Learn SQL with MySQL

Retrieve and Manipulate Data Using SQL Commands with Ease

ASHWIN PAJANKAR

bpb

Learn SQL with MySQL

Retrieve and Manipulate Data Using SQL Commands with Ease

ASHWIN PAJANKAR

bpb

Learn SQL with MySQL

Retrieve and Manipulate Data Using SQL Commands with Ease

Ashwin Pajankar

FIRST EDITION 2020

ISBN: 978-93-89898-088

Distributors:

BPB PUBLICATIONS

20, Ansari Road, Darya Ganj

New Delhi-110002

Ph: 23254990/23254991

MICRO MEDIA

Shop No. 5, Mahendra Chambers,

150 DN Rd. Next to Capital Cinema,

V.T. (C.S.T.) Station, MUMBAI-400 001

Ph: 22078296/22078297

DECCAN AGENCIES

4-3-329, Bank Street,

Hyderabad-500195

Ph: 24756967/24756400

BPB BOOK CENTRE

376 Old Lajpat Rai Market,
Delhi-110006
Ph: 23861747

Published by Manish Jain for BPB Publications, 20 Ansari Road, Darya Ganj, New Delhi-110002 and Printed by him at Repro India Ltd, Mumbai

Dedicated to

Jayant Narlikar

Prominent Indian Astrophysicist

About the Author

Ashwin is an experienced veteran who, for the past 25 years, has been working with STEM (Science, Technology, Engineering, and Mathematics). In his career, Ashwin has worked for more than 7 years as an employee for various IT companies and Software Product Companies. He has written more than 2 dozen books on Arduino, Python programming, Computer Vision, IoT, databases, and other popular topics with BPB and other international publications. He has also reviewed many other technical books. He also creates courses for BPB and other platforms and teaches to 60000 students online.

He has been working as a freelancer since 2017. He got his first taste in writing in 2015 when he wrote his first book on Raspberry Pi. In his free time, Ashwin makes videos for his Youtube channel, which has 10000 subscribers now.

Outside work, Ashwin volunteers his spare time as a STEM Ambassador, helping, coaching, and mentoring young people in taking up careers in technology.

About the Reviewer

Md Abdul Aziz, currently working as Staff Engineer at VMware Software India Pvt Ltd graduated from IIIT-Hyderabad with CSE degree (Honors in Data Engineering) in 2007. Aziz is passionate about open-source projects and in his leisure time works on Linux kernel projects, open stack, and cloud computing.

Aziz delivered lectures and talk on topics of IPv6, Linux Networking stack, Cloud Technologies at prestigious institutions like NIT-Warangal, PESIT, RVCE. Aziz has a patent, Multi-node Virtual Switching System (USPTO DN/20140204805). In the past, Aziz held the position of Software Development Engineer and Lead at Cisco, Microsoft in the areas of Virtualization, Data Center Technologies, Multi-layer Switching, and Internet Technologies.

Acknowledgement

There are a few people I want to thank for the continued and ongoing support they have given me during the writing of this and other books with BPB Publications.

This book would not have happened if I had not had the support from some of the key people from BPB Publications. My gratitude goes to the team at BPB. I could have never completed this book without their support. This is my fourth book with BPB, and I plan to write more.

Finally, I would like to thank the technical reviewer for his valuable input.

Preface

I have been professionally working with various DBMS ever since I started my formal professional career in June 2007. I have extensively worked with various ORDBMS (Object Relational Database Management Systems) like Oracle, IBM DB2, Teradata, MySQL, and MariaDB. And I have been working with MySQL since 2010.

We use relational databases and ORDMBS everywhere, from enterprise applications to mobile apps. They are an integral part of programmers' life and livelihood. While everyone does not spend most of his working hours directly working with databases, programmers routinely access database services for their applications. Most of the organizations employ RDBMS for the structured storage of data.

SQL (Structured Query Language) is the lingua franca of the world of RDBMS and ORDBMS. Almost all the major ORDBMS software has a flavor of SQL implemented in the Engine so that the administrators and common users can interact with the ORDMBS and the data with ease. SQL has seen action on almost all the operating systems and computing platforms, including servers, desktop computers, handheld computers, and mobile devices. There is ORDBMS software available for all the types of computing devices, and the SQL is the common ground amongst them.

With these many flavors of SQL provided by various RDBMS software, beginners and novice developers are often confused with the choice of RDBMS for learning SQL. While software vendors like Oracle have provided a free to use, but non-enterprise version of their proprietary software free to use for non-commercial purposes, MySQL and its fork MariaDB are free to use projects for even commercial

purpose. We can use their full features for production level and realtime usage. It is often advised for beginners to start learning SQL with MySQL or MariaDB. We can learn the SQL with the latest ANSI syntax with the help of MySQL and MariaDB databases. These databases are available with Windows and Linux. In this book, we will learn how to install MySQL and MariaDB on the Windows Platform. We will also learn how to install MariaDB on Raspberry Pi Raspbian OS (recently rechristened as Raspberry Pi OS). We will learn various interfaces like MySQL prompt and MySQL Workbench for connecting with MySQL and MariaDB servers.

This book is extremely useful for beginners as it starts from scratch and covers the most basic part in the beginning. If you have sufficient technical background and lack detailed knowledge of SQL, then this book is the right choice to get started with SQL. This book is very useful for developers, engineers, data scientists, and entry-level DBAs.

Chapter 1 introduces to MySQL and MariaDB projects as well as the concepts in the domain of databases.

Chapter 2 discusses the tools to connect to MySQL and MariaDB. It also explains how to download and install sample schemas on MySQL and MariaDB servers for practice.

Chapter 3 discusses the SELECT query in detail. It also explains the concept of NULL and DISTINCT rows.

Chapter 4 is a key chapter that discusses, in-depth, what WHERE clause can accomplish. It focuses on various Operators. It explains the selection operation in detail.

Chapter 5 is also a key chapter that discusses, in-depth, what a single row function is. It discusses the NULL handling and nested function calls for single-row function.

Chapter 6 describes some of the more advanced functions known as group functions. It explores Group by and Having clauses in SQL.

Chapter 7 introduces the readers to the technique of combining data from various sources. This technique is known as the Join. The chapter explores the ANSI syntax and older syntax for various types of joins.

Chapter 8 describes how to write a query within a query. It explores two types of subqueries in detail.

Chapter 9 describes how to perform DDL and DML. It also explores the concept of transactions in detail.

Chapter 10 describes how to create views. It explores both the types of views in detail.

Chapter 11 describes how to connect Python 3 with MySQL and MariaDB. It explores how to load a pandas dataframe with data from the MySQL table.

Downloading the code bundle and coloured images:

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/pywh61l

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors if any, occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Table of Contents

CHAPTER 1

Introduction and Installation

I recommend all the readers to read the preface and the table of contents. It has a lot of information about what we can expect in the book. So, if you have not read it, I recommend going through it line by line.

In this chapter, we are going to start the exciting journey of learning SQL with widely used and adopted open-source database software MySQL and MariaDB. We will also have a look at a few essential concepts that are important before we start hands-on. Finally, we will learn in detail how to install MySQL and MariaDB on major OS platforms. So, let us start an incredible journey of learning SQL with MySQL and MariaDB.

Structure

We will learn the following topics in this chapter:

- Basic concepts related to Databases, DBMS, and SQL
- Hands-on with SQL using “Tryit Online SQL Editor.”
- MySQL and its installation on Windows
- MariaDB and installation on Windows and Linux

Objective

The objective of this chapter is to make the readers comfortable with the basic concepts related to the topic of the Database. Also, the readers will be able to install the MySQL and MariaDB on the different Operating Systems. Readers will also learn to use an online SQL Editor, the Tryit

Online SQL Editor. All the software that we learn to install in this chapter will be used throughout the rest of the book.

Databases, DBMS, and SQL

A database is data collected and stored in an organized form. It can be stored, accessed, and processed in electronic format or in the paper-format too. Before the advent of computers, people and organizations used to store records in tabular forms in books and papers. These were precursors to modern relational database systems. Today, in the 21st century, we store and process almost all the databases in the electronic format.

Databases that use a relational model for storing and processing data are known as relational databases. *E. F. Codd* first proposed the relational data model in his research paper **"A Relational Model of Data for Large Shared Data Banks"**in 1970. Almost all the relational databases use **Structured Query Language (SQL)** for processing the data stored in the Database. In the relational model, the related entities are stored in a tabular data structure known as a table. Modern relational databases are often Object-Relational Databases. They can be easily interfaced with the programming languages that support the concept of Objects. There are other historical and current data models too. Following is the list of a few of them:

- Hierarchical database model
- Network model
- Object model
- Document model
- Key–value model
- Associative model
- Correlational model

- Multidimensional model
- Multivalue model
- Semantic model
- XML database
- Named graph (GraphDB)
- Triplestore/Resource Distribution Framework (RDF)

A Database Management System is a software that interacts with the Database, Operating System, programming languages, and end-users to collect, store, process, and analyze the data stored in the Database. A DBMS that works with relational databases is Relational DBMS, and similarly, a DBMS that works with the object-relational data model is Object-Relational DBMS.

DBMS and Databases are vast topics that require several dedicated books themselves. We will be encountering a lot of DBMS concepts while learning SQL. We will absorb those concepts wherever we encounter them in this book.

A Bit of Hands-on with SQL

We have learned in the earlier section that SQL is a query-based languagefor interacting with the Database. We can try demonstrations with SQL without even installing any of the DBMS software. All we need is a computer with an internet connection. **w3schools.com**has a very nice online SQL editor and a sample database with a few tables. It is known as Tryit Online SQL Editor. It is intended for quickly practicing SQL skills. While it is not a full-featured database product, we can certainly practice a lot of queries here. So, let us get started. Open a web browser on your computer and visit the URL **https://www.w3schools.com/sql/trysql.asp?filename=trysql_op_in**. It will show the following page:

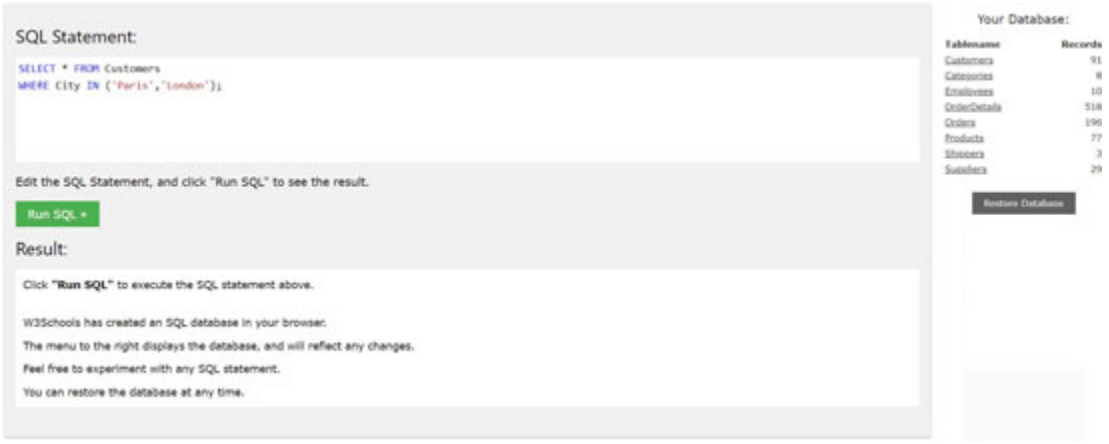


Figure 1.1

We can see a text area occupying a big part of the page. We also already have the text `SELECT * FROM Customers WHERE City IN ('Paris', 'London');` already present in the text area. This text is known as a query. A query is a SQL statement that a database engine runs to perform some operation on the Database. In this case, the Database is local and is in our browser and memory (RAM). We can see a green button that says `Run SQL`. To the right, it displays the member tables in the Database and the number of records in those tables listed against them.

Let us run the query in the text area. It will show the following output:

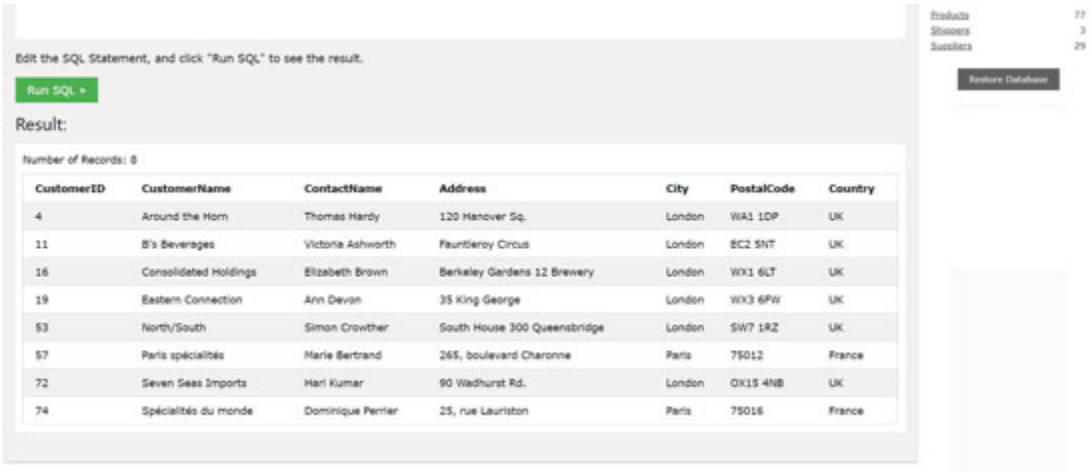


Figure 1.2

The query is simple. It selects all the records from the table `Customers` where the column `City` has values `London` and `Paris`. Modify the query like `SELECT * FROM Customers;` execute it again. It will show all the records in the table `Customers`. A semicolon, in the end, is not mandatory. Remove the semicolon and run the query.

The queries are not case sensitive except when they are referring to the data in the tables. We can write the above query such as se`lect * from customers`, and it will run without any problem. However, the string data in the Database is case sensitive. The query `select * from customers where city in ('paris', 'london')` is not the same as sel`ect * from customers where city in ('Paris', 'London')`. Run both the queries separately and observe that the earlier(the one with the names of cities in lowercase) does not yield any rows in the output as in the column `City`, the table `Customer`, all the city names have the first letter of the name in capital. As we learned earlier, the data stored in the database tables is always case sensitive, whereas the keywords and table name in a query are case insensitive. We can practice more queries with this. But we need real database products to learn all the features SQL.

MySQL

MySQL is a free and open-source relational DBMS. It was created by a Swedish company **MySQL AB.** MySQL AB was founded by *David Axmark, Allan Larsson,* and *Michael "Monty" Widenius. My* is *Michael's* daughter's name. That is why the product is christened as MySQL. MySQL is an important component of **Linux, Apache, MySQL, Perl / Python / PHP (LAMP)** web development stack. It is used by many government agencies, including NASA (Reference: **https://www.mysql.com/fr/customers/industry/?id=69**) and big organizations like Facebook, Twitter, and Youtube. We can find details about MySQL at

https://www.mysql.com/. The development of MySQL began in 1994. The following timeline shows the major milestones:

- First internal release on *23 May 1995*
- Windows version was released on *8 January 1998* for Windows 95 and NT
- **Version 3.21:**production release *1998*
- **Version 4.0:**beta from *August 2002,* production release *March 2003*
- **Version 5.0:** beta from *March 2005,* production release *October 2005*
- Sun Microsystems acquired MySQL AB in *2008*
- Oracle acquired *Sun Microsystems* on *27 January 2010* and *Michael Widenius* announced a fork of MySQL, MariaDB
- MySQL Server 5.5 was generally available
- MySQL Server 6.0.11-alpha was announced on *22 May 2009*
- MySQL Server 8.0 was announced in *April 2018*

Currently, MySQL is further developed and managed by Oracle Corporation. We will discuss the community-developed fork of MySQL, MariaDB, later in the chapter.

Install MySQL on Windows

Let us see how to install MySQL on Windows. Visit **https://www.mysql.com** and go to the download page. MySQL has many editions. Go to the download page of the community edition (**https://dev.mysql.com/downloads/installer/)** as it is the only free edition of MySQL. Following is the screenshot of the page:

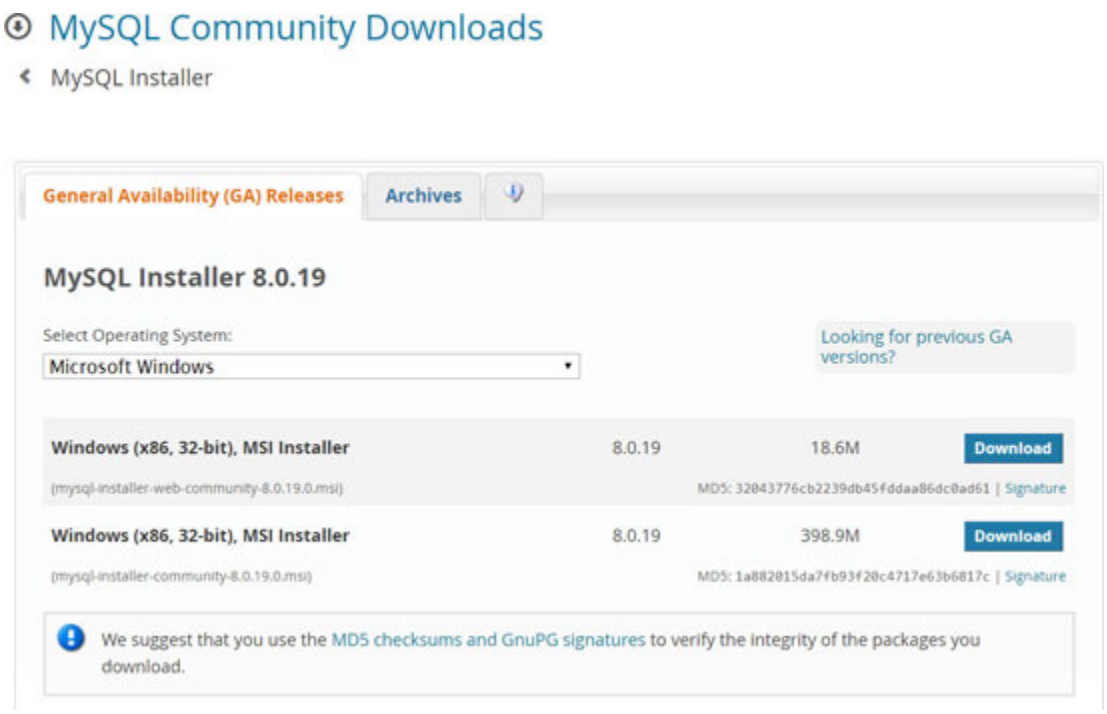


Figure 1.3

The latest version as of writing the books is 8.0.19. There are two types of installers. The first is a web installer that downloads the selected components from the web. That is why its size is smaller. Download it, and you can find it in the `Downloads` directory of your user. Double click to run it. It requires admin privileges. Enter the privileges, and it might show the following message box:

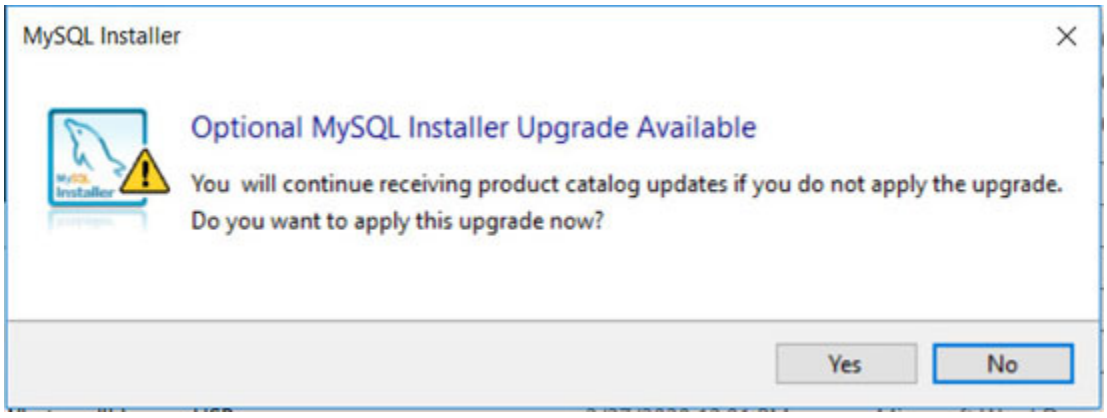


Figure 1.4

Click on the **`Yes`** button and upgrade the installer. Once the installation begins, choose the option **`Custom`** as shown in the following screenshot:

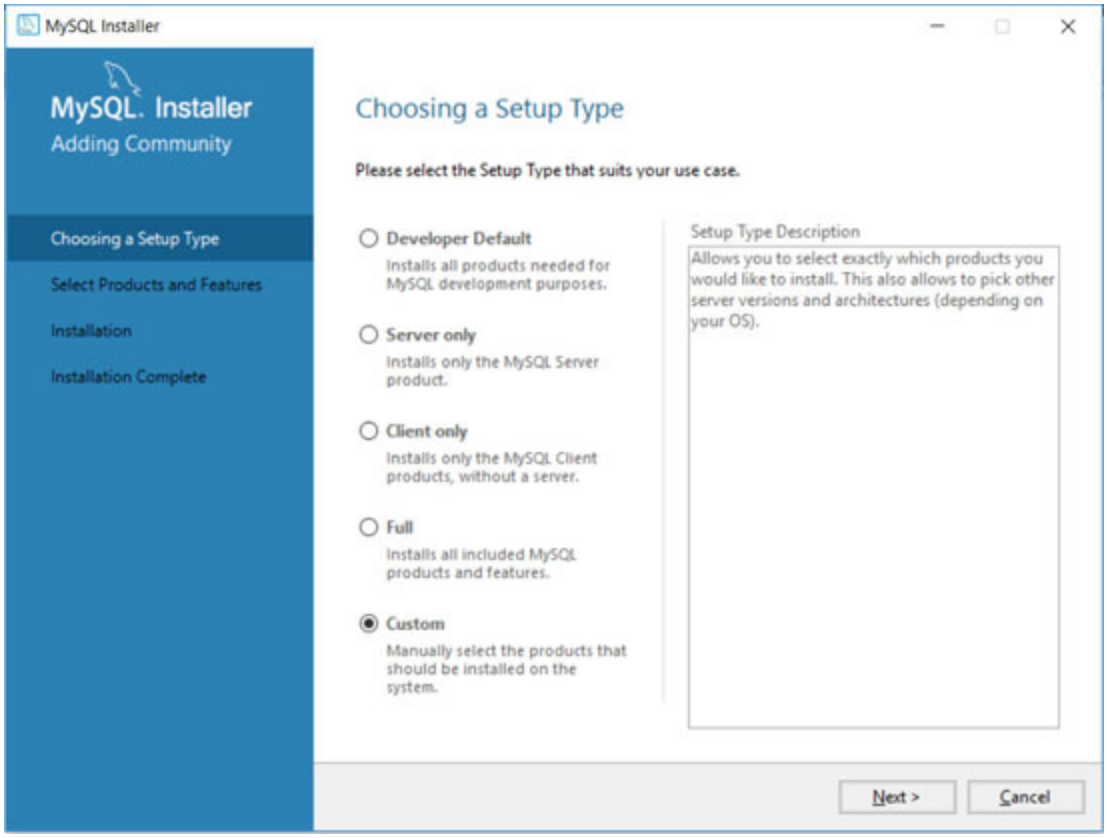


Figure 1.5

In the next window of the wizard, we get to choose what components to install. Choose all the components, as shown in the following screenshot:

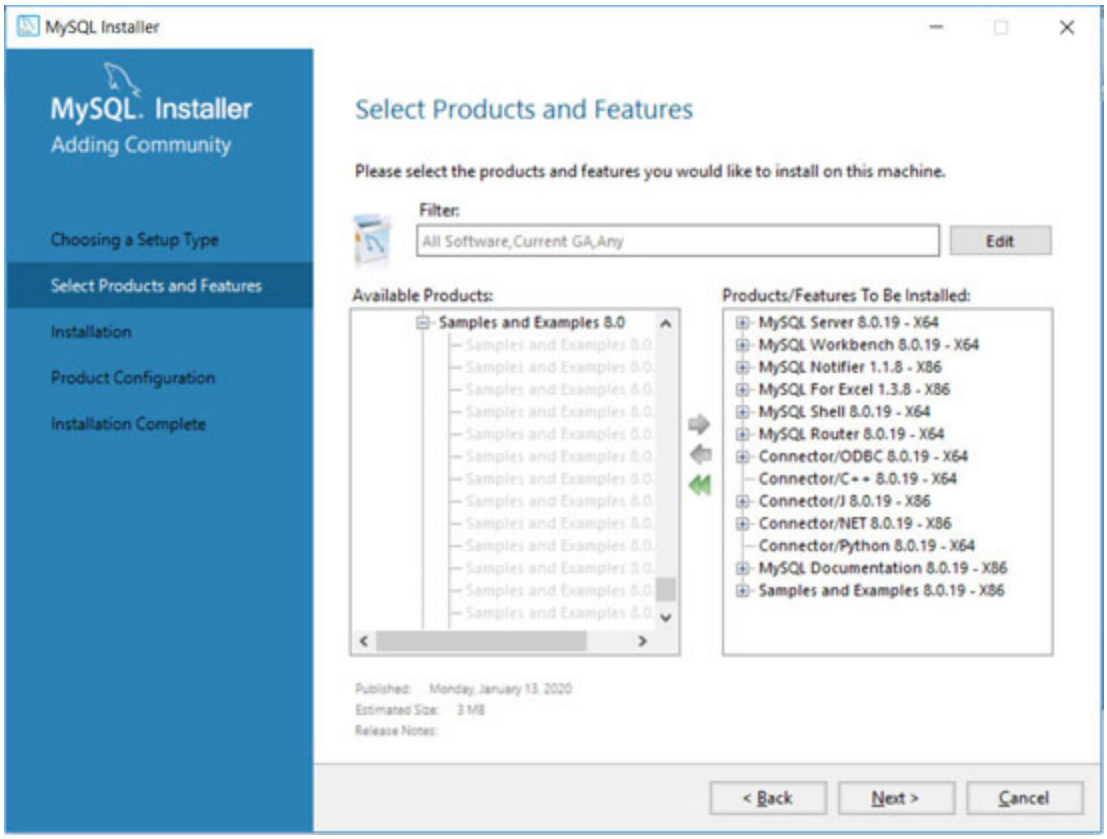


Figure 1.6

In the next window, it will show us the list of selected components:

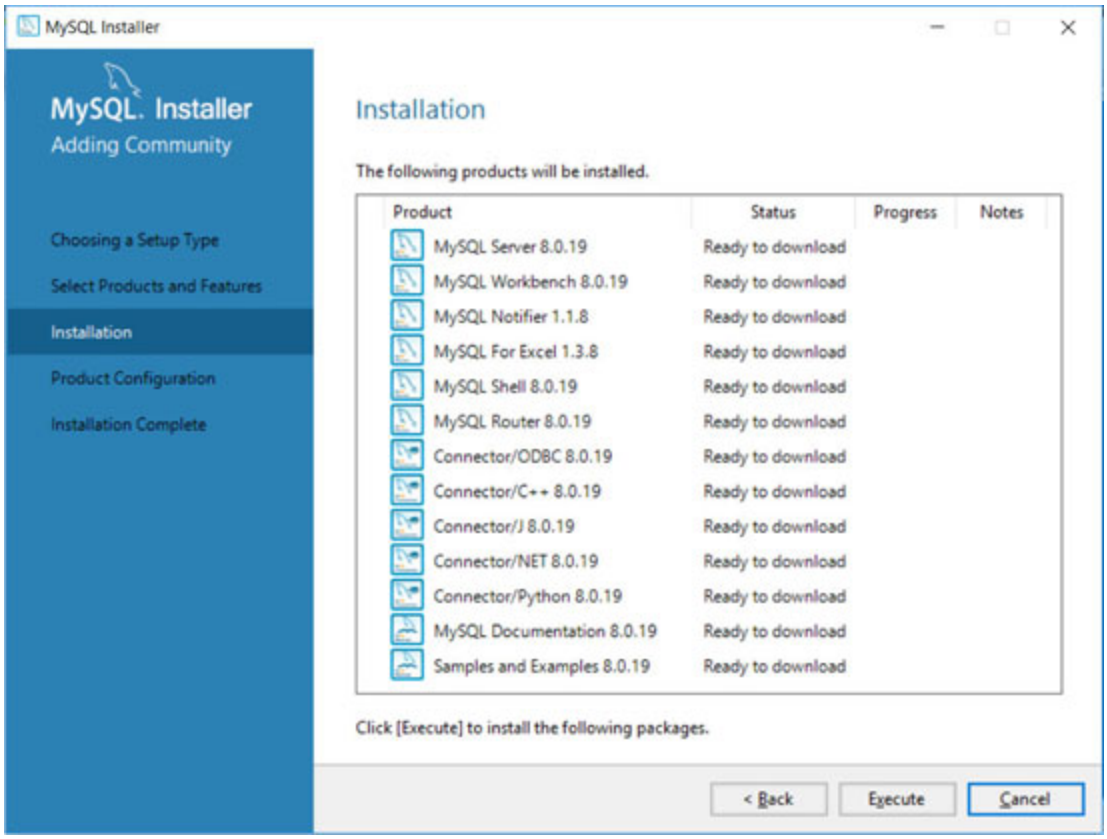


Figure 1.7

Verify the list. In case we miss anything, we can go to the earlier window and add the component that we want to install. After verifying the list, click on the **Execute** button. It will start the download and installation process, as shown in the following screenshot:

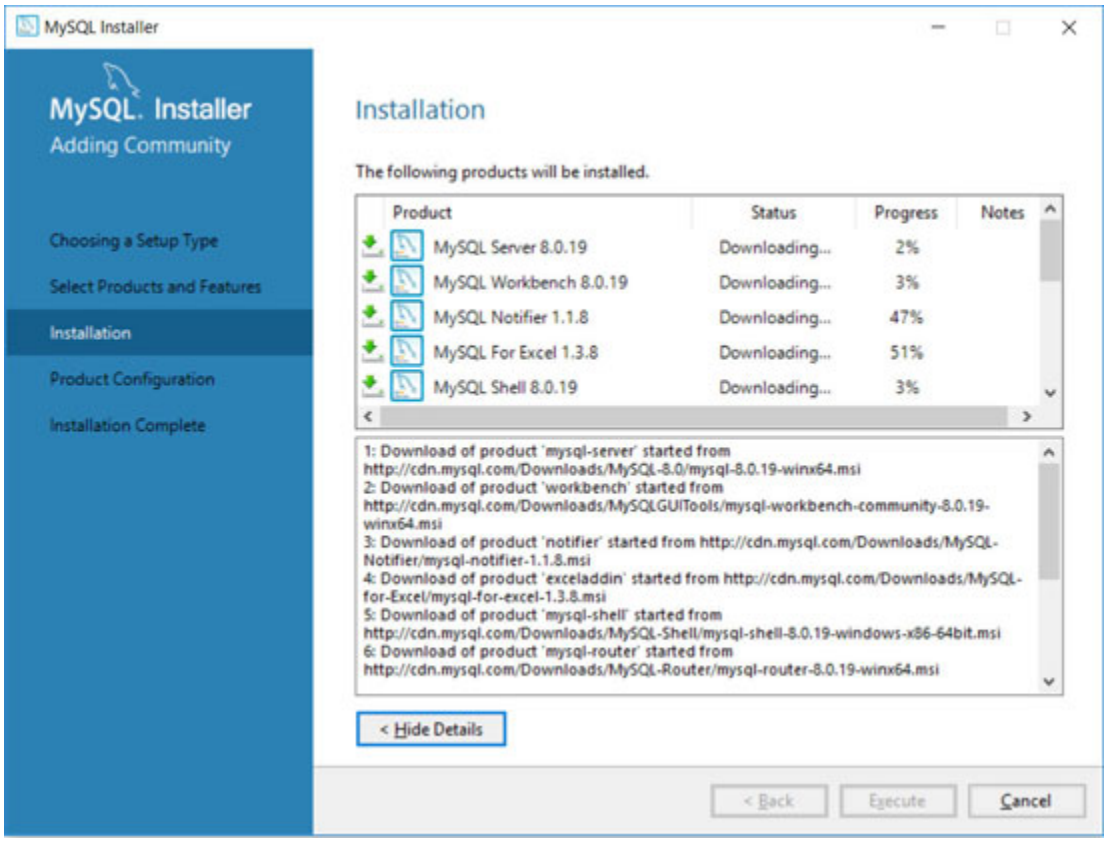


Figure 1.8

Note that I have clicked on the **`Show Details`** button. That is why it is showing us the installation log, as shown in the screenshot above. Depending on the speed of the internet, it will take some time to download the components. In case any download fails, it will show an option to try the download again for that component. Once the download and installation are completed, it will show us the **`Next`** button in place of the **`Execute`** button. Click on it, and it will show us the list of components ready to be configured as follows:

