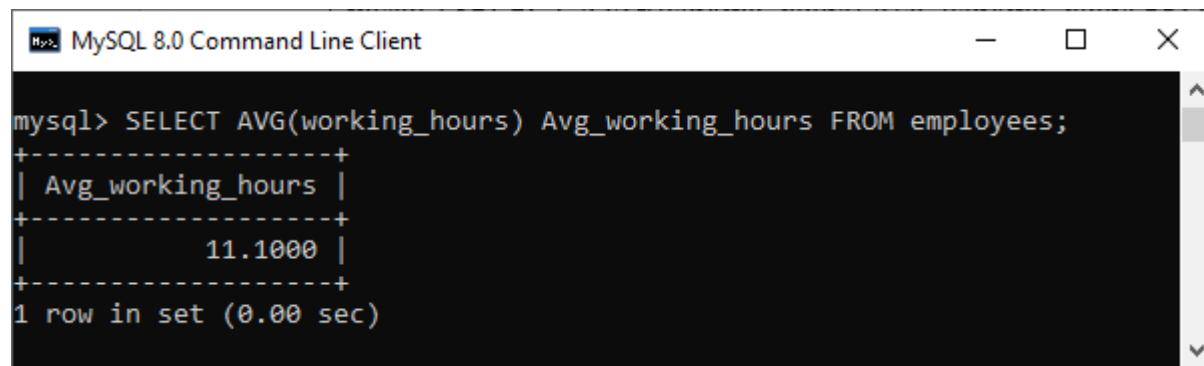
### 1. Basic Example

Execute the following query that calculates the **average working hours** of all employees in the table:

1. mysql> **SELECT AVG(working\_hours) Avg\_working\_hours FROM** employees;

## Output:

We will get the result as below:



```
MySQL 8.0 Command Line Client

mysql> SELECT AVG(working_hours) Avg_working_hours FROM employees;
+-----+
| Avg_working_hours |
+-----+
|      11.1000 |
+-----+
1 row in set (0.00 sec)
```

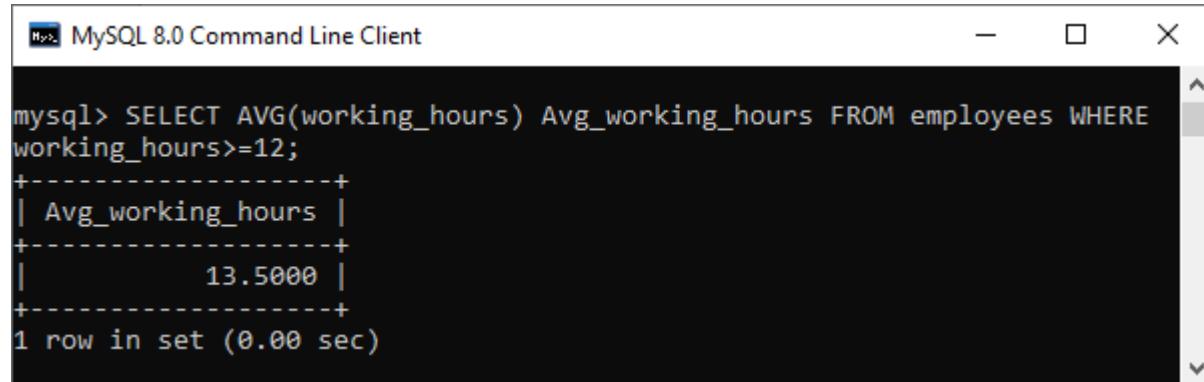
## 2. MySQL AVG() function with WHERE clause

The WHERE clause specifies the conditions that must be fulfilled for the selected records. Execute the following query to calculate the total average working hours of employees whose **working\_hours >= 12**.

1. mysql> **SELECT AVG(working\_hours) Avg\_working\_hours FROM employees WHERE working\_hours >= 12;**

## Output:

It will give the following output:



```
MySQL 8.0 Command Line Client

mysql> SELECT AVG(working_hours) Avg_working_hours FROM employees WHERE
working_hours>=12;
+-----+
| Avg_working_hours |
+-----+
|      13.5000 |
+-----+
1 row in set (0.00 sec)
```

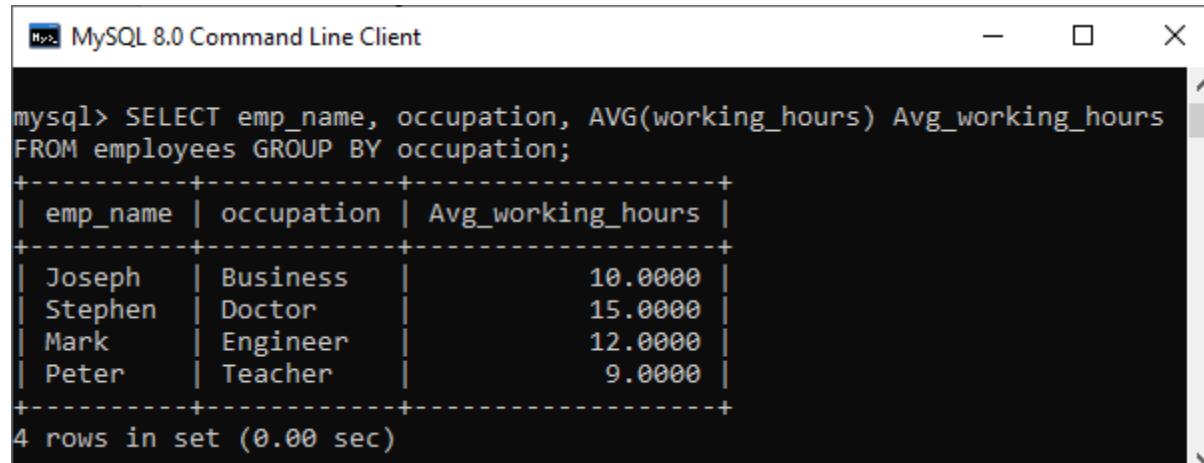
## 3. MySQL AVG() function with GROUP BY clause

The GROUP BY clause is used to return the result for each group by one or more columns. For example, this statement calculates the average working hours of each employee using the AVG() function and then group the result with the GROUP BY clause:

1. mysql> **SELECT emp\_name, occupation, AVG(working\_hours) Avg\_working\_hours FROM employees GROUP BY occupation;**

## Output:

Here, we can see that the total working hours of each employee calculates by grouping them based on their **occupation**.



```
MySQL 8.0 Command Line Client

mysql> SELECT emp_name, occupation, AVG(working_hours) Avg_working_hours
FROM employees GROUP BY occupation;
+-----+-----+-----+
| emp_name | occupation | Avg_working_hours |
+-----+-----+-----+
| Joseph   | Business   |      10.0000 |
| Stephen  | Doctor     |      15.0000 |
| Mark     | Engineer   |      12.0000 |
| Peter    | Teacher    |       9.0000 |
+-----+-----+-----+
4 rows in set (0.00 sec)
```

## 4. MySQL AVG() function with HAVING clause

The HAVING clause is used to **filter** the average values of the groups in MySQL. Execute the following statement that calculates the average working hours of all employees, grouping them based on their occupation and returns the result whose **Avg\_working\_hours > 9**.

1. mysql> **SELECT emp\_name, occupation,**
2. **AVG(working\_hours) Avg\_working\_hours**
3. **FROM employees**

4. **GROUP BY** occupation
5. **HAVING AVG(working\_hours)>9;**

**Output:**

```
MySQL 8.0 Command Line Client
mysql> SELECT emp_name, occupation, AVG(working_hours) Avg_working_hours FROM employees GROUP BY occupation HAVING AVG(working_hours)>9;
+-----+-----+-----+
| emp_name | occupation | Avg_working_hours |
+-----+-----+-----+
| Joseph   | Business   |      10.0000 |
| Stephen  | Doctor     |      15.0000 |
| Mark     | Engineer   |      12.0000 |
+-----+-----+-----+
3 rows in set (0.00 sec)
```

## 5. MySQL AVG() function with DISTINCT clause

MySQL uses the **DISTINCT** keyword to remove the **duplicate** rows from the column name. This clause is used with this **avg()** function to return the average value of a unique number of records present in the table.

