

LEONARD BASE

SQL CODING FOR BEGINNERS

**A SMART GUIDE FOR ABSOLUTE BEGINNERS TO
LEARN SQL DATABASE ANAD SERVER. LEARN
IT FASTER AND REMEMBER IT LONGER.**

SQL Coding For Beginners

*A Smart Guide For Absolute Beginners To Learn SQL
Database And Server. Learn It Faster And Remember
It Longer*

Leonard Base

Table of Contents

[introduction](#)

[Chapter 1](#)

[Introduction To Sql And Data Definition Language](#)

[Chapter 2](#)

[Creating A Database Using Sql And Maintaining Data Integrity](#)

[Chapter 3](#)

[Sql Joins And Union Commands](#)

[Chapter 4](#)

[Sql Views And Transactions](#)

[Chapter 5](#)

[Database Security And Administration](#)

[Conclusion](#)

Introduction

Congratulations on downloading SQL Coding For Beginners.

The following chapters will discuss the fundamental concepts of the “structured query language (SQL)” to help the beginners looking to learn and master this analytical language for the “relational database management systems”. This book will provide you everything you need to know to efficiently and effectively use “SQL” for retrieving and updating information on SQL databases and servers, primarily focusing on the MySQL server, which is one of the highly touted interface for relational database management.

In the first chapter of this book titled “Introduction to SQL and the data definition language”, you will start with an overview of the database management systems along with the different types of database management systems and their advantages. You will be introduced to the SQL language and the five fundamental types of SQL queries namely, “Data definition language (DDL)”, “Data manipulation language (DML)”, “Data control language (DCL)”, “Data query language (DQL)” and “Transaction control language (TCL)”. This chapter will also provide you with the thumb rules required to build SQL syntax or query, which pertains to the chord structure that can be processed by the database server. You will also learn the most commonly used data types, which are integral to their learning of SQL. We will kick start the actual programming language learning with the “SQL CREATE” statements and continue to build up on these concepts. The vital concept of SQL constraints used with the “SQL ALTER” statements is also explained in detail with multiple examples and hands-on exercises.

In the chapter 2 titled, “Creating a database using SQL and maintaining data integrity”, you will be introduced to “MySQL”, which is a “free and open source relational database management system”. There are a variety of user interfaces available with “MySQL” servers including “MySQL workbench”, “phpMyAdmin”, “Adminer”, “ClusterControl”, “Database workbench”, “DBeaver”, “DBEdit”, “HeidiSQL”. “Sequel Pro”, “Toad” among others; which are used by anyone looking to execute SQL queries on the MySQL database server. You will be walked through the process of installing MySQL on your operating system(s). Since, this is an open and free program you can easily download and install it on your system, so you can make the best use

of this book and included hands-on exercises and example to effectively learn the SQL programming language. You will be learning how to generate a whole new database and subsequently create tables and insert data into those tables on the “MySQL” server. You will also learn the concept of temporary tables, derived tables and how you can generate a new table from a table already present in the database.

In the chapter 3 titled, “SQL Joins and Union commands”, you will be introduced to the “SQL SELECT” statement along with the various Data manipulation clauses including “ORDER BY”, “WHERE”. The key concept of SQL Joins is presented in detail including different “SQL JOIN” functions such as “INNER JOIN”, “LEFT JOIN”, “RIGHT JOIN”, “CROSS JOIN” and “SELF JOIN”. The syntaxes of these clauses are explained in exquisite detail the with examples and pictures of the result set that you can expect to obtain while performing hands-on execution of the examples on your own MySQL instance. This effort has been made to bolster your understanding and allow hands on practice of the SQL programming language. In this chapter, you will also learn about the “SQL union” function that is used to collate a number of outputs into a single comprehensive result set. The “MySQL UNION” and “MySQL UNION ALL” statements are presented in detail along with the distinction between MySQL join and union functions.

In the chapter 4 titled, “SQL Views and Transactions”, you will learn the concept of “Database View” in SQL, which is simply a virtual table defined using the “SQL SELECT” statements with the “SQL JOIN” clause(s). There are a wide variety of advantages of using the “Database View” or “SQL View” as explained in this chapter. The “CREATE VIEW” statement is explained along with the underlying processing algorithms used in MySQL such as, “MERGE”, “TEMPTABLE” and “UNDEFINED”. You will also learn how to create a view from another existing view which may or may not include a sub query. The concept of “Updatable SQL Views” is also explained with examples and hands on exercises, so you can learn how to modify SQL views using “ALTER VIEW” and “CREATE OR REPLACE VIEW” statements. In SQL, transaction is defined as “a unit of work that is performed against a database”. The different properties of SQL transactions and various SQL transaction statements with controlling clauses such as, “START TRANSACTION”, “COMMIT”, “ROLLBACK” among others are included in this chapter. Another important concept of database recovery

models is explained here along with different types of recovery models and “SQL BACKUP” statements. The process of restoring database in case of a failure can be crucial to maintain the valuable data, as you will learn in this book.

The last chapter titled, “Database Security and Administration”, will primarily focus on the user access privileges to manage and secure the data on a MySQL server. You will be given a step-by-step walk-through on how to create new user accounts, update the user password as needed, grant and revoke access privileges to ensure that only permitted users have authorized and required access to the database.

There are plenty of books on this subject on the market, thanks again for choosing this one! Every effort was made to ensure it is full of as much useful information as possible, please enjoy!

Chapter 1

Introduction to SQL and Data Definition Language

A database could be defined as “an organized collection of information or data, which can be stored electronically on a computer system and accessed as needed”. A Database Management System (DBMS) is defined as “a software that interacts with the end users, applications and the database itself to capture and analyze the data”. “DBMS” is an integrated computer software package enabling users to communicate with a number of databases and offering access to the information stored on the database. The DBMS offers multiple features which enable big volumes of data to be entered, stored and retrieved as well as offers methods for managing how the data can be organized. Considering this close association, the term “database” is frequently used to refer to both databases as well as “DBMS”.

Existing DBMSs provide multiple features to manage and classify a database and its information into four primary functional groups as follows:

- **Data definition** – Definitions that represent the organization of data are created, modified and eliminated.
- **Update** – Inserting, modifying and deleting the actual data.
- **Retrieval** - Provision of data in a form that can be used directly or processed further by other applications. Data collected may be accessible in the same form as the one in the database or in a different form, acquired through alteration or combination of data sets from various databases.
- **Management** – User registration and tracking, data security enforcement, performance tracking, data integrity, competence control, and the recovery of data damaged by a certain event, for example, an accidental glitch in the system.

A database and its DBMS are in accordance with a given model database. "Database system" can be defined as a collection of the database, DBMS and database model.

Database servers are physically connected computers which only run DBMS and associated software, holding real databases. Multiprocessor machines

with extensive memory and “RAID disk arrays” used for long term storage of the data are called as "database servers". If one of the disks fails RAID can be used to recover the data. In a heavy volume transaction processing setting, “hardware database accelerators” linked to single or multiple servers via the high-speed channel are being utilized. Most database applications have DBMS at their core. DBMS may be created with built-in network assistance around a custom multi-tasking kernel, but contemporary DBMSs typically use a normal operating system to offer the same functionalities.

As DBMS represents an important market, DBMS requirements are often taken into consideration by computers and storage providers in their own production plans.

“Databases” and “DBMSs” can be grouped by on the basis of following parameters:

- The supported database models, for example, relational or XML databases.
- The variety of computers on they can operate on, for example, “server cluster to a mobile device”.
- The query language in use, for example, “SQL” or “XQuery”.
- The “internal engineering” which drives the efficiency, scalability, resiliency, and safety of the system.

We will be primarily focusing on relational databases in this book, which pertains to the SQL language. In the beginning of the 1970s, "IBM" began to work on a prototype model based on the ideas of the computer scientist named Edgar Codd as “System R”. The first version was introduced in 1974 and 1975 and followed by work on multi-table systems that allowed splitting of the data so as to avoid storage of all the data as a single big slice for individual record. Customers tested subsequent multi-user versions in 1978 and 1979, after a standardized query language called "SQL", was added to the database. Codd's concepts forced “IBM” to create a real version of “System R”, known as “SQL / DS” and subsequently “Database 2 (DB2)”.

The "Oracle database” developed by Larry Ellison, began from a distinct chain on the basis of the work published by “IBM” on “System R”. Whilst the implementation of "Oracle V1" was finished in 1978, but it was with the release of "Oracle Version 2" in 1979 that Ellison succeeded over “IBM” in entering the market.

For the further development of a new database, "Postgres", now referred to as "PostgreSQL", Stonebraker had used the "INGRES classes". "PostgreSQL" is widely used for worldwide mission critical apps. The registrations of '.org' and '.info' domain names are primarily using "PostgreSQL" for data storage and similarly done by various big businesses and financial institutions. The computer programmers at the "Uppsala University in Sweden", were also inspired by Codd's work and developed "Mimer SQL" in the mid-1970s. In 1984, this project was transitioned into an independent company.

In 1976, the "entity relationship model" was developed and became highly popular for designing databases as it produced a relatively better known description compared to the preceding "relational model". Subsequently, the structure of the entity-relationship was rebuilt as "a data modeling construct for the relational model", and the distinction between the two became insignificant.

"Relational databases" consist of a collection of tables that can be matched to a predetermined category. "Each table has at least one data category in a column, and each row has a certain data instance for the categories which are defined in the columns". Relational databases have multiple names such as "Relational Database Management Systems (RDBMS)" or "SQL databases". The "Microsoft SQL Server", "Oracle Database", "MySQL" and "IBM DB2" have historically been the most popular of Relational databases. The "Relational databases" are mainly used in big corporate settings with the exception of "MySQL", which can also be used in web-based data storage.

All relation databases can be utilized for management of "transaction oriented applications (OLTP)". On the other hand, most non-relational databases in the classifications of "Document Stores" and "Column Store", may also be used for OLTP, thus causing confusion between the two. OLTP databases can be considered "operational" databases, distinguished by regular quick transactions, including data updates, small volumes of data, and simultaneously processing hundreds and thousands of transactions, such as, online reservations and banking apps.

Advantages of Database Management System

The database management system (DBMS) is described as “software system that enables users to identify, develop, maintain and regulate access to the database”. DBMS allows end customers to generate information in the database as well as to read, edit and delete desired data. It can be viewed as a layer between the information and the programs utilizing that data.

DBMS offers several benefits in comparison to the "file-based data management system". Some of these advantages are listed below:

- **Reduction in the redundancy of data** - The “file based DBMS” contain a number of files stored in a variety of location on a system and even across multiple systems. Due to this, several copies of the same file can often result in data redundancy. This can be easily avoided in a database since there is only a single database holding all the data and any modifications made to the database are immediately reflected across the entire system. Hence, there is no possibility that duplicate information will be found in the database.
- **Seamless data sharing** - Users can share all existing data with each other found within a database. Different levels of authorizations exist within the database for selective access to the information. Therefore, it is not possible to share the information without following the proper authorization protocols. Multiple remote users can concurrently access the database and share desired information as needed.
- **Data integrity** - The integrity of data implies that the information in the database is reliable and accurate. The integrity of data is very crucial as a DBMS contains a variety of databases. The information contained in all of these databases is available to all the users across the board. It is therefore essential to make sure that all the databases and customers have correct and coherent data available to them at all times.
- **Data security** - Data security is a vital in creation and maintenance of a database. Only authorized users are allowed to access the database by authenticating their identity using a valid username and password. Unauthorized users can not, under any conditions, be permitted to access the database, as it

infringes upon the data integrity rules.