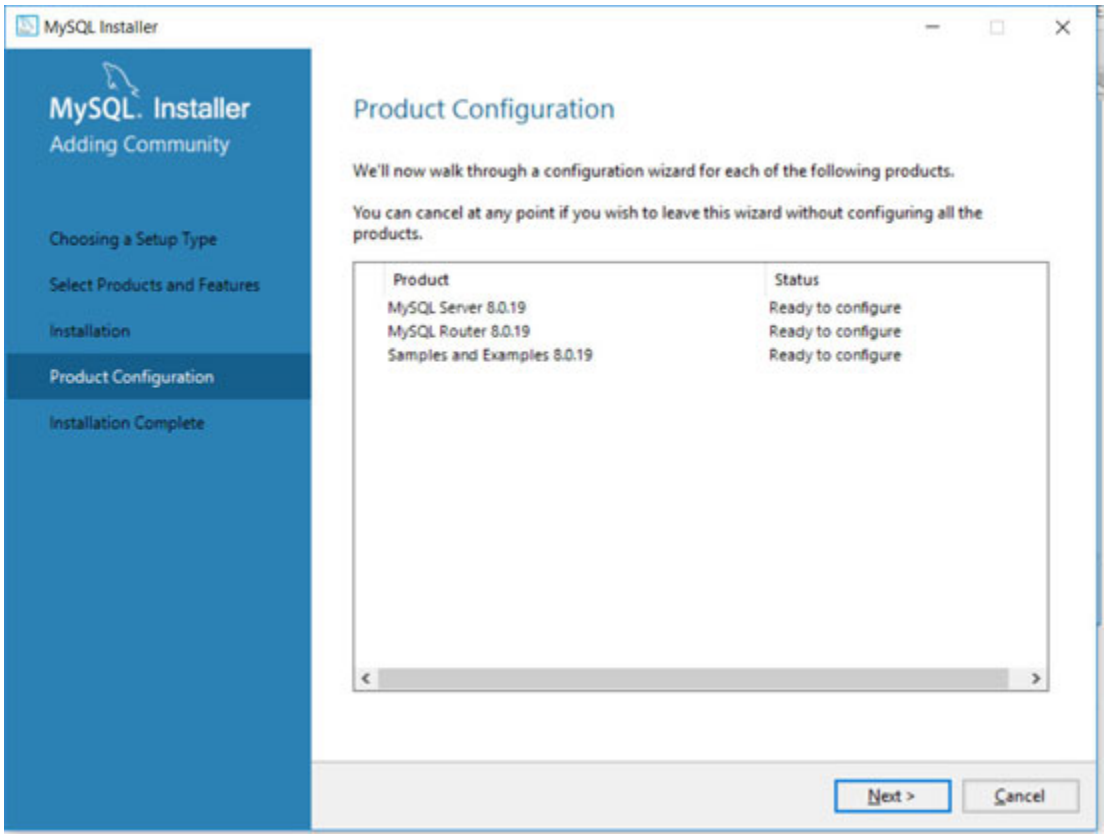


Figure 1.9

Clicking on the **`Next`** button will launch the configuration wizard for the MySQL Server component. MySQL database has many components. MySQL Server is the component that performs the tasks of an RDBMS, and it needs to be configured. Following is the screenshot of the first window of **`Configuration Wizard:`**

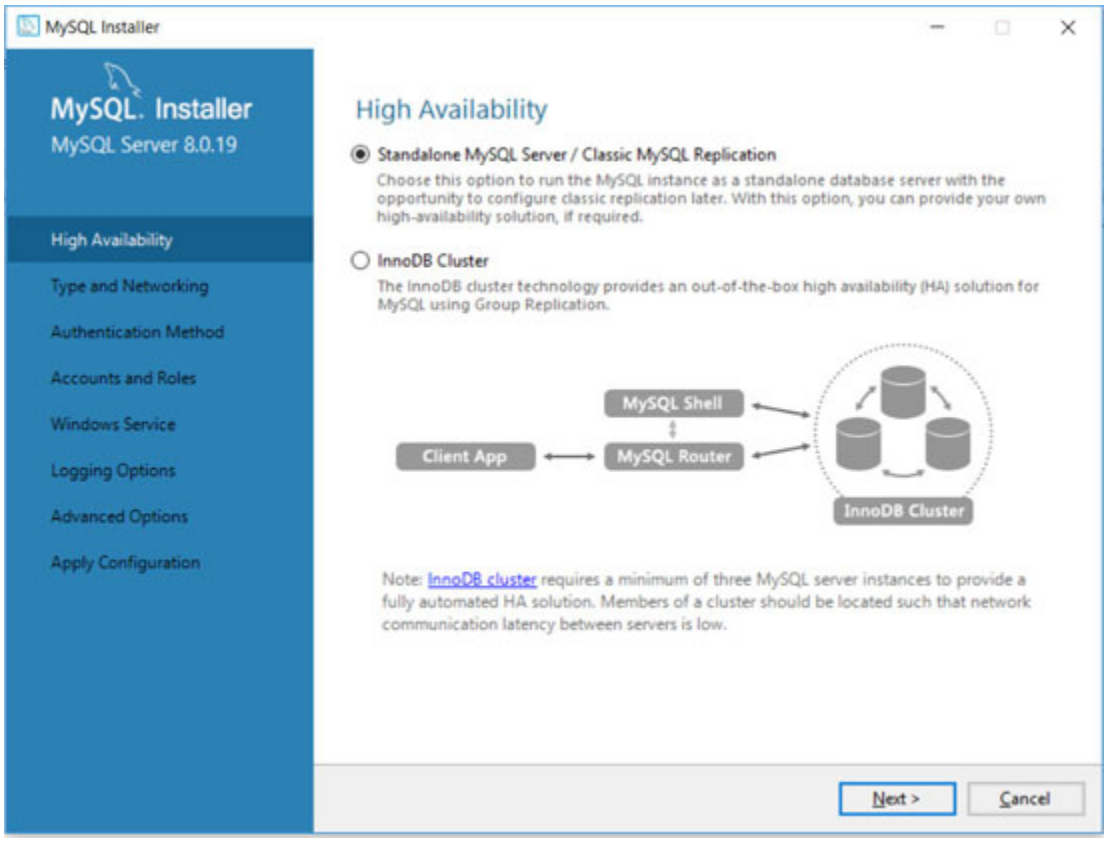


Figure 1.10

Choose the first radio button (Standalone MySQL Server/Classic MySQL Replication), as shown in the screenshot above, and click on the Next button. It will show us the configuration options as follows:

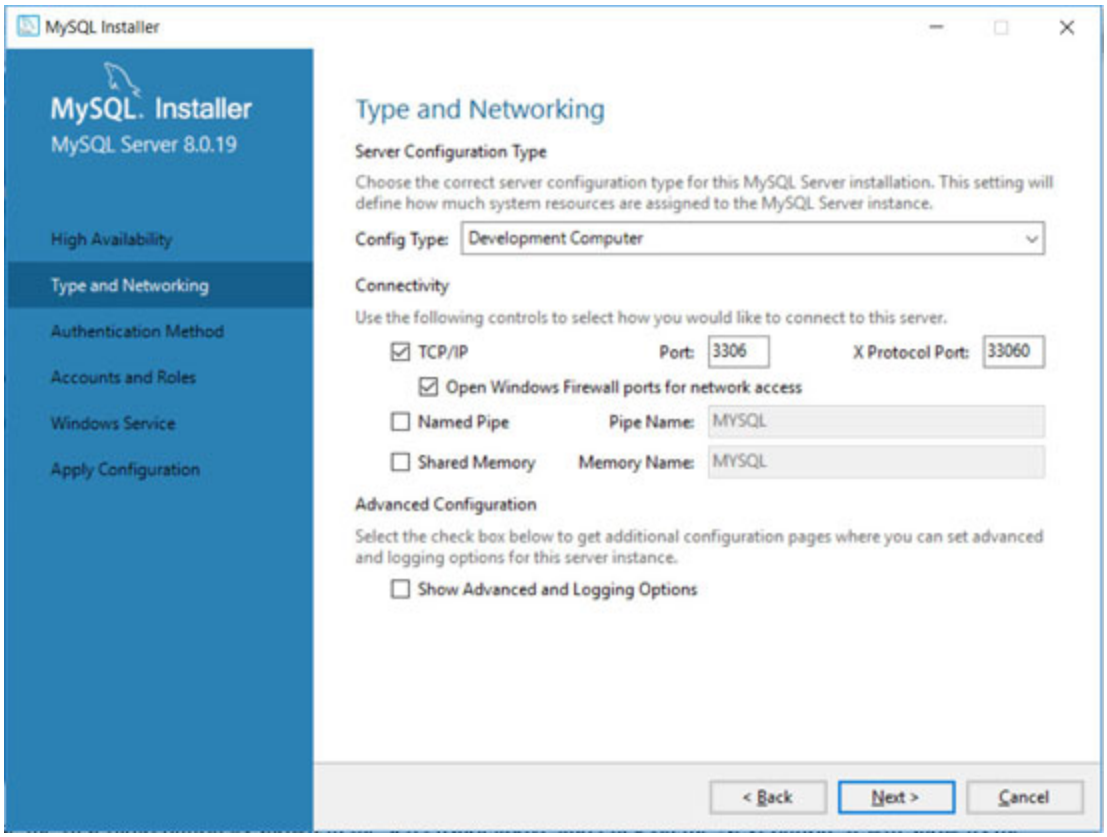


Figure 1.11

Keep all the options as they are and click on the **`Next`** button. It will take us to the following options:

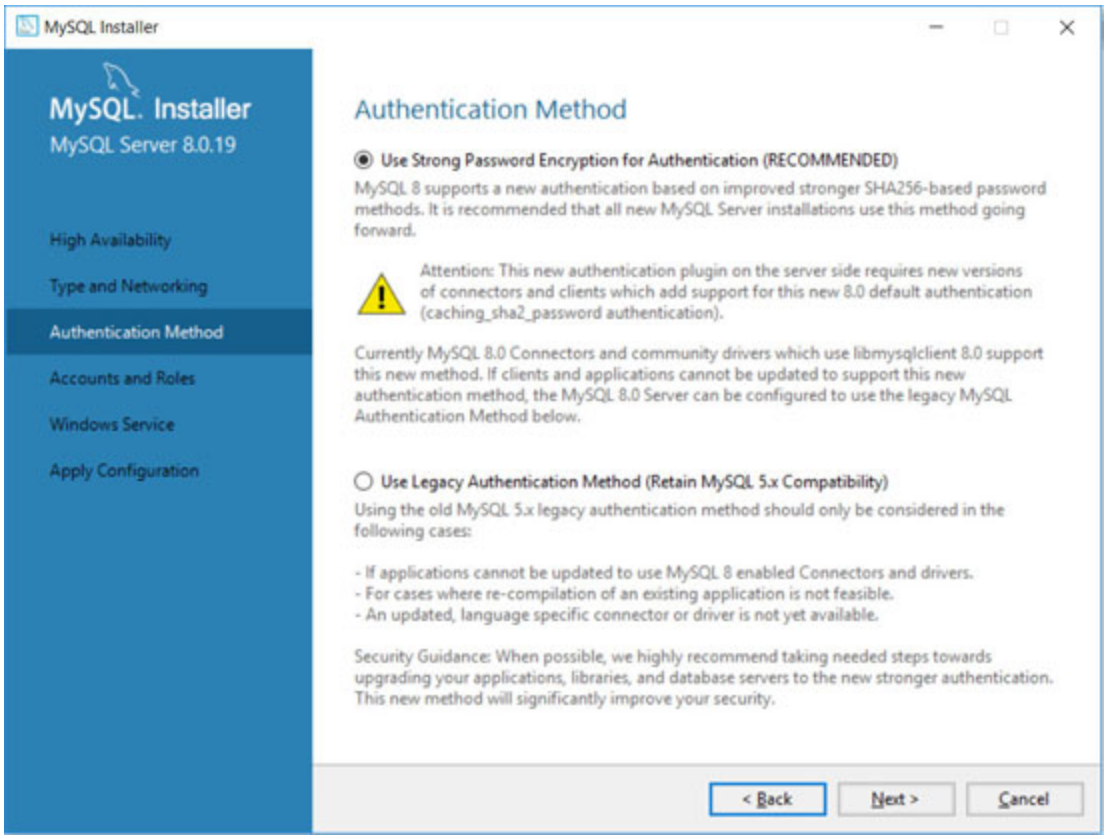


Figure 1.12

Choose the first radio button, as shown in the screenshot above, and then click the **`Next`** button. It takes us to the accounts and roles configuration as follows:

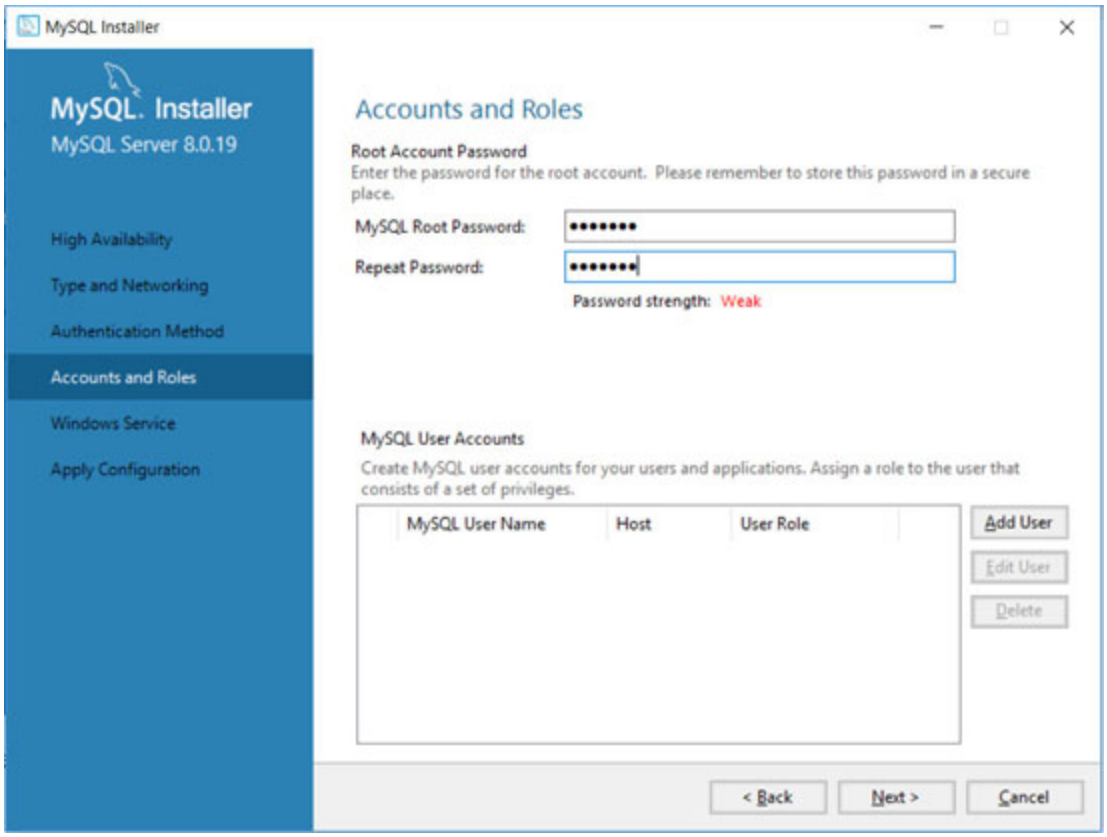


Figure 1.13

Enter any password of your choice for the root user of MySQL. Since this is a training database, I have chosen an easy password, **`test123`**. In the case of a production database, the password must be strong. Click on the **`Next`** button, and it will show the following options:

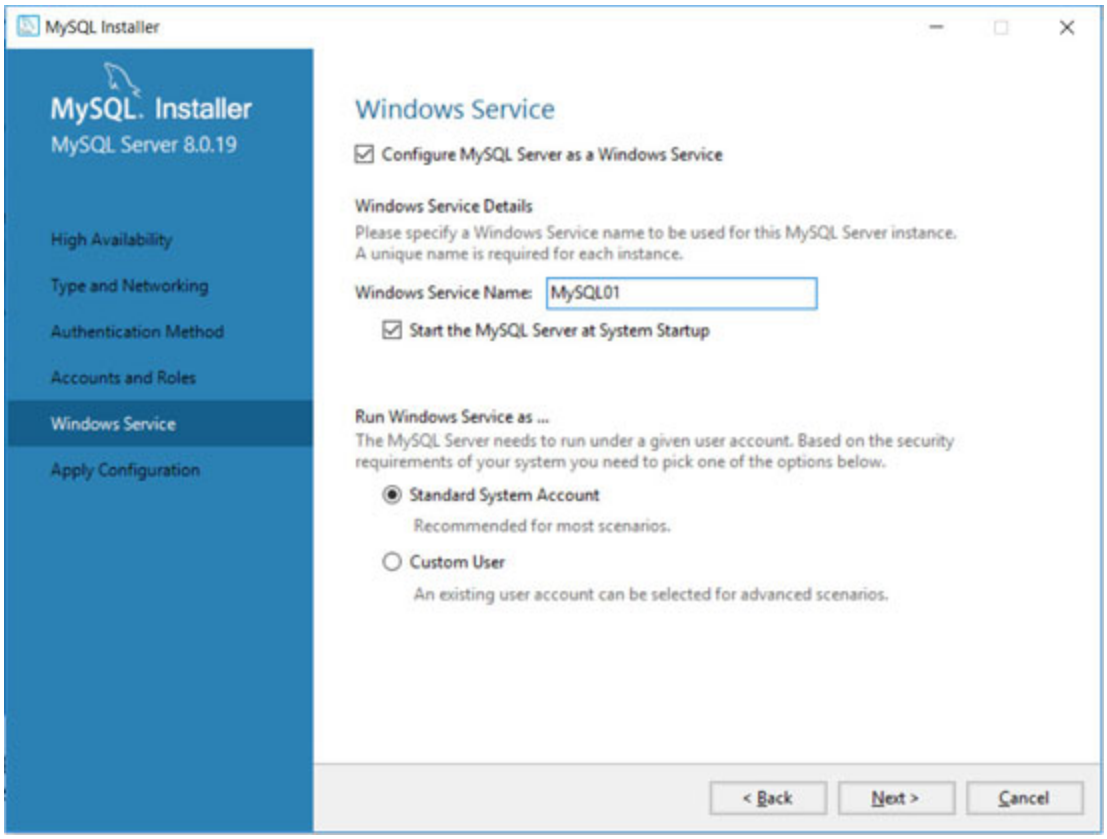


Figure 1.14

Keep all the options as they are. We can modify the string in the textbox Windows Service Name. Click on the **`Next`** button, and in the next window, click on the **`Execute`** button. It will start the service, and you will see a notification:

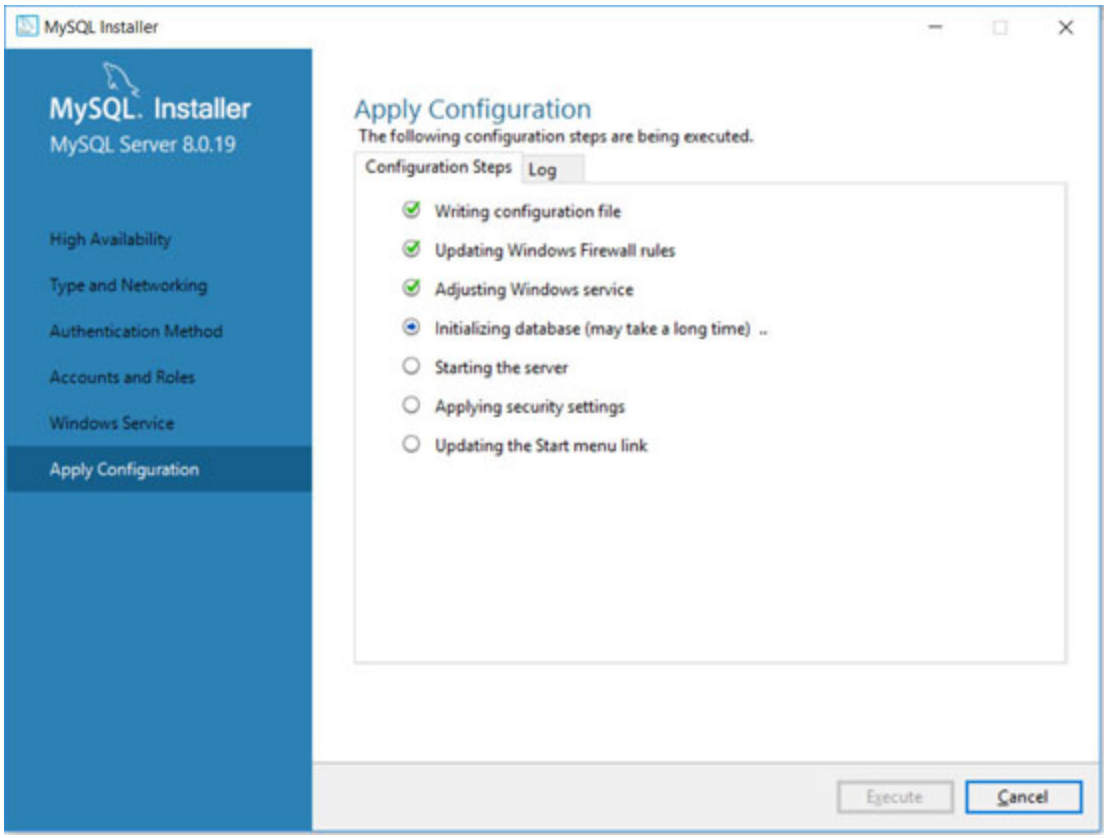


Figure 1.15

After finishing the setup, the **`Execute`** button will turn into the **`Finish`** button. Click on it, and it will take us to the router configuration. We do not need it as of now, so we can just click on the **`Finish`** button. Finally, we need to configure samples and examples as follows:

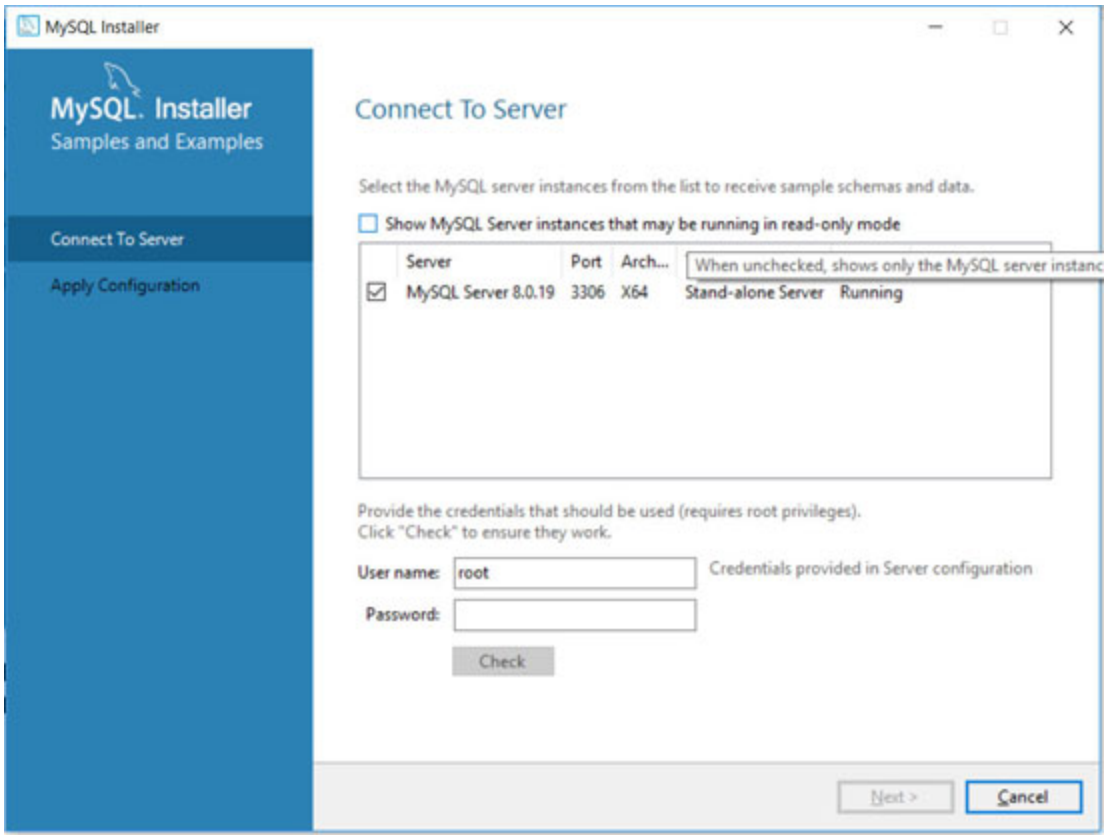


Figure 1.16

Input the **`root`** password and click on the **`Check`** button. Once the check is completed successfully, the **`Next`** button will be enabled. Click on it, and it shows the following window:

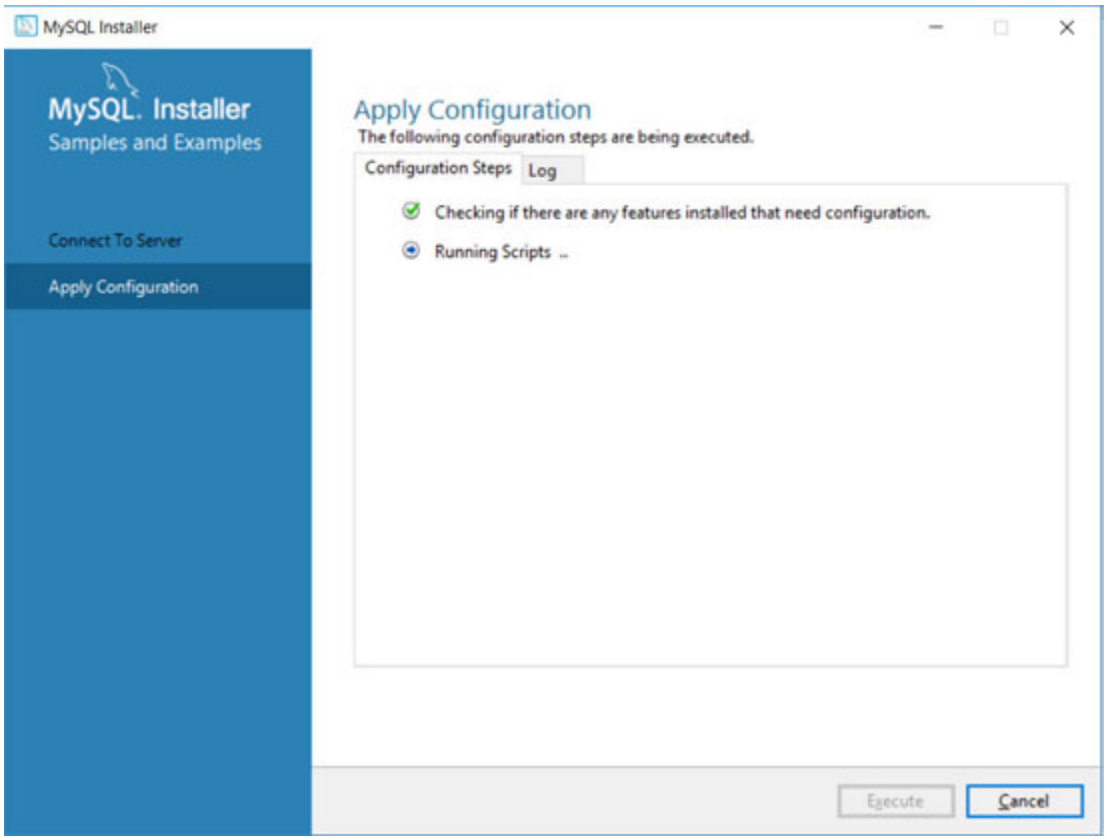


Figure 1.17

This concludes the installation of Windows. In case we need to reconfigure the installation or upgrade or install new components, then we must use MySQL Installer utility that is installed automatically with this setup. We can find it in the Windows search bar. Following is a screenshot of that:

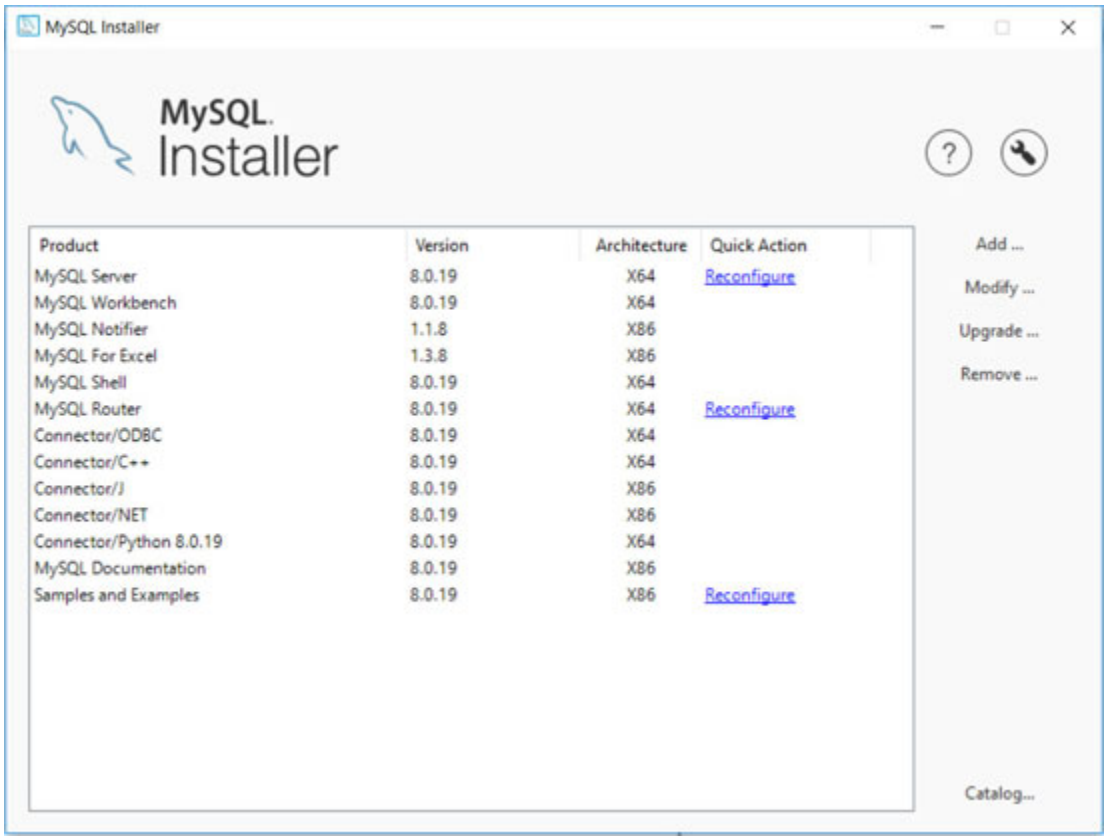


Figure 1.18

MySQL server process is automatically started once we start our Windows computer.