Execute the following query that removes the duplicate records in the working\_hours column of the employee table and then returns the average value:

1. mysql> **SELECT** emp\_name, occupation,
2. **AVG(DISTINCT** working\_hours) Avg\_working\_hours
3. **FROM** employees
4. **GROUP BY** occupation;

**Output:**

```
MySQL 8.0 Command Line Client
mysql> SELECT emp_name, occupation, AVG(DISTINCT working_hours) Avg_working_hours FROM employees GROUP BY occupation;
+-----+-----+-----+
| emp_name | occupation | Avg_working_hours |
+-----+-----+-----+
| Joseph   | Business   |      10.0000 |
| Stephen  | Doctor     |      15.0000 |
| Mark     | Engineer   |      12.0000 |
| Peter    | Teacher    |      9.0000  |
+-----+-----+-----+
4 rows in set (0.00 sec)
```

## MySQL MIN() Function

The **MIN()** function in MySQL is used to return the **minimum value** in a set of values from the table. It is an aggregate function that is useful when we need to find the smallest number, selecting the least expensive product, etc.

### Syntax

The following is the basic syntax of **MIN()** [function in MySQL](#):

1. **SELECT MIN ( DISTINCT aggregate\_expression)**
2. **FROM table\_name(s)**
3. **[WHERE conditions];**

### Parameter explanation

This function uses the following parameters:

**aggregate\_expression:** It is the required expression. It specifies the column or expression name from which the minimum value will be returned.

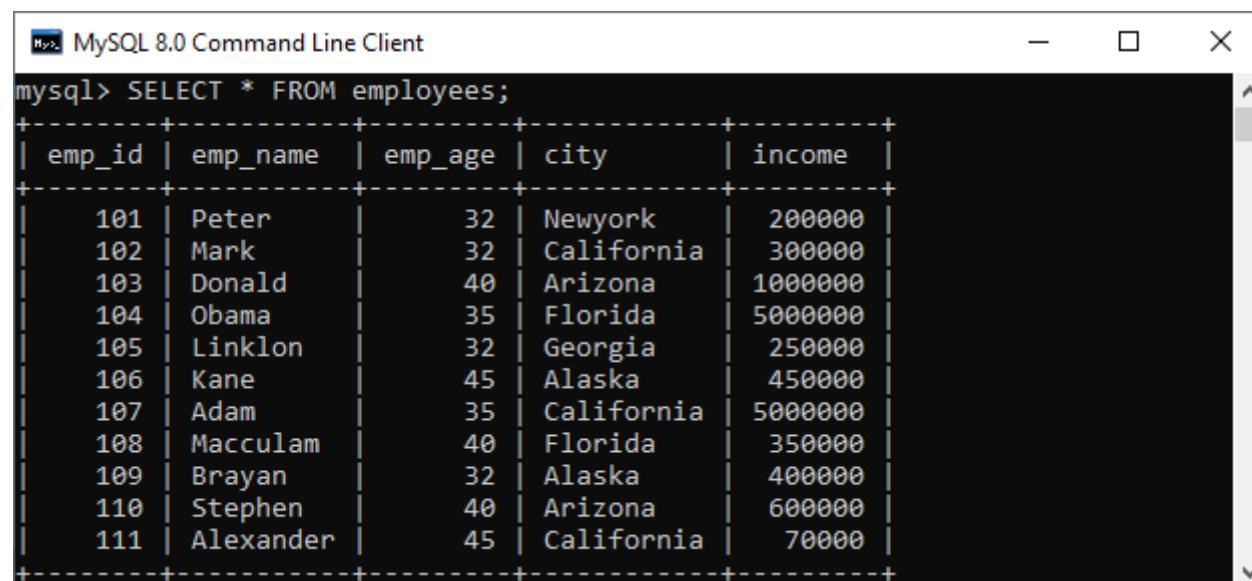
**Table\_name(s):** It specifies the tables from where we want to retrieve records. There must be at least one table listed in the FROM clause.

**WHERE conditions:** It is optional. It specifies the conditions that must be fulfilled for the records to be selected.

**DISTINCT:** It allows us to return the minimum of the distinct values in the expression. However, it does not affect the MIN() function and produces the same result without using this keyword.

## MySQL MIN() Function Example

Let us understand how MIN function works in [MySQL](#) with the help of various examples. Consider our database has a table named "**employees**" that contains the following data.



The screenshot shows the MySQL 8.0 Command Line Client window. The command entered is "SELECT \* FROM employees;". The resulting table has columns: emp\_id, emp\_name, emp\_age, city, and income. The data is as follows:

emp_id	emp_name	emp_age	city	income
101	Peter	32	Newyork	200000
102	Mark	32	California	300000
103	Donald	40	Arizona	1000000
104	Obama	35	Florida	5000000
105	Linklon	32	Georgia	250000
106	Kane	45	Alaska	450000
107	Adam	35	California	5000000
108	Macculam	40	Florida	350000
109	Brayan	32	Alaska	400000
110	Stephen	40	Arizona	600000
111	Alexander	45	California	70000

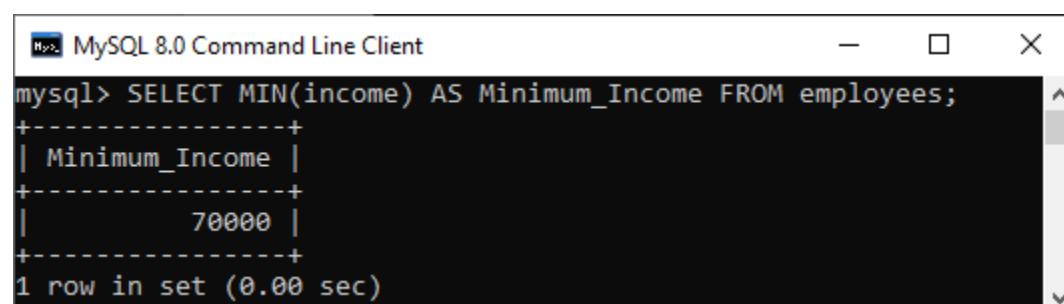
### 1. Basic Example

Execute the following query that uses the MIN function to find the **minimum income** of the employee available in the table:

1. mysql> **SELECT MIN(income) AS Minimum\_Income FROM employees;**

#### Output

The above query produces the result of minimum values in all rows. After execution, we will get the output as below:



The screenshot shows the MySQL 8.0 Command Line Client window. The command entered is "SELECT MIN(income) AS Minimum\_Income FROM employees;". The resulting table has one column: Minimum\_Income. The value is 70000. The output also includes the message "1 row in set (0.00 sec)".

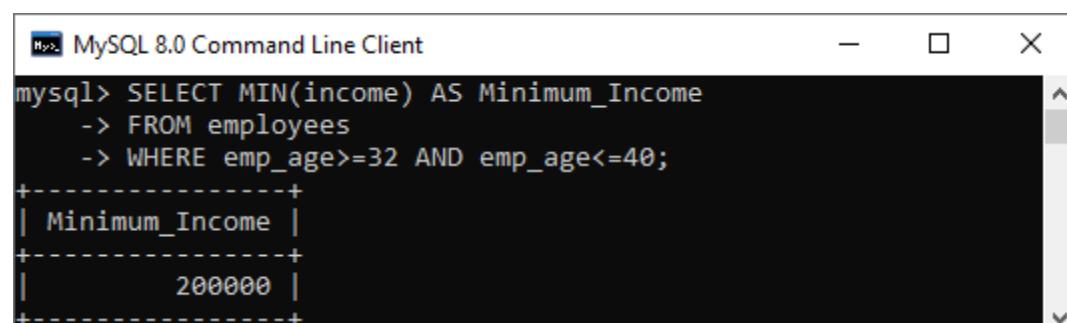
### 2. MySQL MIN() Function with WHERE Clause

The [WHERE clause](#) allows us to filter the result from the selected records. The following statement finds the minimum income in all rows from the employee table and WHERE clause specifies all those rows whose **emp\_age column** is greater than or equal to 32 and less than or equal to 40.