- **Privacy** - The "privacy rule" in a database dictates that only authorized users are allowed to access the database on the basis of the predefined privacy constraints of the database. There are multiple database access levels and only permitted data can be viewed by the user. For example, various access limitations on the social networking sites for accounts that a user may want to access.
- **Backup and Recovery** - DBMS is capable of generating automated backup and recovery of the database. As a result, the users are not required to backup data regularly since the DBMS can efficiently handle this. In addition, it will also restore the database to its preceding state, if a technical error or system failure occurs.
- **Data Consistency** - In a database, the consistency of data is guaranteed, due to lack of any redundant data. Entire database contains all data consistently, and all the users accessing the database receive the same data. In addition, modifications to the database are instantly reported to all the users to avoid any inconsistency of the existing data.

Structured Query Language

The "Structured Query Language (SQL)" in context of relational databases is considered a standard user and application program interface. In 1986, "SQL" became a standard of the "American National Standards Institute (ANSI)", and in 1987, it was added to the "International Organization for Standardization (ISO)". Since then, the standards have been frequently improved and are endorsed by all mainstream commercial relational DBMSs (with different extents of compliance).

For the relational model, SQL was one of the initial commercial languages, even though it is distinct from the relational structure in certain aspects, according to Codd. For instance, SQL allows organization of data rows and columns. The relational databases are highly extensible. After the initial database has been created, a new data category can be easily introduced without needing to alter any current apps.

Types of SQL Queries

Database languages are defined as “special-purpose languages”, that enable execution of one or more of the tasks listed below and often called as “sublanguages”:

“Data definition language (DDL)” – It is used to define data types including their creation, modification, or elimination as well as the relationships among them. (As you go through this chapter, you will learn more about this topic)

“Data manipulation language (DML)” – Only after the database is created and tables have been built using DDL commands, the DML commands can be used to manipulate the data within those tables and databases. The convenience of using DML commands is that they can readily be changed and rolled back if any incorrect modifications to the data or its values have been made. The DML commands used to perform specific tasks are:

- **“Insert”** – For insertion of new rows in the table.
- **“Update”** – For modification of the data values contained in the rows of the table.
- **“Deletion”** – For deletion of selected rows or complete table within the database.
- **“Lock”** – To define the user access to either “read only” or “read and write” privilege.
- **“Merge”** – To merge couple of rows within a table.

“Data control language (DCL)” – As the name indicates, the DCL commands pertain to data control problems in a database. DCL commands provide users with unique database access permissions and are also used to define user roles as applicable. There are two DCL commands that are frequently used:

- **“Grant”** – To give access permissions to the users.
- **“Revoke”** – To remove the access permission given to the users.

“Data query language (DQL)” – DQL comprises of a single command that drives data selection in SQL. In conjunction with other SQL clauses, the "SELECT" command is used for collection and retrieval of data from databases or tables based on select user-applied criteria. A "SELECT" statement is used to search for data and computing derived information from table and/or database.

“Transaction control language (TCL)” – As indicated by its name, TCL administers transaction related issues and problems in a database. They are used to restore modifications made to the original database or confirm them.

Roll back implies the modifications "Undo," and Commit means the modifications "Apply." The 3 main TCL commands available are:

- **“Rollback”** – Used to “cancel or undo” any updates made in the table or database.
- **“Commit”** – Used to “deploy or apply or save” any updates made in the table or database.
- **“Savepoint”** – Used to temporarily “save” the data in the table or database.

“Database languages” are limited to a specific data model. Some of the examples are:

- **“SQL”** – It offers functions such as data definition, data manipulation and query as a unified language.
- **“OQL”** - It is an “object model language standard developed by the Object Data Management Group". This language inspired the development of a few of the modern query languages such as "JDOQL" and "EJBQL".
- **“XQuery”** – It is a “standard XML query language”, which was introduced by "XML database technologies like MarkLogic and eXist, relational databases with XML capabilities like Oracle and DB2, as well as in memory XML processors like Saxon".
- **“SQL / XML”** – It is a combination of "SQL" and "XQuery".

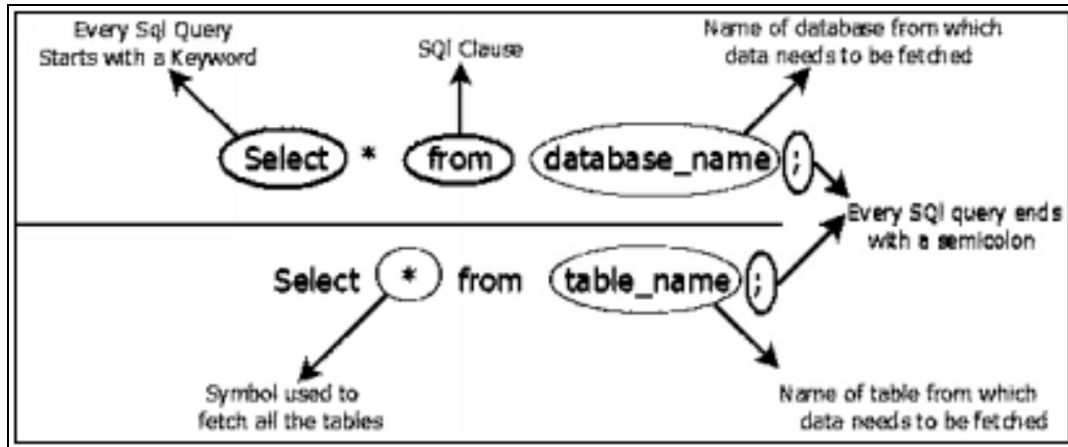
SQL SYNTAX

The picture below depicts the most common syntax (code structure that can be understood by the server) of a “SQL query”. Before we jump into the actual SQL commands that you can execute for a hands-on learning experience. Here are some thumb rules that you must memorize first:

1. A “semicolon” must be used at the end of each command.
2. A “keyword” must be used at the beginning of the command.
You will learn all the keywords as you progress through this

book.

3. The case structure of the alphabets is irrelevant to the commands.
4. You must remember the name or title of the tables and databases, used at the time of creation and make sure you always use those specific names for all your commands.



SQL Data Types

Another important concept in learning SQL, is the various data types that can be used to “define the type of data values that can be entered in specific columns”. The commonly used data types are listed in the table below:

DATA TYPE	DESCRIPTION
<code>“char(size)”</code>	This is used to define a “fixed length character string”. The size of the data value can be specified using desired number in the parenthesis as shown. Maximum allowed size is “255 characters”.
<code>“varchar(size)”</code>	This is used to define a “variable length character string”. The size of the data value can be specified using desired number in the parenthesis as shown. Maximum allowed size is “255 characters”.
<code>“number(size)”</code>	This is used to define a “numerical value” with a specified in parenthesis. The maximum number of digits in the column can be specified using desired number in the parenthesis as shown.
	This is used to define a “date value” for pertinent

<i>“date”</i>	column.
<i>“number(size, d)”</i>	This is used to define a “numerical value” with a specified in parenthesis. The maximum number of digits in the column can be specified using desired number in the parenthesis, along with the number of digits to the right of the decimal “d” as shown.

Data definition language (DDL)

Along with the introduction of the “Codasyl database model”, the term “data definition language” was coined. The concept of DDL required the database scheme to be developed in a syntax that described the “records, fields and sets of the user data model”. Subsequently, DDL was considered a subset of “SQL” used to declare tables, columns, data types and constraints. The third major revision of SQL called “SQL – 92”, laid the foundation for “schema manipulation language and schema information tables to query schemas”. In “SQL – 2003” the data tables were described as “SQL/Schemata”.

DDL can be used to make or perform modifications to the physical structure of a desired table in given database. All such commands can be inherently auto committed upon execution, and all modifications to the table are immediately reflected and stored across the database. Here are the most known and widely used DDL commands:

- **“Create”** – For creation of new table(s) or an entire database.
- **“Alter”** – To modify the data values of rows that already exist in the table.
- **“Rename”** – To rename the table(s) or the database.
- **“Drop”** – To delete the table(s) from a database.
- **“Truncate”** – To delete an entire table from the database.

Now, we will explore various uses of these DDL commands to execute required operations. For ease of learning all the fixed codes or keywords will be written in CAPITAL LETTERS, so you can easily differentiate between the dummy data and command syntax.

SQL “CREATE” command

Usage – As the name suggests, “CREATE” command is primarily used to

build database(s) and table(s). This always tends to be the first step in learning SQL.

Syntax:

`“CREATE DATABASE database_name;”`

`“CREATE TABLE table_name;”`

`“CREATE TABLE table_name (column1 datatype, column2 datatype, column3 datatype,...);”`

Examples:

Let’s assume you would like to create a database named as “database_football”, which contains one table that stores details of the teams called “team_details” and another table that stores details of the players called “player_details”.

The first step here would be to build the desired database, with the use of the command below:

`“CREATE DATABASE database_football;”`

Now, that you have your database in the system, you can create the desired tables, using the command below:

`“CREATE TABLE team_details;”`

`“CREATE TABLE player_details;”`

If you wanted to create the table “player_details” with some columns like “player_firstname” with column size as 15, “player_lastname” with column size as 25 and “player_age”. You will use the command below:

`“CREATE TABLE player_details (player_firstname varchar (15), player_lastname varchar (25), player_age int);”`

Remember the size of the “varchar” data type can be mentioned in the statement with the maximum size as “255 characters” and in case the column size has not been specified, the system will “default” you to the maximum size

Hands-on Exercise

For exercise purposes, imagine that you are a brand new business owner and looking to hire some employees. Now, you would first have to create a database for your company, let’s name your company “Fit Life” and create a

database named “database_FitLife”. You can then add a table titled “New_Employees” with columns to store employees’ “First Name”, “Last Name”, “Job Title”, “Age” and “Salary”.

****Use your discretion and write your SQL statements first!****

Now you can verify your statement against the right command as given below:

```
“CREATE DATABASE database_FitLife;”
```

```
“CREATE TABLE New_Employees (employee_firstname varchar,  
employee_lastname varchar, employee_jobtitle varchar, employee_age int,  
employee_salary int);”
```

SQL “ALTER” command

Usage – This command is primarily used to “add a column, remove a column, add different data constraints and remove predefined data constraints”.

Data Constraints – In terms of SQL, “constraints” are nothing but specific rules applied to the data in the table. They can be applied at a “column level”, which is only applicable to the selected column or at a “table level”, which is applicable to all the columns in the table. They “limit” the type of data values that are allowed to be added to the table, to ensure the data in the table is accurate and reliable. Violating these constraints will abort the command that you are trying to execute.

The data constraints can be added while creating the table (using the “CREATE” command) or once the table is generated (utilizing the “ALTER” command).