MariaDB

As explained earlier, MariaDB is a community-developed fork of MySQL. *Maria* is the name of the younger daughter of *Michael Widenius*. MariaDB is intended to be free and open-source drop-in alternative and replacement of MySQL, and it is under GNU General Public License. Just like MySQL, MariaDB is also used by government organizations and other big companies. MariaDB and MySQL are fully compatible with each other. All the SQL queries of MySQL are executed by MariaDB as it is. Even the data files where the physical data is stored in the Database are compatible. If we are using MySQL, we can just replace it with MariaDB with minimal changes and impact. All the external APIs and

tools that support MySQL also support MariaDB. For many Linux distribution MariaDB is the default database product. We can find more information about MariaDB at **https://mariadb.org/.**

Installation on Windows

Let us see how to install MariaDB on Windows. Remember that we have already installed a MySQL instance on Windows. Download the setup from **https://downloads.mariadb.org/.** Following is the screenshot of the download options:

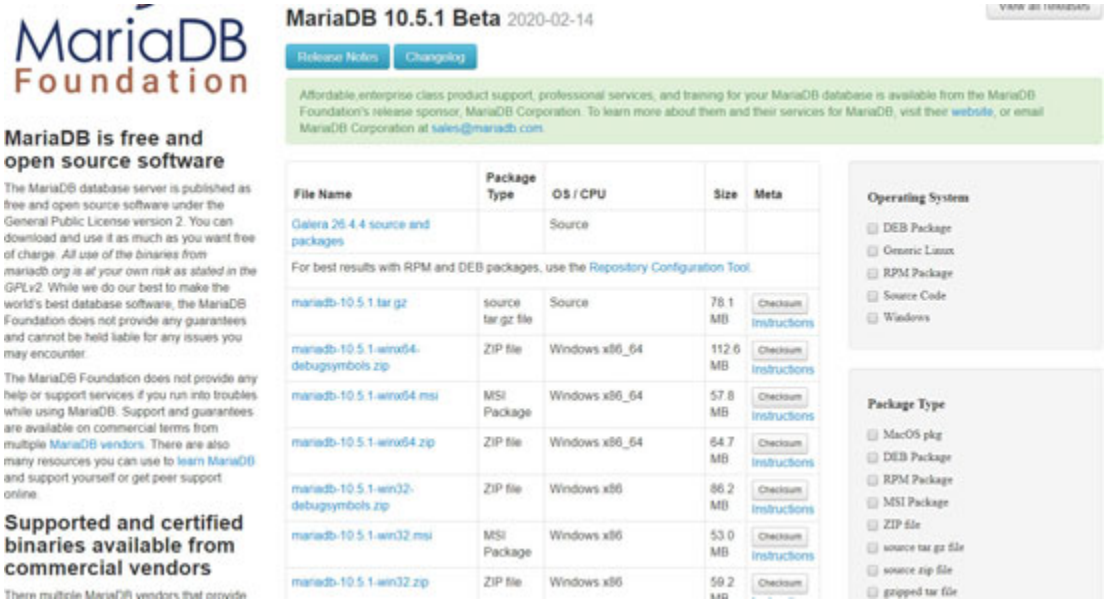


Figure 1.19

Download the appropriate file (a file with `.msi` extension for Windows) for your architecture (x64/win32). We can find it in the Downloads directory for our user. Double click to launch it. It requires admin privileges. The installation and configuration of MariaDB are much simpler than MySQL. Once we launch it, it shows the following window:

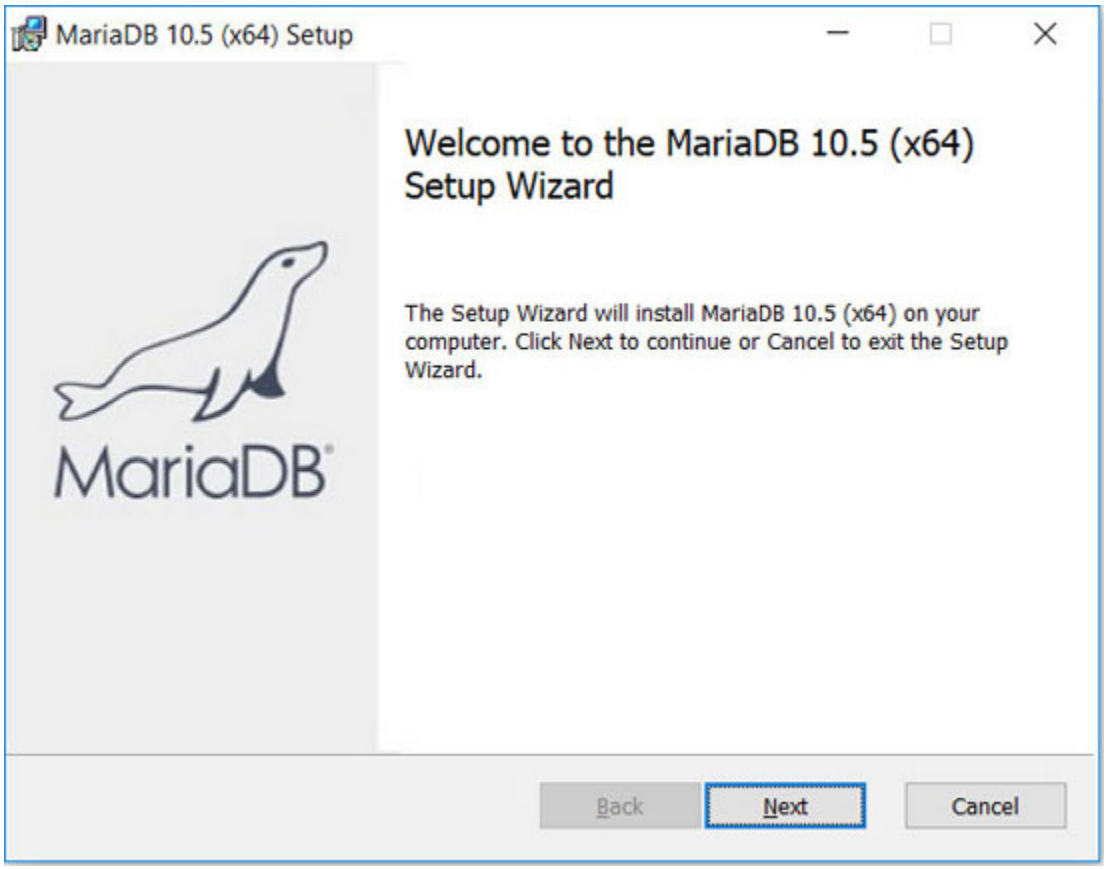


Figure 1.20

Click on the **`Next`** button, and it shows the following window:

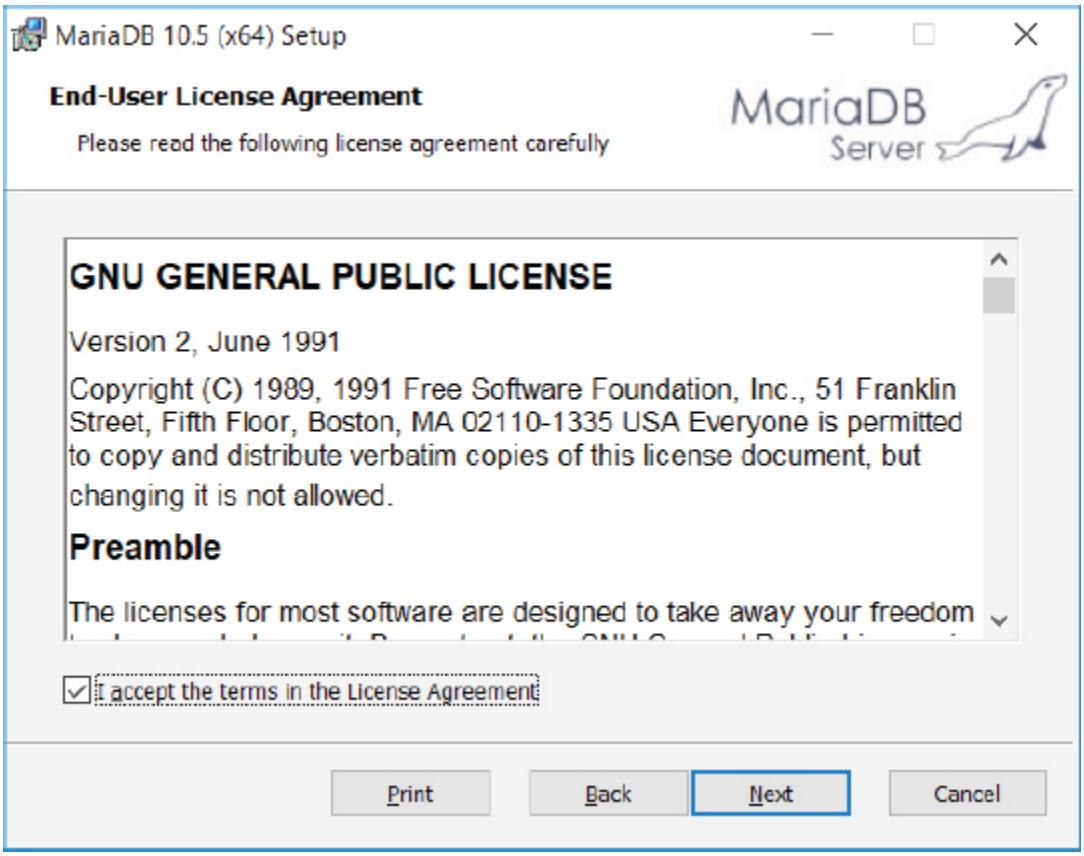


Figure 1.21

Accept the **`License Agreement`** and click on the **`Next`** button here too and it shows the following window:

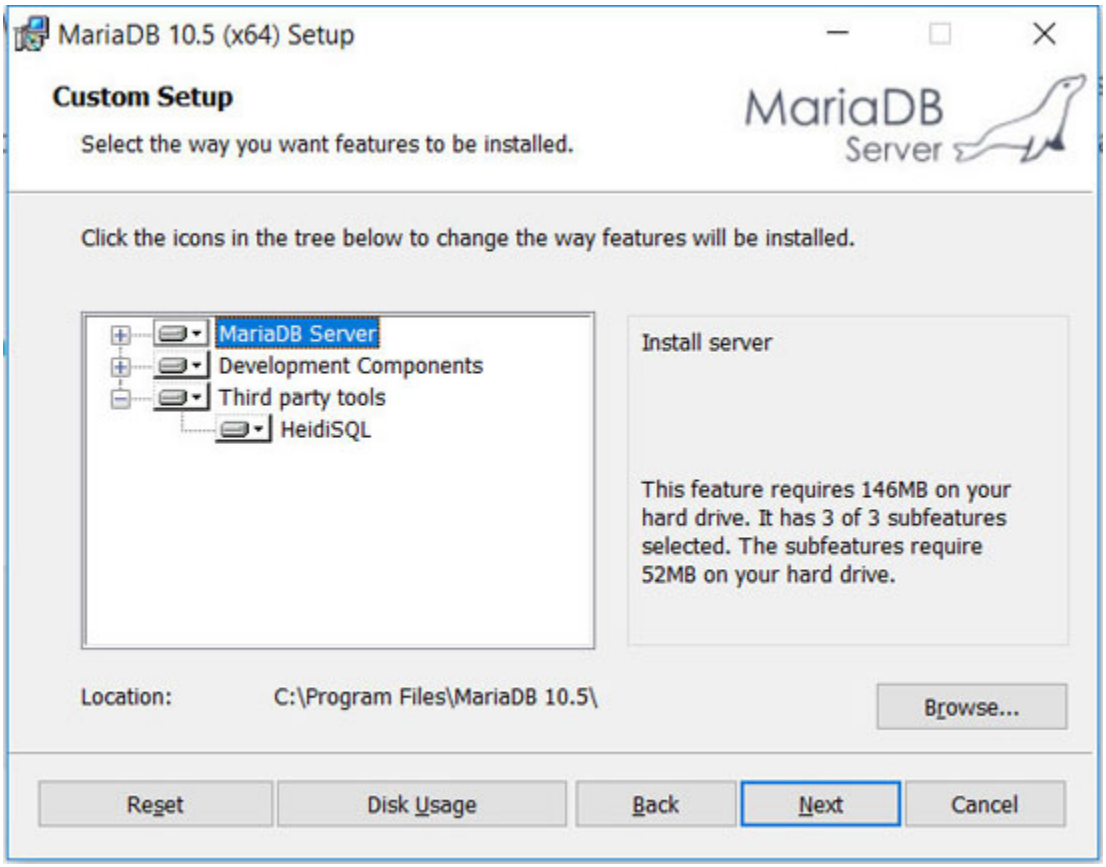


Figure 1.22

In the window above, we can choose the installation path. I recommend to keep the default options and then click on the **`Next`** button. It will show a window as follow:

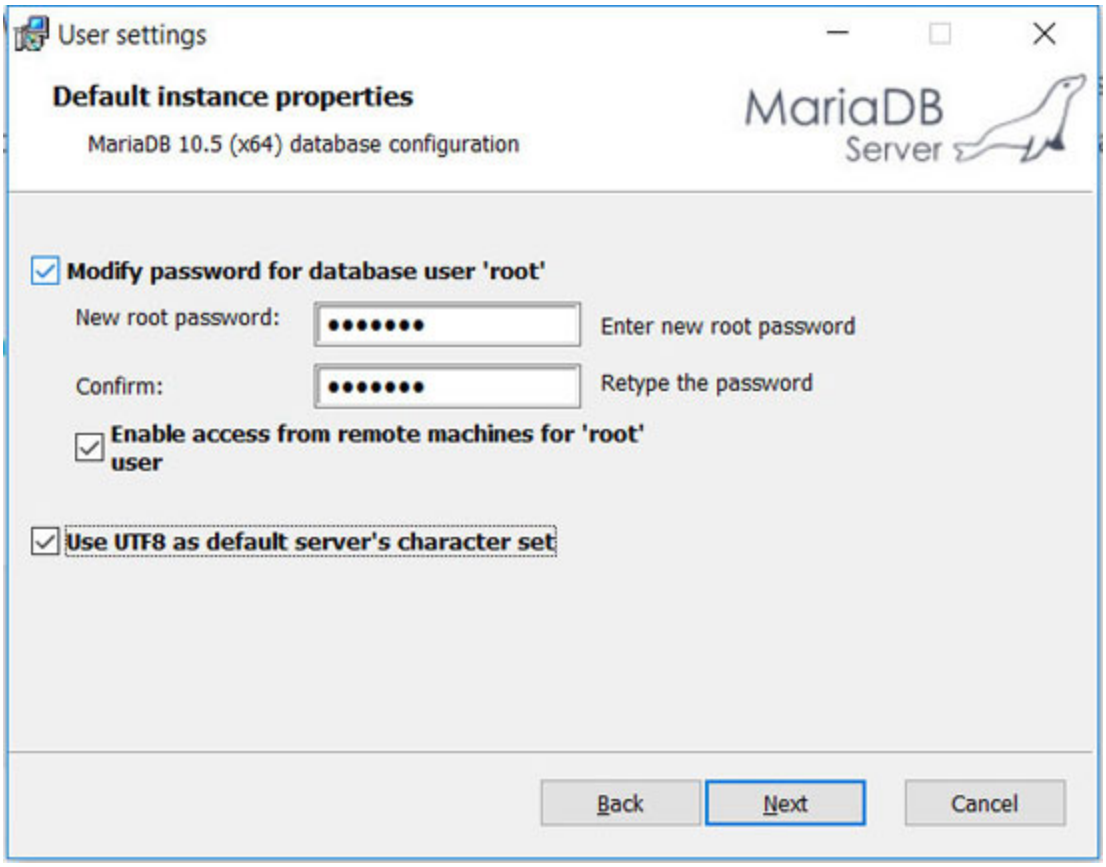


Figure 1.23

Here we can set the password for the root account. Check all the checkboxes and click on the **`Next`** button. It shows the following configuration window:

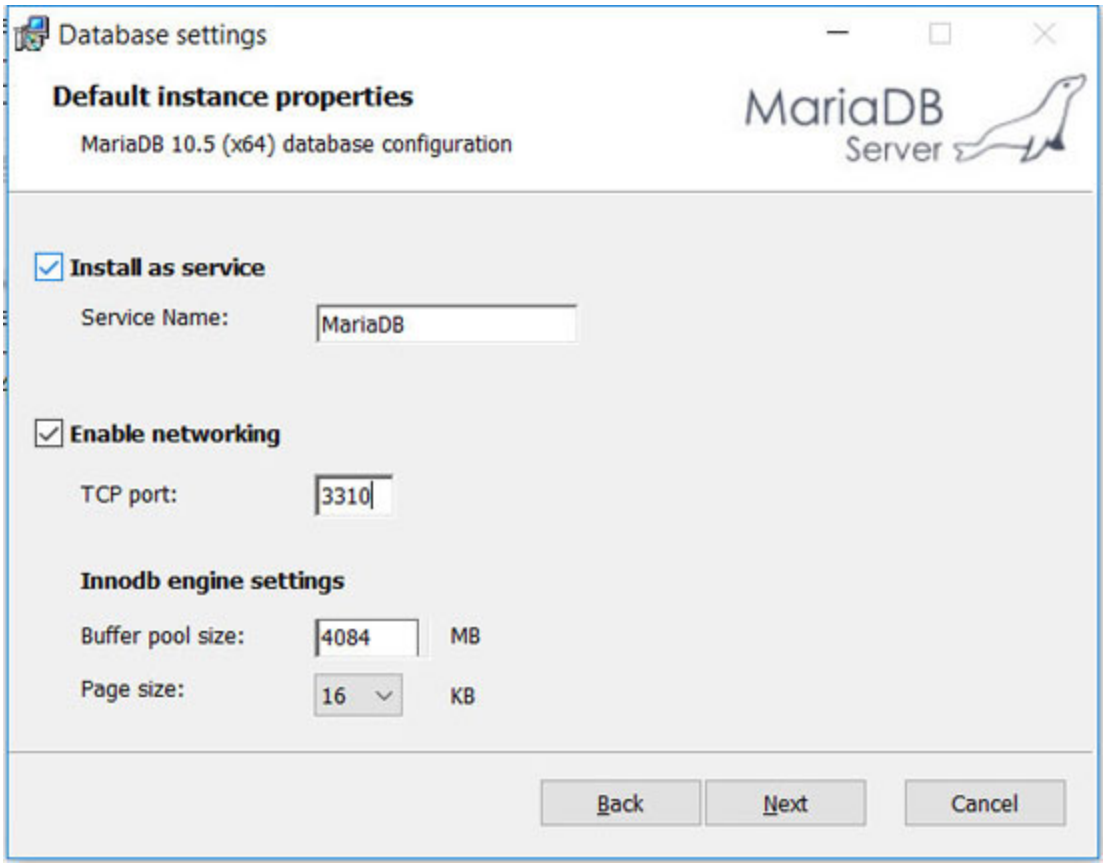


Figure 1.24

If you remember, we had assigned port number 3306 to MySQL Server for networking and remote connection. By default, MariaDB too uses the same port; just change it to the number 3310 to avoid the conflict. Keep the rest of the options as they are and click on the **`Next`** button.

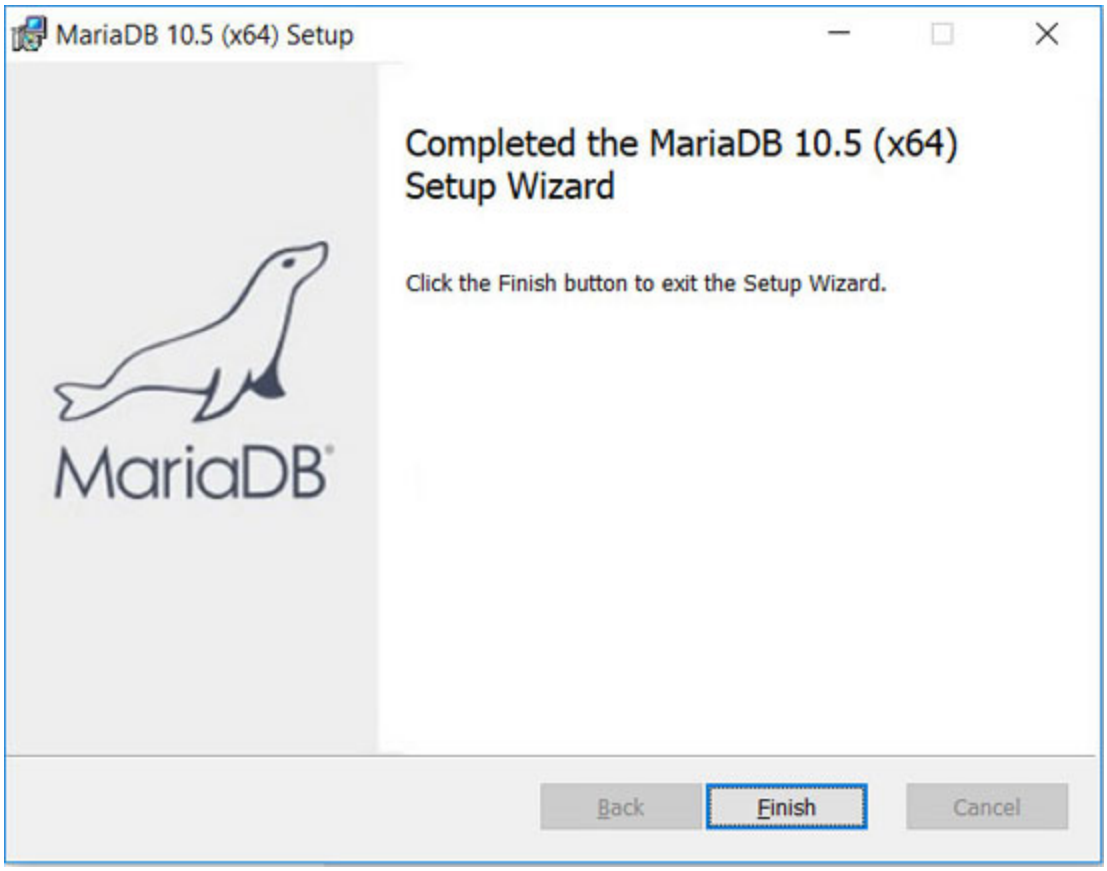


Figure 1.25

Click the **`Finish`** button to finish the installation process. MariaDB Server and other utilities like HeidiSQL are installed. Whenever the computer boots up, the MariaDB server process automatically starts up.

Installation on Debian and Raspberry Pi Raspbian Linux

We can install MariaDB on Debian and Raspbian. I am using a Raspberry Pi running on Raspbian. MariaDB is the default MySQL server. Open the command prompt of Raspbian OS and run the following commands in sequence:

```
sudo apt-get update
sudo apt-get install mariadb-server --fix-missing -y
sudo apt-get install mariadb-client -y
```

These will install MariaDB Server and MySQL Client packages on the Raspbian/Debian. After installation is done, we can run the command **`sudo mysql_secure_installation`** to configure the MariaDB server. We can set the password for the root account and answer no for all the other options.

Conclusion

In this chapter, we have learned the basic concepts required to get started. We learned about MySQL and MariaDB projects. We also installed MySQL and MariaDB servers and clients on Windows and Debian/Raspbian Linux. We also got started with hands-on with SQL with w3schools Tryit online SQL Editor.

In the next chapter, we will get started with the installation of the sample databases on MySQL and MariaDB on Windows and Raspbian OS. We will also see how to create another user for the MySQL and MariaDB databases.

Points to Remember

- MariaDB is the default MySQL database environment that comes with many Linux flavors.
- MariaDB and MySQL are highly compatible with each other in terms of SQL Syntax, interfaces, and tools.

MCQ

1. Which of the following databases use SQL?
   a) Hierarchical Databases
   b) Network Databases
   c) Relational Databases
   d) Non-relational Databases

Answers to MCQ

1. *a)*

Questions

1. What is a database?
2. What is a relational database?
3. Name a few tools like MySQL workbench, which are used for querying MySQL and MariaDB.

Key Terms

MariaDB, MySQL, Installation, MySQL Workbench, HeidiSQL, SQL, Database, Relational Database, DBMS, RDBMS, ORDBMS

CHAPTER 2

Getting Started with MySQL

In the last chapter, we learned a few basic concepts related to database and DBMS. We had hands-on with basic SQL queries with Tryit Online SQL Editor. We learned the details and history of MySQL and MariaDB projects. We also learned how to install MySQL and MariaDB on Windows and Debian/Raspbian platforms. In this chapter, we will learn and demonstrate the basics of MySQL. We will not be getting started with the full-fledged SQL queries in this chapter. However, we will learn how to connect to MySQL, how to use the command line and graphical utilities, how to install sample databases, and how to create users. All these topics are needed before we dive deeper into the syntax of SQL.

Structure

In this chapter, we will learn the following topics:

- Connecting to the MySQL and the MariaDB Servers using the different ways
- Basic SQL Queries and installation of sample databases

Objective

The main objective of this chapter is to prepare the MariaDB and the MySQL server instances we installed on Windows and Debian/Raspbian platforms for the demonstrations. Following the instructions in this chapter will prepare readers and the setup they have for running the sample queries and exercises in this and the further chapters.

Connecting to MySQL and MariaDB Server Instances

As we have seen earlier, MySQL and MariaDB have server components, and these services get launched automatically whenever we boot up/restart the machine.

These server processes take care of the organization of the database, physical data files, and other administrative tasks. To work with the database and run SQL queries, we need to connect to these server processes using many tools available. In this section, we will learn how to connect to a server instance of MariaDB or MySQL with the tools that come by default at the time of installation.

Connection with MySQL Workbench

MySQL comes with a graphical tool known as MySQL Workbench that can be used to connect with any MySQL or MariaDB instance. We can use MySQL Workbench for running the queries and basic administrative tasks. We can find it in the results of Windows Search Menu by searching for MySQL as shown in the screenshot below:

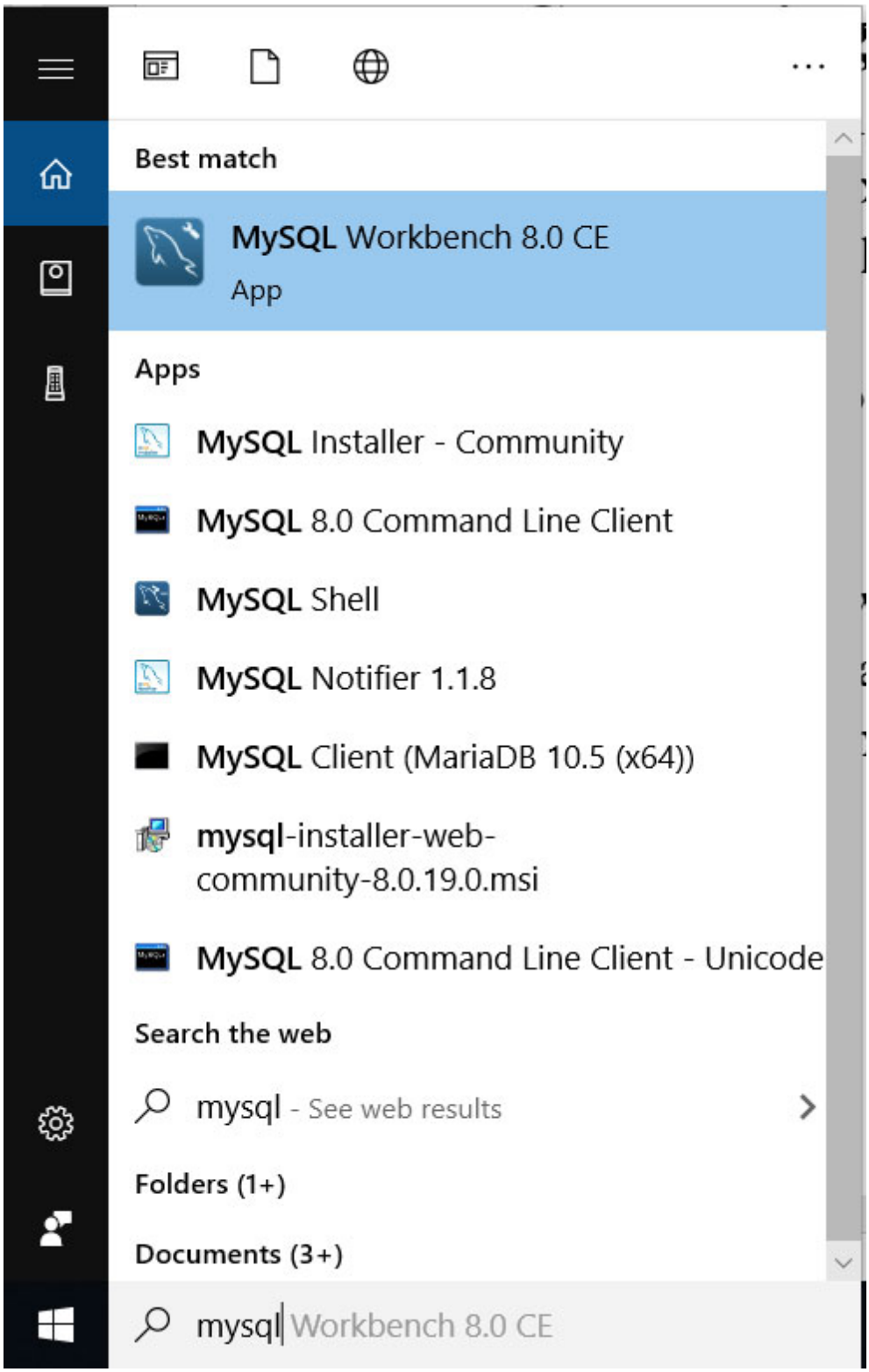

Best match
MySQL Workbench 8.0 CE
App
Apps
MySQL Installer - Community
MySQL 8.0 Command Line Client
MySQL Shell
MySQL Notifier 1.1.8
MySQL Client (MariaDB 10.5 (x64))
mysql-installer-web-community-8.0.19.0.msi
MySQL 8.0 Command Line Client - Unicode
Search the web
mysql - See web results
Folders (1+)
Documents (3+)
mysql Workbench 8.0 CE

Figure 2.1

Click that option to launch **`MySQL Workbench`**. Following is the screenshot of the MySQL workbench window:

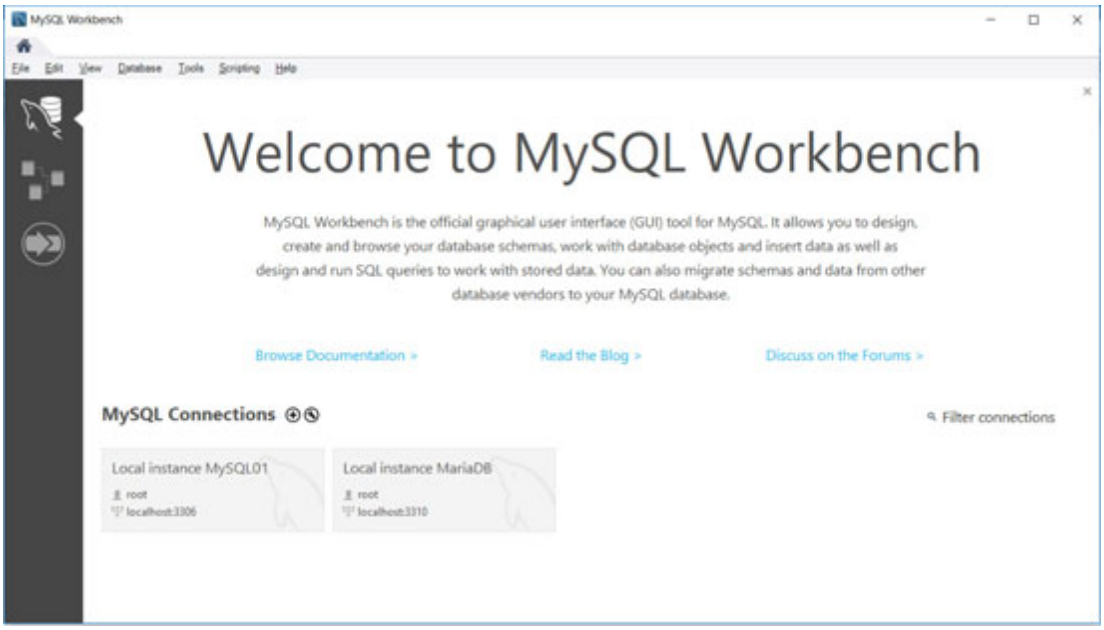


Figure 2.2

As we can see in the screenshot above, we already have two connections available. This is because the MySQL Workbench detected instances of MySQL and MariaDB server processes running locally (on the same computer). It created the connection for the root accounts of both the server instances found running on the computer. Let us connect to the MySQL server instance. Double click the connection corresponding to MySQL server instance (in this case, it is MySQL01; we had set this name during installation). It will show a dialogue box as follows:

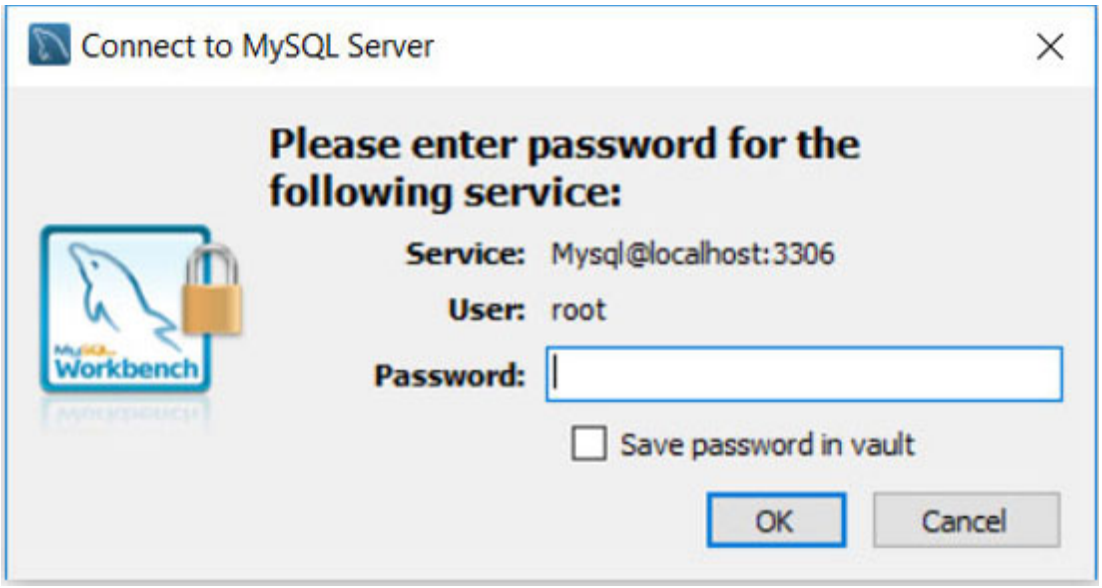


Figure 2.3

Enter the password and check the checkbox if you want to login without a password from now onwards. Once done, the following window opens.