1. mysql> **SELECT MIN(income) AS Minimum\_Income**
2. **FROM employees**
3. **WHERE emp\_age >= 32 AND emp\_age <= 40;**

#### Output

The above statement will get the output as below:



The screenshot shows the MySQL 8.0 Command Line Client window. The command entered is "SELECT MIN(income) AS Minimum\_Income FROM employees WHERE emp\_age >= 32 AND emp\_age <= 40;". The resulting table has one column: Minimum\_Income. The value is 200000. The output also includes the message "1 row in set (0.00 sec)".

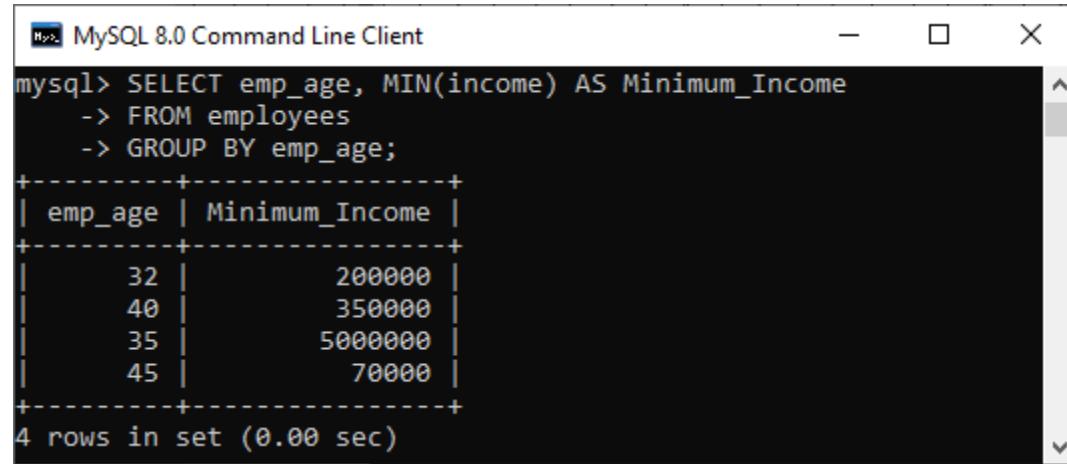
### 3. MySQL MIN() Function with GROUP BY Clause

The GROUP BY clause allows us to collect data from multiple rows and group it based on one or more columns. For example, the following statement uses the MIN() function with the GROUP BY clause to find the minimum income in all rows from the employee table for each emp\_age group.

1. mysql> **SELECT** emp\_age, **MIN**(income) **AS** Minimum\_Income
2. **FROM** employees
3. **GROUP BY** emp\_age;

#### Output

After the successful execution, we can see that the income of each employee returns by grouping them based on their age:



```
MySQL 8.0 Command Line Client
mysql> SELECT emp_age, MIN(income) AS Minimum_Income
-> FROM employees
-> GROUP BY emp_age;
+-----+-----+
| emp_age | Minimum_Income |
+-----+-----+
|      32 |        200000 |
|      40 |        350000 |
|      35 |        5000000 |
|      45 |         70000 |
+-----+-----+
4 rows in set (0.00 sec)
```

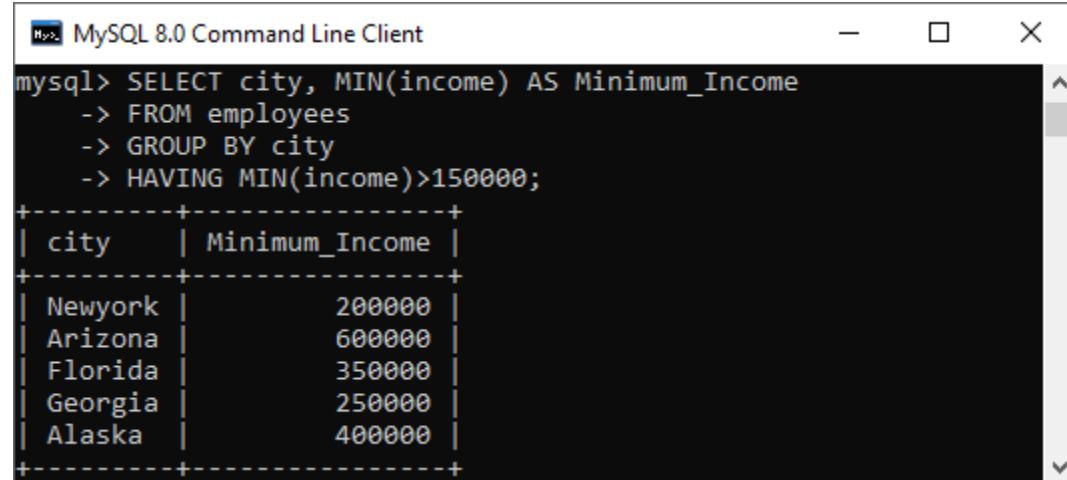
### 4. MySQL MIN() Function with HAVING Clause

The [HAVING clause](#) is always used with the GROUP BY clause to filter the records from the table. For example, the below statement returns the minimum income of all employees, grouping them based on their city and returns the result whose MIN(income)>150000.

1. mysql> **SELECT** city, **MIN**(income) **AS** Minimum\_Income
2. **FROM** employees
3. **GROUP BY** city
4. **HAVING** **MIN**(income) > 150000;

#### Output

This statement will return the output as below:



```
MySQL 8.0 Command Line Client
mysql> SELECT city, MIN(income) AS Minimum_Income
-> FROM employees
-> GROUP BY city
-> HAVING MIN(income)>150000;
+-----+-----+
| city | Minimum_Income |
+-----+-----+
| Newyork |        200000 |
| Arizona |        600000 |
| Florida |        350000 |
| Georgia |        250000 |
| Alaska |        400000 |
+-----+-----+
```

### 5. MySQL MIN() Function with DISTINCT Clause

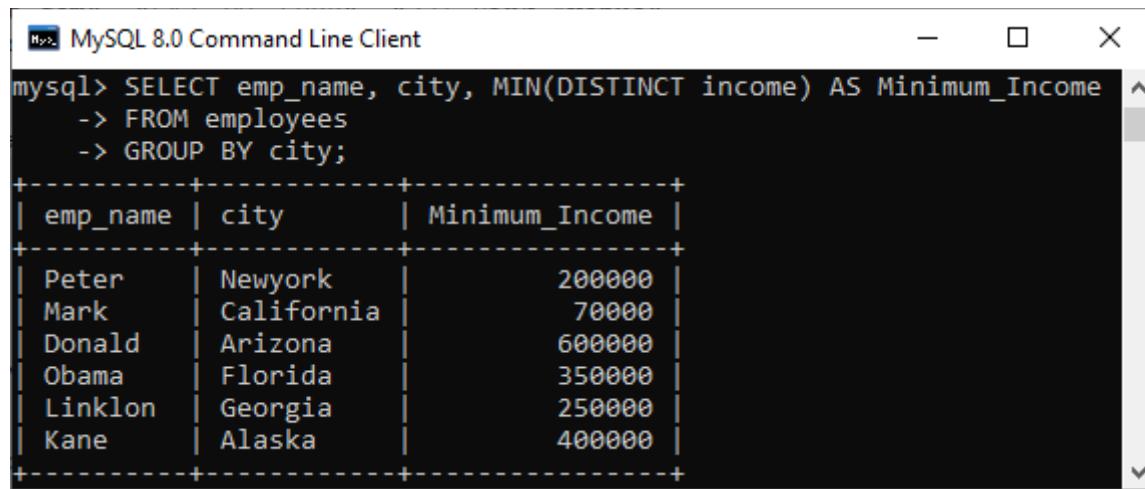
MySQL uses the [DISTINCT](#) keyword to remove the duplicate rows from the column name. We can also use this clause with MIN() function to return the minimum income value of a unique number of records present in the table.