The most widely used SQL constraints are:

- **“NOT NULL”** – It is utilized to make sure that the column always contains a value and does not hold a “NULL value”. Meaning, a new record cannot be added or a record that already exists cannot be modified, if there are no values added to this column.
- **“UNIQUE”** - This constraint is used to make sure that every single data value in a column is distinct from one another.
- **“PRIMARY KEY”** - This constraint is essentially a

“combination of a NOT NULL and UNIQUE” constraints. It serves as a unique identification for each record in the table, which is required to contain a data value. Only one “primary key” can be indicated for individual tables.

- **“FOREIGN KEY”** - This constraint serves as a unique identification for a record/row existing in a different table that has been linked to the table you are working on. This helps in aborting commands that will break the link between the two tables i.e. invalid data cannot be added to the “foreign key” column as it has to be a value as contained in the table that is holding that specific column.
- **“CHECK”** - This constraint is used to make sure that all the data values in a column meet a specific condition and are limited to the predefined value range. Applying “Check” constraint on a column will allow only select values to be added to that column. However, applying “Check” constraint on a table will put restrictions on the values that can be added in specific columns, on the basis of the values defined in other columns of a row.
- **“DEFAULT”** – It is utilized to set a “default data value” for specific column, in case no other value has been specified for that column.
- **“INDEX”** - This constraint is utilized to “create and retrieve” data values from the database at a quick pace, using indexes which are not visible to the users. This should only be used for columns that will be searched more frequently than the others.

Syntax:

Adding Column

```
“ALTER TABLE table_name  
ADD column_name datatype;”
```

Dropping Column

```
“ALTER TABLE table_name
```

DROP column_name;”

Adding “Not Null” Constraint

“ALTER TABLE table_name
MODIFY column_name datatype NOT NULL;”

Adding “Unique” Constraint

“ALTER TABLE table_name
ADD UNIQUE column_name;”

Adding “Primary Key” Constraint

“ALTER TABLE table_name
ADD PRIMARY KEY column_name;”

Adding “Foreign Key” Constraint

“ALTER TABLE table_name
ADD FOREIGN KEY (column_name) REFERENCES table_name
(column_name);”

Adding “Check” Constraint

“ALTER TABLE table_name
ADD CHECK (column_name <= ‘numeric value’);”

Adding “Default” Constraint

“ALTER TABLE table_name
ADD CONSTRAINT (column_name) DEFAULT ‘data_value’ FOR
(column_name);”

Adding “Index” Constraint

“CREATE INDEX index_name
ON table_name (column1, column2,...);”

Dropping “Not Null” Constraint

“ALTER TABLE table_name
DROP CONSTRAINT column_name datatype;”

Dropping “Unique” Constraint

“ALTER TABLE table_name

DROP CONSTRAINT column_name datatype;”

Drop “Primary Key” Constraint

“ALTER TABLE table_name

DROP PRIMARY KEY;”

Dropping “Foreign Key” Constraint

“ALTER TABLE table_name

DROP FOREIGN KEY (column_name);”

Dropping “Check” Constraint

“ALTER TABLE table_name

DROP CHECK (column_name);”

Dropping “Default” Constraint

“ALTER TABLE table_name

ALTER COLUMN (column_name) DROP DEFAULT;”

Dropping “Index” Constraint

“DROP INDEX table_name.index_name;”

Examples:

Now, assume that you would like to alter the “player_details” table, that was generated in the section above. To add a column titled “player_ID” and then drop the “player_age” column, you could execute the commands below:

“ALTER TABLE player_details

ADD player_ID number;”

“ALTER TABLE player_details

DROP player_age;”

Let’s now look at how various “constraints” can be added with the “player_details” table!

You can specify that the column “player_firstname” has the “Not Null” constraint. The column “player_lastname” is unique. The column “player_ID” can be defined as the primary key and can be utilized as the foreign key in our other table titled “team_details”. We can define the “check constraint” on the column titled “player_age” to make sure all the values

entered are above or equal to 20. We can also use the default value for “player_age” column as “20”. Since, we will be frequently using the “player_lastname” column, let’s add an index to it.

Step 1:

```
“ALTER TABLE player_details  
MODIFY player_firstname varchar (15) NOT NULL;”
```

Step 2:

```
“ALTER TABLE player_details  
ADD UNIQUE player_lastname;”
```

Step 3:

```
“ALTER TABLE player_details  
ADD PRIMARY KEY player_ID;”
```

Step 4:

```
“ALTER TABLE team_details  
ADD FOREIGN KEY (player_ID) REFERENCES player_details  
(player_ID);”
```

Step 5:

```
“ALTER TABLE player_details  
ADD CHECK (player_age <= “20”);”
```

Step 6:

```
“ALTER TABLE player_details  
ADD CONSTRAINT (player_age) DEFAULT ‘20’ FOR (player_age);”
```

Step 7:

```
“CREATE INDEX idx_name  
ON player_details (player_lastname, player_firstname);”
```

Now, lets drop all these constraints we just defined above, in the same order.

Step 1:

```
“ALTER TABLE player_details
```

```
DROP CONSTRAINT player_firstname varchar (15) NOT NULL;"
```

Step 2:

```
"ALTER TABLE player_details  
DROP CONSTRAINT player_lastname;"
```

Step 3:

```
"ALTER TABLE player_details  
DROP PRIMARY KEY;"
```

Step 4:

```
"ALTER TABLE team_details  
DROP FOREIGN KEY (player_ID);"
```

Step 5:

```
"ALTER TABLE player_details  
DROP CHECK (player_age);"
```

Step 6:

```
"ALTER TABLE player_details  
ALTER COLUMN (player_age) DROP DEFAULT;"
```

Step 7:

```
"DROP INDEX player_details.idx_name;"
```

Hands-on Exercise

For exercise purposes, we will continue to use the “database_FitLife” that we created in the previous section, which contains a table titled “New_Employees” with columns to store “First Name”, “Last Name”, “Job Title”, “Age” and “Salary” of the employees.

Now, let’s add another table to this database titled “Current_Employees”. Then add a new column for employee ID, which will be the “primary key” for the “New_Employees” table and “foreign key” for “Current_Employees” table. The salary column can be dropped for this exercise.

Make the column for the first name as “Not Null” and last name as “Unique”. Add a “check” constraint on the employee age column to be “greater than or equal to 21”. Then set the default value for the same column as “21”. And

lastly, add an index for the last name column to the “New_Employees” table.

*****Use your discretion and prep your SQL statements first!*****

Now you can verify your statement against the right command as given below:

Step 1:

```
“CREATE TABLE Current_Employees (employee_firstname varchar,
employee_lastname varchar, employee_jobtitle varchar, employee_age int,
employee_salary int);”
```

Step 2:

```
“ALTER TABLE New_Employees
ADD employee_ID number;”
```

Step 3:

```
“ALTER TABLE New_Employees
DROP employee_salary;”
```

Step 4:

```
“ALTER TABLE New_Employees
ADD PRIMARY KEY employee_ID;”
```

Step 5:

```
“ALTER TABLE Current_Employees
ADD FOREIGN KEY (employee_ID) REFERENCES New_Employees
(employee_ID);”
```

Step 6:

```
“ALTER TABLE New_Employees
MODIFY employee_firstname varchar NOT NULL;”
```

Step 7:

```
“ALTER TABLE New_Employees
ADD UNIQUE employee_lastname;”
```

Step 6:

```
“ALTER TABLE New_Employees  
ADD CHECK (employee_age <= “21”);”
```

Step 7:

```
“ALTER TABLE New_Employees  
ADD CONSTRAINT (employee_age) DEFAULT ‘21’ FOR (employee  
_age);”
```

Step 8:

```
“CREATE INDEX idx_name  
ON New_Employees (employee_lastname);”
```

Viola! Now you are tasked to drop all these constraints and then add the salary column back to the “New_Employees” table.

*****Use your discretion and write your SQL statements first!*****

Now you can verify your statement against the right command as given below:

Step 1:

```
“ALTER TABLE New_Employees  
DROP PRIMARY KEY;”
```

Step 2:

```
“ALTER TABLE Current_Employees  
DROP FOREIGN KEY (employee_ID);”
```

Step 1:

```
“ALTER TABLE New_Employees  
DROP CONSTRAINT employee_firstname varchar NOT NULL;”
```

Step 2:

```
“ALTER TABLE New_Employees  
DROP CONSTRAINT employee_lastname;”
```

Step 5:

```
“ALTER TABLE New_Employees
```

DROP CHECK (employee_age);”

Step 6:

“ALTER TABLE New_Employees
ALTER COLUMN (employee_age) DROP DEFAULT;”

Step 7:

“DROP INDEX New_Employees.idx_name;”

Step 8:

“ALTER TABLE New_Employees
ADD employee_salary number;”

SQL “DROP DATABASE” command

Usage – This command is primarily used to delete an already existing SQL database.

Syntax:

“DROP DATABASE database_name;”

Examples:

Let’s assume you would like to delete the database “database_football” for any business reason like irrelevant data in the database or duplicate databases etc.

You can simply use the command below and the entire database will be wiped out from the system.

“DROP DATABASE database_football;”

Hands-on Exercise

For exercise purposes, imagine that your company “Fit Life” has gone under or renamed and you want to discard all the data you have available related to the company. What would you do?

*****Use your discretion and write your SQL statements first!*****

Now you can verify your statement against the right command as given below:

“DROP DATABASE database_FitLife;”

SQL “CREATE VIEW” command

Usage – This command is primarily used to generate a “virtual view of the tables” in the database on the basis of the “result-set” of a SQL command. The view resembles an actual database table, in that it also has columns and rows but the data values can be called from a single or multiple tables in the database using SQL functions “WHERE” and “JOIN”. “SQL Views” are used to enhance the structure of the data making it more user friendly.

In SQL, the “WHERE” function is used to selectively filter the data or records on the basis of specific conditions. It is a highly versatile function and can also be used with other SQL statements including “UPDATE” and “DELETE”.

Only latest and greatest data will be used for generation of a view!

Syntax:

```
“CREATE VIEW view_name AS  
SELECT column1, column2, ...  
FROM table_name  
WHERE condition;”
```

Examples:

Let’s go back to the database “database_football” and assume you would like to view details of the players who are 25 years old.

You can simply use the command below and view desired information.

```
“CREATE VIEW young_players AS  
SELECT player_firstname, player_lastname  
FROM player_details  
WHERE player_age = ‘25’;”
```

Hands-on Exercise

For exercise purposes, imagine that your company “Fit Life” is undergoing an organizational change and you would like to view a list of all the employees with the job title as “managers”.

*****Use your discretion and write your SQL statements first!*****

Now you can verify your statement against the right command as given below:

```
“CREATE VIEW employee_managers AS  
SELECT employee_firstname, employee_lastname  
FROM Current_Employees  
WHERE employee_jobtitle = ‘manager’;”
```

Chapter 2

Creating a Database using SQL and maintaining Data integrity

“MySQL” has always been a popular “free and open source SQL based Relational Database Management System (RDMS)”. In 1995, a Swedish company called "MySQL AB" originally developed, marketed and licensed the MySQL data management system, that was eventually acquired by “Sun Microsystems” (now called “Oracle Co.”). MySQL is a "LAMP software stack web application component, an acronym for Linux, Apache, MySQL, Perl / PHP / Python". Several database-controlled web applications, including “Drupal”, “Joomla”, “PhpBB”, and “WordPress”, are using “MySQL”. It is one of the best applications for RDBMS to create various online software applications and has been used in development of multiple renowned websites including “Facebook”, “YouTube”, “Twitter” and “Flickr”.