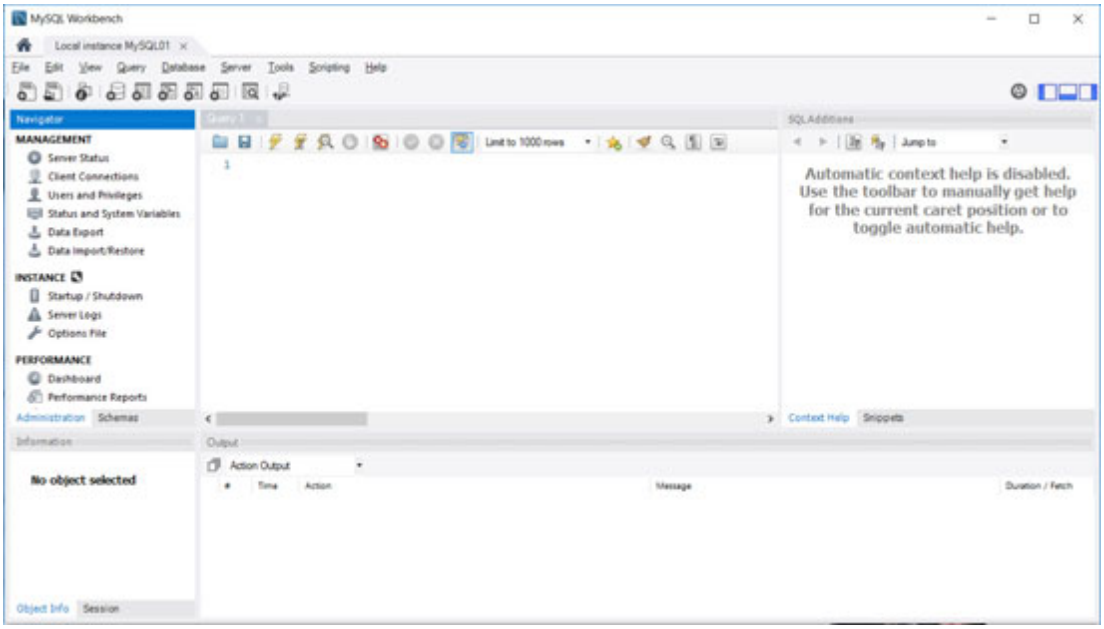


Figure 2.4

As we can see, the window is divided into various tabs. In the navigator tab, we have two options, **`Administration`** and **`Schemas`**. We will not be using the Administration tab for the

book as it is used for Database Administration, which is a separate topic that requires dedicated books. We are more interested in schemas. So click on the option for schemas and the navigator tab will as follows:

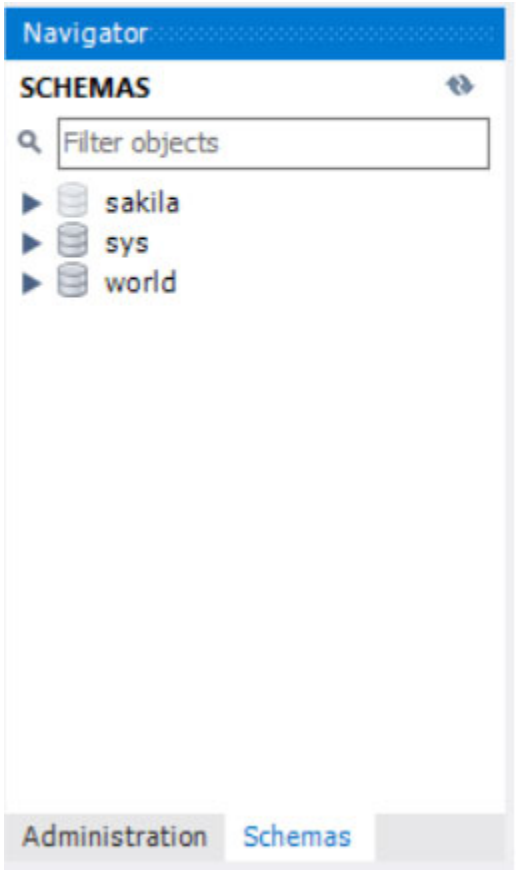


Figure 2.5

In MySQL (and MariaDB, too), the terms schema and database are used interchangeably. If you have worked with the other databases before this, a few databases like Oracle Database treat a schema as a user and database are different from that. In MySQL (and MariaDB), a server instance can have various databases/schemas, and users are not tied to a single schema alone. We will learn all these things in detail further down the road.

Now, let us explore the **`Query`** tab. Following is the partial screenshot of it:

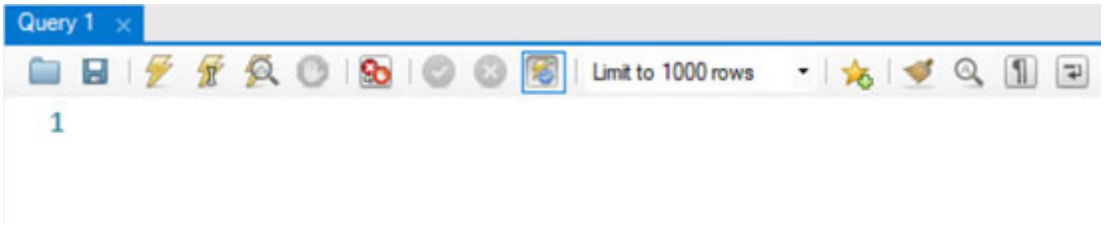


Figure 2.6

We can see a series of icons in the top section. The first icon is to open and load a saved SQL script for execution. We can save our SQL queries as scripts on the disk using the save icon (Floppy Disk). The adjacent lightning symbol is to run the selected portion of the script or everything if there is no selection. Then next lightning symbol with a cursor sign executes the query under the current cursor. We can adjust the font of the query text by pressing *Ctrl* and + or – keys simultaneously. The tab just below the **`Query`** tab, we have the output tab where we can see the output of the query.

Connecting with HeidiSQL

MariaDB comes with a similar GUI tool. It is known as HeidiSQL. We can create a new connection as follows:

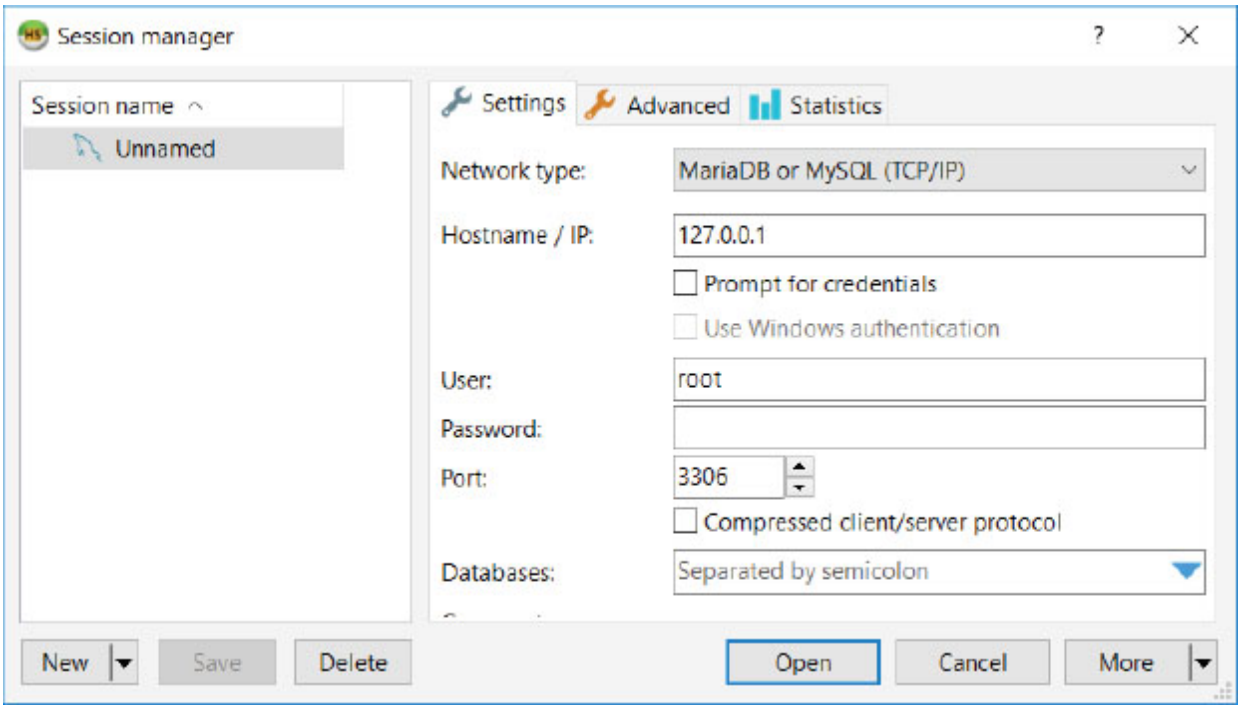


Figure 2.7

Provide the details relevant to an instance of MariaDB or MySQL servers, and it will connect. Try to explore HeidiSQL yourselves. Following is a screenshot of a GUI of HeidiSQL:

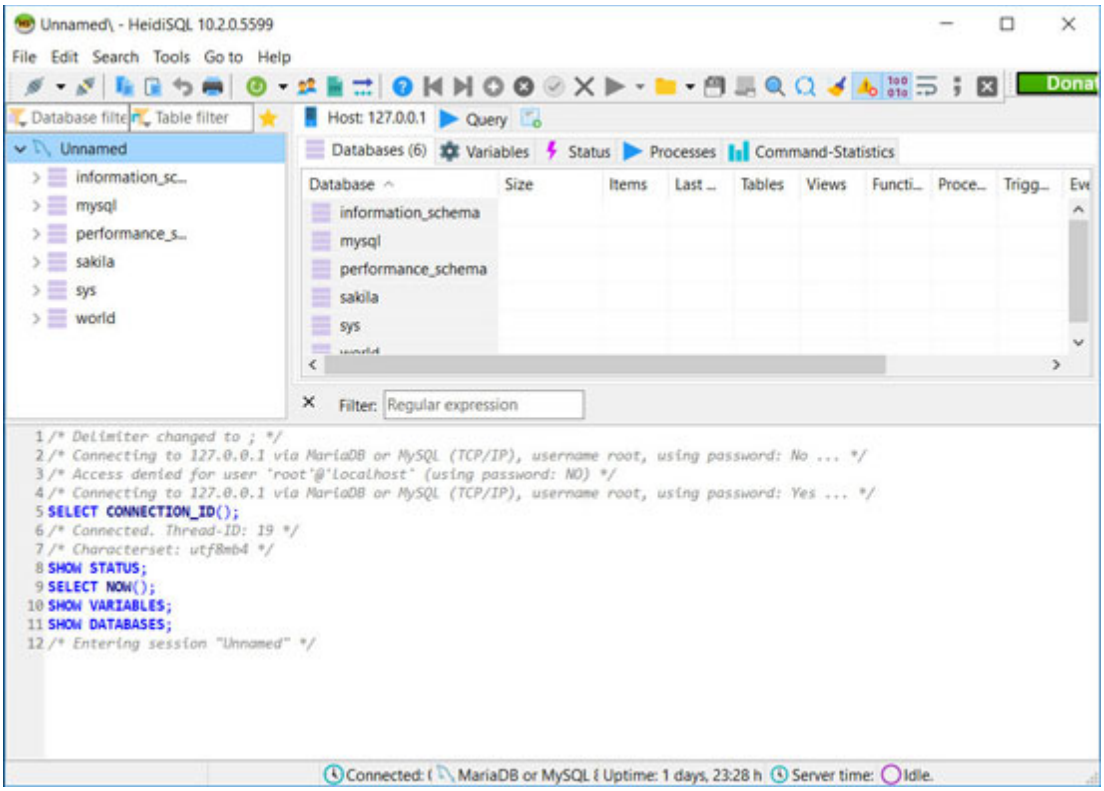


Figure 2.8

Connecting with Command Prompt

We can connect to an instance with the Windows command prompt. Just run the following command on cmd to connect to an instance of MariaDB:

```
mysql -u root -ptest123 -h localhost -P 3310
```

To connect to an instance of MySQL:

```
mysql -u root -ptest123 -h localhost -P 3306
```

There is no space between `-p` and the password. This is how the connection with command prompt looks similar to the following screenshot:

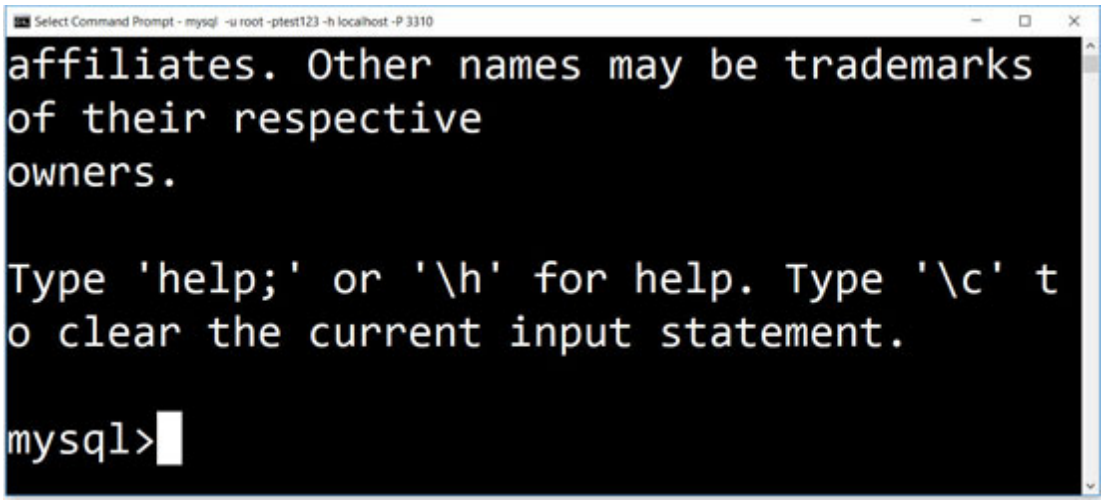


Figure 2.9

NOTE: If the command fails with 'mysql' not found, please make sure to check the PATH environment variable has the path 'C:\Program Files\MySQL\MySQL Server 8.0\bin' added.

Similarly, we can run the following command on lxterminal of Debian / Raspbian to connect to a local instance of MariaDB:

```
sudo mysql -u root -ptest123 -h localhost -P 3310
```

sudo is only needed when we are connecting to the root user. Otherwise, we do not need it.

Connecting to a Remote Instance

We have already seen how to connect with local instances of MariaDB and MySQL servers. We can even connect to the remote instances with the GUI and the command prompts. All we need is a username, password, hostname/IP address, and port number. We can use these details with the command prompts or the GUIs to connect with a remote server process of MariaDB or MySQL.

A few Basic Queries

We can create a new user with the username of '`testuser`' and the password '`test123`', with the following query

:

```
CREATE USER 'testuser'@'localhost' IDENTIFIED BY 'test123';
```

The following query grants all the privileges on all the databases/schemas to the newly created user '**testuser':**

```
GRANT ALL PRIVILEGES ON *.* TO 'testuser'@'localhost';
```

We can run these queries with any interface (GUI or command prompt), and the result will be the same;

If we are using Debian / Raspbian, we can log in to our database without using sudo as follows:

```
mysql -u testuser -ptest123 -h localhost -P 3310
```

Now, to see what sample schemas are installed already, run the following query:

```
SHOW DATABASES;
```

The result will show us a few systems and a few sample databases. MySQL comes with a sample database sakila. We need to install this on MariaDB.

Download the zip file of the database from **https://github.com/datacharmer/test_db/archive/master.zip** and then unzip it. Then browse to the directory named *sakila* in the extracted directory. There we will find the scripts for creating a schema and loading the schema with the data. The names of those scripts commence with the string sakila. Run the following commands on the command prompt of windows:

```
mysql -u root -ptest123 -h localhost -P 3310 < sakila-mv-schema.sql
mysql -u root -ptest123 -h localhost -P 3310 < sakila-mv-data.sql
```

Run the following commands on the Raspbian / Debian command prompt:

```
sudo mysql -u root -ptest123 -h localhost -P 3310 < sakila-mv-schema.sql
sudo mysql -u root -ptest123 -h localhost -P 3310 < sakila-mv-data.sql
```

We also need to install another sample database, employees, on MySQL and MariaDB both. We can find the scripts for installation in the parent directory of the directory sakila in the same extracted location. The names of those scripts commence with the string employees. Run the following commands in the command prompt of windows after navigating to that directory:

```
mysql -u root -ptest123 -h localhost -P 3310 < employees.sql
```

Run the following command on the command prompt of Debian / Raspbian after navigating to the directory where the scripts for the *employees*' database are stored:

```
sudo mysql -u root -ptest123 -h localhost -P 3310 < employees.sql
```

Once all the installation completes, run the query `SHOW DATABASES` for every connection to verify that the databases sakila and employees show in the output of the query. Anyway, in the GUI tools, we can see them in the schema browser.

Exercise

As an exercise to this section, find out how to enable remote login for the MariaDB installed on the Debian / Raspbian distribution of Linux.

Conclusion

In this chapter, we have learned to connect to the database server instances with various GUI tools like MySQL Workbench and HeidiSQL. We also learned how to connect with command prompts like lxterminal and cmd. Then we installed the sample databases. In that process, we have learned how to run SQL scripts with command prompt. In all these demonstrations, we saw how to write a few SQL queries to perform certain tasks with databases. With these topics covered, we can start learning the SQL with MySQL and MariaDB from the next chapter onwards.

Points to Remember

- Database installations do not come with a lot of sample schemas. We must install them ourselves.
- The GUI tools like HeidiSQL, Oracle SQL Developer, and MySQL Workbench provide easy to use interface for MySQL and MariaDB.

MCQ

1. Which one of the following is the correct command to display a list of all the database schemas installed on the current instance?

   a) DISPLAY DATABASES

   b) LIST DATABASES

   c) VISUALIZE DATABASES

   d) SHOW DATABASES

Answer to MCQ

1. *d)*

Questions

1. What are the other free and open-source alternatives for MySQL Workbench and HeidiSQL?

Key Terms

Connection, Command Prompt, HeidiSQL, MySQL Workbench, Sample Databases, employees, sakila

CHAPTER 3

Getting Started with SQL Queries

In the last chapter, we learned how to connect to the MySQL and the MariaDB server instances with command prompt and GUI tools to run SQLs. We also installed the sample database employees and sakila, which will be used throughout the book for the demonstrations of SQL queries. In this chapter, we will dive further into the syntax of SQL and learn a few essential concepts.

Structure

In this chapter, we will learn the following topics:

- Getting Started with the simple SQL queries
- Selection and Projection
- Arithmetic Operators and Precedence
- Column Aliases
- NULL
- DISTINCT

Objective

The objective of this chapter is to get started with the basic SQL statements. After following this chapter, readers can select the various schemas and tell the difference between the system catalog schemas and other schemas that have data. Readers will also be comfortable with Selection, Projection, and arithmetic operations. Finally, the readers

will have an introduction to the concepts of `NULL` and `DISTINCT`.

Getting Started with Simple SQL Statements

Let us get started with a few basic yet essential queries. We have briefly seen these queries earlier too. However, this time, we will learn these queries in detail and with the proper context. Log in to the local MySQL server instance or the local MariaDB server instance with the root user using the MySQL Workbench. We can see the list of all the databases (also known as **schemas**) in the **`Schemas`** tab of the **`Navigator`** panel located on the left-hand side. If we double click on the database name there, we can see the options for various objects such as tables and views. If we double click those options, it shows us the list of relevant objects accessible to the root user. If we double click those objects, we see more details about the objects. Following is a screenshot of the same:

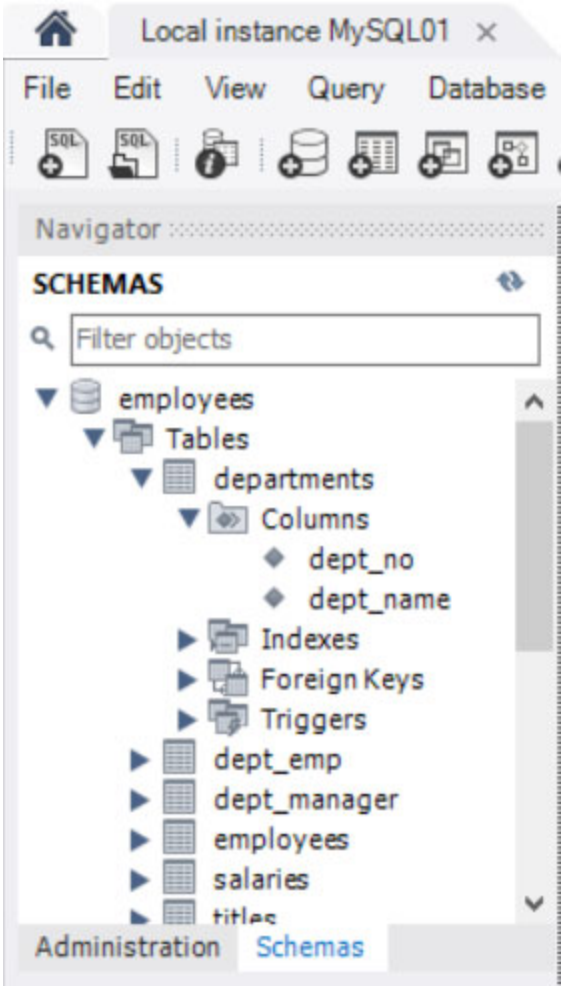


Figure 3.1

We can even collapse these details by clicking the triangular symbols in the tab. We can also retrieve the list of databases by running the following SQL command or statement:

```
SHOW DATABASES;
```

It returns the following rows as a result:

```
employees
information_schema
mysql
performance_schema
sakila
```

```
sys
world
```

As we can see in the output above, the output has three schemas more than those shown in the schema browser. They are known as metadata or data dictionary or system catalog schemas. Let us have a look at them one by one:

- The mysql schema is the system schema. It has the tables that have the information required by the MySQL server process to execute itself. It stores the metadata related to the objects and operational data.
- The `INFORMATION_SCHEMA` stores the metadata of the databases. It stores the information about the MySQL server, such as the name of a database and table, the data type of a column, and access privileges.
- The `Performance_Schema` monitors the execution of the MySQL Server process at a low level. The `Performance_Schema` allows us to inspect the internal execution of the server at runtime. It focuses on the data related to the performance.

To query the objects such as tables of a schema, we need to switch to that schema. The following query selects the schema sakila:

```
USE SAKILA;
```

To see the list of tables in a schema, we run the following query:

```
SHOW TABLES;
```

It will show the list of tables present in the schema sakila. All these queries, as discussed earlier, are case insensitive.

To select all the rows and all the columns from a table named `ACTOR` in the schema, run the following query:

```
SELECT * FROM ACTOR;
```

To see the structure of table ACTOR, we can run any of the following queries:

```
DESC ACTOR;
DESCRIBE ACTOR;
```

These are a few basic, yet very important queries and we will use them regularly in the demonstrations throughout the entire book.

Selection and Projection

We have seen the SELECT statement earlier. It chooses the rows in a database table (or a view, but for now, we will focus on tables). We can restrict the rows with the WHERE clause in a SELECT query as follows:

```
SELECT * FROM ACTOR WHERE FIRST_NAME = 'BURT';
```

Run the query, and you will observe that the output is limited to the rows that have BURT as the value of the column FIRST_NAME. This operation of selecting specific rows from a table is known as the **selection** in the relational terminology.

We also saw that the following query chooses all the rows and columns from the table:

```
SELECT * FROM ACTOR;
```

If we want to select all the rows but only a few columns, then the operation is known as the **projection** in the relational terminology. Following is an example:

```
SELECT FIRST_NAME, LAST_NAME FROM ACTOR;
```

We can even change the order of the columns in the Projection operations as follows:

```
SELECT LAST_NAME, FIRST_NAME FROM ACTOR;
```

We can use the combination of the Selection and the Projection operations as follows:

```
SELECT FIRST_NAME, LAST_NAME FROM ACTOR WHERE FIRST_NAME =
'BURT';
```

Arithmetic Operators and Precedence

We can use the arithmetic operations with the numerical columns and constants. These operators follow the BODMAS rule when it comes to the precedence. The implementation of SQL of MySQL supports the following operators:

Operator	Description	Example Query
+	Addition Operator	`SELECT 10 + 3;`
-	Subtraction Operator	`SELECT 10 - 3;`
*	Multiplication Operator	`SELECT 10 * 3;`
/	Division Operator	`SELECT 10 / 3;`
DIV	Integer Division Operator	`SELECT 10 DIV 3;`
% (or MOD)	Modulus Operator	`SELECT 10 MOD 3; SELECT 10 % 3;`

Table 3.1

In the table above, we can see that we are using constant numerical values in the `SELECT` statements. Run all the queries one by one to understand how these basic operators work. We can use the numeric columns of a table and numeric constants as operands of these arithmetic operators. Following is a simple example:

```
SELECT title, 12 * rental_rate + 200 FROM film;
```

The output of the above query is computed with the precedence operations using the **BODMAS** rule. By that rule, the multiplication is computed before the addition. We

can force the MySQL to compute the addition first by putting a pair of simple brackets around the operands as follows:

```
SELECT title, 12 * (rental_rate + 200) FROM film;
```

This is how we can work with the arithmetic operators with numerical columns of a table and constants as operands.

Column Aliases

You must have observed that when the output of a query is displayed, it shows the names of the columns in the heading. We can choose what those headings can be. This concept is known as **Column Aliases**. We can mention them after the column name. If there is a space in the string denoting the alias, then the use of double quotes around the string denoting the alias is mandatory. We can also use the keyword AS between the name of the column and the alias. The following query covers all the possible options for the alias:

```
SELECT
title AS "Film Title", release_year "Year of Release",
description AS Summary, original_language_id Language
FROM
film;
```

NULL

NULL is a special marker in the database which represents the absence of any value. It does not mean zero (in case of numeric columns) or an empty string (for character or string columns). It just means there is no value at all. It was introduced by *E.F.Codd*, who proposed the relation data model first. It is a keyword in any relational database. `NULL` is a state, not a value. The output of the following query shows us the `NULL` in the last column mentioned in the `SELECT`:

```
SELECT title, description, original_language_id FROM film;
```

The following screenshot shows the output:

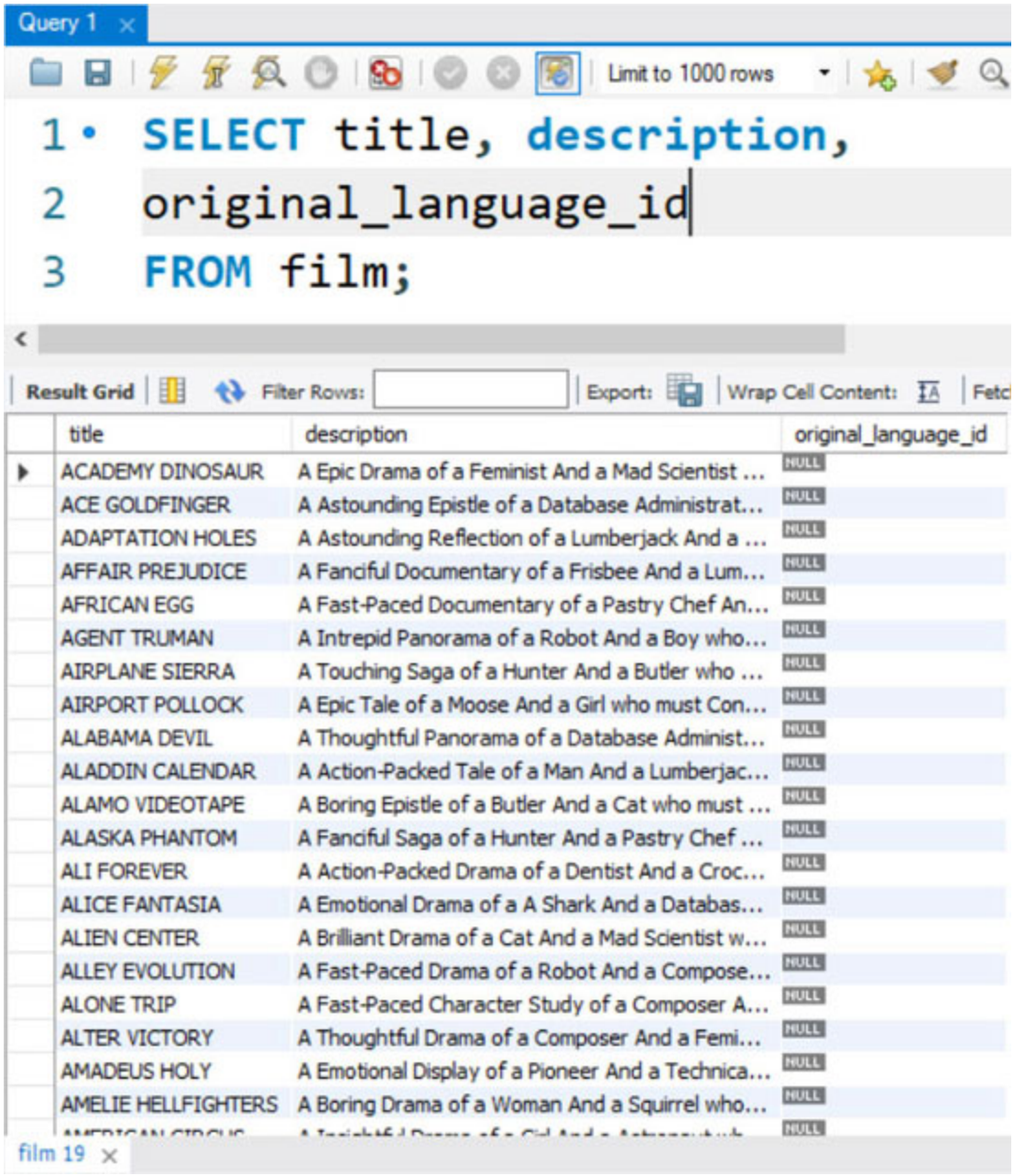


Figure 3.2

We can see the `NULL` in the output for the last column in the `SELECT` statement. As an exercise, check what other tables are storing `NULL`.

DISTINCT

In case we want to see distinct values for a column or a group of columns, we use the `DISTINCT` keyword. The results include `NULL` too. Let us see its working step-by-step. Run the following query to see the list of all the values in a column of a table:

```
SELECT film_id FROM film_actor;
```

You must have noticed that the values are not unique. To see the list of all the unique values, run the following query:

```
SELECT DISTINCT film_id FROM film_actor;
```

Let us see how it works with the combination of columns. Run the following query:

```
select store_id, active from customer;
```

We see a lot of duplicate pairs of values in the output. We can see the unique combination of the values for both the columns with the following query:

```
select DISTINCT store_id, active from customer;
```

We can see the four distinct combinations of values in the output of the above query. We can extend this logic to more than two columns.

Conclusion

In this chapter, we got started with the basic syntax of SQL queries with MySQL and MariaDB. We are now comfortable with the basic syntax of SQL. We learned and demonstrated column aliases, BODMAS, and arithmetic operations. We can now write queries for choosing a unique combination of columns and arithmetic operations. We also saw the meaning of the keyword `NULL` and learned how to use the keyword `DISTINCT` to fetch the unique combination of records.

In the next chapter, we will discuss the different ways we can use the `WHERE` clause in the SQL queries.

Points to Remember

- Arithmetic operators in the implementation of SQL in MySQL follow the rules of precedence defined by BODMAS
- `NULL` is a state, not a value. It means the state of absence of any value

MCQs

1. Which one of the following is the correct command to select a schema named sakila?
   a) `Choose sakila;`
   b) `LIST sakila;`
   c) `Show sakila;`
   d) `Use sakila;`
2. A `NULL` value is:
   a) `An empty string`
   b) `Zero`
   c) `Absence of any value`
   d) `infinity`
3. What keyword should we use to choose the unique combination of values from a table?
   a) `UNIQUE`
   b) `DISTINCT`
   c) `SORT`
   d) `DIFFERENT`

Answer to MCQ

1. *d)*
2. *c)*
3. *b)*

Questions

1. How to retrieve unique values for a column or a combination of columns from a database table?
2. Explain the meaning of the keyword NULL.
3. What is metadata? What are the metadata schemas in MySQL and MariaDB?
4. State the difference between the operations of selection and projection.

Key Terms

Selection, Projection, `NULL`, `DISTINCT`, Operators, Precedence

CHAPTER 4

The WHERE Clause in Detail

In the last chapter, we learned and demonstrated the basic SQL syntax and a few relatively simple concepts and their demonstration with the SQL queries. We are now comfortable with connecting to MySQL and MariaDB instances as well as running simple queries.

In this chapter, we will focus on the selection operation and learn the intricacies of the `WHERE` clause in detail.

Structure

Following is the detailed list of topics we will learn and demonstrate with the SQL queries in this chapter:

- The WHERE Clause
- The comparison operators
- LIKE Operator
- Comparing with NULL
- Logical Operations
- Sorting the resultset
- Working with Dates

Objective

The objective of this chapter is to make readers comfortable with the selection operation. After following this chapter, the readers will be able to effectively use the `WHERE` clause in the SQL queries to filter the data. This chapter forms base for many types of queries we will learn throughout the coming

chapters in the book. For example, we can use the `WHERE` clause in the SQL queries about the join operation.