Execute the following query that removes the duplicate records in the income column of the employee table, group by city, and then returns the minimum value:

1. mysql> **SELECT** emp\_name, city, **MIN(DISTINCT** income) **AS** Minimum\_Income
2. **FROM** employees
3. **GROUP BY** city;

#### Output

This statement will give the output as below:



```
MySQL 8.0 Command Line Client
mysql> SELECT emp_name, city, MIN(DISTINCT income) AS Minimum_Income
-> FROM employees
-> GROUP BY city;
+-----+-----+-----+
| emp_name | city      | Minimum_Income |
+-----+-----+-----+
| Peter    | Newyork   |      200000    |
| Mark     | California |      70000     |
| Donald   | Arizona    |      600000    |
| Obama    | Florida    |      350000    |
| Linklon  | Georgia   |      250000    |
| Kane     | Alaska    |      400000    |
+-----+-----+-----+
```

## MySQL MAX() Function

The MySQL MAX() function is used to return the maximum value in a set of values of an expression. This aggregate function is useful when we need to find the maximum number, selecting the most expensive product, or getting the largest payment to the customer from your table.

### Syntax

The following is the basic syntax of MAX() function in MySQL:

1. **SELECT MAX(DISTINCT aggregate\_expression)**
2. **FROM table\_name(s)**
3. **[WHERE conditions];**

### Parameter Explanation

This function uses the following parameters:

**aggregate\_expression:** It is the required expression. It specifies the column, expression, or formula from which the maximum value will be returned.

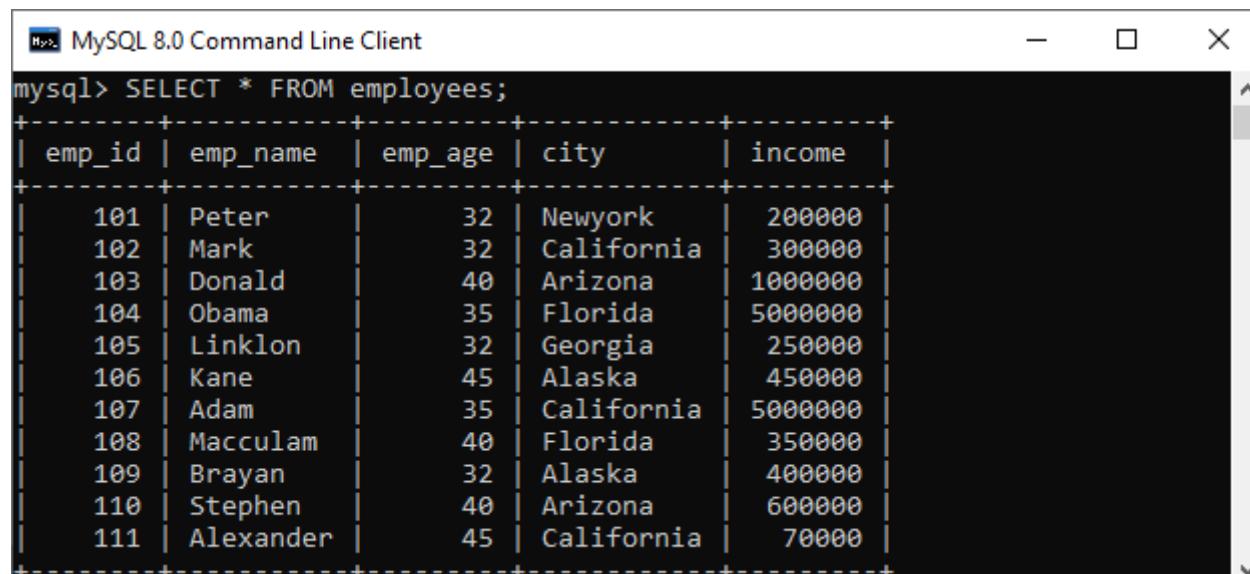
**table\_name(s):** It specifies the tables from where we want to retrieve records. There must be at least one table listed in the FROM clause.

**WHERE conditions:** It is optional. It specifies the conditions that must be fulfilled for the records to be selected.

**DISTINCT:** It allows us to return the maximum of the distinct values in the expression. However, it does not affect the MAX() function and produces the same result without using this keyword.

## MySQL MAX() Function Example

Let us understand how the MAX function works in [MySQL](#) with the help of various examples. Consider our database has a table named "**employees**" that contains the following data.



```
MySQL 8.0 Command Line Client
mysql> SELECT * FROM employees;
+-----+-----+-----+-----+-----+
| emp_id | emp_name | emp_age | city      | income |
+-----+-----+-----+-----+-----+
| 101    | Peter    |    32   | Newyork   | 200000  |
| 102    | Mark     |    32   | California | 300000  |
| 103    | Donald   |    40   | Arizona    | 1000000 |
| 104    | Obama    |    35   | Florida    | 5000000 |
| 105    | Linklon  |    32   | Georgia   | 250000  |
| 106    | Kane     |    45   | Alaska    | 450000  |
| 107    | Adam     |    35   | California | 5000000 |
| 108    | Macculam |    40   | Florida    | 350000  |
| 109    | Brayan   |    32   | Alaska    | 400000  |
| 110    | Stephen  |    40   | Arizona    | 600000  |
| 111    | Alexander |    45   | California | 70000   |
+-----+-----+-----+-----+-----+
```

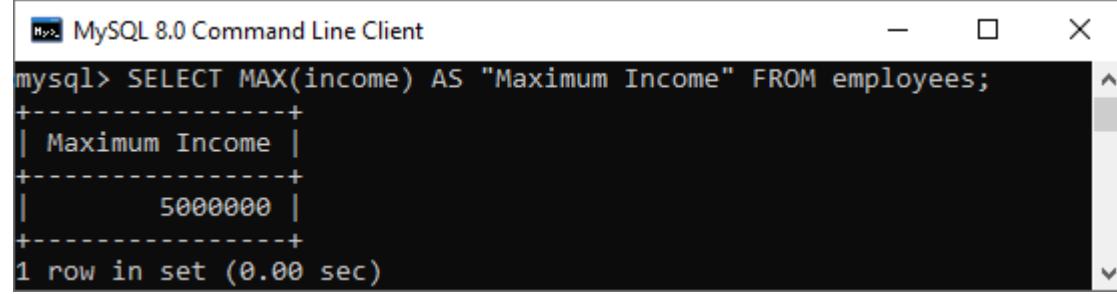
### 1. Basic Example

Execute the following query that uses the MAX function to find the **maximum income** of the employee available in the table:

```
1. mysql> SELECT MAX(income) AS "Maximum Income" FROM employees;
```

#### Output

The above query produces the result of maximum values in all rows. After execution, we will get the output as below:



```
MySQL 8.0 Command Line Client
mysql> SELECT MAX(income) AS "Maximum Income" FROM employees;
+-----+
| Maximum Income |
+-----+
|      5000000 |
+-----+
1 row in set (0.00 sec)
```