MySQL has been used with both "C" and "C++". MySQL can be operated on several systems, including "AIX, BSDi, FreeBSD, HP-Ux, eComStation, i5/OS, IRIX, Linux, macOS, Microsoft Windows, NetBSD, Novell NetWare, Open Solaris, OS/2 Warp, QNX, Oracle Solaris, Symbian, SunOS, SCO Open Server, SCO UnixWare, Sanos and Tru64". Its SQL parser is developed in "yacc" and it utilizes an in house developed "lexical analyzer". A MySQL port for "OpenVMS" has also been developed. Dual-license distribution is utilized for the "MySQL server" and its "client libraries", which are available under "version 2 of the GPL" or a proprietary license. Additional free assistance is accessible on various IRC blogs and channels. With its "MySQL Enterprise goods", Oracle is also providing paid support for their software. The range of services and prices vary. A number of third party service providers, including "MariaDB" and "Percona", offer assistance and facilities as well.

MySQL has been given highly favorable feedback from the developer community and reviewers have noticed that it does work exceptionally well on most cases. The developer interfaces exist and there's a plethora of excellent supporting documentation, as well as feedback from web locations in the real world. “MySQL” has also been successfully passed the test of a "fast, stable and true multi-user, multi-threaded SQL database server".

“MySQL” can be found in two different editions: the "MySQL Community Server" (open source) and the "Enterprise Server", which is proprietary. MySQL Enterprise Server" differentiates itself on the basis of a series of proprietary extensions that can be installed as "server plugins", but nonetheless share the numbering scheme of versions and are developed using the same basic code.

Here are some of the key characteristics provided by “MySQL v5.6”:

- “A broad subset of ANSI SQL 99, as well as extensions.”
- “Cross-platform support.”
- “Stored procedures, using a procedural language that closely adheres to SQL/PSM.”
- “Triggers”; “Cursors”; “Updatable views”; “SSL support”; “Query caching”; “Sub-SELECTs (i.e. nested SELECTs)”; “Built-in replication support”; “Information schema.”
- “Online Data Definition Language (DDL) when using the InnoDB Storage Engine.”
- “Performance Schema that collects and aggregates statistics about server execution and query performance for monitoring purposes.”
- “A set of SQL Mode options to control runtime behavior, including a strict mode to better adhere to SQL standards.”
- “X/Open XA distributed transaction processing (DTP) support; two phase commit as part of this, using the default InnoDB storage engine.”
- “Transactions with savepoints when using the default InnoDB Storage Engine.” “The NDB Cluster Storage Engine also supports transactions.”
- “ACID compliance when using InnoDB and NDB Cluster Storage Engines.”
- “Asynchronous replication: master-slave from one master to many slaves or many masters to one slave.”
- “Semi synchronous replication: Master to slave replication where the master waits on replication.”
- “Synchronous replication: Multi-master replication is provided in MySQL Cluster.”
- “Virtual Synchronous: Self managed groups of MySQL servers

with multi master support can be done using: Galera Cluster or the built in Group Replication plugin.”

- “Full-text indexing and searching”; “Embedded database library”; “Unicode support”.
- “Partitioned tables with pruning of partitions in optimizer.”
- “Shared-nothing clustering through MySQL Cluster.”
- “Multiple storage engines, allowing one to choose the one that is most effective for each table in the application.”
- “Native storage engines InnoDB, MyISAM, Merge, Memory (heap), Federated, Archive, CSV, Blackhole, NDB Cluster.”
- “Commit grouping, gathering multiple transactions from multiple connections together to increase the number of commits per second.”

User Interfaces for MySQL

A graphical user interface (GUI) can be defined as “a type of interface that allows users to interact with electronic devices or programs through graphical icons and visual indicators such as secondary notation, as opposed to text-based interfaces, typed command labels or text navigation”. “GUIs” tend to be easier to learn in comparison to the “command-line interfaces (CLIs)”, where commands must be entered on your keyboard. “MySQL” offers a variety of interfaces that you can leverage to best meet your need. Some of the widely popular “MySQL interfaces” are:

MySQL Workbench

MySQL Workbench is the “official built-in environment for MySQL database”. It has been created by "MySQL AB" and allows the user to supervise MySQL databases graphically and to design the databases visually. It has substituted "MySQL GUI Tools", which was the preceding software suite. “MySQL Workbench” enables users to perform database designing & model creation, “SQL implementation” (replacing "MySQL Query Browser") and “Database administration” (replacing "MySQL Administrator"). These functionalities are also available from other third-party applications but “MySQL Workbench” is still regarded as the official MySQL front end. “MySQL Workbench” can be downloaded from the official MySQL website, in two separate versions: a regular "free and open source community" version, and a "proprietary standard" version, which continues to

expand and enhance the community version feature set.

phpMyAdmin

PhpMyAdmin is “a free and open source application developed in PHP programming language, that allows use of a web browser to manage MySQL administration”. It allows undertaking of multiple functions such as generation, modification, or deletion of database, table, field, or row by running SQL queries as well as management of users and permissions. phpMyAdmin is easily accessible in whopping 78 different languages and managed by "The phpMyAdmin Project". The data can be imported from "CSV" and "SQL" and used with a set of predefined features to convert stored data into a desirable format, such as "representing BLOB data as pictures or download links".

Adminer

Adminer (previously referred to as "phpMinAdmin") is another “free and open source MySQL front end for maintaining information in the MySQL databases”. The release of version 2, the "Adminer" can also be used on "PostgreSQL", "SQLite" and "Oracle" databases. The “Adminer” can be used to manage several databases with multiple "CSS skins" that are easily accessible. It is supplied under the "Apache License" or "GPL v2" as a single “PHP file”. Jakub Vrána, began developing "Adminer", in July 2007, as a light weight option to the original "phpMyAdmin" application.

ClusterControl

ClusterControl is defined as “an end-to-end MySQL management system or GUI for managing, monitoring, scaling and deploying versions of MySQL from a single interface”. It has been engineered by "Severalnines". The "ClusterControl community version” can be accessed for free and allow deployment and tracking of the MySQL instances, directly by the user. Advanced characteristics such as load balance, backup and restoration, among other are additional features that can be paid for.

Database Workbench

“Database Workbench” application was created by "Upscene Productions", for the creation and management of various relational databases that are interoperable within separate database systems, using SQL. “Databases Workbench” can support several database systems and offers a cross-database tool for computer programmers with a similar interface and

production environment, for a variety of databases. The relational databases that can be supported by “Database Workbench” are: "Oracle Database, Microsoft SQL Server, SQL Anywhere, Firebird, NexusDB, InterBase, MySQL and MariaDB". “Database Workbench 5” can be operated on Windows systems that are either "32-bit or 64-bit" as well as on "Linux, FreeBSD, or MacOS" operating systems by leveraging the "Wine" application.

DBeaver

DBeaver is an “open source and free software application marketed under "Apache License 2.0", used as a database administration tool and a SQL client with the source code hosted on GitHub”. DBeaver incorporates extensive assistance for the following databases: "MySQL and MariaDB, PostgreSQL, Oracle, DB2 (LUW), Exasol, SQL Server, Sybase, Firebird, Teradata, Vertica, Apache Phoenix, Netezza, Informix, H2, SQLite and any other JDBC or ODBC driver database".

DBEdit

DBEdit is a "database editor, which can connect to an Oracle, DB2, MySQL and any database that provides a JDBC driver, and its source code is hosted on SourceForge ". It is also defined as “a free and open source software, which is distributed under the GNU General Public License”. It is available on "Windows, Linux and Solaris".

HeidiSQL

HeidiSQL, earlier referred to as "MySQL-Front", is another free and open source application and serves as a front end for MySQL, operable with "MariaDB", "Percona Server", "Microsoft SQL Server" and "PostgreSQL". German computer scientist Ansgar Becker and several other contributors at "Delphi", developed the "HeidiSQL" interface. To operate “HeidiSQL” databases, customers need to log in and create a session to a MySQL server locally or remotely with acceptable credentials. This session enables the users to connect and manage “MySQL Databases” on the “MySQL server” and to disconnect from the server once they have completed their task. “HeidiSQL” function set is suitable for most popular and sophisticated activities pertaining to the database, table and data records, but it continues to actively grow towards the complete feature desired in a MySQL user interface.

LibreOffice Base

LibreOffice Base enables databases to be created and managed, forms to be prepared and reports providing simple access to information for the end users. It also serves as an interface for different database systems like "Microsoft Access", including "Access databases (JET), ODBC information sources, and MySQL or PostgreSQL".

Navicat

Navicat is a "series of graphical database management and development software for MySQL, MariaDB, Oracle, SQLite, PostgreSQL and Microsoft SQL Server manufactured by PremiumSoft CyberTech Ltd". It has a graphical user interface similar to the "Microsoft Internet Explorer" and supports various local and remote database connections. It has been designed to satisfy the demands of a range of customers, from database admins and developers to various corporations that are serving the public and share data with their partner companies. "Navicat" is a cross-platform software that can be operated on systems such as "Microsoft Windows, OS X and Linux". Once the software has been purchased, the user can select one of the available eight languages to work with their software, which are: "English, French, German, Japanese, Polish, Simplified Chinese, and Traditional Chinese".

Sequel Pro

Sequel Pro is also free and open source "Macintosh" operating system software that can be operated locally or remotely with "MySQL databases" and hosted on "Sourceforge". It utilizes the "freemium model", which effectively provides "Gratis users" with most of the fundamental features. These applications need to be managed by a SQL Table itself. For newer unicode, it can manage the latest "fun" UTF-8 functions as well as having various GB tables with little to no difficulty.

SQLBuddy

SQLBuddy is a "web-based open-source software published in PHP that manages MySQL and SQLite administration using a web browser". The objective of this design is easy software set-up and an enhanced and convenient interface for the users.

SQLyog

SQLyog is another "MySQL GUI" application which is available for free but

also offers paid software versions of their platform. There is a spreadsheet resembling interface that allows data manipulation (e.g. insert, update and delete) to be accomplished easily.

Its editor offers multiple choices for automatic formatting such as "syntax highlighting". A query can be used to manipulate both raw table data and outcome set. Its search function utilizes "Google-like search syntax", which can be translated into SQL for users transparently. It is provided with a backup utility to execute unmonitored backups. Backups can be compressed and stored, if needed, as "a file-per-table as well as identified with a timestamp".

Toad

Toad for MySQL was developed by Dell Software, as a computer application utilized by database programmers, database admins as well as data analysts, using SQL to operate on both "relational and non-relational databases". Toad can be used with numerous databases and environments. It can be easily installed on all "32-bit/64-bit Windows platforms, including Microsoft Windows Server, Windows XP, Windows Vista, Windows 7 and 8(32-Bit or 64-Bit)". A Toad Mac Edition has also been recently published by the "Dell Software" and is offered as business version and trial / freeware version. The freeware version of Toad can be accessed from the "ToadWorld.com" community.