The WHERE Clause

Let us see the usage of the `WHERE` clause. We can use it to filter the resultset returned by the `SELECT` query. Let us see how to filter the data. Run the following queries:

```
USE sakila;
```

Now let us write a simple `SELECT` query:

```
SELECT * FROM film;
```

This query, as we already know, returns all the rows from the table film. We can filter the rows returned based on a criterion of our choice using the `WHERE` clause. Following is an example:

```
SELECT * FROM film WHERE rental_rate = 0.99 ;
```

This query returns all the rows where the value of the column `rental_rate is 0.99`. The following screenshot shows the query and the output:

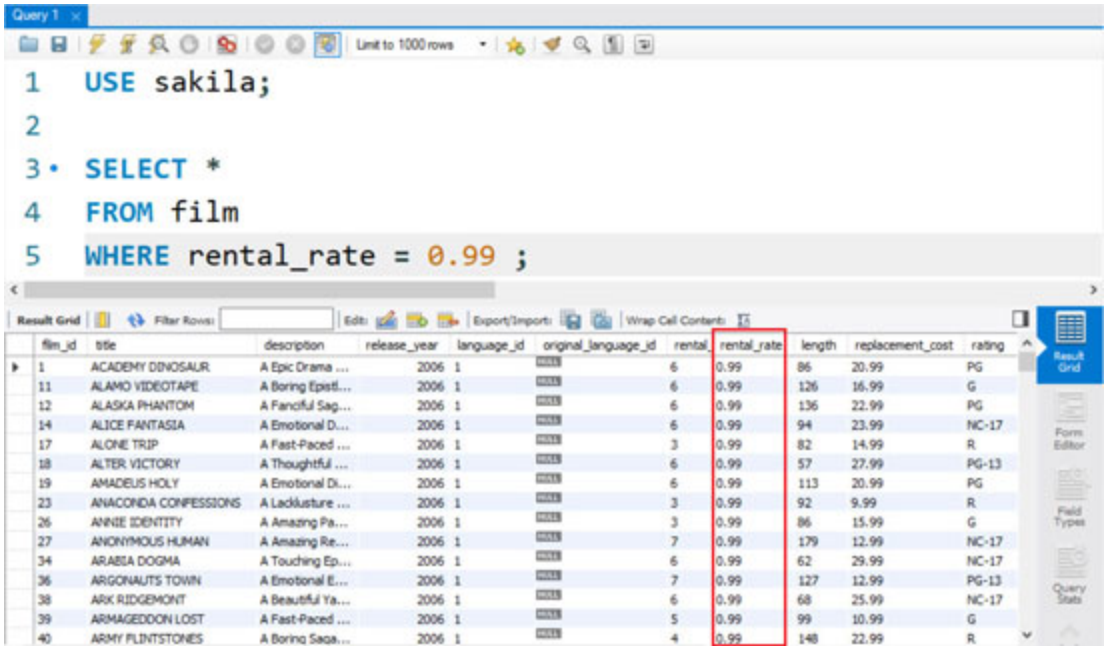


Figure 4.1

We can see that all the rows in the output above match the filter condition. This is an example of the numerical data. We can also use the `WHERE` clause to filter character strings as follows:

```
SELECT * FROM film WHERE title = 'ALONE TRIP';
```

This query returns the one row where the film name matches with the string mentioned in the `WHERE` clause of the query.

The comparison operators

We can use the comparison operators in the `WHERE` clause of the `SELECT` query. Let us see an example:

```
SELECT * FROM film WHERE rental_rate > 1.99;
```

This query used the greater than comparison operator. Similarly, we can use other comparison operators as follows:

```
SELECT * FROM film WHERE rental_rate >= 1.99;
SELECT * FROM film WHERE rental_rate < 1.99;
SELECT * FROM film WHERE rental_rate <= 1.99;
```

We also have `Not Equal To` operator. It can be written in the following two ways:

```
SELECT * FROM film WHERE rental_rate <> 2.99;
SELECT * FROM film WHERE rental_rate != 2.99;
```

We also can compare the data of a column with a numerical range using `BETWEEN` and `AND` keywords as follows:

```
SELECT title, rental_rate FROM film WHERE rental_rate BETWEEN
0.99 AND 2.99;
```

The above query returns all the data that falls in the given range (including `0.99` and `2.99`) as follows:

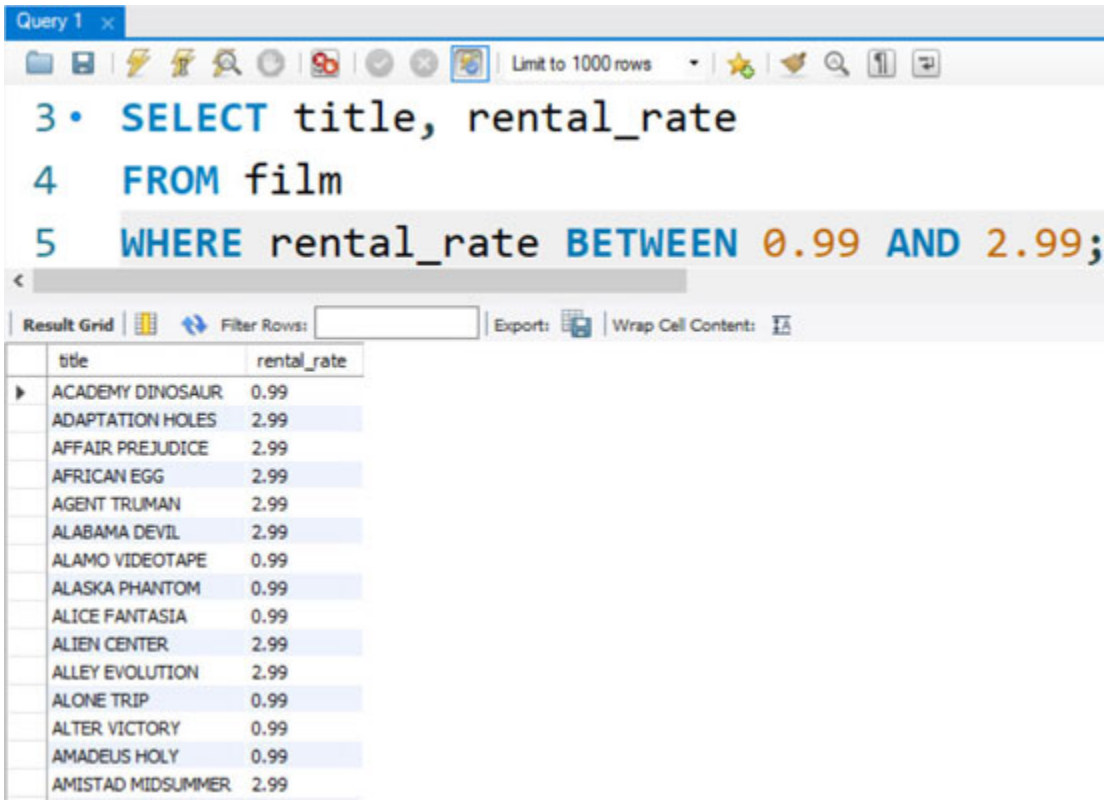


Figure 4.2

We can even compare the values of a numerical and character column with multiple values using the `IN` operator as follows:

```
SELECT title, rental_rate FROM film WHERE rental_rate IN (0,
0.99, 2.99, 4.99);
SELECT title, rental_rate FROM film WHERE title IN ('ARIZONA
BANG', 'ARABIA DOGMA');
```

LIKE Operator

We have seen how to compare character strings using the equality operator. We can even match the patterns of strings with the `LIKE` operator. We must use wildcards _ and % in addition to characters. _ means exactly one character, and % means zero or more characters. Let us see an example. Have a look the following query:

```
SELECT * FROM film WHERE title LIKE 'A%';
```

This query will return all the rows where the value of the column '`title`' starts with the character *A*. The wildcard pattern here means *A* followed by any number of characters.

```
SELECT * FROM film WHERE title LIKE '_C%';
```

This query returns all the rows where the value of the column '`title`' has the second character as C. The wildcard pattern here means a single character followed by C followed by any number of characters.

```
SELECT * FROM film WHERE title LIKE '%E';
```

This query returns all the rows where the value of the column '`title`' has the last character as *E*. The wildcard pattern here means any number of characters and the last character as *E*.

```
SELECT * FROM film WHERE title LIKE '%B%';
```

This query returns all the rows where the value of the column title has the character *B* anywhere. The wildcard pattern here means any number of characters followed by character *B* followed any number of characters.

Comparing with NULL

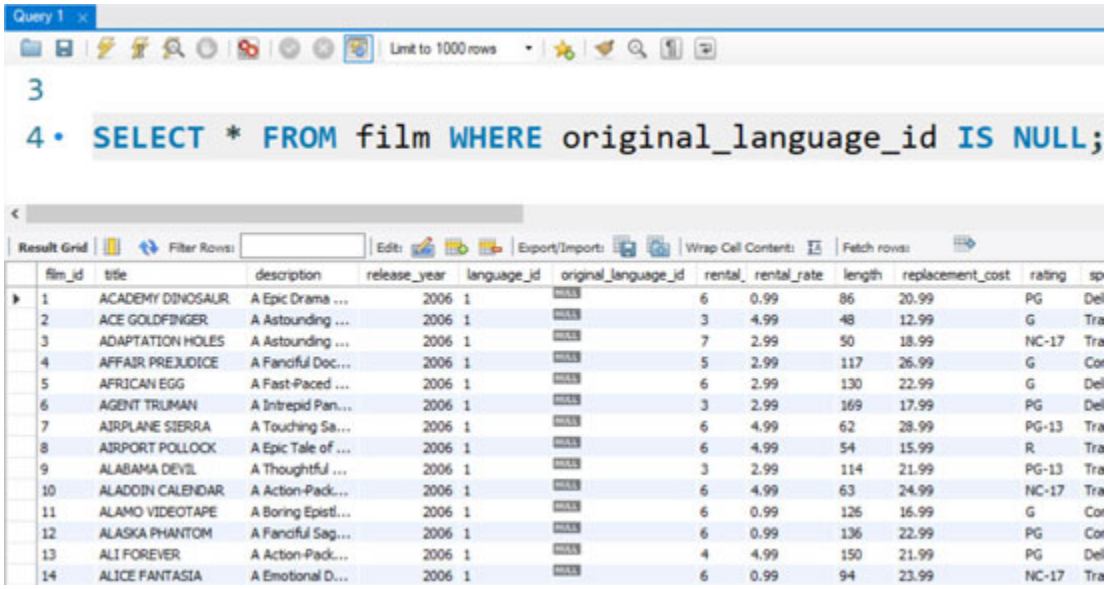
We know that the `NULL` means the absence of any value. If we try to use the equality operator to compare the values in a column with the `NULL`, then it does not return anything. Following is the example of the syntactically correct, yet logically wrong query:

```
SELECT * FROM film WHERE original_language_id = NULL;
```

This query returns nothing. Following is the logically correct query:

```
SELECT * FROM film WHERE original_language_id IS NULL;
```

This query returns all the rows where the value of the column `original_language_id` is `NULL` as follows:



film_id	title	description	release_year	language_id	original_language_id	rental_	rental_rate	length	replacement_cost	rating	sp
1	ACADEMY DINOSAUR	A Epic Drama ...	2006	1	NULL	6	0.99	86	20.99	PG	Del
2	ACE GOLDFINGER	A Astounding ...	2006	1	NULL	3	4.99	48	12.99	G	Tra
3	ADAPTATION HOLES	A Astounding ...	2006	1	NULL	7	2.99	50	18.99	NC-17	Tra
4	AFFAIR PREJUDICE	A Fanciful Doc...	2006	1	NULL	5	2.99	117	26.99	G	Co
5	AFRICAN EGG	A Fast-Paced ...	2006	1	NULL	6	2.99	130	22.99	G	Del
6	AGENT TRUMAN	A Intrepid Pan...	2006	1	NULL	3	2.99	169	17.99	PG	Del
7	AIRPLANE SIERRA	A Touching Sa...	2006	1	NULL	6	4.99	62	28.99	PG-13	Tra
8	AIRPORT POLLOCK	A Epic Tale of ...	2006	1	NULL	6	4.99	54	15.99	R	Tra
9	ALABAMA DEVIL	A Thoughtful ...	2006	1	NULL	3	2.99	114	21.99	PG-13	Tra
10	ALADDIN CALENDAR	A Action-Pack...	2006	1	NULL	6	4.99	63	24.99	NC-17	Tra
11	ALAMO VIDEOTAPE	A Boring Epistl...	2006	1	NULL	6	0.99	126	16.99	G	Co
12	ALASKA PHANTOM	A Fanciful Sag...	2006	1	NULL	6	0.99	136	22.99	PG	Co
13	ALI FOREVER	A Action-Pack...	2006	1	NULL	4	4.99	150	21.99	PG	Del
14	ALICE FANTASIA	A Emotional D...	2006	1	NULL	6	0.99	94	23.99	NC-17	Tra

Figure 4.3

Logical Operations

We can also use logical operations like `AND`, `OR`, and `NOT` to combine multiple conditions to filter the data. Let us have a look at the following examples:

```
SELECT * FROM film WHERE length >= 30 AND title LIKE 'A%';
```

This query returns all the films where the length of the film is more than 30 minutes, and the title starts with the character A as follows:

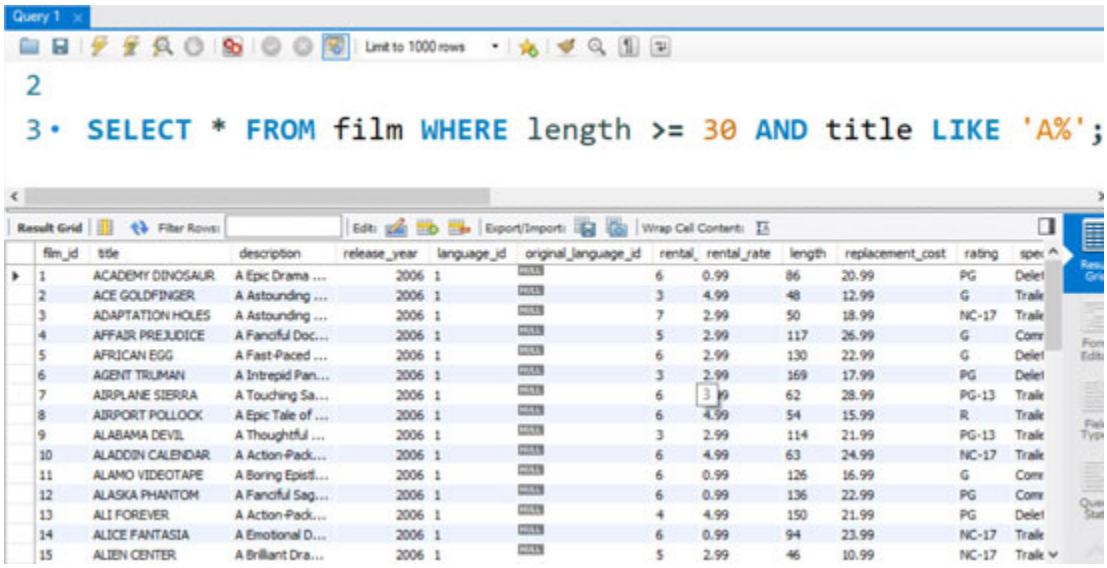


Figure 4.4

Following is an example of the OR operation:

```
SELECT * FROM film WHERE length >= 30 OR title LIKE 'A%';
```

Similarly, following is an example of `NOT` operation:

```
SELECT * FROM film WHERE rental_duration NOT IN (1, 2, 3, 5,
6);
```

Let us see how we can write the `BETWEEN AND` style query with the logical operations. Following is the query:

```
SELECT title, rental_rate FROM film WHERE rental_rate BETWEEN
0.99 AND 2.99;
```

We can rewrite this with the logical operations as follows:

```
SELECT title, rental_rate FROM film WHERE rental_rate >= 0.99
AND rental_rate <= 2.99;
```

We can also rewrite the queries that use the `IN` operator using the logical operator. Following is a query with the `IN` operator:

```
SELECT title, rental_rate FROM film WHERE title IN ('ARIZONA
BANG', 'ARABIA DOGMA');
```

Following is the rewritten query with the logical operator OR:

```
SELECT title, rental_rate FROM film WHERE title = 'ARIZONA
BANG' OR title = 'ARABIA DOGMA';
```

Sorting the Resultset

We can sort the data retrieved by the `SELECT` query with the clause `ORDER BY` as follows:

```
SELECT * FROM film ORDER BY title;
```

The query returns the resultset sorted by alphabetically ascending order of the character column title.

By default, all the data returned with the clause ORDER BY is ascending alphabetically in case of character columns and numerically in case of numerical columns. We can retrieve it in the ascending or the descending order by explicitly using the keywords `ASC` and `DESC`.

```
SELECT * FROM film ORDER BY title ASC;
SELECT * FROM film ORDER BY title DESC;
```

We can even use the aliases of columns for sorting as follows:

```
SELECT title AS NAME, description FROM film ORDER BY NAME;
SELECT title AS NAME, description FROM film ORDER BY NAME DESC;
```

We can also sort by multiple columns as follows:

```
SELECT rental_rate, title, description FROM film ORDER BY
rental_rate, title;
SELECT rental_rate, title, description FROM film ORDER BY
rental_rate, title DESC;
```

Note that this sorts only the retrieved data and does not sort the data stored in the tables in the database.

Working with Dates

We have seen how to use various operators with the numerical and character types of columns. Now, we will see how to work with the datetime types of columns. We will see the datatypes in the MySQL in detail while learning about Data Definition **Language (DDL)**queries.

The following query filters the output based on the exact datetime:

```
SELECT * FROM rental WHERE rental_date = '2005-05-24 22:53:30';
```

The following is an example of a datetime column with a comparison operator:

```
SELECT * FROM rental WHERE rental_date > '2005-05-24';
```

We can also use BETWEEN AND for extracting the data in a range of dates as follows:

```
SELECT * FROM rental WHERE rental_date BETWEEN '2005-05-24' AND
'2005-05-29';
```

We can even use the `LIKE` operator to extract all the records of the year *2005* as follows:

```
SELECT * FROM rental WHERE rental_date LIKE '2005%';
```

Conclusion

In this chapter, we have learned and demonstrated the detailed usage of the `WHERE` clause to filter the resultset returned by a `SELECT` query. Now we can write a variety of `SELECT` queries with the `WHERE` clause to retrieve the data as per the needs of the business or scientific applications downstream.

The `WHERE` clause is an important aspect of SQL, and it is one of the most frequently used features of SQL.

In the next chapter, we will get started with the concept of the built-in functions in MySQL and explore the single row

functions in detail.

Points to Remember

- `NULL` cannot be compared with any value. We must use `IS NULL` and `IS NOT NULL` to use it in the `WHERE` clause.
- We can work with comparison operators and `datetime` columns.
- We can use the `ORDER BY` clause to sort the `resultset` with ascending and descending orders of `datetime`, `numerical`, and `character` columns.

MCQs

1. Which clause filters the resultset?
   a) `GROUP BY`
   b) `FILTER`
   c) `WHERE`
   d) `ORDER BY`
2. Which operator is used to compare string patterns with character columns?
   a) `COMPARE`
   b) `LIKE`
   c) `=`
   d) `REGEX`
3. What clause is used to sort the resultset?
   a) `FILTER`
   b) `SORT`
   c) `WHERE`
   d) `ORDER BY`

Answer to MCQs

1. *c)*
2. *b)*
3. *d)*

Questions

1. How to sort the result set returned by the query?
2. How to compare `NULL`?
3. What are the logical operators?
4. How to match string patterns in the character columns?

Key Terms

`WHERE, LIKE, ORDER BY, IS NULL`

CHAPTER 5

Single Row Functions

In the last chapter, we learned and demonstrated the `WHERE` clause. We learned how to filter rows from output resultset using the `WHERE` clause.

In this chapter, we will focus on the basics of functions in MySQL and MariaDB. We will learn and demonstrate the single row functions in detail. We will see the various categories of single-row functions such as case and character manipulation, number manipulation, and data manipulation. We will also learn to handle the `NULL` values with these functions.

Structure

Following is the detailed list of topics we will learn and demonstrate with the SQL queries in this chapter:

- Single Row Functions
- The Dual Table
- The case manipulation functions
- The character manipulation functions
- Number Manipulation functions
- Date Manipulation Functions
- Datetime to string and vice versa
- NULL Handling
- CASE statement
- Nested Single Row functions

Objective

The objective of this chapter is to make readers comfortable with the functions in SQL. We will learn and demonstrate all types of single-row functions in detail. After following this chapter, the readers will be able to work with single row functions of various types and apply those on the resultset returned by the `SELECT` query.

Single Row Functions

MySQL has many built-in functions to perform various operations on data. Functions are routines that perform computation on the data we pass to them as arguments. For databases, the data can be constant data or columns. These functions are useful while performing operations such as mathematical calculations, string concatenations, and sub-strings, and so on. SQL functions are divided into two types:

- Single Row Functions (also known as, Scalar Functions)
- Aggregate Functions (also known as, Group Functions)

In this chapter, the focus will be on single-row functions. Single row functions, accept n rows as input and return the same n number of rows in the output resultset. They are applied to every row to produce results. There are various types of single-row functions, and we will learn and demonstrate those categories one by one.

The Dual Table

Let us begin by understanding the concept of the table `DUAL`. It is a dummy table that has one row and one column. It is mostly used as a placeholder in SQL queries that do not need a real table. The examples of the queries using this table are as follows:

```
USE Sakila;
```

```
SELECT 1 from DUAL;
SELECT 1 + 1 from DUAL;
SELECT 'Oracle' from DUAL;
```

The first and the second query select numbers, and the third query selects the string.

NOTE: In MySQL and MariaDB we do not require `DUAL` table and same queries can be run without the from `DUAL` option. `i.e. SELECT 1 from DUAL` and `SELECT 1` would work the same in MySQL and MariaDB. In Oracle always a table is required hence the concept of dummy table.

The Case Manipulation Functions

The case manipulation functions are used for manipulating the case of the characters in character strings. We can use them with static strings or character columns. Let's demonstrate this. If we run the following query:

```
SELECT title, description FROM film;
```

The output is as follows:

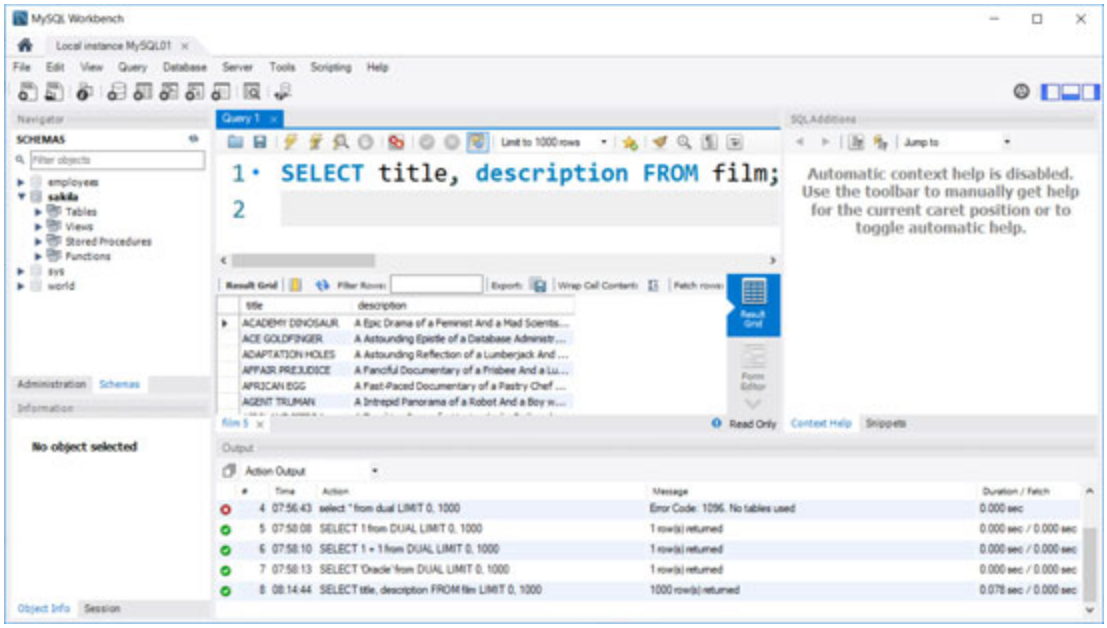


Figure 5.1

We can see that all the characters in the first column in the output are uppercase, and in the second column, they are in mixed case. We can change that by running the following query:

```
SELECT title, LOWER(title) FROM film;
```

The output will have the title as the first column and all the characters in lowercase in the second column of the output as follows:

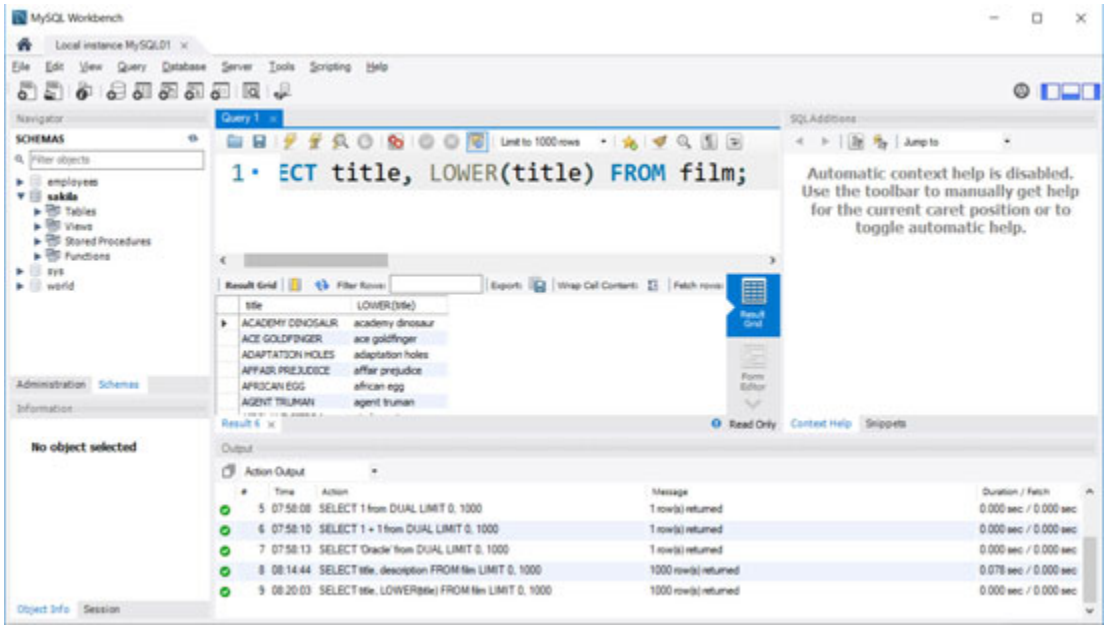


Figure 5.2

Let’s demonstrate another function:

```
SELECT description, Upper(description) FROM film;
```

Run this query; the output will have the description column and the uppercase of that column.

We can even use it with strings as follows:

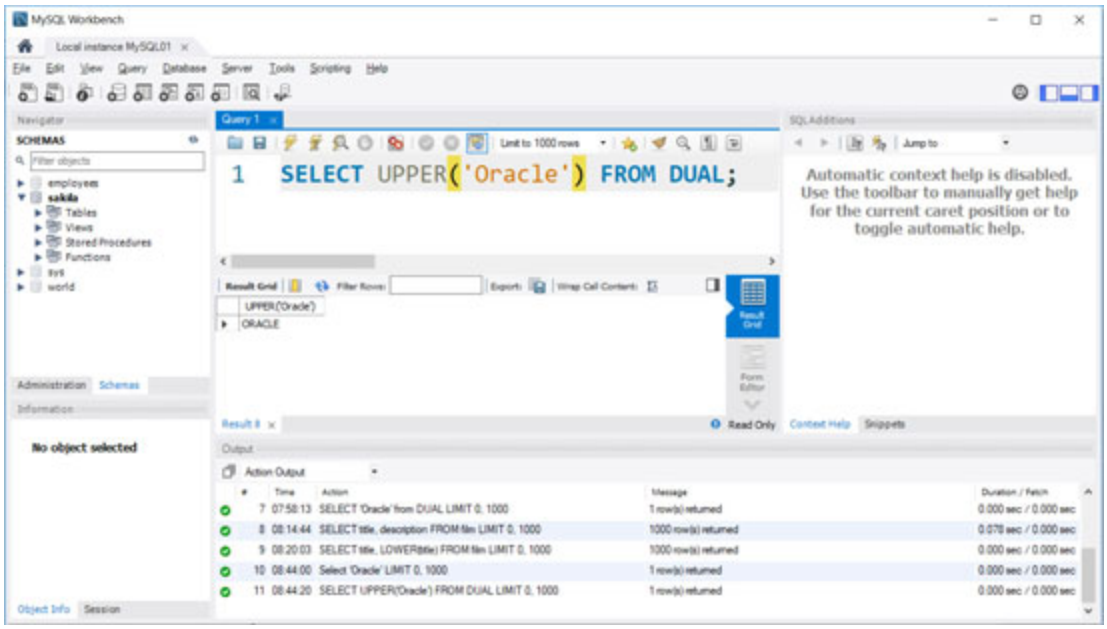


Figure 5.3

The Character Manipulation Functions

The character manipulation functions can be used to manipulate character strings. There are many such functions, and we will have demonstrations of them one by one in this section. We can use strings or character columns with these functions.

We can concatenate two strings with the function `CONCAT()`. It works in the same way as the operator `||`. Both the arguments to the call of this function are stings. Following are the examples of the same:

```
SELECT CONCAT('Oracle', ' SQL') FROM DUAL;
SELECT CONCAT(title, description) FROM film;
```

In the first example, we are concatenating two strings, and in the second example, we are concatenating two different character columns. This is one of the simplest character manipulation functions.

We can extract a substring from a string with the function `SUBSTR()` as follows:

```
SELECT SUBSTR('Oracle SQL', 1, 6) FROM DUAL;
```

The indexing of the strings starts from 1. The first argument is the string or the column from which we need to extract the substring. The second and third arguments are starting and ending indices of the substring from the string. The query above will output the string '`Oracle`'.

We can use the function `length`() to determine the length of a string or a character column as follows:

```
SELECT title, LENGTH(title) FROM film;
```

We can know the location of a substring within a string using the function `INSTR`() as follows:

```
SELECT INSTR('Oracle SQL', 'S') FROM DUAL;
SELECT INSTR('Oracle SQL', '1') FROM DUAL;
```

In both examples, we are using the function to determine the location of the substring (which is passed as the second argument).

We can use the functions `LPAD()` and `RPAD()` to pad a string with a particular character. Run the following query to see `LPAD()` in action:

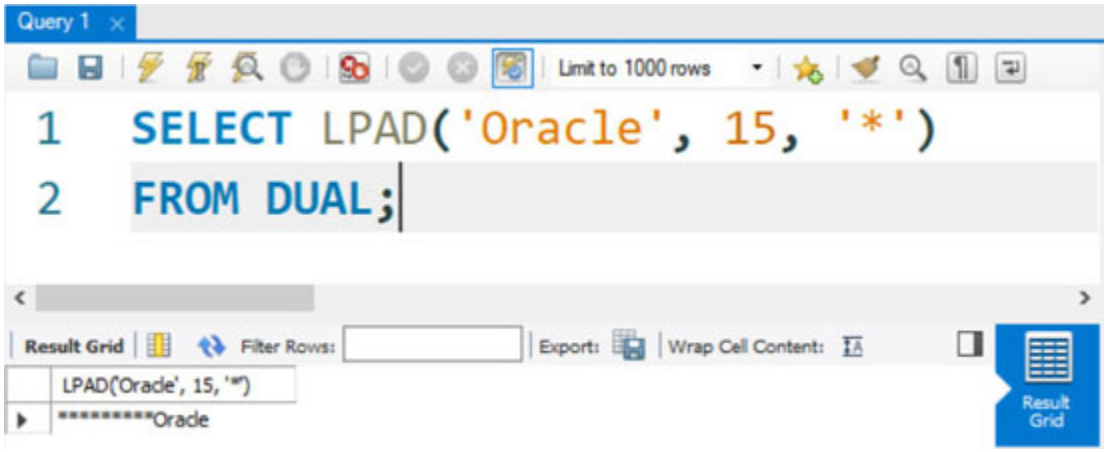


Figure 5.4

Following is an example of `RPAD()` in action:

```
SELECT RPAD('Oracle', 15, '*') FROM DUAL;
```

Number Manipulation Functions

We can round off and truncate numbers. MySQL has functions `ROUND()` and `TRUNCATE()` for these. They accept a number and position of the digit as arguments. The negative integer position means the position is before the decimal point, 0 means the decimal point, and the positive integer means it is the place after the decimal point. The function `ROUND()` rounds off the number whereas the function `TRUNCATE()` truncates the number. Following are the example queries:

```
SELECT ROUND(67.1234, 1), ROUND(67.1234, 0), ROUND(67.1234, -1)
FROM DUAL;
SELECT ROUND(67.1234, 2), ROUND(67.1234, 0), ROUND(67.1234, -2)
FROM DUAL;
SELECT TRUNCATE(67.1234, 1) from dual;
SELECT TRUNCATE(67.1234, 0) from dual;
SELECT TRUNCATE(67.1234, -1) from dual;
```

Run the queries above and see the output.

We can also use the function `MOD()` to compute the modulus:

```
SELECT MOD(15, 6) from dual;
```

Run the query above and see the output.