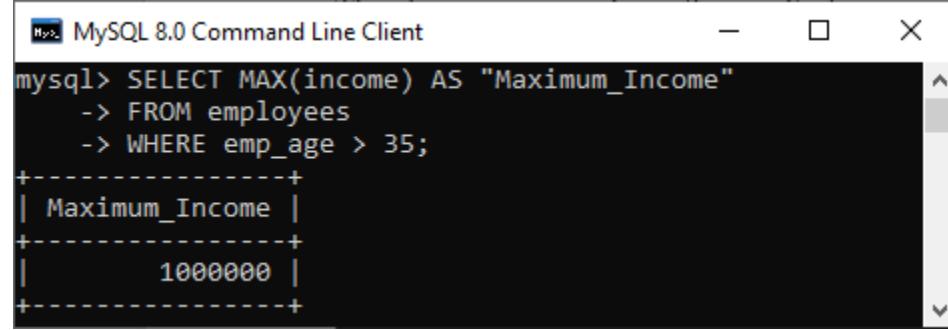
## 2. MySQL MAX() Function with WHERE Clause

The [WHERE clause](#) allows us to filter the result from the selected records. The following statement finds the maximum income in all rows from the employee table. The WHERE clause specifies all those rows whose **emp\_age column** is greater than 35.

1. mysql> **SELECT MAX(income) AS "Maximum\_Income"**
2. **FROM** employees
3. **WHERE** emp\_age > 35;

#### Output

The above statement will get the output as below:



```
MySQL 8.0 Command Line Client
mysql> SELECT MAX(income) AS "Maximum_Income"
-> FROM employees
-> WHERE emp_age > 35;
+-----+
| Maximum_Income |
+-----+
|      1000000 |
+-----+
```

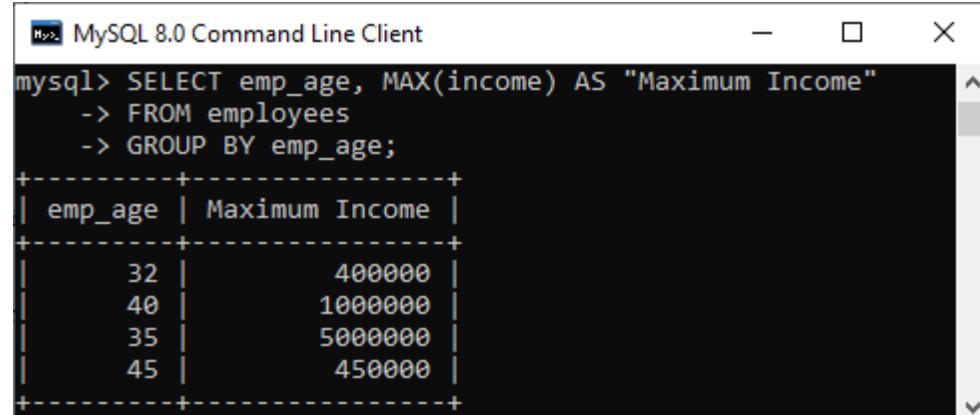
## 3. MySQL MAX() Function with GROUP BY Clause

The [GROUP BY clause](#) allows us to collect data from multiple rows and group it based on one or more columns. For example, the following statement uses the MAX() function with the GROUP BY clause to find the maximum income in all rows from the employee table for each emp\_age group.

1. mysql> **SELECT emp\_age, MAX(income) AS "Maximum Income"**
2. **FROM** employees
3. **GROUP BY** emp\_age;

#### Output

After the successful execution, we can see that the maximum income of the employee returns by grouping them based on their age:



```
MySQL 8.0 Command Line Client
mysql> SELECT emp_age, MAX(income) AS "Maximum Income"
-> FROM employees
-> GROUP BY emp_age;
+-----+
| emp_age | Maximum Income |
+-----+
|      32 |        400000 |
|      40 |       1000000 |
|      35 |      5000000 |
|      45 |        450000 |
+-----+
```

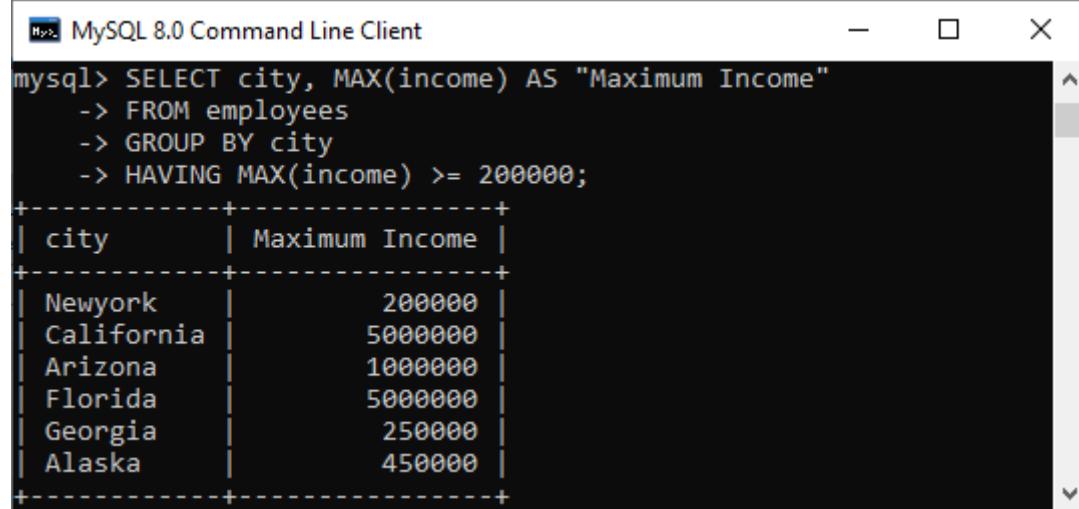
## 3. MySQL MAX() Function with HAVING Clause

The [HAVING clause](#) is always used with the GROUP BY clause to filter the records from the table. For example, the following statement returns the maximum income among all employees, grouping them based on their city and returns the result whose MAX(income) >= 200000.

1. mysql> **SELECT** city, **MAX**(income) **AS** "Maximum Income"
2. **FROM** employees
3. **GROUP BY** city
4. **HAVING MAX**(income) >= 200000;

#### Output

This statement will return the output as below:



```
MySQL 8.0 Command Line Client
mysql> SELECT city, MAX(income) AS "Maximum Income"
-> FROM employees
-> GROUP BY city
-> HAVING MAX(income) >= 200000;
+-----+-----+
| city      | Maximum Income |
+-----+-----+
| Newyork   |      200000 |
| California | 5000000 |
| Arizona   | 1000000 |
| Florida   | 5000000 |
| Georgia   | 250000 |
| Alaska    | 450000 |
+-----+-----+
```

## 5. MySQL MAX() Function with DISTINCT Clause

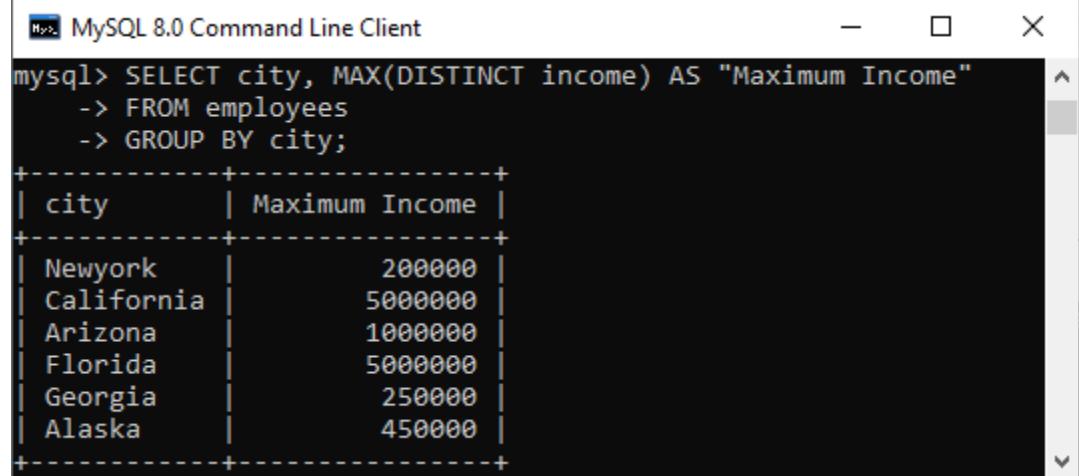
MySQL uses the **DISTINCT** keyword to remove the duplicate rows from the column name. We can also use this clause with MAX() function to return the maximum income value of a unique number of records present in the table.