Webmin

Webmin was originally developed as "a web-based system configuration tool for Unix-like systems", but the newer versions are available for installation and can be operated on "Windows operating system" as well. It allows customization of internal "operating system configurations such as users, disk quotas, utilities or configuration files" as well as "open source applications such as Apache HTTP Server, PHP or MySQL" can be modified and controlled.

"Webmin" is primarily built on "Perl", and one that runs as its own internet server and process. For communication, it would default to the "TCP port 10000" and configured to utilize "SSL" (when "OpenSSL" has already been installed on the system with all "Perl" modules that are needed for execution). It is developed around modules that have an interface with the "Webmin server" and the "configuration files". This makes adding latest

features much more convenient. Because of the modular design of “Webmin”, custom plugins can be created for desktop setup for those who are keen. “Webmin” also enables control on many devices on the same subnet or LAN via one comprehensive user interface, or seamless connection to other “Webmin” hosts.

Installing MySQL on Linux/UNIX

MySQL should be installed using "RPM" on the "Linux system". The following RPMs are accessible on the "MySQL AB" official site for download:

- **MySQL** – “The MySQL database server manages the databases and tables, controls user access and processes the SQL queries”.
- **MySQL client** – “MySQL client programs, which make it possible to connect to and interact with the server”.
- **MySQL devel** – “Libraries and header files that come in handy when compiling other programs that use MySQL”.
- **MySQL shared** – “Shared libraries for the MySQL client”.
- **MySQL bench** – “Benchmark and performance testing tools for the MySQL database server”.

In order to continue with your setup, you must follow the instructions below:

1. Use the root user to log into the system.
2. Change to the RPM-containing directory.
3. Execute the command below to install the MySQL database server. Keep in mind to substitute “the filename in italics with a desired file name of your RPM”.

```
“[root@host] # rpm -i MySQL-5.0.9-0.i386.rpm”
```

This function is responsible for the installation of the “MySQL server”, the creation of a “MySQL” user, the required setup and automated startup of the “MySQL server”.

You will be able to locate all “MySQL” associated binaries in “/usr / bin” and “/usr / sbin” directories. Every database and table would be generated in the “/var / lib / mysql” directory.

The commands below are optional and can be used to install the other RPMs using the same approach, but it is highly recommended to install all these

RPMs on your system:

```
“[root@host]# rpm -i MySQL-client-5.0.9-0.i386.rpm”
```

```
“[root@host]# rpm -i MySQL-devel-5.0.9-0.i386.rpm”
```

```
“[root@host]# rpm -i MySQL-shared-5.0.9-0.i386.rpm”
```

```
“[root@host]# rpm -i MySQL-bench-5.0.9-0.i386.rpm”
```

Installing MySQL on Windows Operating System

The standard installation of MySQL on any Windows version has been made much easier than in the past, as MySQL is now offered as an installation package. All you have to do is “download the installer package, unzip and run the setup file”.

Installation processes in the default setup.exe are insignificant and all of them are installed under “C:\mysql” by default.

To test the server for the first time simply open user interface using the command prompt. Go to the MySQL server location that is likely going to be “C:\mysql\bin” and type “mysqld.exe –console”.

After successful installation some startup and “InnoDB” messages will be displayed on your screen. A failed installation may be related to “system permissions” issue. It is important to ensure that the directory holding the data is available to every user (likely MySQL) under which the procedures of the database are run.

MySQL can not be added to the start menu with installation and no user friendly way is offered to exit the server. So you should consider stopping the process manually with the use of “mysql admin”, “task list”, “task manager”, or other “Windows-specific” methods, instead of double-clicking on the “mysqld executable” to launch the server.

Installing MySQL on Macintosh Operating System

To install “MySQL” on “Mac OS X”, follow the step below:

- Download the “disk image file (.dmg)” containing the “MySQL package installer”. For mounting the “disk image” and viewing the contents, click twice on the “.dmg” file.
- Now, click twice on the “MySQL installer package”. The name of the “OS X MySQL installer package version” would be in

accordance with the "MySQL" and the "OS X" version you are working on. For instance, "if you have downloaded the package for MySQL 5.6.46 and OS X 10.8, double-click mysql-5.6.46-osx-10.8-x86_64.pkg".

- The opening installer message will be displayed on the screen. To start installation, click "Continue".
- A copy of the accompanying "GNU General Public License" will be displayed for all downloads of the "MySQL community version". Click "Continue" and then "Agree" to proceed.
- You can select "Install" to run the installation wizard from the "Installation Type" page with all default settings. Then click "Customize" to select the components you desire to install ("MySQL server", "Preference Pane" or "Launchd Support", all of which are activated by default).
- To start the installation process, click "Install".
- After successful installation has been done, an "Install Succeeded" message with a brief overview will be displayed on the screen. You can exit the installation assistant at this point and start using the "MySQL server".

MySQL has been configured now, but can not be launched automatically. You can either select "launchctl" from the command line or click on "Start" on the MySQL preference pane to fire up the MySQL server.

Installations done using the package installer will configure the files within "/usr / local" directory, that matches the name of the installation version and operating system being used. For instance, "the installer file mysql-5.6.46-osx10.8-x86_64.dmg installs MySQL into /usr/local/mysql-5.6.46-osx10.8-x86_64". A symbolic connection to a particular directory depending on the version or platform will be generated during the setup phase from "/usr / local / mysql", which is automatically updated while you are installing the package. The table below illustrates the installation directory layouts.

Directory	Contents of Directory
bin, scripts	"mysqld server, client and utility programs"
data	"Log files, databases"

docs	“Helper documents, like the Release Notes and build information”
include	“Include (header) files”
lib	“Libraries”
man	“Unix manual pages”
mysql-test	“MySQL test suite”
share	“Miscellaneous support files, including error messages, sample configuration files, SQL for database installation”
sql-bench	“Benchmarks”
support-files	“Scripts and sample configuration files”
/tmp/mysql.sock	“Location of the MySQL Unix socket”

Creating a new database using “MySQL server”

In “MySQL server”, databases are implemented as directories that contain all the files corresponding to the tables in a particular database.

Now that you have installed the “MySQL Server”, you can follow the instructions below to generate new databases with the “CREATE” command, using the syntax below:

“CREATE DATABASE [IF NOT EXISTS] database_name”

“[CHARACTER SET charset_name]”

“[COLLATE collation_name]”

In the syntax above, the first step is specifying the “database_name” right after the “CREATE DATABASE” command. In the “MySQL server” instance the name of the new database must be unique. When you're trying to build a database with an existing name, the server will abort the action. Next you can indicate the choice "IF NOT EXISTS" to prevent an infringement, if you erroneously generate a new database that shares its name with a database that already exists on the server. In this scenario, MySQL will not generate an issue but will instead terminate the "CREATE DATABASE" statement. Lastly, when the new database is created, you are able to indicate the character set and combination requirements. If the clauses "CHARACTER

SET" and "COLLATE" are omitted, then "MySQL" utilizes the default settings for the new database.

Alternatively, you can build a new database using **"MySQL Workbench"** and following the steps below:

1. Open "MySQL Workbench" and click on the "setup new connection" button on the screen.
2. Enter desired name for your connection and click on the "Test Connection" button. A window pane requiring the "root user" password would be displayed in "MySQL Workbench". You must enter the "root user password", check the "Save password in vault" and select "OK".
3. To connect to the "MySQL server", click twice on the "connection name Local". MySQL Workbench opens the window containing four different components namely: "Navigator, Query, Information, and Output".
4. Click the "create a new schema in the connected server" button from the toolbar. The "schema" is synonymous with the "database" in MySQL. Therefore, developing a new schema just implies that you are generating a new database.
5. In the subsequent window you must enter the "schema name" and modify the "character set and collation" as required and click on the "Apply" button.
6. "MySQL Workbench" will open up a new window displaying the SQL script that needs to be executed. Keep in mind that the "CREATE SCHEMA" query will lead to the same result as the "CREATE DATABASE" query. If all goes well, the new database will be generated and displayed in the "schemas tab of the Navigator section".
7. To select the "testdb2 database", simply right click on the database name and select "Set as Default Schema".
8. The "testdb2 node" would be open so you can run queries on "testdb2" using the MySQL Workbench.

Managing a database using "MySQL server"

Once all the databases containing your desired data have been created, you can selectively display and use the content needed using the instructions

below.

Display Databases

The "SHOW DATABASES" statement is utilized to list all the databases currently existing on the "MySQL server". You can inspect only your database or every database on the server through the use of a "SHOW DATABASES" statement before building a new database. For example, if you assume that the databases we have created so far are all on the "MySQL Server" and the "*SHOW DATABASES;*" statement is executed, you will see the result as "*information_schema; database_football; database_FitLife; mysql*".

Now you know that there are four different databases on your MySQL server. The "information_schema" and "mysql" are default databases that are created upon installation of the MySQL server and the other two databases, namely "*database_football*" and "*database_FitLife*" were created by us.

Selection of a database

Before operating with a specific database, you are required to first inform MySQL, which database you would like to operate by utilizing the "USE" command as below:

"USE database_name;"

To select the "*database_FitLife*", you can write the command as below:

"USE database_FitLife;"

Creating a new database table using "MySQL server"

You will be using the "MySQL CREATE TABLE" statement to build a new table in a database. The "CREATE TABLE" command is considered as one of the most complicated statements in SQL. A simple "CREATE TABLE" syntax can be written as below:

```
"CREATE TABLE [IF NOT EXISTS] table_name(  
    column_list  
) ENGINE=storage_engine"
```

Start by indicating the table name after the "CREATE TABLE" clause, that you would like to create. The name of the new table must be unique. The "IF NOT EXISTS" is an auxiliary clause for verification as to whether the table

you are trying to build already happens to be in that database. If there is a duplication in the table name, the entire statement will be ignored by MySQL and the new table will not be generated. You are advised to use the "IF NOT EXISTS" clause in every "CREATE TABLE" statement, in order to prevent an accidental creation of a new table name that can be found on the server.

Next, in the "column list" section, a list of columns for the table can be specified, and each column name can be distinguished with the use of commas.

Lastly, in the "ENGINE" clause, you have the option to indicate the table storage engine. Any storage engine like "InnoDB" and "MyISAM" could be used. If the storage engine is not specifically declared, MySQL will be using the default "InnoDB" engine.

The following syntax can be utilized to describe a table column in the "CREATE TABLE" statement:

“column_name data_type(length) [NOT NULL] [DEFAULT value] [AUTO_INCREMENT]”

In the syntax above, the column name has been specified by the “column_name” clause. Each column contains a particular type and the desired length of the data, for example, “VARCHAR (255)”. The "NOT NULL" clause specifies that NULL value is not permitted by this column.

The value "DEFAULT" can be utilized to indicate the default value for specific column. The "AUTO_INCREMENT" shows that when a new row is added to a table, the column value will be automatically created by the column. The “AUTO_INCREMENT” columns in each table are unique. For instance, you can create a table named “tasks”, with the command below:

```
“CREATE TABLE IF NOT EXISTS tasks (  
    task_id INT AUTO_INCREMENT,  
    title VARCHAR(255) NOT NULL,  
    start_date DATE,  
    due_date DATE,  
    status TINYINT NOT NULL,  
    priority TINYINT NOT NULL,
```

```
description TEXT,  
PRIMARY KEY (task_id)  
) ENGINE=INNODB;"
```

In the syntax above, the "task Id" is an “auto increment column”. If the "INSERT" query is used to add a new record to the table, with the "task Id" column value not being specified, the "task Id" column will be given an “auto-generated” integer value starting with '1'. The "task Id" column has been specified as the “primary key” for the table.