Date Manipulation Functions

We can retrieve the current date as follow:

```
SELECT sysdate() from DUAL;
```

We can compute the difference between two dates as follows:

```
SELECT TIMESTAMPDIFF(DAY, '2012-05-05', sysdate()) FROM DUAL;
```

```
SELECT TIMESTAMPDIFF(MONTH, '2012-05-05', sysdate()) FROM DUAL;
```

The first argument is the unit of measure of the difference, and the rest two arguments are the dates. They could be the date columns or the absolute dates, as shown above. Finally, we can add several days to date as follows:

```
SELECT DATE_ADD(sysdate(), INTERVAL 10 DAY);
SELECT DATE_ADD(sysdate(), INTERVAL -10 DAY);
```

The above function calls return a date. Note that we can change the unit of measure of time to `MONTH` or `YEAR` or any other desired unit as per our requirement. A negative value means subtraction.

Datetime to String and Vice Versa

We can convert datetime to string and vice versa. We can use the function `STR_TO_DATE()` to convert any formatted string to a date. The first argument is the string, and the second argument is the format used to represent the date in the string. Following are the examples of the function calls:

```
SELECT STR_TO_DATE('25/08/1986', '%d/%m/%Y') FROM DUAL;
SELECT STR_TO_DATE('25-08-1986', '%d-%m-%Y') FROM DUAL;
SELECT STR_TO_DATE('15-AUG-1947', '%d-%M-%Y') FROM DUAL;
```

Similarly, we can use the function `DATE_FORMAT()` to convert a date to a string in the desired format as follows:

```
SELECT DATE_FORMAT(SYSDATE(), '%Y-%m-%d') FROM DUAL;
SELECT DATE_FORMAT(SYSDATE(), '%Y-%m-%d %H:%i:%s') FROM DUAL;
SELECT DATE_FORMAT(SYSDATE(), '%d-%b-%Y') FROM DUAL;
```

The first argument is the date to be converted to the string, and the second argument is the format in which the string is to be displayed.

NULL Handling

Let us have a look at a few functions that handle the `NULL` value. If a value is `NULL`, then we can replace it with the function `IFNULL()`. Following is an example:

```
select original_language_id, IFNULL (original_language_id,
'English') from film;
```

In the query above, if the value of the first argument is `NULL`, then it is substituted by the second argument.

The function `NULLIF()` returns `NULL` if two expressions are equal else will return the first argument passed. We can use it as follows:

```
Select first_name, last_name, length(first_name),
length(last_name),
NULLIF(length(first_name), length(last_name)) from actor;
```

In the example above, we are comparing the lengths of two different character columns. The result looks as follows:

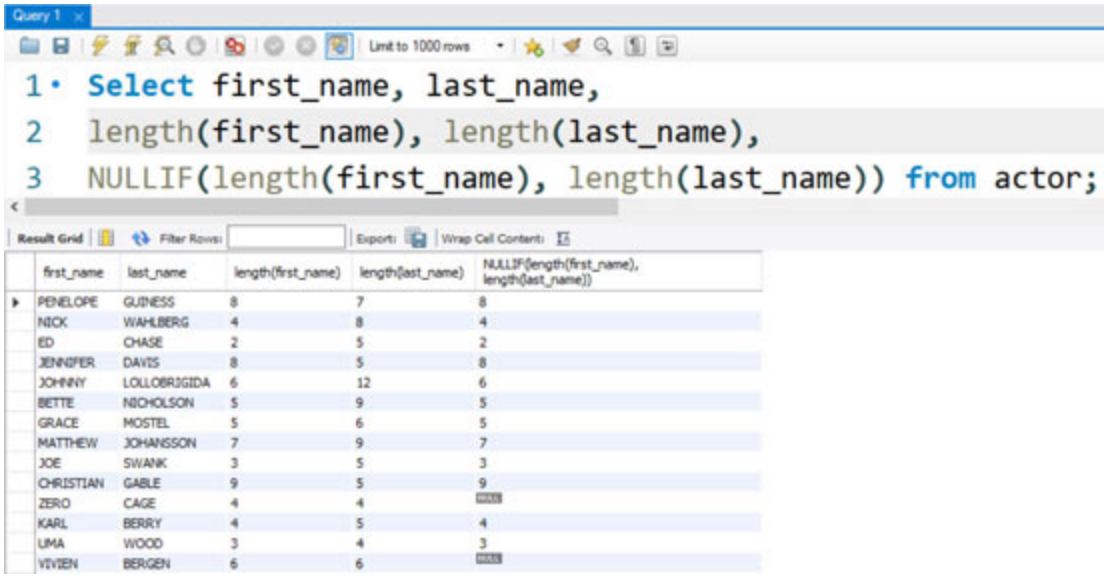


Figure 5.5

If we want to choose the first non-NULL expression from multiple expressions, then we have to use the function `coalesce()`:

```
select coalesce(address2, address, district) from address;
```

CASE Statement

We can have multiple if the style code block with the `CASE` statement. It allows us to handle multiple possibilities for expression as follows:

```
SELECT film_id, title, description, release_year, rental_rate,
CASE rental_rate
WHEN 0.99 THEN 1.00
WHEN 1.99 THEN 2.5
WHEN 2.99 THEN 4.5
ELSE rental_rate END AS "Revised Rental"
FROM film;
```

In the query above, we can handle multiple cases for the value of a column. The output is as follows:

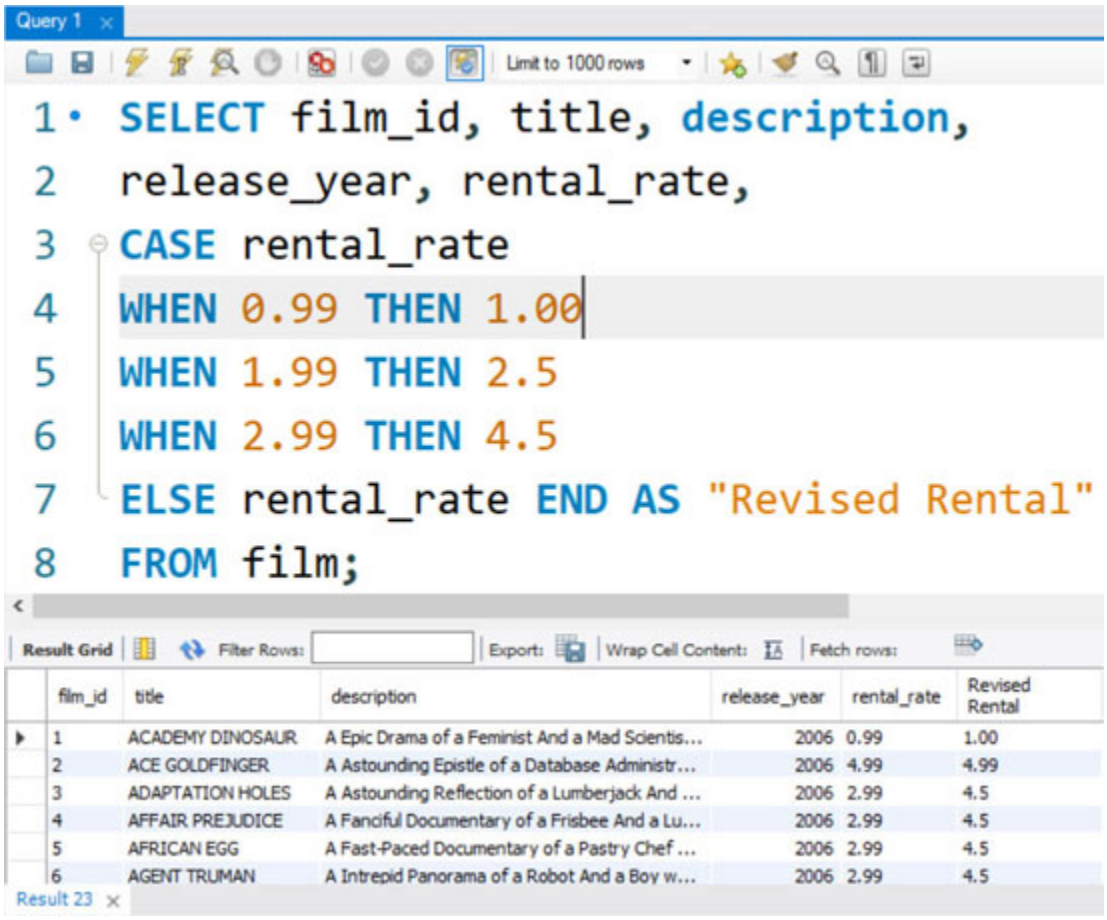


Figure 5.6

Nested Single Row functions

We can use the single row functions in a nested way. In this technique, we pass call of a function to the call of the other function as follows:

```
Select LOWER(SUBSTR(title, 1, 5)) from film;
```

In the query above, we are calling the function `SUBSTR()` within the function `LOWER()`. Run the query and see the output for yourself.

Conclusion

In this chapter, we have learned and demonstrated the detailed usage of the single row functions in MySQL. We

have seen almost all the categories of single row function. These single-row functions are very handy in the activities related to data transformation and report generation. Many times, we may want to properly format and crunch the data or derive new data based on the existing data in the database. These functions fulfill that need. In the next chapter, we will have a look at aggregate functions and discuss them in detail.

Points to Remember

- Single row functions accept the resultset with n number of rows and return n number of rows.
- Single row functions are used in deriving new data or data transformation or report generation.
- We can use multiple single row functions in the same SQL query. We can also use them in a nested way.

MCQs

1. What is the synonym of the term **Single Row Function?**

   a) Scalar function

   b) Vector function

   c) Group function

   d) Aggregate function

2. For what task should we `NOT` use single-row functions?

   a) Character manipulation

   b) Number manipulation

   c) Data grouping and aggregation

   d) NULL Handling

3. What is the synonym of the term **Group Function?**

   a) Scalar function

b) Vector function

c) Single Row function

d) Aggregate function

Answer to MCQs

1. *a)*
2. *c)*
3. *d)*

Questions

1. What are single row functions?
2. How to use the `CASE` keyword?
3. Explain the scenarios in which we can use single-row functions.
4. How to handle `NULL` values of an expression?
5. How can we add one year to the current date? Explain with the help of SQL query.

Key Terms

Single Row Functions, Nested Single row functions, character manipulation, CASE, case manipulation, numerical functions

CHAPTER 6

Group Functions

In the last chapter, we learned and demonstrated the single row functions (also known as **scalar functions**). We saw various categories of single-row functions and learned how to use them. We also discussed the scenarios when we could use the single-row functions. In this chapter, we will continue our quest to explore the functions in MySQL and MariaDB further in detail.

In this chapter, we will learn and demonstrate group functions. Group functions are useful features of any RDBMS product. They are frequently used in the requirement about the data analytics where the grouping of the data is required. The `GROUP BY` clause is used to group the data. We will also learn the `HAVING` clause and demonstrate it to filter the groups. We will learn and demonstrate all these topics in detail in this chapter.

Structure

Following is the detailed list of topics we will learn and demonstrate with the SQL queries in this chapter:

- Group Functions
- The GROUP BY Clause
- HAVING

Objective

We will learn and demonstrate all the group functions in detail. After following this chapter, the readers will be able

to work with group functions and use the `HAVING` clause.

Group Functions

Group functions are also known as aggregate functions. These functions operate on sets of rows to return one result per set. The sets of rows are decided on some criteria. Let us study how to use group by functions.

We have the following group functions in MySQL:

- **`AVG()`:** computes the average
- **`MAX()`:** returns maximum of the set
- **`MIN()`:** returns minimum of the set
- **`SUM()`:** returns sum of the set
- **`STDDEV()`:** computes the standard deviation
- **`VARIANCE()`:** computes the variance

The following queries demonstrate the usage of these functions:

```
USE sakila;
SELECT AVG(address_id) FROM staff;
SELECT MAX(address_id) FROM staff;
SELECT MIN(address_id) FROM staff;
SELECT SUM(address_id) FROM staff;
SELECT STDDEV(address_id) FROM staff;
SELECT VARIANCE(address_id) FROM staff;
```

All the above queries treat the entire table as a single set, and we get only one row as output per query.

All the group functions ignore `NULL` values by default. For example, the query `SELECT AVG(postal_code) FROM address` ignores null values. The average is calculated only for the rows where the value for the column for postal code is not `NULL`.

The workaround for this is to use the following query:

```
SELECT AVG(NULLIF(postal_code, 0)) FROM address;
```

The above query replaces the `NULL` values with zeroes; thus, the function `AVG()` considers the rows with the `NULL` values too in the final computation.

Another special group function is `COUNT()`. It returns the count of rows. If we run the queries `SELECT COUNT(*) FROM address or SELECT COUNT(1) FROM address`, then it returns the count of the total number of rows irrespective of `NULL` values. If we run the query such as `SELECT count(address2) FROM address`, it returns the count of non-NULL rows for that column.

We can even find out the count of the distinct values for a column as follows:

```
SELECT COUNT( DISTINCT postal_code) FROM address;
```

This is how we can use the group functions on a complete table.

The GROUP BY Clause

Till now, we have seen how to use the group functions on the entire table. We can divide the result set into various sets to retrieve one value per set using the clause `GROUP BY`. For example, we can use it as follows:

```
SELECT SUM(rental_rate)
FROM film
GROUP BY rental_duration ;
```

In the query above, we are computing the summation of the rental rates grouped by rental duration. To put it in a better context, we can use the following query:

```
SELECT rental_duration, SUM(rental_rate)
FROM film
GROUP BY rental_duration ;
```

The output is as follows:

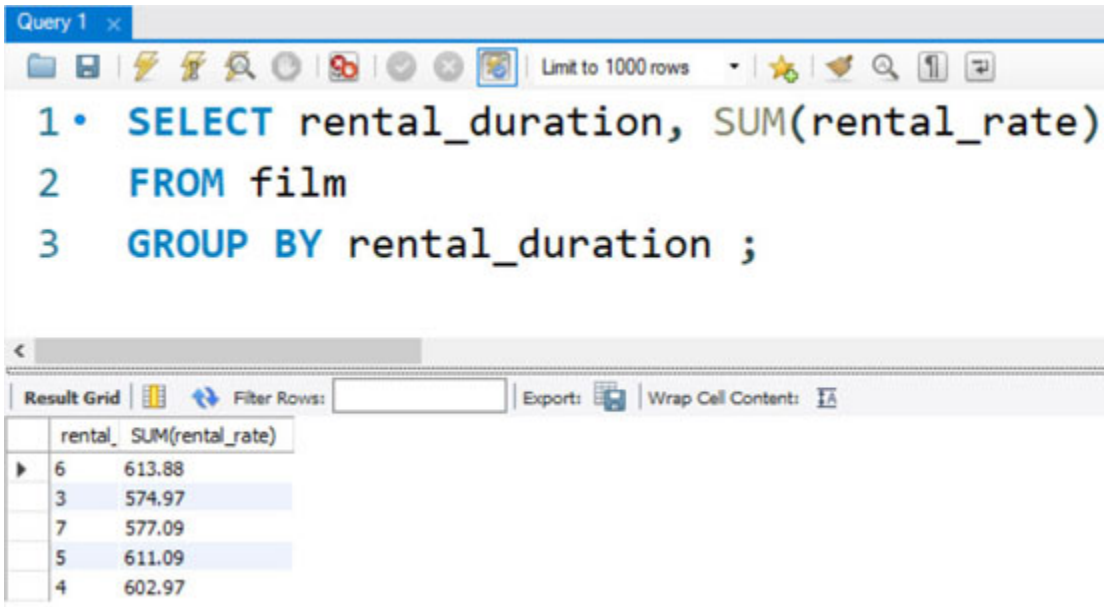


Figure 6.1

We can also use the other group functions in the same manner as follows:

```
SELECT rental_duration, AVG(rental_rate)
FROM film
GROUP BY rental_duration;
```

We can also use more columns in the `GROUP BY` clause. The only condition is that those all columns must also be in the `SELECT` column list as follows:

```
SELECT rental_duration, replacement_cost, SUM(rental_rate)
FROM film
GROUP BY rental_duration, replacement_cost ;
```

The output will be as follows for multiple columns in the `GROUP BY` clause:

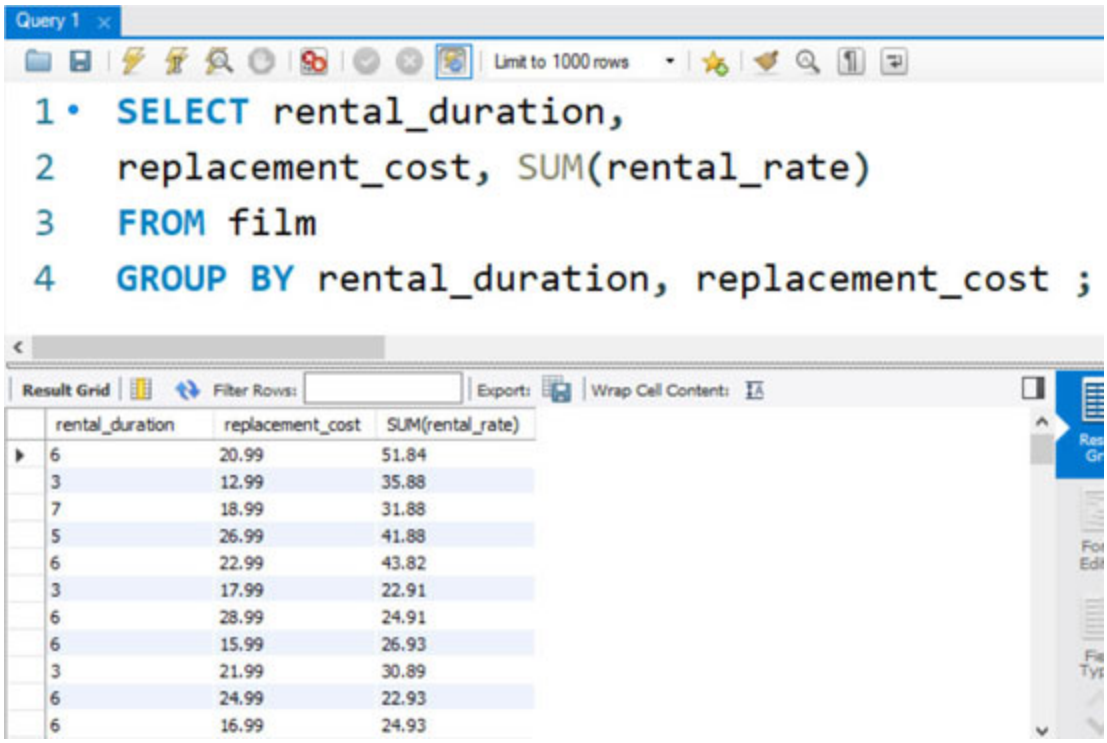


Figure 6.2

HAVING

We have already seen how to filter rows based on some conditions with the `WHERE` clause. The limitation of the `WHERE` clause is that it only filters rows, not groups. To filter groups, we need to use the `HAVING` clause. Following is an example query that demonstrates this:

```
SELECT rental_duration, SUM(rental_rate)
FROM film
GROUP BY rental_duration
HAVING SUM(rental_rate) > 600;
```

We are filtering the output groups based on the criteria mentioned in the `HAVING` clause. It is not mandatory to use the same aggregate function in the `HAVING` clause. We can use different functions too. The following query demonstrates this:

```
SELECT rental_duration, SUM(rental_rate)
```

```
FROM film
GROUP BY rental_duration
HAVING AVG(rental_rate) > 3;
```

Conclusion

In this chapter, we have learned and demonstrated the detailed usage of the group by (also known as **aggregate**) functions in MySQL. We have seen all the aggregate functions. The group functions, just like single-row functions, are also used for report generation and data transformation. They are useful in the grouping of data. We can filter the data generated by them with the `HAVING` clause.

Points to Remember

- Aggregate functions are also known as group functions
- Aggregate functions are also used in deriving new data or data transformation or report generation
- We can use multiple group functions in the same SQL query
- We must use the `HAVING` clause to filter the groups

MCQs

1. What clause filters the grouped data?
   a) `HAVING`
   b) `WHERE`
   c) `GROUP BY`
   d) `ORDER BY`
2. For what task should the aggregate functions are best used?
   a) Character manipulation
   b) Number manipulation

c) Data grouping and aggregation

d) NULL Handling

Answer to MCQs

1. *a)*
2. *c)*

Questions

1. What are the aggregate functions? How are they different from single-row functions?
2. Explain the scenarios in which we can use aggregate functions.
3. How do the aggregate functions handle NULL values?

Key Terms

Aggregate functions, group functions, data aggregating, grouping, `HAVING`, `GROUP BY`.

CHAPTER 7

Joins in MySQL

In the last chapter, we learned and demonstrated the group functions (also known as aggregate functions). We also discussed the scenarios when we could use the group functions.

In this chapter, we will learn and demonstrate the techniques to combine columns from various sources. These techniques are known as Joins. They are very useful in many business analytics requirements, where we need data from multiple sources. People working in the projects related to data analysis and visualization must use joins frequently. If you want to work as a DBA, Data Analyst, or Data Scientist, then the knowledge of joins is a must.

Structure

Following is the detailed list of topics we will learn and demonstrate with the SQL queries in this chapter:

- Concept of Joins
- Cross Join
- Simple / Natural / Inner Join
- Outer Join
- Multiple tables and conditions in Join
- Self-Join
- Non equijoin

Objective

We will learn and demonstrate all the types of joins in detail. After following this chapter, the readers will be able to successfully combine columns from various sources to derive the data as per their business and analytics needs.

Concept of Joins

Joins are an important feature of any RDBMS. They are frequently used in data analytics queries. They are employed to combine data from multiple sources (tables or views). We combine the data from various sources based on some condition. And based on what condition we use in joins, they are broadly categorized into the following types:

- Equijoins
- Non-equijoins

Equijoins use an equality operator (=) in the condition for the join. Non-equijoins use other operators such as `LIKE` and >= for the condition for the join. We will see both the types of joins and the sub-types of equijoin in detail. We have learned earlier that the joins combine the columns from multiple sources. The sources can be tables and views or a combination of both; that is, one of the sources can be a table, and others can be a view. We have been using tables in our `SELECT` queries until now. We will learn and demonstrate views in the 10th chapter dedicated to the concept of views in detail.

Cross Join

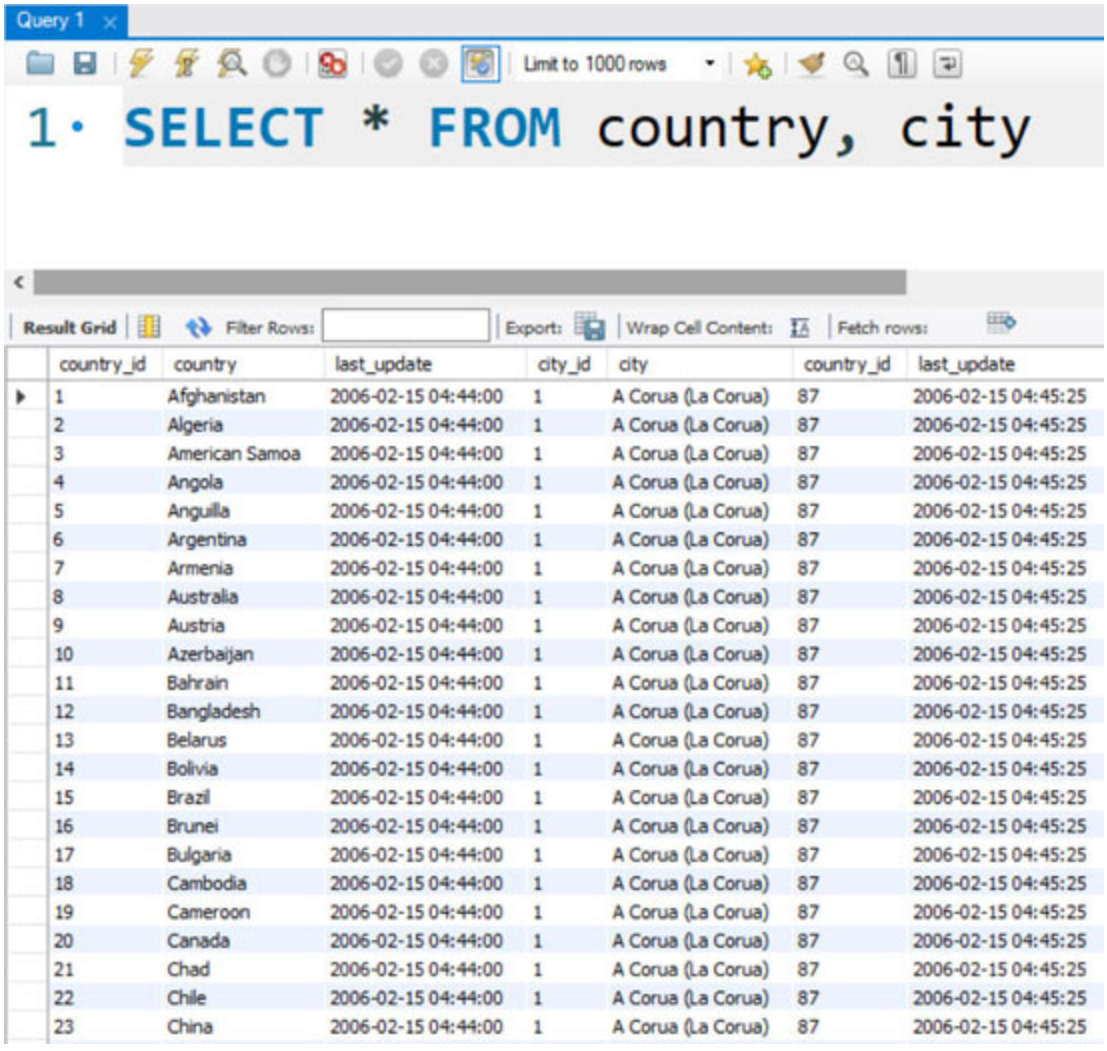
Cross join is also known as **Cartesian Product**. It is the set of every combination of all the rows from all the sources mentioned in the query. If the query has two tables as source with *n* and *m* as the number of rows respectively, then the resultset of the cross join will have *n x m* number of rows.

There are two ways to specify joins. The older syntax does not specify it in the query. The ANSI standard allows us to specify it in the query itself. For every type of join, we will see both the syntaxes.

We can cross join two tables with the following syntax:

```
Use sakila;
SELECT * FROM country, city;
```

The result is as follows:



country_id	country	last_update	city_id	city	country_id	last_update
1	Afghanistan	2006-02-15 04:44:00	1	A Corua (La Corua)	87	2006-02-15 04:45:25
2	Algeria	2006-02-15 04:44:00	1	A Corua (La Corua)	87	2006-02-15 04:45:25
3	American Samoa	2006-02-15 04:44:00	1	A Corua (La Corua)	87	2006-02-15 04:45:25
4	Angola	2006-02-15 04:44:00	1	A Corua (La Corua)	87	2006-02-15 04:45:25
5	Anguilla	2006-02-15 04:44:00	1	A Corua (La Corua)	87	2006-02-15 04:45:25
6	Argentina	2006-02-15 04:44:00	1	A Corua (La Corua)	87	2006-02-15 04:45:25
7	Armenia	2006-02-15 04:44:00	1	A Corua (La Corua)	87	2006-02-15 04:45:25
8	Australia	2006-02-15 04:44:00	1	A Corua (La Corua)	87	2006-02-15 04:45:25
9	Austria	2006-02-15 04:44:00	1	A Corua (La Corua)	87	2006-02-15 04:45:25
10	Azerbaijan	2006-02-15 04:44:00	1	A Corua (La Corua)	87	2006-02-15 04:45:25
11	Bahrain	2006-02-15 04:44:00	1	A Corua (La Corua)	87	2006-02-15 04:45:25
12	Bangladesh	2006-02-15 04:44:00	1	A Corua (La Corua)	87	2006-02-15 04:45:25
13	Belarus	2006-02-15 04:44:00	1	A Corua (La Corua)	87	2006-02-15 04:45:25
14	Bolivia	2006-02-15 04:44:00	1	A Corua (La Corua)	87	2006-02-15 04:45:25
15	Brazil	2006-02-15 04:44:00	1	A Corua (La Corua)	87	2006-02-15 04:45:25
16	Brunei	2006-02-15 04:44:00	1	A Corua (La Corua)	87	2006-02-15 04:45:25
17	Bulgaria	2006-02-15 04:44:00	1	A Corua (La Corua)	87	2006-02-15 04:45:25
18	Cambodia	2006-02-15 04:44:00	1	A Corua (La Corua)	87	2006-02-15 04:45:25
19	Cameroon	2006-02-15 04:44:00	1	A Corua (La Corua)	87	2006-02-15 04:45:25
20	Canada	2006-02-15 04:44:00	1	A Corua (La Corua)	87	2006-02-15 04:45:25
21	Chad	2006-02-15 04:44:00	1	A Corua (La Corua)	87	2006-02-15 04:45:25
22	Chile	2006-02-15 04:44:00	1	A Corua (La Corua)	87	2006-02-15 04:45:25
23	China	2006-02-15 04:44:00	1	A Corua (La Corua)	87	2006-02-15 04:45:25

Figure 7.1

The ANSI syntax for the same is as follows:

```
SELECT * FROM country CROSS JOIN city;
```

It is often said that a Cartesian product or a cross join is nothing but a normal join with no condition. Thus, we can get the same result by running the following query:

```
SELECT * FROM country JOIN city;
```

Simple/Inner and Natural Joins

The other type of join where the data from multiple tables is combined based on a match of a condition is known as **Simple** or **Inner join**. The old or theta style syntax for the Simple/Inner join is:

```
select * from city, country where city.country_id =
country.country_id;
```

In this query, we are matching data of city and country tables based on the equality of values of the common column `country_id`, present in both the tables. All the records of both tables are combined where the values for the columns mentioned in the condition are matched. It is not necessary for the columns used in the condition for the join to have the same name, but it is must have the same datatype and size for the columns. In the query above, we are using the table names to address the columns as the names of the columns are the same in both the tables. We can write the same query by using the table aliases as follows:

```
select * from city ci, country co where ci.country_id =
co.country_id;
```

This was the old-style syntax. The ANSI style syntax is as follows:

```
SELECT city, country FROM city INNER JOIN country ON
city.country_id = country.country_id;
```

Another ANSI style syntax for the same query is as follows:

```
SELECT city, country FROM city INNER JOIN country USING
(country_id);
```

For understanding the concept of natural join, we need to switch to another database by using the following query:

```
USE employees;
```

A natural join is the join taken on the columns that have the same name, data types, and sizes across the sources. The following is a natural join between two tables:

```
SELECT * FROM employees NATURAL JOIN dept_emp;
```

If we describe both the tables, with the queries `desc employees` and `desc dept_emp`, we will find that the column `emp_no` is the matching column in both the tables. The above query is equivalent to the following queries:

```
select * from employees, dept_emp where employees.emp_no =
dept_emp.emp_no;
SELECT * FROM employees INNER JOIN dept_emp ON employees.emp_no
= dept_emp.emp_no;
SELECT * FROM employees INNER JOIN dept_emp USING (emp_no);
```

Outer Joins

In inner join and natural join, the matching rows are included in the resultset. The outer join is the join where even non-matching records are included in the resultset. Two types of outer joins are present in MySQL. The first one is Left outer join where all the records from the left table in the join query are present in the output, and only the matching records from the right table in the query are present in the output. We will again switch to another database with the following query:

```
USE Sakila;
```

We can write the ANSI style query for the left outer join as follows:

```
SELECT
c.customer_id, c.first_name, c.last_name,
a.actor_id, a.first_name, a.last_name
FROM customer c
LEFT OUTER JOIN actor a
ON (c.last_name = a.last_name);
```

We can also write the query as follows:

```
SELECT
c.customer_id, c.first_name, c.last_name,
a.actor_id, a.first_name, a.last_name
FROM customer c
LEFT JOIN actor a
ON (c.last_name = a.last_name);
```

In the output, we can see the `NULL` values corresponding to the non-matching record on the right-hand side as follows:

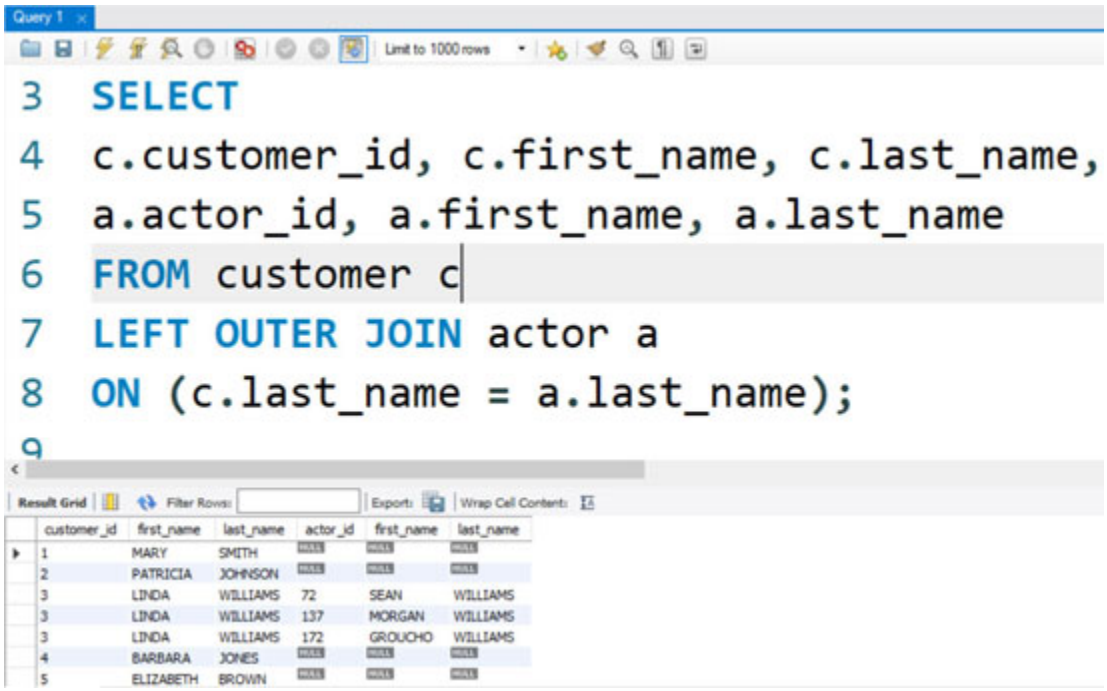


Figure 7.2

Similarly, in the right outer join, only all the records from tables at the right side in the query are shown, and only the matching records from the left side table are shown. We can write the query for right outer join in the following way:

```
SELECT
c.customer_id, c.first_name, c.last_name,
a.actor_id, a.first_name, a.last_name
FROM customer c
RIGHT OUTER JOIN actor a
ON (c.last_name = a.last_name);
```

We can also write the query as follows:

```
SELECT
c.customer_id, c.first_name, c.last_name,
a.actor_id, a.first_name, a.last_name
FROM customer c
RIGHT JOIN actor a
ON (c.last_name = a.last_name);
```

In the output, we can see the `NULL` values corresponding to the non-matching record on the left-hand side as follows:

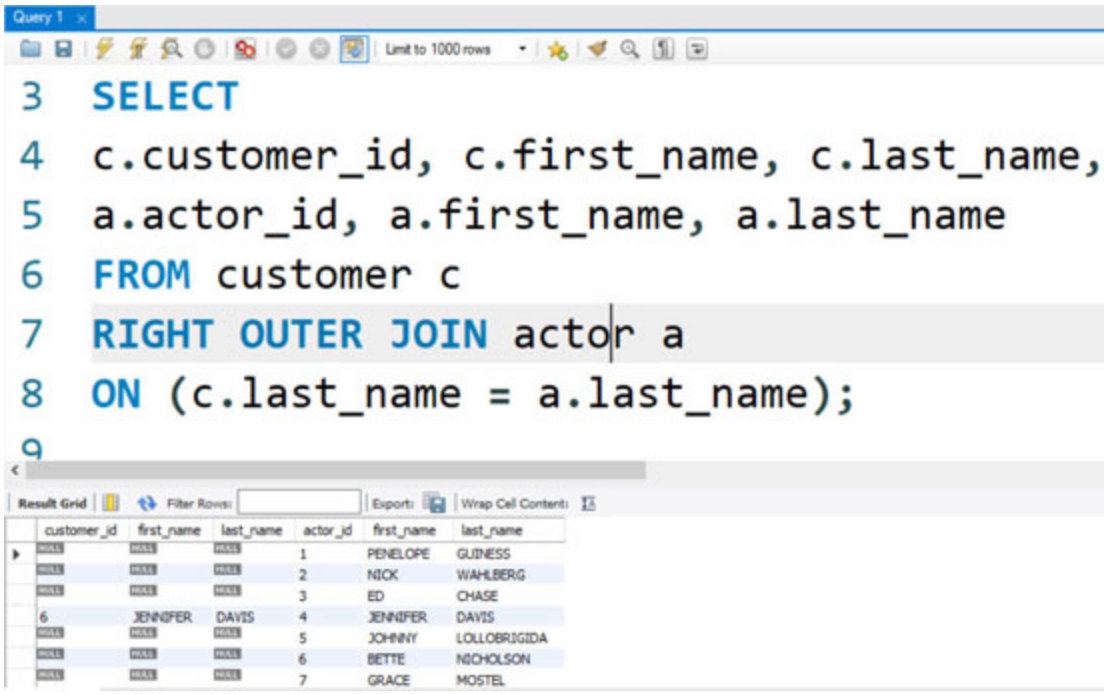


Figure 7.3

Many database products like Oracle database support another type of outer join known as the full outer join. MySQL and MariaDB both do not support this type of join.