Execute the following query that removes the duplicate records in the employee table's income column, group by city, and then returns the maximum value:

1. mysql> **SELECT** city, **MAX(DISTINCT** income) **AS** "Maximum Income"
2. **FROM** employees
3. **GROUP BY** city;

#### Output

This statement will give the output as below:



```
MySQL 8.0 Command Line Client
mysql> SELECT city, MAX(DISTINCT income) AS "Maximum Income"
-> FROM employees
-> GROUP BY city;
+-----+-----+
| city      | Maximum Income |
+-----+-----+
| Newyork   |      200000 |
| California | 5000000 |
| Arizona   | 1000000 |
| Florida   | 5000000 |
| Georgia   | 250000 |
| Alaska    | 450000 |
+-----+-----+
```

## 6. MySQL MAX() Function in Subquery Example

Sometimes it is required to use the subquery for returning the maximum value in the table. In that case, we use the following query:

1. mysql> **SELECT** \* **FROM** employees **WHERE**
2. emp\_age = (**SELECT MAX**(emp\_age) **FROM** employees);

The subquery first finds the maximum age of employees from the table. Then, the main query (outer query) returns the result of age being equal to the maximum age returned from the subquery and other information.

#### Output

```

MySQL 8.0 Command Line Client
mysql> SELECT * FROM employees WHERE
    -> emp_age = (SELECT MAX(emp_age) FROM employees);
+-----+-----+-----+-----+
| emp_id | emp_name | emp_age | city      | income |
+-----+-----+-----+-----+
|   106  | Kane     |     45 | Alaska   | 450000  |
|   111  | Alexander |     45 | California | 70000   |
+-----+-----+-----+-----+

```

## MySQL GROUP\_CONCAT() Function

The GROUP\_CONCAT() function in MySQL is a type of an aggregate function. This function is used to concatenate string from multiple rows into a single string using various clauses. If the group contains at least one non-null value, it always returns a string value. Otherwise, you will get a null value.

The following are the syntax of the GROUP\_CONCAT() function:

1. GROUP\_CONCAT(
2. **DISTINCT** expression
3. **ORDER BY** expression
4. SEPARATOR sep
5. );

OR,

1. mysql> **SELECT** c1, c2, ...., cN
2. GROUP\_CONCAT (
3. [DISTINCT] c\_name1
4. [ORDER BY]
5. [SEPARATOR] )
6. **FROM** table\_name **GROUP BY** c\_name2;

In this syntax,

- o The c1, c2,...,cN are the table columns.
- o The c\_name1 is the table column whose values will be concatenated into a single string for each group.
- o The c\_name2 is the table column from which grouping is performed.

The options of GROUP\_CONCAT() function are explained below:

**Distinct:** This clause removes the duplicate values in the group before doing concatenation.

**Order By:** It allows us to sorts the group data in ascending or descending order and then perform concatenation. By default, it performs the sorting in the ascending order. But, you can sort values in descending order using the DESC option explicitly.

**Separator:** By default, this clause uses comma(,) operator as a separator. If you want to change the default separator, you can specify the literal value.

**NOTE:** This function always returns a result in binary or non-binary string value that depends on the specified arguments. By default, it returns maximum length of string value equal to 1024. If you want to increase this length, you can use the group\_concat\_max\_len system variable.

## GROUP\_CONCAT() Example

Let us create a table employee to understand how this function works in [MySQL](#) using different queries.

emp_id	emp_fname	emp_lname	dept_id	designation
1	David	Miller	2	Engineer
2	Peter	Watson	3	Manager
3	Mark	Boucher	1	Scientist
2	Peter	Watson	3	BDE
1	David	Miller	2	Developer
4	Adam	Warner	4	Receptionist
3	Mark	Boucher	1	Engineer
4	Adam	Warner	4	Clerk

## 1. Using a Simple Query

1. mysql> **SELECT** emp\_id, emp\_fname, emp\_lname, dept\_id,
2. GROUP\_CONCAT(designation) **as** "designation" **FROM** employee **group by** emp\_id;

This statement will give the following output:

emp_id	emp_fname	emp_lname	dept_id	designation
1	David	Miller	2	Engineer,Developer
2	Peter	Watson	3	Manager,BDE
3	Mark	Boucher	1	Scientist,Engineer
4	Adam	Warner	4	Receptionist,Clerk

## 2. Using DISTINCT Clause

1. mysql> **SELECT** emp\_fname, dept\_id,
2. GROUP\_CONCAT(**DISTINCT** designation) **as** "designation" **FROM** employee **group by** emp\_id;

After successful execution of the above statement, we will get the following output:

emp_fname	dept_id	designation
David	2	Developer,Engineer
Peter	3	BDE,Manager
Mark	1	Engineer,Scientist
Adam	4	Clerk,Receptionist

## 3. Using Separator Clause

1. mysql> **SELECT** emp\_fname,
2. GROUP\_CONCAT(**DISTINCT** designation SEPARATOR ';' ) **as** "designation" **FROM** employee **group by** emp\_id;

Here, the separator clause changes the default returning string comma(,) to a semicolon(;) and a whitespace character.

The above statement will give the following output:

emp_fname	designation
David	Developer; Engineer
Peter	BDE; Manager
Mark	Engineer; Scientist
Adam	Clerk; Receptionist

## GROUP\_CONCAT() and CONCAT\_WS()

Now, you are aware of the working of the GROUP\_CONCAT() function. Sometimes, we can use this function with the CONCAT\_WS() function that gives the more useful result. The following statement explains it more clearly:

1. mysql> **SELECT** GROUP\_CONCAT(CONCAT\_WS(' ', emp\_lname, emp\_fname) SEPARATOR ';') **as** employename **FROM** employee;

In this statement, the CONCAT\_WS() function first concatenates the first name and last name of each employee and results in the full name of the employees. Next, we use the GROUP\_CONCAT() function with a semicolon (;) separator clause to make the list of all employees in a single row. Finally, execute the statement. After successful execution, we will get the following output:

employename
Miller, David; Watson, Peter; Boucher, Mark; Watson, Peter; Miller, David; Warner, Adam; Boucher, Mark; Warner, Adam

This function returns the result in a single row, not a list of values. Therefore, we cannot work GROUP\_CONCAT() function with the **IN** operator. If we use an IN operator with this function, then the query will not work because the IN operator accepts a list of values, not a string.

# MySQL first function

The MySQL first function is used to return the first value of the selected column. Here, we use limit clause to select first record or more.

## Syntax:

1. **SELECT** column\_name
2. **FROM** table\_name

3. LIMIT 1;

## MySQL first function example

### To SELECT FIRST element:

Consider a table named "officers", having the following data.