The "title" column has been given a "variable character string" data type with maximum allowed length of “255 characters”. This implies that a string with a length larger than “255 characters” can not be inserted in this column. The "NOT NULL" suggests that a value is required for the column. I.e. when inserting or updating this column, a value must be provided.

The "start date" and "due date" are date columns that will allow “NULL” values.

The "status" and "priority" are the "TINYINT" columns and will not permit “NULL” values.

The "description" column is a "TEXT" column that will allow NULL values.

Inserting data into a table on the “MySQL server”

To add one or more records into the table, the “INSERT” statement is used, as shown in the syntax below:

```
“INSERT INTO table (c1, c2,...)  
VALUES (v1, v2,...);”
```

In the statement above, the table name must be specified followed by a list of desired columns separated by “commas” written within “parentheses”, after the "INSERT INTO" clause. After which, a list of values separated by “commas” corresponding to the columns within the parentheses must be written, after the "VALUES" function. Make sure there is same quantity of columns and corresponding column values. Furthermore, the column positions must correspond with the position of the column values.

Use the syntax below to add a number of rows to a table, using only one "INSERT" statement:

```
“INSERT INTO table(c1,c2,...)
VALUES
    (v11,v12,...),
    (v21,v22,...),
    ...
    (vnn,vn2,...);”
```

The example below will generate three rows or records in the table "tutorials tbl" and display the confirmation that the “row has been affected”:

```
“root@host# mysql -u root -p password;
Enter password:*****
mysql> use TUTORIALS;
Database changed
mysql> INSERT INTO tutorials_tbl
    ->(tutorial_title, tutorial_author, submission_date)
    ->VALUES
    ->('Learn PHP', 'John Poul', NOW());
Query OK, 1 row affected (0.01 sec)
mysql> INSERT INTO tutorials_tbl
    ->(tutorial_title, tutorial_author, submission_date)
    ->VALUES
    ->('Learn MySQL', 'Abdul S', NOW());
Query OK, 1 row affected (0.01 sec)
mysql> INSERT INTO tutorials_tbl
    ->(tutorial_title, tutorial_author, submission_date)
    ->VALUES
    ->('JAVA Tutorial', 'Samer', '2007-05-06');
Query OK, 1 row affected (0.01 sec)
mysql>”
```

Here is another example, the query below can be utilized to add a new record to the “tasks” table that was created earlier:

```
"INSERT INTO
tasks (title, priority)
VALUES
('Learn MySQL INSERT Statement',1);
1 row(s) affected;"
```

The output should look like the image below:

	task_id	title	start_date	due_date	priority	description
►	1	Learn MySQL INSERT Statement	NULL	NULL	1	NULL

We indicated the values, in the example above, only for the "title" and "priority" columns. MySQL will use the "default values" for the other non-specified columns.

The "task_id" column has been specified as an auto increment column. This implies that upon addition of a new record to the table, MySQL will generate a subsequent integer.

The “start_date”, “due_date”, and “description” columns have been specified to hold NULL as the default value, so if no values are specified in the "INSERT" statement, MySQL will insert 'NULL' into those columns.

To “Insert a Default Value” into a specific column, either simply disregard the column name and its corresponding values or indicate the column name in the “INSERT INTO” function and type “DEFAULT” in the “VALUES” clause. For example, in the syntax below the “priority” column has been specified with the “DEFAULT” keyword.

```
"INSERT INTO
tasks (title, priority)
VALUES
```

```
('Understanding DEFAULT keyword in INSERT statement', DEFAULT);"
```

You can insert dates into the table using the “YYYY-MM-DD” format, which represents year, month and date in the mentioned order. For example, you can add the “start date” and “due date” values to the “tasks” table using the command below:


```
“INSERT INTO tasks (title, start_date, due_date)  
VALUES ('Insert date into table','2018-01-09','2018-09-15');”
```

The “VALUES” clause also supports expressions as shown in the example below, where the syntax will add a new task utilizing the “current date for start date and due date columns” using the "CURRENT_DATE()" function. Remember the "CURRENT_DATE()" function will always return the current date in the system.

```
“INSERT INTO tasks (title, start_date, due_date)  
VALUES  
('Use current date for the task', CURRENT_DATE (), CURRENT_DATE ());”
```

To insert multiple rows or records in the, take a look at the example below of the “tasks” table, where each record is indicated as values listed in the “VALUES” clause:

```
“INSERT INTO tasks (title, priority)  
VALUES  
('My first task', 1),  
('It is the second task', 2),  
('This is the third task of the week', 3);”
```

The resulting output should be: “3 row(s) affected Records: 3 Duplicates: 0 Warnings: 0”, which implies that the records have been added to the table and there are no “duplicates or warnings” in the table.

Creating a “Temporary table” in the database using “MySQL server”

A "temporary table" in MySQL is a peculiar kind of table that enables storage of a temporary outcome set for reuse over multiple times over the same session.

A "temporary table" is extremely useful when querying information that needs a single "SELECT" statement with the "JOIN" clauses which is not feasible or costly. In such scenario, the temporary table can be used as a storage for the immediate result which can be processed further using a different query.

A "temporary table" in MySQL may have the characteristics listed below:

- A "temporary table" is generated using the "CREATE TEMPORARY TABLE" statement. Note that between "CREATE" and "TABLE" functions, the "TEMPORARY" keyword must be added.
- When the session is completed or the connection is ended, MySQL server will automatically remove the temporary table from its memory. Another thing to remember is that once you are done with the temporary table, it can be explicitly deleted utilizing the "DROP TABLE" command.
- A "temporary table" will only be available to and accessed by the user that generates it. Different users can potentially generate multiple "temporary tables" with the same name and still not cause any error, since they can only be seen by the user who produced that specific temporary table. Two or more "temporary tables" can not, however, share the same name over the same session.
- In a database, a "temporary table" may be given the same name as a standard table. The existing "employee table", for instance, cannot be accessed if you create a "temporary table" called "employees", within the sample database. All queries you make against the "employees table" will now be referred to the "temporary employees table". When you delete the "employees temporary table", the permanent "employees table" would still be in the system and can be accessed again.

While a "temporary table" can be given the same name as a "permanent table" without triggering any error, it is advised to have a different name for the "temporary table". As the same name can lead to confusion and may result in an unexpected loss of information. For example, you can not distinguish between the "temporary table" and the "permanent table", if the connection to the database server is dismissed for some reason and then automatically reconnected to the server. Then, rather than the "temporary table", you could enter a "DROP TABLE" statement to delete the "permanent table" unexpectedly.

Now, a "temporary table" can be created in MySQL with the use of the "TEMPORARY" keyword in the "CREATE TABLE" statement. For instance, the command below will create a "temporary table" storing the top

ten clients by revenue with the given table name as “top10customers”:

```
"CREATE TEMPORARY TABLE top10customers
SELECT p.customerNumber,
       c.customerName,
       ROUND(SUM(p.amount),2) sales
FROM payments p
INNER JOIN customers c ON c.customerNumber = p.customerNumber
GROUP BY p.customerNumber
ORDER BY sales DESC
LIMIT 10;"
```

Using this “temporary table”, you can further run queries on this table like you would on a standard or permanent table in the database.

```
"SELECT
    customerNumber,
    customerName,
    sales
FROM
    top10customers
ORDER BY sales;"
```

“Creating a new table from an existing table in the database using MySQL server”

You may also generate a copy of a table that already exists in the table using the "CREATE TABLE" command, as shown in the syntax below:

```
"CREATE TABLE new_table_name AS
SELECT column1, column2,...
FROM existing_table_name
WHERE ....;"
```

The new table will have the same specifications for the columns as the parent table. It is possible to select all columns or a particular column. If a new table

is generated using a table that already exists in the database, then the current values from the parent table will be automatically loaded into the new table. For example, in the syntax below a table called “TestTables” will be created as a replica of the parent “Customers” table:

```
"CREATE TABLE TestTable AS
SELECT customername, contactname
FROM customers;"
```

You might encounter a scenario, when you require a precise copy or "clone" of a table and "CREATE TABLE... SELECT" does not meet the requirement, since you would like the the clone to contain the identical indexes, default values, etc as the parent table.

This scenario can be addressed by executing the measures provided below:

- Use "SHOW CREATE TABLE" to get a "CREATE TABLE" query specifying the structure, indexes and other feature of the source or parent table.
- Adjust the query to alter the table name to be same as the "clone table" and run the query. You would be able to get the precise clone of the table with this method.
- Optionally, if you would also like to copy the table contents, you can execute a "INSERT INTO... SELECT" statement.

For example, you can generate a “clone table” for “tutorials_tbl” using the syntax below:

FIRST, copy the entire schema of the table:

```
"mysql> SHOW CREATE TABLE tutorials_tbl \G;
```

```
***** 1. row *****
```

```
Table: tutorials_tbl
```

```
Create Table: CREATE TABLE `tutorials_tbl` (
  `tutorial_id` int (11) NOT NULL auto_increment,
  `tutorial_title` varchar (100) NOT NULL default ' ',
  `tutorial_author` varchar (40) NOT NULL default ' ',
  `submission_date` date default NULL,
```

```
PRIMARY KEY (`tutorial_id`),  
UNIQUE KEY `AUTHOR_INDEX` (`tutorial_author`)  
) TYPE = MyISAM
```

1 row in set (0.00 sec)

ERROR:

No query specified"

SECOND, rename the parent table to generate a new "clone table":

```
"mysql> CREATE TABLE clone_tbl (  
-> tutorial_id int (11) NOT NULL auto_increment,  
-> tutorial_title varchar (100) NOT NULL default ' ',  
-> tutorial_author varchar (40) NOT NULL default ' ',  
-> submission_date date default NULL,  
-> PRIMARY KEY (tutorial_id),  
-> UNIQUE KEY AUTHOR_INDEX (tutorial_author)  
-> ) TYPE = MyISAM;
```

Query OK, 0 rows affected (1.80 sec)"

LASTLY, if you would like to replicate the data from the source table to the clone table, you can use the command below:

```
"mysql> INSERT INTO clone_tbl (tutorial_id,  
-> tutorial_title,  
-> tutorial_author,  
-> submission_date)  
  
-> SELECT tutorial_id, tutorial_title,  
-> tutorial_author, submission_date  
-> FROM tutorials_tbl;
```

Query OK, 3 rows affected (0.07 sec)

Records: 3 Duplicates: 0 Warnings: 0"

DERIVED TABLES in MySQL Server

A "derived table" can be defined as "a virtual table returned upon execution of a SELECT statement". A "derived table" seems to be comparable to a "temporary table", however, in the "SELECT" statement it is much easier and quicker to use a "derived table" than a "temporary table" as it would not entail additional steps to create the "temporary table".