Multi-Condition and Multi-Source Join

Till now, we have learned and demonstrated the joins with a single condition for join. We can also have multiple conditions in the join statements as follows:

```
SELECT c.customer_id, c.first_name, c.last_name,
a.actor_id, a.first_name, a.last_name
FROM customer c
Inner JOIN actor a
ON (c.last_name = a.last_name AND c.first_name = a.first_name);
```

In the query above, we can see that there are two conditions for the inner join. Similarly, we can write a query with one join condition and another condition for `WHERE` clause:

```
SELECT * FROM city INNER JOIN country
ON city.country_id = country.country_id
WHERE city.country_id IN ( 1, 4, 7);
```

Also, we have seen the queries with only two tables as sources. We can have multiple sources in the query. The rule is that is we have n sources, then the join has *n-1* conditions. The following is an example of such a query with 3 distinct tables as sources:

```
select first_name, last_name, title from film_actor, actor,
film
WHERE actor.actor_id = film_actor.actor_id AND
film_actor.film_id = film.film_id
```

We can write the same query as follows with the ANSI syntax:

```
select first_name, last_name, title from film_actor
JOIN actor USING (actor_id) JOIN film USING (film_id)
```

Self-Join

In many requirements, we must have a single table or a view act as multiple sources. These types of situations require a special type of join known as the self-join. The following is an example:

```
SELECT a.customer_id, b.customer_id, a.first_name,
b.first_name, a.last_name, b.last_name
FROM customer a
INNER JOIN customer b
ON a.last_name = b.first_name;
```

In the query above, we can see that we are using a single table with two different aliases with an inner join. We can even use outer joins if the situation requires it.

Non-equijoin

We can use other operators in the condition to join. Such joins are known as non-equijoin. Following is a simple example of a non-equijoin:

```
select * from city, country where city_id = 2
AND city.country_id < country.country_id
```

We must use non-equality in certain situations. This join fits those situations perfectly.

Conclusion

In this chapter, we have learned the technique to combine the data from multiple sources. It is known as the join operation. We have learned all the types of joins. We can use joins when we need to combine the data from multiple sources. We can use multiple sources and multiple conditions, too, in the query for join.

In the next chapter, we will learn subquery in detail. We will see multiple types of subqueries and all the scenarios when subqueries are used.

Points to Remember

- Joins combine data from multiple sources.
- Simple and natural joins combine data based on equality.
- Non-equijoins combine data using operators other than equality.
- MySQL and MariaDB do not have Full Outer Join.
- In a multi-source join query, if there are n sources, then there are *n-1* conditions for join.

MCQ

1. What is another name for Simple Join?
   a) Outer Join
   b) Self-Join
   c) Inner Join
   d) Non-equijoin
2. How many conditions in the join are required if the join query has n sources?
   a) n
   b) n+1
   c) n * n
   d) n-1

Answer to MCQ

1. *c)*
2. *d)*

Questions

1. What are the joins? What are the types of joins?
2. Explain Outer Join in detail with examples.
3. How to join multiple tables?
4. Explain non-equijoins with examples.

Key Terms

Joins, Equijoins, Non-Equijoins, Self Joins, outer joins, multi-source joins, Inner join, natural join, simple join, cross join, cartesian product

CHAPTER 8

Subqueries

In the last chapter, we learned the concept of joins. We saw that the joins assist in combining the data from different sources and produce a single resultset which we can use for our analytical requirement.

We also saw the different categories of joins and joins offered by MySQL and MariaDB. We learned that the full outer join is not available in MySQL and MariaDB. We also explored the old theta style and the new ANSI style syntax for writing joins.

In this chapter, we will learn the concept of query within a query. This is also known as subqueries. We will learn and demonstrate this in detail. We will see the various types and applications of the concept of the subquery. Subqueries are widely used in various scenarios. They are used mostly in the `WHERE` clause. They are sometimes used for creating tables too (CTAS queries – create table as which we will see in the next chapter). It is essential to know subqueries to a developer and a data scientist.

Structure

Following is the detailed list of topics we will learn and demonstrate with the SQL queries in this chapter:

- Subquery
- Correlated Subquery

Objective

We will learn and demonstrate all the types of subqueries in detail. After following this chapter, the readers will be able to successfully write various types of subqueries for their data requirements and tell the difference between simple subquery and correlated subquery.

Subquery

Subquery stands for query within a query. It is also known as a nested query. It is usually nested inside `SELECT`, `DELETE`, `INSERT`, or `UPDATE` queries. A subquery can be used in the places where an expression is allowed. Let us see a few examples of the subquery used in the select statement.

We will use the Sakila database for this:

```
USE SAKILA;
SELECT * FROM actor
WHERE actor_id IN
(SELECT actor_id FROM film_actor
WHERE film_id = 2);
```

In the above query, we can see two queries. The main query (also known as an outer query) uses the output generated by the inner query. So, the inner query is executed first, and the output is passed to the `WHERE` clause of the outer query for the further evaluation of the resultset. The output is as follows:

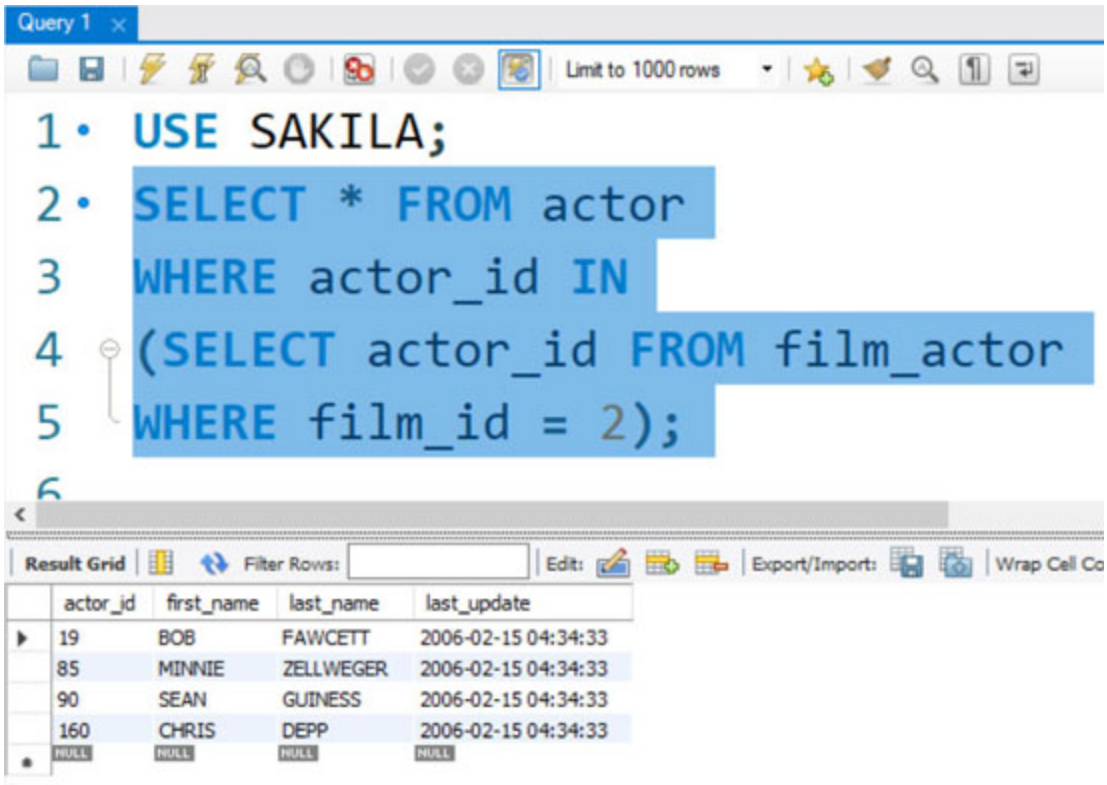


Figure 8.1

The output shows the details of all the actors where the ID of the film in the database is equal to 2.

We can have many levels of subqueries. It is known as nested subqueries:

```
SELECT * FROM actor
WHERE actor_id IN
(SELECT actor_id FROM film_actor
WHERE film_id =
(SELECT film_id FROM film
WHERE title = 'Ace Goldfinger')
);
```

In the query above, we can see that there are two inner queries. The innermost query is executed first, and the outermost query is executed in the end. We can also use a subquery as a derived table as follows:

```
SELECT AVG(a) FROM
(SELECT
customer_id,
SUM(amount) a
FROM payment
GROUP BY customer_id) AS totals;
```

Here, we are using the inner query to create a derived table that acts as the source of data for the outer `SELECT` query.

Correlated Subquery

A correlated subquery is also known as a synchronized subquery. It is a subquery that uses values from the outer query. In the correlated subquery, the outer and inner both the queries are executed at every iteration. That's why it can be very slow. Following is an example of a correlated subquery:

```
SELECT
first_name, last_name,
(SELECT count(*)
FROM film_actor fa
WHERE fa.actor_id = a.actor_id) AS "Total Films"
FROM actor a
```

The output is as follows:

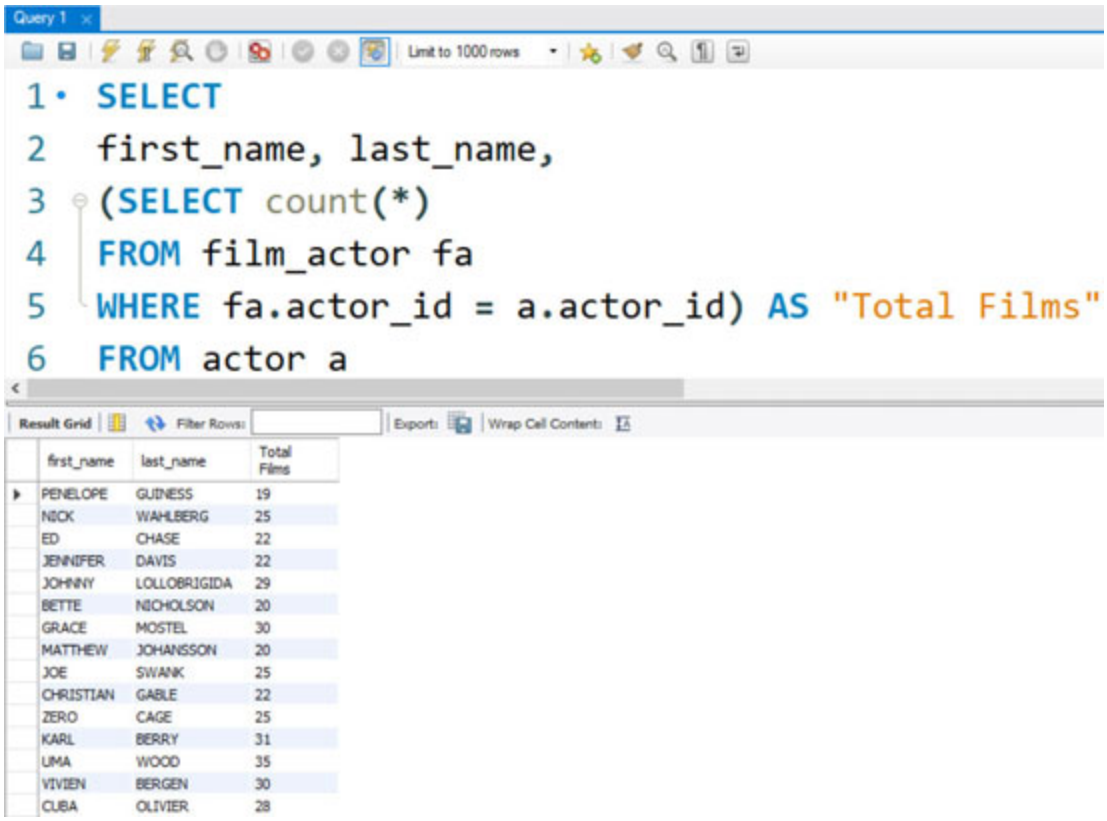


Figure 8.2

Due to the slowness of the correlated subqueries, developers mostly prefer to use joins to fetch the same data in less time. We can read more about it at **https://www.zentut.com/sql-tutorial/understanding-correlated-subquery/.**

Conclusion

In this chapter, we have learned another technique to combine the data from multiple sources. It is known as a subquery. Subqueries are specifically used when the number of rows is very less. Otherwise, it takes a lot of time for the execution. When the number of rows is more, we use joins to combine data from multiple sources.

In the next chapter, we will learn **Data Definition Language (DDL),** and **Data Modification Language (DML)**queries**.**

Points to Remember

- Subqueries combine data from multiple sources.
- Subqueries are slower than joins.
- Subqueries should be used when the number of rows is less.
- A correlated subquery can be very expensive in terms of processing time.

MCQ

1. What is another name for subquery?
   a) Outer Join
   b) Super query
   c) mega query
   d) Nested Query
2. What is another name for correlated subquery?
   a) Synchronized Query
   b) Super query
   c) mega query
   d) Nested Query

Answer to MCQ

1. *d)*
2. *a)*

Questions

1. What is subquery?
2. What is correlated subquery?

Key Terms

Subquery, Correlated Subquery, Nested query

CHAPTER 9

DDL, DML, and Transactions

In the last chapter, we learned and demonstrated the concept of subqueries. We learned the types of subqueries and the scenarios in which we can use subqueries. We have also learned that the subqueries are computationally expensive.

In this chapter, we will learn how to create tables in MySQL and MariaDB. We will also learn how to execute DDL and DML queries on tables. We will learn the basics of transactions and how to commit or rollback them.

Structure

Following is the detailed list of topics we will learn and demonstrate with the SQL queries in this chapter:

- Tables and Data Dictionary
- DDL Statements
- Transactions and DML statements
- The Truncate operation
- Create Table As
- Constraints
- Drop a database

Objective

After following this chapter, we will be able to create our tables in a schema in MySQL or MariaDB. We will be able to

run DDL, and DML queries on the tables. We will also be able to explain the transaction in detail.

We will also see a small application of subquery and how to drop a database.

Tables and Data Dictionary

In relational databases, a table is a collection of data. It is organized as rows and columns. A table is also known as a relation and a row as a tuple. Tables are usually used to store information about similar entities. Suppose that there is an educational institution where there are different groups of people. In the database of the institute, we can have separate tables for the staff, the students, and the faulty.

We can see the list of tables using the information schema as follows:

```
SELECT table_schema, table_name, table_type FROM
information_schema.tables;
```

The following is the output:

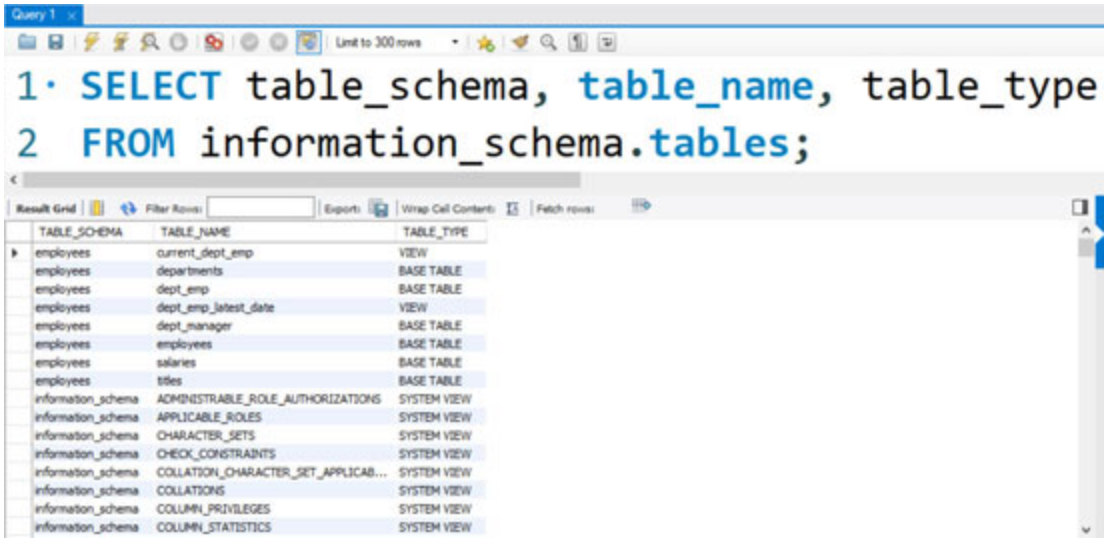


Figure 9.1

Data dictionary tables in MySQL and MariaDB are not visible. So, they cannot be accessed directly with SQL queries.

MySQL and MariaDB support access to data stored in data dictionary tables through `INFORMATION_SCHEMA` tables and `SHOW` statements. We will see how to retrieve the relevant data from the data dictionary as and when needed.

DDL Statements

DDL means Data Definition Language. DDL statements create, modify, or delete database objects. In this chapter, we will learn quite a lot of DDL statements. In this section, we will learn the DDL statements related to the table object.

We will use employees schema to demonstrate the DDLs related to tables:

```
USE employees;
```

The following statement creates a table in the currently selected schema,

```
CREATE TABLE performance_records
(
empno INT,
first_name VARCHAR(20),
last_name VARCHAR(20),
review_date DATETIME,
rating CHAR(5) DEFAULT NULL
)
```

We know that we can use the following statement to select records from the table:

```
SELECT * FROM performance_records;
```

We can see the structure of the table with the following statement:

DESC performance_records;

We can remove the entire table structure and all the records in the table with the following statement:

```
DROP TABLE IF EXISTS performance_records;
```

Now, as the table structure does not exist in the schema, the following statements will return errors when executed:

```
DROP TABLE performance_records;
SELECT * FROM performance_records;
DESC performance_records;
```

You must have observed that we are assigning data types to the columns. We can find a detailed list of data types at the following URLs:

https://www.w3schools.com/sql/sql_datatypes.asp
https://dev.mysql.com/doc/refman/8.0/en/data-types.html

Let us create the original table again:

```
CREATE TABLE performance_records
(
empno INT,
first_name VARCHAR(20),
last_name VARCHAR(20),
review_date DATETIME,
rating CHAR(5) DEFAULT NULL
)
```

We can rename the table object as follows:

```
RENAME TABLE performance_records TO performance_records_backup;
```

Now, again, the following queries on the original table will return errors:

```
DROP TABLE performance_records;
SELECT * FROM performance_records;
DESC performance_records;
```

We can see the structure of the renamed table with the following statement:

```
DESC performance_records_backup;
```

Now, create the original table again with the following query for further demonstrations:

```
CREATE TABLE performance_records
(
empno INT,
first_name VARCHAR(20),
last_name VARCHAR(20),
review_date DATETIME,
rating CHAR(5) DEFAULT NULL
)
```

We can add a column with the following statement and see the changes to the table structure:

```
ALTER TABLE performance_records ADD external_reviewer_name
VARCHAR(5) ;
DESC performance_records;
```

We can alter an existing column as follows:

```
ALTER TABLE performance_records MODIFY external_reviewer_name
VARCHAR(15) ;
DESC performance_records;
```

We can delete an existing column as follows:

```
ALTER TABLE performance_records DROP external_reviewer_name;
DESC performance_records;
```

Let us drop the backup table we created with the following statement:

```
DROP TABLE IF EXISTS performance_records_backup;
```

Transactions and DML Statements

A transaction is a unit of work. Whenever we make any changed to the data in the database, it is known as a

transaction. The statements that cause the transactions to happen are known as **DML** or **Data Modification Language statements.** Before we go any further, we need to disable auto-commit in our session. In the main menu bar of MySQL workbench, go to the menu Query and disable the option Auto-Commit Transactions as shown in the following diagram:

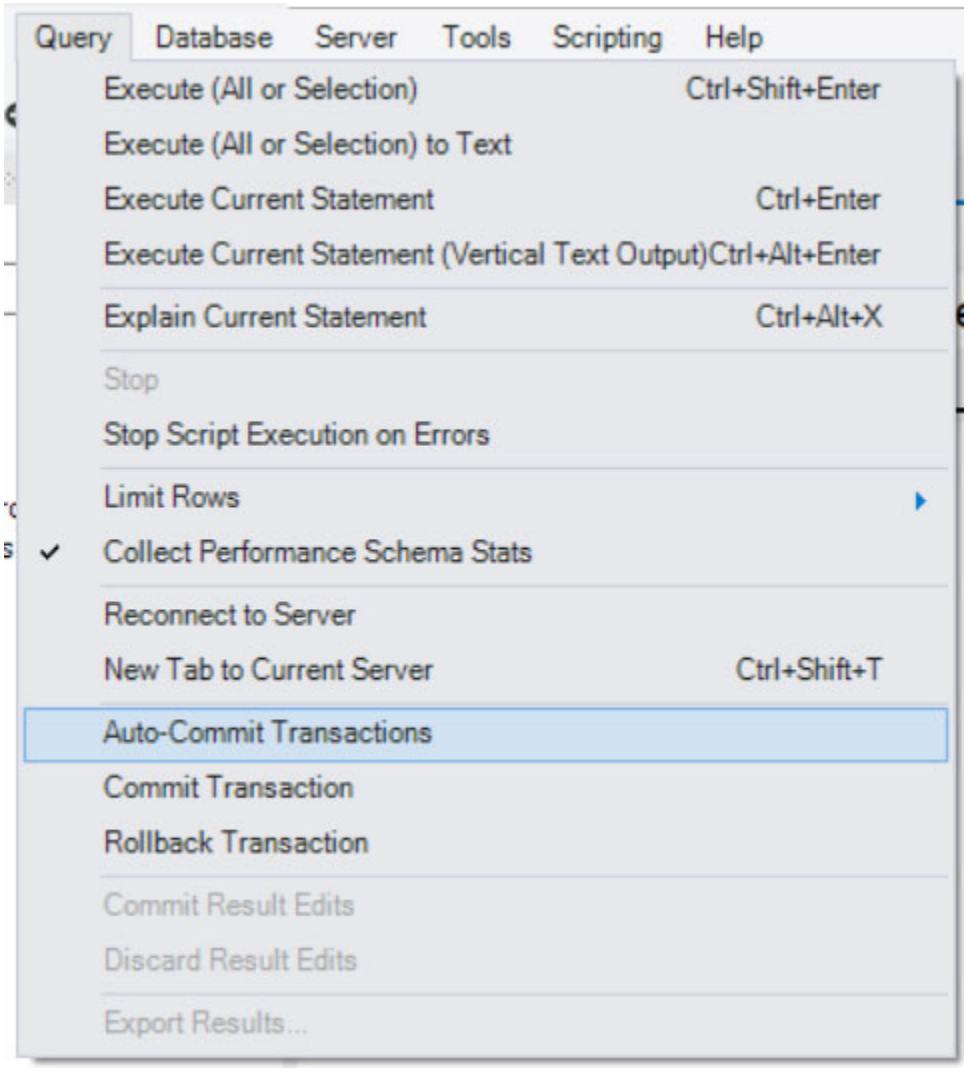


Figure 9.2

We will see what it means soon. The very first DML statement we will learn is `INSERT`. We will use an already existing table for this exercise. Run the following query:

```
SELECT * FROM departments;
```

The query will show the following output:

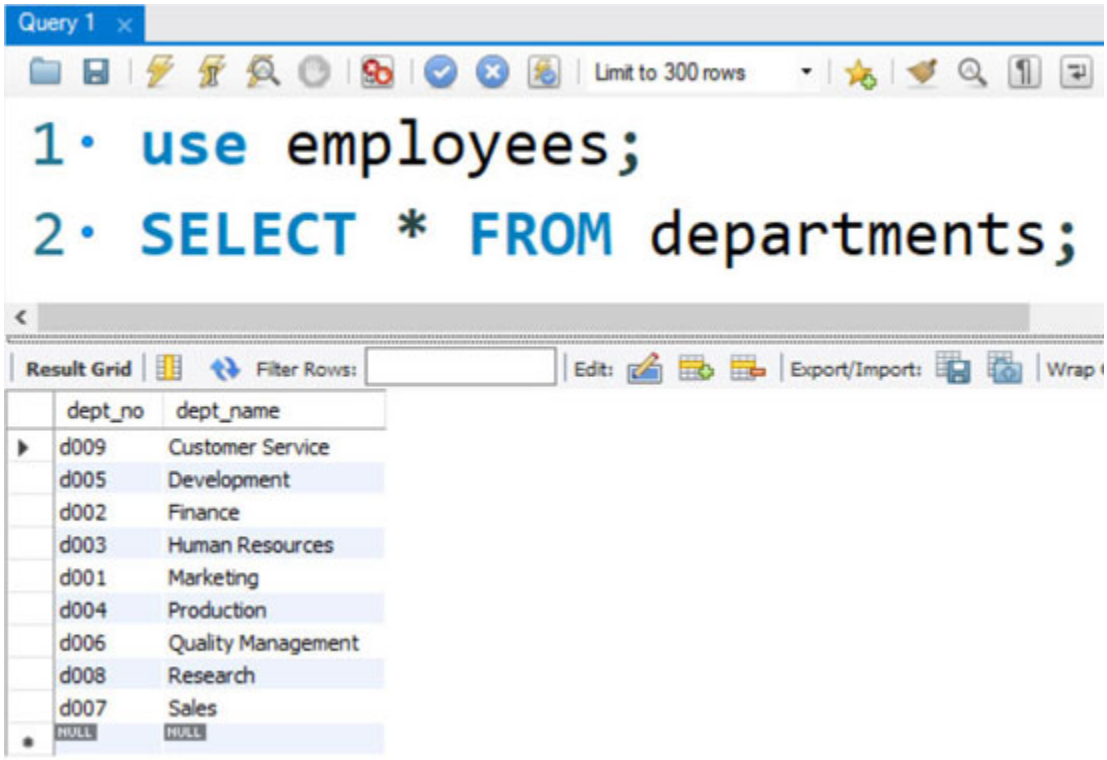


Figure 9.3

We can insert values to the columns using the column name in the statement as follows:

```
INSERT INTO departments (dept_no, dept_name) VALUES ('d099',
'IT');
```

We can insert the values without mentioning column names, but the values are to be in the order of the columns in the table structure. The following is an example:

```
INSERT INTO departments VALUES ('d100', 'Purchase');
```

After running both the queries, run the following statement to see the values in the table:

```
SELECT * FROM departments;
```

We talked about the transaction at the beginning of this section. Whenever we run a DML statement first time in a session, it begins a transaction. A transaction can be completed in two ways. We either commit it or rollback. Whenever we initiate a transaction, if it is not completed, the changes are not made permanent. So, if we try to check the values in the table from some other session, those will not be reflected. If we want to rollback to the state when the transaction began, then we use the following command:

```
ROLLBACK;
```

If we run the select query now, it will not show the added rows. Let us insert the new rows again and commit the changes to make them permanent:

```
INSERT INTO departments (dept_no, dept_name) VALUES ('d099',
'IT');
INSERT INTO departments VALUES ('d100', 'Purchase');
COMMIT;
```

Now, if we see this from another session, we can see the added rows. Thus, a transaction is completed after a commit or a rollback.

We can update existing rows in a table with the `UPDATE` statement. We can update a single row as follows:

```
UPDATE employees SET first_name = 'ASHWIN' WHERE emp_no =
10018;
```

The `WHERE` clause in the query above returns a single row when used in a `SELECT` query. This the query above updates a single row. We can also update multiple rows as follows:

```
UPDATE employees SET birth_date = birth_date + 100 WHERE emp_no
IN (10001, 10002);
```

We can conclude this transaction by either a commit or a rollback operation.

We can delete a single or a few records similarly:

```
DELETE FROM employees WHERE emp_no = 10004;
```

We can even try to delete all the records with the following query:

```
DELETE FROM employees;
```

However, due to the safe update mode, it will not run the query and return an error. To run this, we must change our preferences.

The Truncate Operation

We can delete all the records in a table using a DDL command known as `TRUNCATE` as follows:

```
SELECT * FROM titles;
TRUNCATE TABLE titles;
SELECT * FROM titles;
```

First, we are selecting all the records from the table. Then we are truncating it. Afterward, the `SELECT` statement does not return any row as all the records are truncated. Also, the rollback statement will not restore the changes we made. We can try this as follows:

```
ROLLBACK;
SELECT * FROM titles;
```

The above query will not return anything as the rollback operation does not restore the truncated rows. This is because truncate is a DDL operation, and all the DDL operations are (unlike DML operations) auto-commit in nature.

Create Table As

We will now learn and demonstrate a special DDL statement **CREATE TABLE AS**. It is also abbreviated as **CTAS**. It uses a simple subquery to create a table structure and populate it with existing data. The following is an example:

```
CREATE TABLE EMPLOYEES_BACKUP AS (SELECT * FROM EMPLOYEES)
```

Constraints

Let us study the concept of constraints. Constraints mean limitations. In the context of the area of DBMS, Constraints are the limitations rules enforced on the columns of a table. These rules determine the data that can be stored in tables.

There are two types of constraints. Those are table-level constraints and column level constraints. The column level constraints apply only to the column they are defined for. The table-level constraints are applied to the whole table. We can either have a system named constraints, or we can also name them ourselves.