The screenshot shows the MySQL 5.5 Command Line Client window. The command `USE sssit;` is run, changing the database to `sssit`. Then, `SHOW tables;` is run, displaying the tables `Tables_in_sssit`: `courses`, `employees`, `officers`, and `students`. Finally, `SELECT * FROM officers;` is run, displaying the following data:

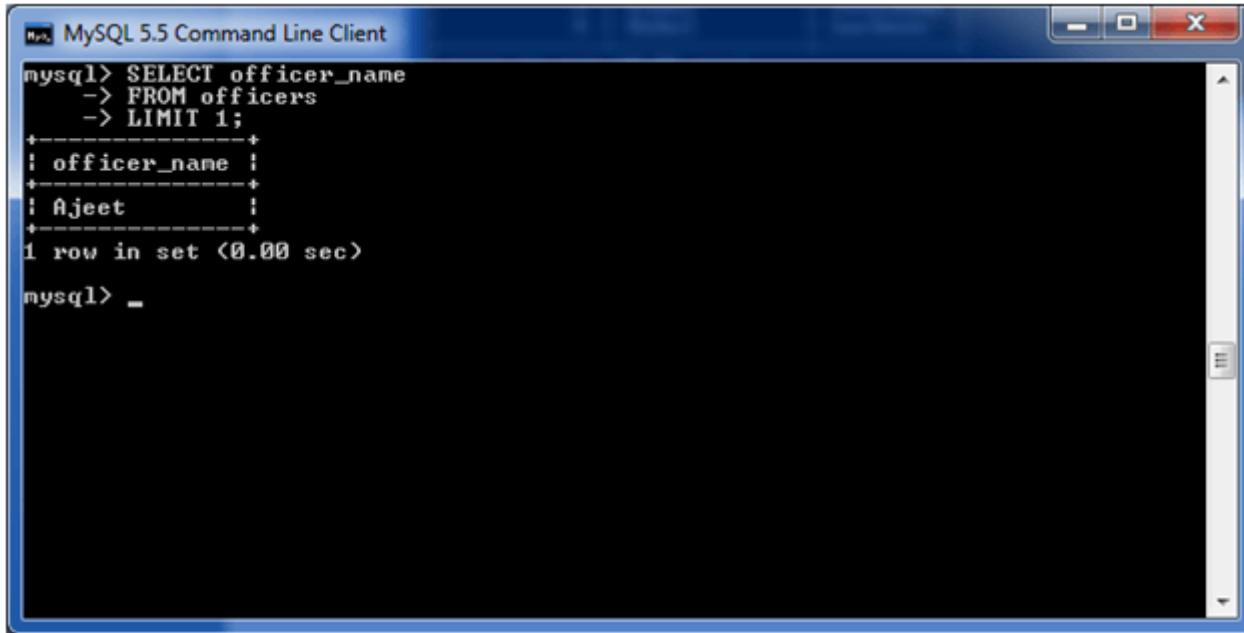
officer_id	officer_name	address
1	Ajeet	Mau
2	Deepika	Lucknow
3	Uimal	Faizabad
4	Rahul	Lucknow

4 rows in set (0.05 sec)

Execute the following query:

1. **SELECT** officer\_name
2. **FROM** officers
3. **LIMIT** 1;

Output:



The screenshot shows the MySQL 5.5 Command Line Client window. The query `SELECT officer_name  
 -> FROM officers  
 -> LIMIT 1;` is run, resulting in the output:

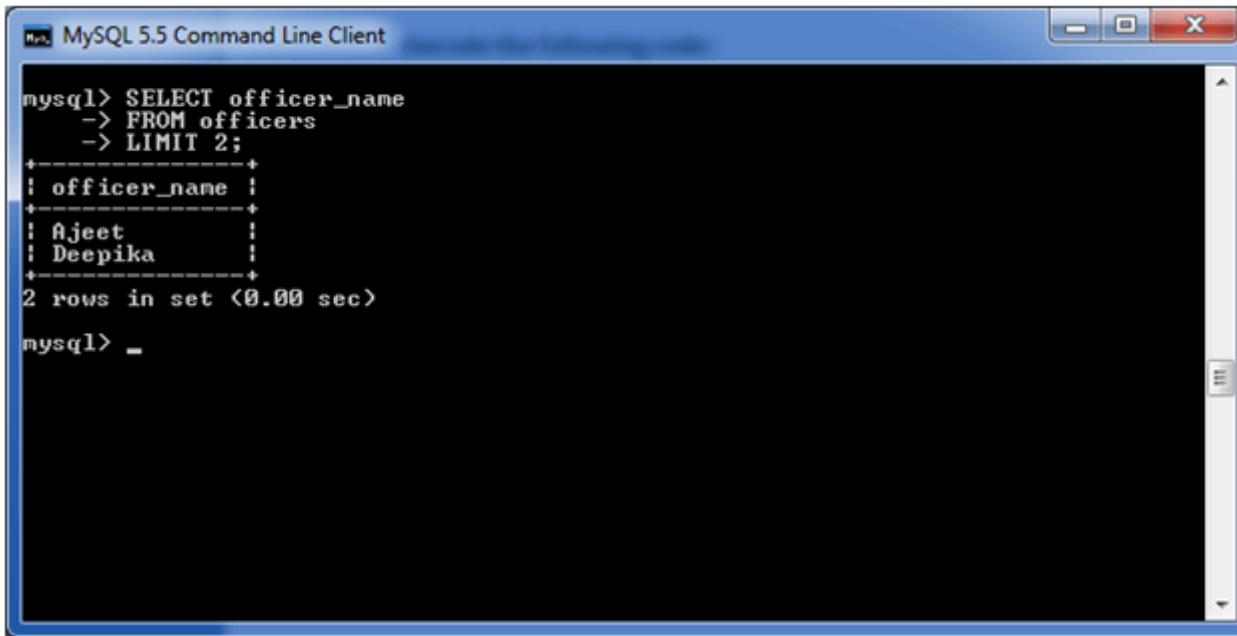
officer_name
Ajeet

1 row in set (0.00 sec)

To SELECT FIRST two records

1. **SELECT** officer\_name
2. **FROM** officers
3. **LIMIT** 2;

Output:



```
MySQL 5.5 Command Line Client
mysql> SELECT officer_name
-> FROM officers
-> LIMIT 2;
+-----+
| officer_name |
+-----+
| Ajeet        |
| Deepika      |
+-----+
2 rows in set (0.00 sec)

mysql>
```

## MySQL last function

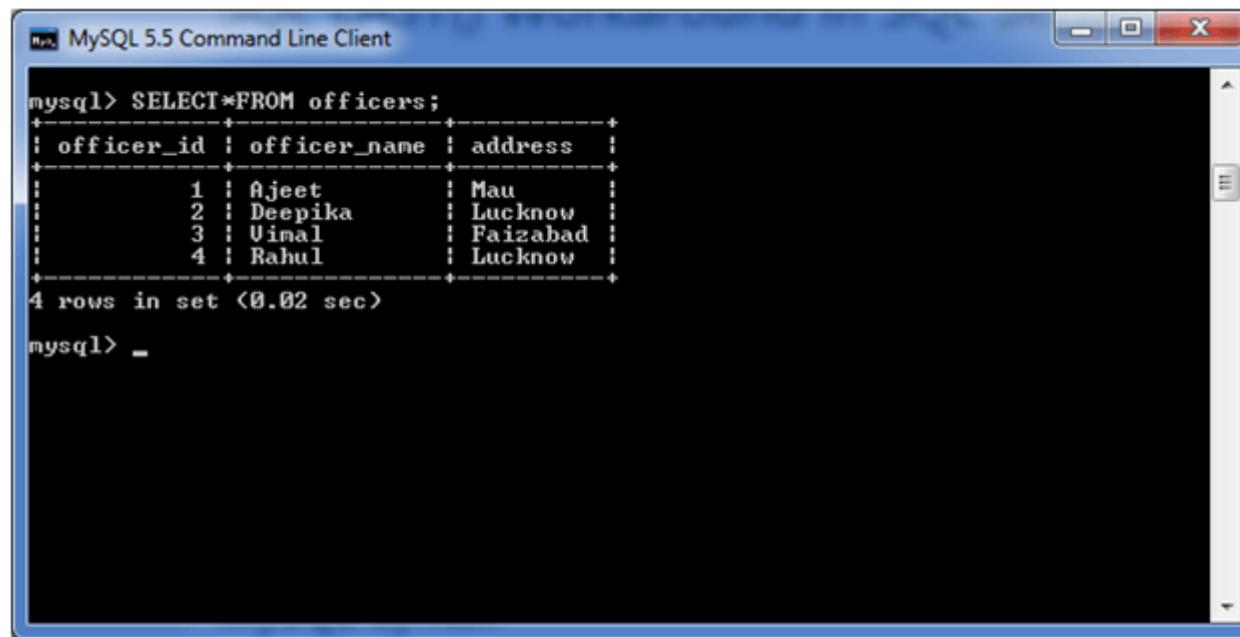
MySQL last function is used to return the last value of the selected column.