There is often interchangeable use of the term "derived table" and "subquery." When the "FROM" clause of a "SELECT" query uses a stand-alone "subquery", it is called as a "derived table". For example, the syntax below can be used to create a "derived table":

```
"SELECT
```

```
    column_list
```

```
FROM
```

```
    (SELECT
```

```
        column_list
```

```
    FROM
```

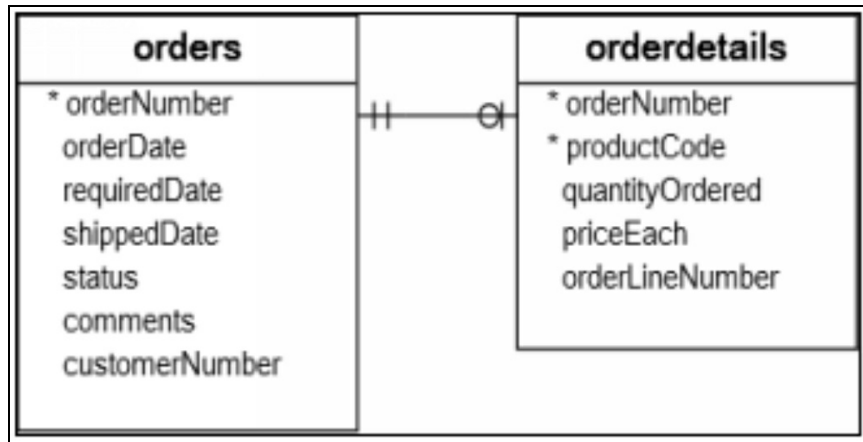
```
        table_1) derived_table_name;
```

```
WHERE derived_table_name.c1 > 0;"
```

A "derived table" is required to have an alias, which can be referenced in future queries, however, there is no such requirement for a subquery. If there is no alias defined for a "derived table", the error message below will be displayed by MySQL:

```
"Every derived table must have its own alias."
```

Review the example below of a query that will generate "top five products by sales revenue in 2003" from the "orders" and "orderdetails" tables in the MySQL server "sample database":



```

"SELECT
    productCode,
    ROUND (SUM (quantityOrdered * priceEach)) sales
FROM
    orderdetails
    INNER JOIN
    orders USING (orderNumber)
WHERE
    YEAR(shippedDate) = 2003
GROUP BY productCode
ORDER BY sales DESC
LIMIT 5;"
  
```

The resulting table would be as shown in the picture below:

	productCode	sales
►	S18_3232	103480
	S10_1949	67985
	S12_1108	59852
	S12_3891	57403
	S12_1099	56462

The result shown in the picture above can be used as a “derived table” and

joined with the “products” table using the syntax below:

products
* productCode
productName
productLine
productScale
productVendor
productDescription
quantityInStock
buyPrice
MSRP

```
"SELECT
    productName, sales
FROM
    (SELECT
        productCode,
        ROUND (SUM (quantityOrdered * priceEach)) sales
    FROM
        orderdetails
    INNER JOIN orders USING (orderNumber)
    WHERE
        YEAR(shippedDate) = 2003
    GROUP BY productCode
    ORDER BY sales DESC
    LIMIT 5) top5products2003
INNER JOIN
    products USING (productCode);"
```

The query above will result in the output as shown in the picture below:

	productName	sales
►	1992 Ferrari 360 Spider red	103480
	1952 Alpine Renault 1300	67985
	2001 Ferrari Enzo	59852
	1969 Ford Falcon	57403
	1968 Ford Mustang	56462

To drive this concept home, review a relatively complicated example of “derived tables” below:

Assume that the clients from the year 2009 have to be classified into three groups: "platinum, gold and silver". Moreover, taking into consideration the criteria below, you would like to calculate the number of clients within each group:

- Platinum clients with orders larger than 100K in quantity.
- Gold clients with quantity orders ranging from 10K to 100K.
- Silver clients with orders less than 10K in quantity.

To generate a query for this analysis, you must first classify each client into appropriate group using “CASE” clause and “GROUP BY” function as shown in the syntax below:

```
"SELECT
    customerNumber,
    ROUND (SUM(quantityOrdered * priceEach)) sales,
    (CASE
        WHEN SUM (quantityOrdered * priceEach) < 10000 THEN 'Silver'
        WHEN SUM (quantityOrdered * priceEach) BETWEEN 10000 AND
100000 THEN 'Gold'
        WHEN SUM (quantityOrdered * priceEach) > 100000 THEN
'Platinum'
    END) customerGroup
FROM
```

```

orderdetails
  INNER JOIN
  orders USING (orderNumber)
WHERE
  YEAR(shippedDate) = 2003
GROUP BY customerNumber;"

```

The query output is shown in the picture below:

	customerNumber	sales	customerGroup
▶	103	14571	Gold
	112	32642	Gold
	114	53429	Gold
	121	51710	Gold
	124	167783	Platinum
	128	34651	Gold
	129	40462	Gold
	131	22293	Gold
	141	189840	Platinum

Now, you can execute the code below to generate a “derived table” and group the clients as needed:

```

"SELECT
  customerGroup,
  COUNT(cg.customerGroup) AS groupCount
FROM
  (SELECT
    customerNumber,
    ROUND (SUM (quantityOrdered * priceEach)) sales,
    (CASE

```

```

        WHEN SUM (quantityOrdered * priceEach) < 10000 THEN
'Silver'
        WHEN SUM (quantityOrdered * priceEach) BETWEEN 10000
AND 100000 THEN 'Gold'
        WHEN SUM (quantityOrdered * priceEach) > 100000 THEN
'Platinum'
    END) customerGroup
FROM
    orderdetails
INNER JOIN orders USING (orderNumber)
WHERE
    YEAR(shippedDate) = 2003
GROUP BY customerNumber) cg
GROUP BY cg.customerGroup; "
```

The query output is shown in the picture below:

	customerGroup	groupCount
►	Gold	61
	Silver	8
	Platinum	4

Chapter 3

SQL Joins and Union commands

The most common command used in SQL is the “SELECT” command, which we have already used in the exercises mentioned earlier in this book. So, now let me give you a deep dive into the “SQL SELECT” command and various clauses that can be used with it.

MySQL SELECT

You can selectively fetch desired data from tables or views, using the "SELECT" statement. As you already know, similar to a spreadsheet, a table comprises of rows and columns. You are highly likely to view a subset of columns, a sub-set of rows, or a combination of the two. The outcome of the “SELECT” query is known as a "result set", which is a list of records comprising the same number of columns per record.

In the picture shown below of the "employees table" from the "MySQL sample database", the table has 8 different columns, namely: "employee number", "last name", "first name", "extension", "email", "office code", "reports to", "job title" and several rows or records.

	employeeNum	lastName	firstName	extension	email	officeCode	reportsTo	jobTitle
▶	1002	Murphy	Diane	x5000	dmurphy@classicmodelcars.com	1	NULL	President
	1056	Patterson	Mary	x4611	mpatterson@classicmodelcars.com	1	1002	VP Sales
	1076	Firelli	Jeff	x3273	jfirelli@classicmodelcars.com	1	1002	VP Marketing
	1088	Patterson	William	x4371	wpatterson@classicmodelcars.com	6	1056	Sales Manager (APAC)
	1102	Bondur	Gerard	x5408	gbondur@classicmodelcars.com	4	1056	Sales Manager (EMEA)
	1143	Bow	Anthony	x5423	abow@classicmodelcars.com	1	1056	Sales Manager (NA)
	1165	Jennings	Leslie	x3291	ljennings@classicmodelcars.com	1	1143	Sales Rep
	1166	Thompson	Leslie	x4065	lthompson@classicmodelcars.com	1	1143	Sales Rep
	1188	Firelli	Julie	x2173	jfirelli@classicmodelcars.com	2	1143	Sales Rep
	1216	Patterson	Steve	x4334	spatterson@classicmodelcars.com	2	1143	Sales Rep
	1286	Tseng	Foon Yue	x2248	ftseng@classicmodelcars.com	3	1143	Sales Rep
	1323	Vanauf	George	x4102	gvanauf@classicmodelcars.com	3	1143	Sales Rep
	1337	Bondur	Loui	x6493	lbondur@classicmodelcars.com	4	1102	Sales Rep
	1370	Hernandez	Gerard	x2028	ghernandez@classicmodelcars.com	4	1102	Sales Rep
	1401	Castillo	Pamela	x2759	pcastillo@classicmodelcars.com	4	1102	Sales Rep
	1501	Rott	Larry	x2311	lrott@classicmodelcars.com	7	1102	Sales Rep

The "SELECT" query dictates the columns and records that can be fetched from the table. For instance, if you would like to display just the "first name", "last name", and "job title" of all the employees or you selectively desire to see the information pertaining to employees with "job title" as "sales rep", then you will be able to utilize the "SELECT" query to achieve this.

Here is a standard syntax for “SELECT” statement:

"SELECT

column_1, column_2, ...

FROM

table_1

[INNER | LEFT | RIGHT] JOIN table_2 ON conditions

WHERE

conditions

GROUP BY column_1

HAVING group_conditions

ORDER BY column_1

LIMIT offset, length;"

The "SELECT" query above contains various provisions, as described below:

- The "SELECT" clause followed by a list of columns isolated by “comma” or an “asterisk (*)” suggests that all columns are to be returned.
- The "FROM" clause defines the table or view from which the data should be queried.
- The "JOIN" clause receives associated data from other tables based on the specified join criteria.
- In the "result set", the "WHERE" clause is utilized for selectively filtering the rows or records.
- The "GROUP BY" clause is utilized to selectively group a set of records and apply “aggregate functions” to each group.
- The "HAVING" clause is used to filter a group based on the "GROUP BY" clause specified groups.
- The "ORDER BY" clause can be utilized for specifying a list of

columns to be sorted.

- The "LIMIT" clause can be utilized for restricting the number of rows displayed upon execution of the command.

Note that only "SELECT" and "FROM" clauses are necessary to execute the command and all other clauses can be used on ad-hoc basis.

For instance, if you are interested in displaying only the "first name", "last name", and "job title" of all employees, you can utilize the syntax below:

```
"SELECT
```

```
    lastname, firstname, jobtitle
```

```
FROM
```

```
    employees;"
```

But if you would like to view all the columns in the "employees" table, use the syntax below:

```
"SELECT * FROM employees;"
```

In practice, it is recommended to list the columns you would like to view instead of using the asterisk (*) command, as the asterisk (*) will return all column data, some of which you may not be allowed to use. It will create "non-essential I/O disk and network traffic between the MySQL database server and the application". If the columns are defined explicitly, then the "result set" becomes simpler to predict and handle. Suppose you are using the asterisk (*) and some other user modified the table and generated additional columns, you would receive a "result set" containing different columns than needed. Moreover, the use of asterisk (*) can potentially reveal sensitive data to unauthorized users.

MySQL SELECT DISTINCT

You can receive duplicate rows when searching for information from a table. You could utilize the "DISTINCT" clause in the "SELECT" query to get rid of these redundant rows.

The "DISTINCT" clause syntax is given below:

```
"SELECT DISTINCT
```

```
    columns
```

```
FROM
```

table_name

WHERE

where_conditions;"

EXAMPLE

The syntax below presents an example for the "DISTINCT" clause, used to selectively view “unique last names” of the employees from the "employees" table.