Following are various constraints we can have at the level of the table or column:

1. **`Primary Key:`** The `Primary Key` is a column or set of columns that is used to identify a record in a table uniquely. The key columns must have unique and not null values.
2. **`NOT NULL:`** This means that the column cannot hold a `NULL` value.
3. **`UNIQUE:`** This means that all the values in the column must be unique. It can have a `NULL` value.
4. **`CHECK`**: The values in a column must be valid for logical expression.
5. **`DEFAULT:`** This means when we enter a record into a table, then the column will have the default value if we do not specify it.

6. **`Foreign Key`**: This means that the column could have a value from the set of values stored in some other column in the same or another tables.

We can refer the following URLs to read and practice the constraint in MySQL:

https://www.w3resource.com/mysql/creating-table-advance/constraint.php

Drop a Database

We can drop a database (schema) with the following command:

```
DROP DATABASE IF EXISTS employees;
```

Just like any drop command, once we drop a database, all the objects in that database are permanently deleted. To work with a dropped database, you need to follow the steps to install it from external sources, as we had discussed earlier.

Conclusion

In this chapter, we have learned and demonstrated the concepts of DDL and DML. We are now comfortable creating a new table, modifying them, and dropping them from the schema.

We also can run various DDL and DML queries on them. We can also explain transactions in detail. We now know how to use a subquery to create a table from an existing table and drop a database.

In the next chapter, we will explore the concept of views. We will learn different types of views in SQL and demonstrate how to create them with MySQL and MariaDB.

Points to Remember

- We should disable auto-commit before working with DML statements.
- DDL statements are auto-commit by default.
- `TRUNCATE` is a DDL command to erase all the rows from a table. It cannot be rolled back.
- We can use subqueries to create a table from existing table structures.

MCQ

1. `TRUNCATE` is a DDL command.

   a) True

   b) False

2. What statement do we use to make the changes made by DML statements permeant to the database?

   a) `ROLLBACK`

   b) `FINAL`

   c) `DONE`

   d) `COMMIT`

Answer to MCQ

1. *a)*
2. *d*)

Questions

1. What is a DDL statement?
2. What is a Transaction? Explain transactions with the help of DML, `COMMIT`, and `ROLLBACK` statements.

Key Terms

`COMMIT`, Transactions, `ROLLBACK`, DDL, DML, CTAS, `TRUNCATE`, Constraints

CHAPTER 10

Views

In the last chapter, we learned the concept of tables, data dictionaries, DDL statements, DML statements, and Transaction control statements. We have seen how to create tables, how to insert, update, and delete data, and how to control transaction. We also learned the basics of the data dictionary in the database.

In this chapter, we will learn and demonstrate another important type of database object. It is known as the concept of view.

Structure

Following is the detailed list of topics we will learn and demonstrate with the SQL queries in this chapter:

- Concept of views
- Simple Views
- Complex Views

Objective

We will learn and demonstrate all the types of views in MySQL and MariaDB. After following this chapter, we will be able to create our views based on the requirement. We will also be able to identify scenarios where we can use views.

Concept of Views

A view is the SQL statement with an associated name. It is a named query stored in the database catalog that allows us to refer to it later when needed. To be very specific, a view is a named virtual table. We will use the database sakila for this chapter:

```
USE SAKILA;
```

In the earlier chapter, we learned about the data dictionaries and catalogs. Let us query the data dictionary for the list of all the views with the following statement:

```
SHOW FULL TABLES IN sakila WHERE TABLE_TYPE LIKE 'VIEW';
```

Following is the output:

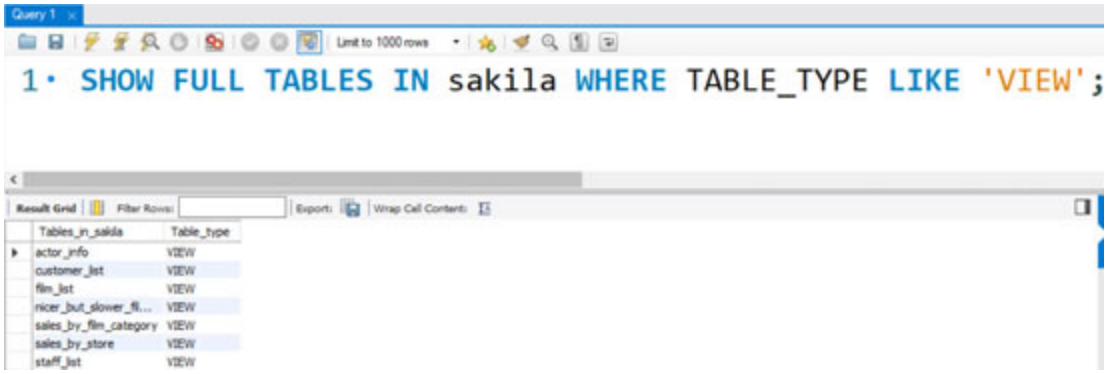
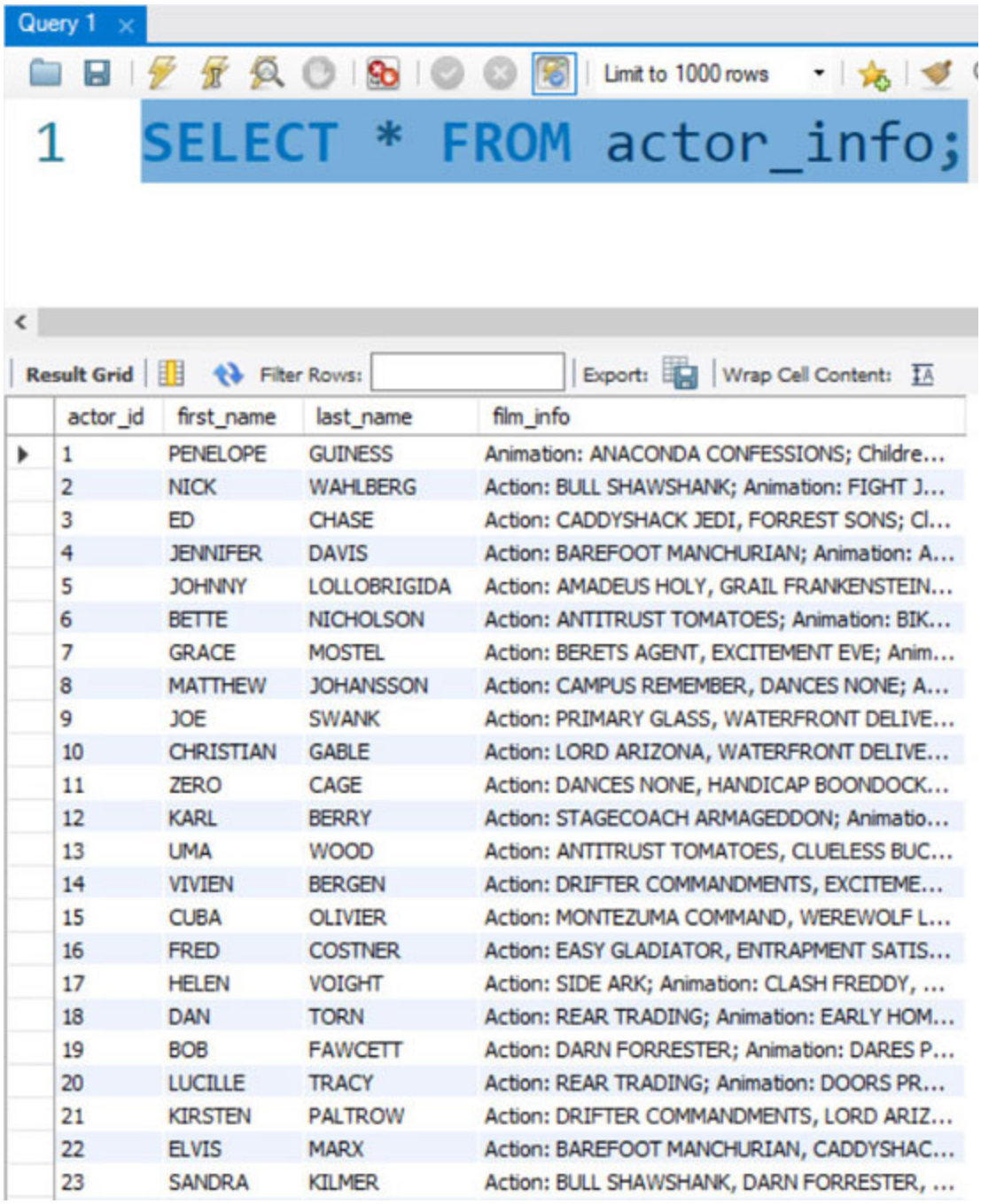


Figure 10.1

This output shows the list of all the views in the current database (schema). We can see the data of a view with the following query:

```
SELECT * FROM actor_info;
```

Following is the output:



actor_id	first_name	last_name	film_info
1	PENELOPE	GUINESS	Animation: ANACONDA CONFESSIONS; Childre...
2	NICK	WAHLBERG	Action: BULL SHAWSHANK; Animation: FIGHT J...
3	ED	CHASE	Action: CADDYSHACK JEDI, FORREST SONS; Cl...
4	JENNIFER	DAVIS	Action: BAREFOOT MANCHURIAN; Animation: A...
5	JOHNNY	LOLLOBRIGIDA	Action: AMADEUS HOLY, GRAIL FRANKENSTEIN...
6	BETTE	NICHOLSON	Action: ANTITRUST TOMATOES; Animation: BIK...
7	GRACE	MOSTEL	Action: BERETS AGENT, EXCITEMENT EVE; Anim...
8	MATTHEW	JOHANSSON	Action: CAMPUS REMEMBER, DANCES NONE; A...
9	JOE	SWANK	Action: PRIMARY GLASS, WATERFRONT DELIVE...
10	CHRISTIAN	GABLE	Action: LORD ARIZONA, WATERFRONT DELIVE...
11	ZERO	CAGE	Action: DANCES NONE, HANDICAP BOONDOCK...
12	KARL	BERRY	Action: STAGECOACH ARMAGEDDON; Animatio...
13	UMA	WOOD	Action: ANTITRUST TOMATOES, CLUELESS BUC...
14	VIVIEN	BERGEN	Action: DRIFTER COMMANDMENTS, EXCITEME...
15	CUBA	OLIVIER	Action: MONTEZUMA COMMAND, WEREWOLF L...
16	FRED	COSTNER	Action: EASY GLADIATOR, ENTRAPMENT SATIS...
17	HELEN	VOIGHT	Action: SIDE ARK; Animation: CLASH FREDDY, ...
18	DAN	TORN	Action: REAR TRADING; Animation: EARLY HOM...
19	BOB	FAWCETT	Action: DARN FORRESTER; Animation: DARES P...
20	LUCILLE	TRACY	Action: REAR TRADING; Animation: DOORS PR...
21	KIRSTEN	PALTROW	Action: DRIFTER COMMANDMENTS, LORD ARIZ...
22	ELVIS	MARX	Action: BAREFOOT MANCHURIAN, CADDYSHAC...
23	SANDRA	KILMER	Action: BULL SHAWSHANK, DARN FORRESTER, ...

Figure 10.2

We can see the underlying query for the view with the following two statements:

```
SHOW CREATE VIEW actor_info;

SELECT VIEW_DEFINITION
```

```
FROM INFORMATION_SCHEMA.VIEWS
WHERE TABLE_SCHEMA = 'sakila'
AND TABLE_NAME = 'actor_info';
```

We will learn how to create our views like this in the next two sections.

Simple Views

Simple views do not apply any type of transformation to the columns in the underlying query. The following query creates a simple view:

```
CREATE OR REPLACE VIEW address_view AS SELECT address,
postal_code FROM address;
```

The above query shows two columns from a table. We can see the data from the view with the following query:

```
SELECT * FROM address_view;
```

We can even redefine the view as follows,

```
CREATE OR REPLACE VIEW address_view AS SELECT address,
postal_code FROM address WHERE postal_code IS NOT NULL;
```

In the query above, we have added a `WHERE` clause to the view. Simple views are mostly created from one table and do not have a complex query. It does not have to join. Simple views do not contain group functions, groups of data, `GROUP BY`, and `DISTINCT`. We can perform DML operations on the underlying base tables in a simple view. We can select the data from the view with a simple `SELECT` query.

Complex Views

Complex views are the views that can have complex underlying queries with joins, group functions, groups of data, `GROUP BY`, and `DISTINCT`. We cannot perform DML

operations on underlying tables. The following is an example of a complex view:

```
CREATE OR REPLACE VIEW city_country_info AS SELECT city,
country FROM city INNER JOIN country USING (country_id);
```

We can select the data from the complex view with a simple `SELECT` query as follows:

```
SELECT * FROM city_country_info;
```

We can see the underlying query of the complex view with the following statement:

```
SHOW CREATE VIEW city_country_info;
```

Conclusion

In this chapter, we have learned the theory and demonstrations of views. We have seen different types of views and data dictionaries associated with views. We have learned that the views are virtual tables created in lieu of very long and frequently used queries to save the time to write the queries from scratch. We can use the views as data sources for other queries too. Now, we are all comfortable with views and can create one when needed.

In the next chapter, we will learn how to use MySQL and MariaDB databases with Python and Pandas.

Points to Remember

- A very big and complex `SELECT` statement can be converted into a view, so we do not have to type in that statement anymore. We can just run a `SELECT` query on the view.
- Simple views allow DML operations on the underlying table.

- Complex views do not allow DML operations on the underlying table.

MCQ

1. Does a simple view allow DML operation on the underlying table?

   a) Yes

   b) No

2. Does a complex view allow DML operation on the underlying table?

   a) Yes

   b) No

3. A Simple view does not allow one of the following in the underlying query.

   a) `WHERE`

   b) `ORDER BY`

   c) `JOIN`

   d) `Column Alias`

Answer to MCQ

1. *a)*
2. *b)*
3. *c)*

Questions

1. What is a view?
2. Prepare a table of differences between simple and complex views.

Key Terms

View, Simple View, Complex View

CHAPTER 11

Python 3, MySQL, and Pandas

In the last chapter, we learned and demonstrated the concept of views. We have seen that views are virtual tables, and we create them when we have a repetitive query. We also have learned that there are two categories of views, simple and complex. We also looked at a few scenarios where we can use views.

Python programming language is one of the prevalently used programming languages of our era. Python is one of the most preferred languages for data science.

In this chapter, we will learn how to interface Python 3 programming language with MySQL and MariaDB. We will also learn how to interface it with a popular data science library in Python 3, known as Pandas. This will give all the readers a fair idea about the role of relational databases play in the domain of Data Science.

Structure

Following is the detailed list of topics we will learn and demonstrate with the SQL queries in this chapter:

- Installation of Python 3
- Running a Python 3 Program
- MySQL and Python 3
- MySQL and Pandas

Objective

After following this chapter, we will be able to install Python 3 on Windows OS. We will also be able to run Python 3 programs from the command prompt of the OS and using the IDLE editor. We will learn the basic features of IDLE and the interactive mode of Python 3. We will also be able to interface MySQL with Python 3 and load the records in a table to a Pandas dataframe.

Installation of Python 3

Python is the most popular programming language as of the writing of this book (the 2020s). And it is going to be one of the most heavily used programming languages in the foreseeable future too. We can check the popularity of programming languages on the web at the following URLs:

- **http://pypl.github.io/PYPL.html**
- **https://www.tiobe.com/tiobe-index/**

Python programming language has two major flavors, Python 2 and Python 3. From the beginning of 2020, Python 2 has been discontinued, and only Python 3 remains under active development.

All the Linux distributions come with Python 3 pre-installed. We just have to install an Editor for Python programming. **Integrated Development and Learning Environment (IDLE)** is one of the most popular Editor for Python 3, and it comes pre-installed in many Linux distributions. We can install it on Debian and derivatives by executing the following command on the command prompt:

```
sudo apt-get install idle3
```

This will install IDLE3, which is the version of IDLE3 for Python 3. We can often find it on the menu of Linux

distribution or shortcuts. We can also launch it from the command prompt of Linux with the following command:

```
idle
```

We will see how to work with IDLE later. Now, let us install Python 3 on Windows. Python 3 does not come pre-installed on Windows. We have to visit the Python's homepage www.python.org and download the installable file from the download section. The website detects the operating system and presents the correct choice for the download of the installable of the latest version of Python 3. As of writing of this book, the version is 3.8.3. It will be something else in the future, but the process to install will not be much different. Following is the screenshot of the download page:

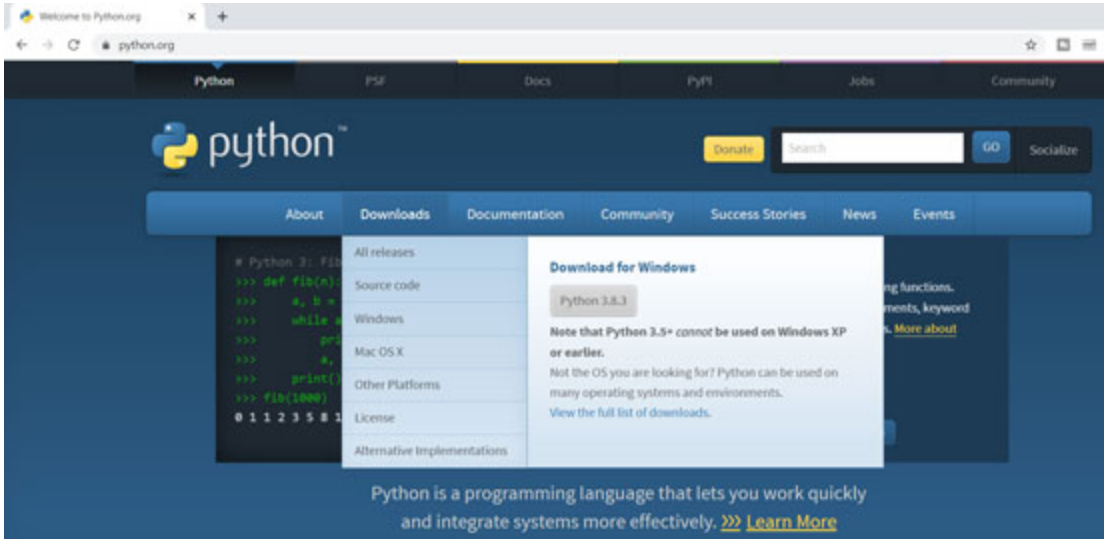


Figure 11.1

Once the download finishes, we can find the downloaded installation file in the Downloads directory of our user on Windows. Double click it to launch it, and it will show the following window:

Figure 11.2

At the bottom of the window, there are two checkboxes. Check both of them by clicking them. The first checkbox installs launcher, and the second adds Python 3 to the PATH variable of Windows. Then, click the option **`Install Now`**. It will ask us for admin credentials, provide it to continue the installation. The installation will proceed forward and will show a success window once it is completed.

Running a Python 3 Program

Python 3 has two modes to run statements. The first one is the interactive mode. This is analogous to the command prompt of the operating system. We can type Python 3 statements on the Python 3 prompt and run them one by one. There are two ways to invoke the interactive mode. We can go the command prompt of the OS (cmd in Windows and lxterminal in Linux) and run the command python3 on Linux and python on Windows to launch Python 3 in interactive mode. The following is a screenshot of Python 3

running in the interactive mode in the command prompt in a Raspberry Pi:

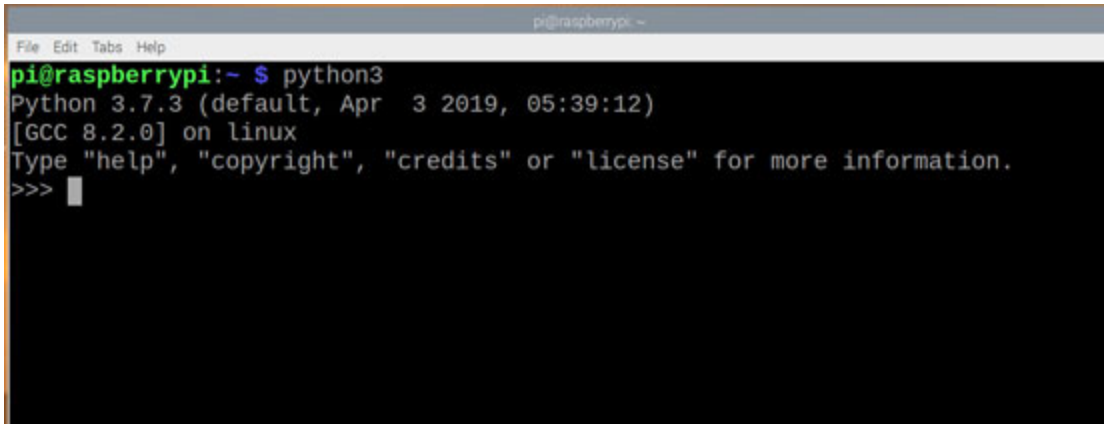


Figure 11.3

The other way is to launch IDLE. We have already seen it earlier on how to launch it on the favors of Linux. We can launch it on Windows by searching for it in the Windows search box.

The interface of the interactive mode of Python is the same in the command prompt and IDLE across various operating systems. The following is a screenshot of IDLE in an interactive mode:

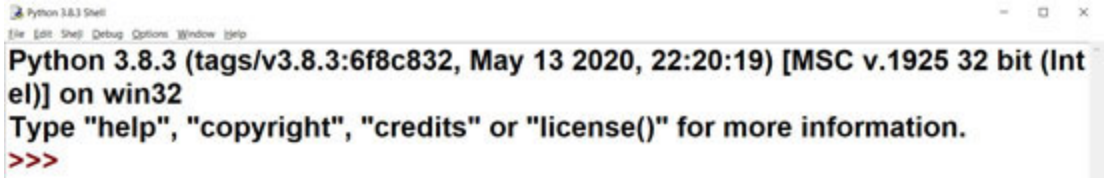


Figure 11.4

We can type in statements here one by one and press enter to execute them. Type in the following statement in the interactive prompt and press *Enter* key on the keyboard to run it:

```
print('Hello World!')
```

The output is as follows:

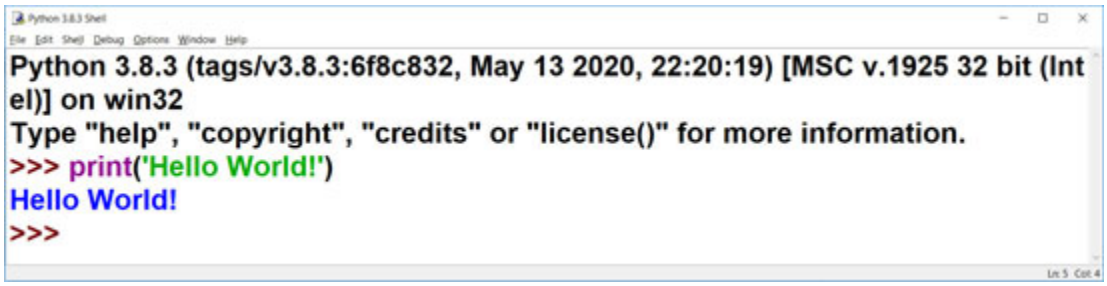


Figure 11.5

The other mode is known as script mode. In this mode, we create a file and save it with .py extension (actually, the extension does not matter as long as it has Python 3 code). Then we add code statements to that. We can do it using IDLE or any other editor of our choice. In IDLE, click **`File > New File`** from the menu. It creates a new blank file. Save it as `prog00.py`. We can add the statement we executed earlier to this file, save it, and run this using **`Run > Run Module`** from the menu. We can also run it by pressing the *F5* key on the keyboard. It will execute in the interactive prompt of IDLE, and the following is the output of the execution:

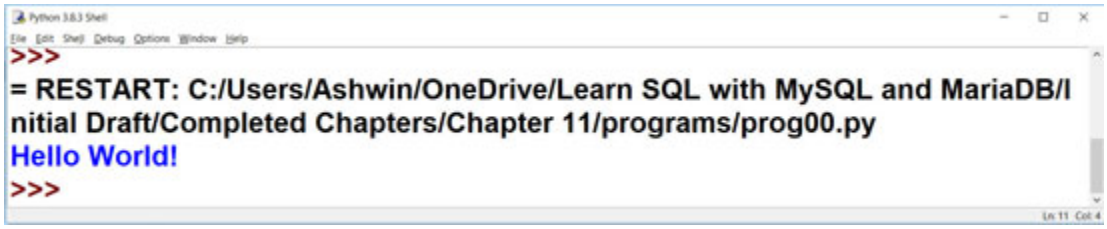


Figure 11.6

We can also launch the program from the command prompt using the Python 3 executable. From the command prompt traverse to the directory where the program is stored and run the following command in cmd in Windows:

```
python prog00.py
```

In Linux, we have to run the following command in lxterminal:

```
python3 prog00.py
```

We can also run the program from any location in the command prompt if we provide the absolute path of it to the Python 3 executable as follows:

```
python "C:\Users\Ashwin\OneDrive\Learn SQL with MySQL and MariaDB\Initial Draft\Completed Chapters\Chapter 11\programs\prog00.py"
```

In Linux, we can run the following command:

```
python3 /home/pi/book/chapter11/prog00.py
```

It will run the program on the command prompt of the OS and print the output.

MySQL and Python 3

We can run MySQL or MariaDB statements from Python 3 programs. For that, we need to install a library known as `pymysql`. We can install it with **`pip`**. `pip` is the package manager of Python, and it means `pip` installs packages (or `pip` installs Python). It is a recursive acronym. We can run the following command on the command prompts of the operating systems to install pymysql and other needed package cryptography:

```
pip3 install pymysql
pip3 install cryptography
```

Once the installation is done, create a new user and grant all the privileges on all the schemas to it using the following SQL statements:

```
CREATE USER 'testuser'@'localhost' IDENTIFIED BY 'test123';
GRANT ALL PRIVILEGES ON *.* TO 'testuser'@'localhost';
```

Now, we can write the following Python 3 program to test the connection:

```
import pymysql
```

```
db =
pymysql.connect("localhost","testuser","test123","sakila",3306)
cursor = db.cursor()
sql = "SELECT VERSION()"
cursor.execute(sql)
data = cursor.fetchone()
print("Database version : %s" % data)
db.close()
```

In the first line, we are importing the needed `pymysql` library. The function `pymysql.connect()` accepts, the `hostname`, `username`, `password`, `database name`, and the port number as arguments and returns a connection object. With the returned object we are creating a cursor object by calling the function `db.cursor()`. We are using the same cursor to execute a string containing a SQL query with the statement `cursor.execute(sql)`. We can then fetch the result using the statement `cursor.fetchone()`. After fetching a record from the resultset and displaying, we are closing the connection with the code `db.close()`.

The program above fetches the version of the database and displays on the interactive prompt of Python 3.

The following is the screenshot of the output:

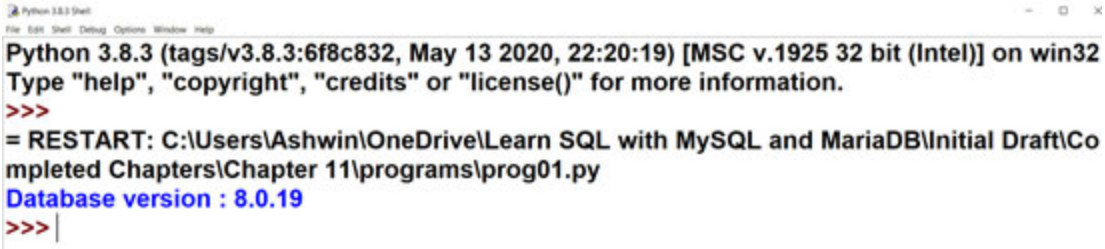


Figure 11.7

Now, let us see an example of a DDL statement with Python 3 as follows:

```
import pymysql
```

```
db =
pymysql.connect("localhost","testuser","test123","sakila",3306)
cursor = db.cursor()
try:
 sql = """CREATE TABLE TEST_TABLE (
 FIRST_NAME CHAR(20) NOT NULL,
 LAST_NAME CHAR(20) NOT NULL,
 AGE INT,
 SEX CHAR(1),
 INCOME FLOAT)"""
 cursor.execute(sql)
 print("The table created successfully!")
except:
 print("The table already exists in the database!")
db.close()
```

In the program above, we are running a DDL statement. If the statement runs successfully, then it shows the success message. If we try to run the program twice, it already finds the object in the database and returns the error message.

We can insert the records to this table with the following code:

```
import pymysql
db =
pymysql.connect("localhost","testuser","test123","sakila",3306)
cursor = db.cursor()
sql1 = """INSERT INTO TEST_TABLE (FIRST_NAME,
 LAST_NAME, AGE, SEX, INCOME)
 VALUES('Ashwin', 'Pajankar', 32, 'M', 1000)"""
sql2 = """INSERT INTO TEST_TABLE (FIRST_NAME,
 LAST_NAME, AGE, SEX, INCOME)
 VALUES('Thor', 'Odinson', 35, 'M', 2000)"""
sql3 = """INSERT INTO TEST_TABLE (FIRST_NAME,
 LAST_NAME, AGE, SEX, INCOME)
 VALUES('Tony', 'Stark', 40, 'M', 40000)"""
```

```
sql4 = """INSERT INTO TEST_TABLE (FIRST_NAME,
 LAST_NAME, AGE, SEX, INCOME)
 VALUES('Jane', 'Foster', 32, 'F', 3000)"""
try:
 cursor.execute(sql1)
 cursor.execute(sql2)
 cursor.execute(sql3)
 cursor.execute(sql4)
 db.commit()
 print("The records inserted successfully!")
except:
 db.rollback()
 print("The records were not inserted!")
db.close()
```

In the program above, we can commit the transaction if all the inserts are successful otherwise we can rollback.

We can fetch the records from the table as follows:

```
import pymysql
db =
pymysql.connect("localhost","testuser","test123","sakila",3306)
cursor = db.cursor()
sql = "SELECT * FROM TEST_TABLE"
try:
 cursor.execute(sql)
 resultset = cursor.fetchall()
 for row in resultset:
  fname = row[0]
  lname = row[1]
  age = row[2]
  sex = row[3]
  income = row[4]
  print("%s %s, %d, %s, %d" %(fname, lname, age, sex, income))

print("The records fetched successfully!")
```

```
except:
 print("The records were not fetched!")
db.close()
```

In the program above, we are using `cursor.fetchall()` to fetch all the records into a variable and then loop over that variable to print all the data.

We can update the records as follows:

```
import pymysql
db =
pymysql.connect("localhost","testuser","test123","sakila",3306)
cursor = db.cursor()
sql = "UPDATE TEST_TABLE SET INCOME = INCOME + 500 WHERE SEX =
'%c'" % ('F')
try:
 cursor.execute(sql)
 db.commit()
 print("The records updated successfully!")
except:
 db.rollback()
 print("The records were not updated!")
db.close()
```

We are creating a string for the update query and running the string with `cursor.execute()`.

Similarly, we can delete the records as follows:

```
import pymysql
db =
pymysql.connect("localhost","testuser","test123","sakila",3306)
cursor = db.cursor()
sql = "DELETE FROM TEST_TABLE"
try:
 cursor.execute(sql)
 db.commit()
 print("The records deleted successfully!")
```

```
except:
 db.rollback()
 print("The records were not deleted!")
db.close()
```

And we can even drop a table as follows:

```
import pymysql
db =
pymysql.connect("localhost","testuser","test123","sakila",3306)
cursor = db.cursor()
sql = "DROP TABLE TEST_TABLE"
try:
 cursor.execute(sql)
 print("The table dropped successfully!")
except:
 print("The object does not exist!")
db.close()
```

MySQL and Pandas

Pandas is a very popular data science library for Python 3. We can install it by running the following command on the command prompt of OS:

```
pip3 install pandas
```

Pandas is a very big topic, and it is difficult to explore it in a small chapter. We can explore the library by visiting the webpage of the project located at the URL **https://pandas.pydata.org**.

Let us write a simple Python program to read a MySQL table into a Pandas data structure known as dataframe. Let us see a very simple example:

```
import pymysql
db = pymysql.connect("localhost", "testuser", "test123",
"world",3306)
```

```
import pandas as pd
df1 = pd.read_sql('select * from country', db)
print(df1)
```

In the program above, we are importing pandas and reading the data from a SQL query with the function `pd.read_sql()`. The read data is loaded into a pandas dataframe which we are printing on the Python 3 prompt.

Conclusion

In this chapter, we have learned how to run simple Python programs. We have learned and demonstrated the interaction between Python 3 and MySQL. We have learned how to connect to a schema and how to execute various statements through Python 3 programs. We have also seen how to load records from a table into a Pandas dataframe. We can conveniently use MariaDB in place of MySQL. The Python 3 code will be the same. We just need to make changes to the connection parameter to connect to MariaDB.

Points to Remember

- We can run MySQL queries from a Python 3 program,
- We can load records of a table into a Pandas dataframe.
- We need to use `pymysql` library to connect Python 3 to a MySQL or a MariaDB database.

MCQ

1. What library do we use to connect MySQL/MariaDB to Python 3?

   a) `pymysql`

   b) `pandas`

c) `numpy`

d) `scipy`

2. Can we perform DMLs on MySQL/MariaDB using Python 3?

   a) Yes

   b) No

Answer to MCQ

1. *a)*
2. *a)*

Questions

1. Explain the difference between script mode and interactive modes of Python 3?
2. How can we load the records of a table in a MySQL instance into a Pandas dataframe?

Key Terms

Python 3, Pandas, MySQL, MariaDB, pymysql