### Syntax:

1. **SELECT** column\_name
2. **FROM** table\_name
3. **ORDER BY** column\_name **DESC**
4. **LIMIT 1;**

## MySQL last function example

Consider a table "officers" having the following data.



```
MySQL 5.5 Command Line Client
mysql> SELECT * FROM officers;
+-----+-----+-----+
| officer_id | officer_name | address   |
+-----+-----+-----+
| 1          | Ajeet       | Mau       |
| 2          | Deepika     | Lucknow   |
| 3          | Vimal       | Faizabad  |
| 4          | Rahul       | Lucknow   |
+-----+-----+-----+
4 rows in set (0.02 sec)

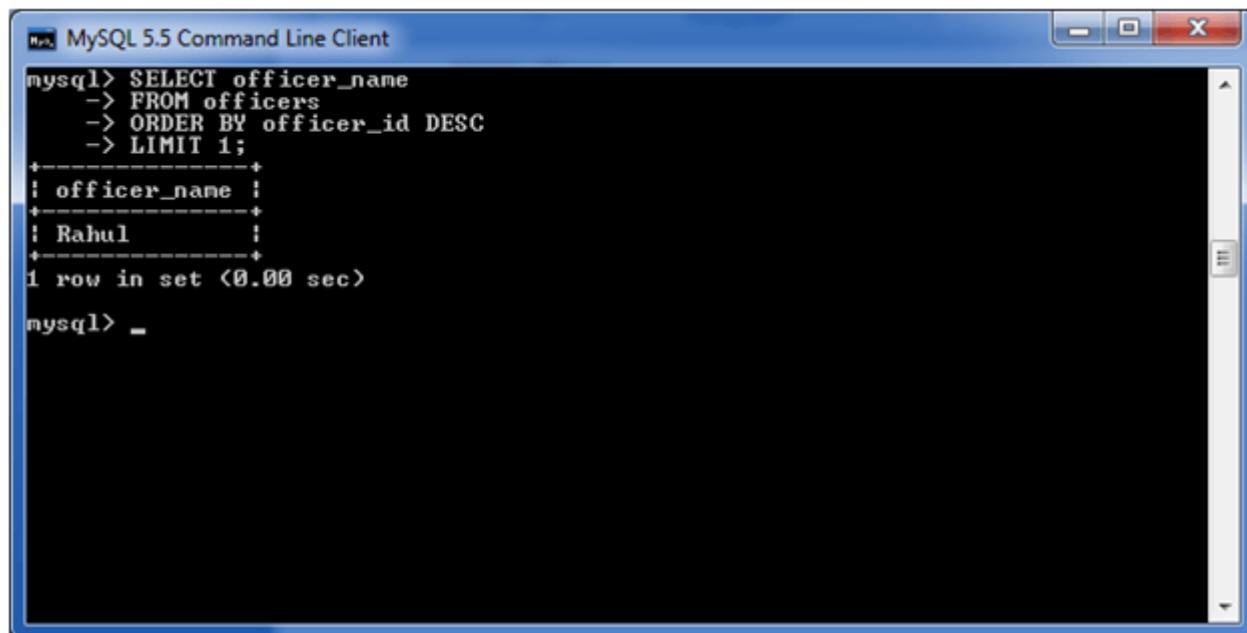
mysql>
```

### Execute the following query:

1. **SELECT** officer\_name
2. **FROM** officers
3. **ORDER BY** officer\_id **DESC**
4. **LIMIT 1;**

This query will return the last officer\_name ordering by officer\_id.

### Output:



MySQL 5.5 Command Line Client

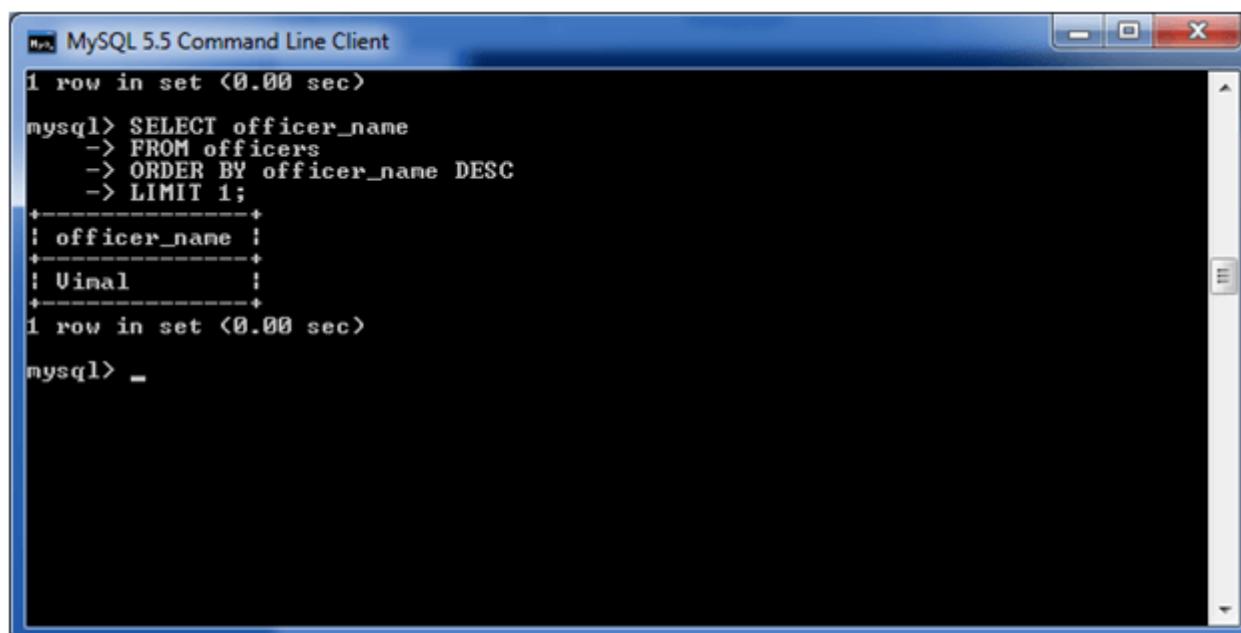
```
mysql> SELECT officer_name
-> FROM officers
-> ORDER BY officer_id DESC
-> LIMIT 1;
+-----+
| officer_name |
+-----+
| Rahul       |
+-----+
1 row in set (0.00 sec)

mysql> _
```

Return the last officer\_name ordering by officer\_name:

1. **SELECT** officer\_name
2. **FROM** officers
3. **ORDER BY** officer\_name **DESC**
4. **LIMIT 1;**

Output:



MySQL 5.5 Command Line Client

```
1 row in set (0.00 sec)

mysql> SELECT officer_name
-> FROM officers
-> ORDER BY officer_name DESC
-> LIMIT 1;
+-----+
| officer_name |
+-----+
| Vimal       |
+-----+
1 row in set (0.00 sec)

mysql> _
```