First, you need to display the “last names” of all employees from the “employees” tables using the syntax below:

"SELECT

last name

FROM

employees

ORDER BY last name;"

Now to get rid of the repeated last names, use the syntax below:

"SELECT DISTINCT

last name

FROM

employees

ORDER BY last name;"

Please note, if a column has been indicated to hold “NULL” values and the “DISTINCT” clause is used. Only one “NULL” value will be retained by the server since the “DISTINCT” clause will consider all “NULL” values as identical.

Using “DISTINCT” clause on multiple columns

To accomplish this “MySQL server” will use a combination of all values in selected columns to identify unique records in the "result set". For example, you can view singular combination of “city” and “state” columns from the “customers” table using the syntax below:

"SELECT DISTINCT

```
state, city
FROM
customers
WHERE
state IS NOT NULL
ORDER BY state, city;"
```

MySQL “ORDER BY”

When using the "SELECT" statement to query the data, the "result set" will not be organized in a particular format. That is where the "ORDER BY" clause can be utilized to organize the "result set" as desired. The “ORDER BY” clause will allow sorting of the “result set” on the basis of one or more columns as well as sorting a number of columns in ascending or descending order.

Here is the "ORDER BY" clause syntax:

```
"SELECT column1, column2,...
FROM tbl
ORDER BY column1 [ASC|DESC], column2 [ASC|DESC],...;"
```

As you would expect the "ASC" means ascending and the "DESC" means descending. By default, if "ASC" or "DESC" has not been specified explicitly, the "ORDER BY" clause will sort the "result set" in ascending order.

EXAMPLE

The syntax below presents an example for the "ORDER BY" clause, used to selectively view contacts from the "customers" table and sort them in ascending order of the “last name”.

```
"SELECT
contact Last name,
contact First name
FROM
customers
```


ORDER BY

contact Last name;"

The ascending “result set” is shown in the picture below:

	contactLastname	contactFirstname
►	Accorti	Paolo
	Altagar,G M	Raanan
	Andersen	Mel
	Anton	Carmen
	Ashworth	Rachel
	Barajas	Miguel
	Benitez	Violeta
	Bennett	Helen
	Berglund	Christina

Now, if you wanted to view the last name in descending order, you will use the syntax below:

"SELECT

contact Last name,

contact First name

FROM

customers

ORDER BY

contact Last name DESC;"

The descending “result set” is shown in the picture below:

	contactLastname	contactFirstname
►	Young	Jeff
	Young	Julie
	Young	Mary
	Young	Dorothy
	Yoshido	Juri
	Walker	Brydey
	Victorino	Wendy
	Urs	Braun
	Tseng	Jerry

Or, if you would like to arrange the “last name” in descending order and the “first name” in the ascending order, you can utilize the “DESC” and “ASC” clause in the same query but with the relevant column as shown in the syntax below:

```
"SELECT
contact Last name,
contact First name
FROM
customers
ORDER BY
contact Last name DESC,
contact First name ASC;"
```

The “result set” is shown in the picture below, where the “ORDER BY” clause will first sort the “last name” in descending order and then subsequently sort the “first name” in the ascending order:

	contactLastname	contactFirstname
►	Young	Dorothy
	Young	Jeff
	Young	Julie
	Young	Mary
	Yoshido	Juri
	Walker	Brydey
	Victorino	Wendy
	Urs	Braun
	Tseng	Jerry

MySQL WHERE

The "WHERE" clause enables the search criteria to be specified for the records displayed after running the query.

The "WHERE" clause syntax is given below:

"SELECT

select_list

FROM

table_name

WHERE

search_condition;"

The “search_condition” is a composite of one or more "predicates" utilizing the “logical operators” like "AND", "OR" and "NOT", as shown in the table below. A "predicate" in SQL, is defined as “an expression that assesses true, false, or unknown”.

Operator	Description
=	“Equal to. You can use it with almost any data types.”
<> or !=	“Not equal to”
<	“Less than. You typically use it with numeric and date/time data types.”

>	"Greater than"
<=	"Less than or equal to"
>=	"Greater than or equal to"

The final "result set" includes any record from the "table_name" that makes the "search_condition" to be evaluated as valid.

In addition to the "SELECT" statement, you may indicate the rows that need to be updated and deleted, utilizing the "WHERE" clause in the "UPDATE" and "DELETE" query.

EXAMPLE

The syntax below presents an example for the "WHERE" clause, used to selectively view employees with job title as "Sales Rep" from the "employees" table in the "MySQL sample database":

"SELECT

lastname,

firstname,

jobtitle

FROM

employees

WHERE

jobtitle = 'Sales Rep';"

Although the "WHERE" clause is defined at the end of the query, "MySQL" first selects the corresponding rows after evaluating the phrase in the "WHERE" clause. It selects the rows with a job title as "Sales Rep" and then chooses the columns in the "SELECT" clause from the "select list". The highlighted rows in the picture below, include the final result set columns and rows.

employeeNumber	lastName	firstName	extension	email	officeCode	reportsTo	jobTitle
1002	Murphy	Diane	x5800	dmurphy@classicmodelcars.com	1	1000	President
1056	Patterson	Mary	x4611	mpatterson@classicmodelcars.com	1	1002	VP Sales
1076	Firelli	Jeff	x5273	jfirelli@classicmodelcars.com	1	1002	VP Marketing
1088	Patterson	William	x4871	wpatterson@classicmodelcars.com	6	1056	Sales Manager (APAC)
1102	Bondur	Gerard	x5408	gbondur@classicmodelcars.com	4	1056	Sales Manager (EMEA)
1143	Bow	Anthony	x5428	abow@classicmodelcars.com	1	1056	Sales Manager (NA)
1165	Jennings	Leslie	x3291	ljennings@classicmodelcars.com	1	1143	Sales Rep
1166	Thompson	Leslie	x4065	lthompson@classicmodelcars.com	1	1143	Sales Rep
1188	Firelli	Julie	x2173	jfirelli@classicmodelcars.com	2	1143	Sales Rep
1216	Patterson	Steve	x4334	spatterson@classicmodelcars.com	2	1143	Sales Rep
1286	Tseng	Foon Yue	x2248	ftseng@classicmodelcars.com	3	1143	Sales Rep
1323	Vanauf	George	x4102	gvanauf@classicmodelcars.com	3	1143	Sales Rep
1337	Bondur	Loui	x5453	lbondur@classicmodelcars.com	4	1102	Sales Rep
1370	Wimmer	Edgar	x3028	ewimmer@classicmodelcars.com	4	1102	Sales Rep

MySQL JOIN Statements

The MySQL "JOIN" function is a way to link information between one or more tables on the basis of common attributes or column values between the selected tables. A relational database comprises of various tables that have been linked by a shared or common column, known as "foreign key". As a result, information in each individual table can be deemed incomplete from the business perspective. For instance, there are two separate tables in the "MySQL sample database" called "orders" and "orderdetails" that have been connected to each other using a column called "orderNumber". You will have to search for information in both the "orders" and the "orderdetails" table to obtain complete order information.

There are 6 different SQL "JOINS" functions, namely, "INNER JOIN", "LEFT JOIN", "RIGHT JOIN", "CROSS JOIN" and "SELF JOIN".

We will be using tables named "t1" and "t2" as described in the syntax below, in order to facilitate your understanding of each type of "JOIN". The "pattern" column in both "t1" and "t2" tables serves as the common column between them.

```
"CREATE TABLE t1 (
    id INT PRIMARY KEY,
```

```

    pattern VARCHAR (50) NOT NULL
);
CREATE TABLE t2 (
    id VARCHAR (50) PRIMARY KEY,
    pattern VARCHAR (50) NOT NULL
);"

```

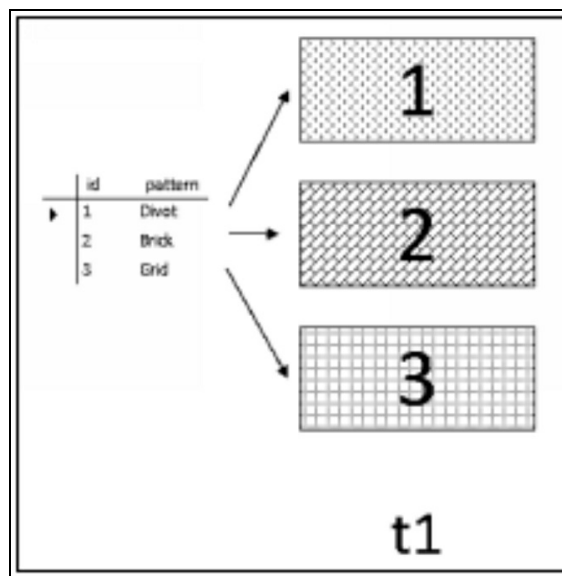
We can insert some data into the two tables using “INSERT” function as shown in the syntax below:

```

"INSERT INTO t1(id, pattern)
VALUES(1,'Divot'),
      (2,'Brick'),
      (3,'Grid');
INSERT INTO t2(id, pattern)
VALUES('A','Brick'),
      ('B','Grid'),
      ('C','Diamond');"

```

The result of the syntax above is shown in the picture below:



Now, let us look at every type of “JOIN” clause in detail!

“INNER JOIN”

The “MySQL INNER JOIN” is used to align “rows in one table with rows in other tables”, so that rows containing columns from the two tables can be queried.

The "SELECT" statement is not required to contain the "INNER JOIN" clause for execution, which appears right next to the "FROM" clause.

You must indicate the following requirements before using the "INNER JOIN" clause:

- Start with the primary table to be included in the "FROM" clause.
- Then, choose the table you would like to connect to the primary table included in the "INNER JOIN" clause, in step 1. Remember theoretically you can join one table to multiple tables at the same time. However, it is recommended to restrict the number of tables to be joined, to achieve higher performance and efficiency.
- Lastly, the "join condition" or "join predicate" must be defined. Following the initial "ON" keyword for the "INNER JOIN", the “join condition” can be indicated. The join condition is the rule that dictates the matching of a row in the “primary table” with the records from the another table.

The "INNER JOIN" clause syntax is given below:

```
"SELECT column_list
```

```
FROM t1
```

```
INNER JOIN t2 ON join_condition1
```

```
INNER JOIN t3 ON join_condition2
```

```
...
```

```
WHERE where_conditions;"
```

Now, for further simplification of the query above, just assume that you are interested in linking only tables “t1” and “t2” using the syntax below:

```
"SELECT column_list
```

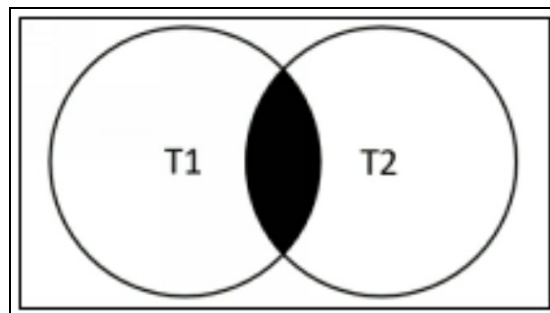
```
FROM t1
```

INNER JOIN t2 ON join_condition;"

The "INNER JOIN" clause will compare each row of the "t1" table with every single record of the "t2" table to determine whether both of them fulfill the defined "join condition". Once the "join condition" has been satisfied, the "INNER JOIN" then returns a new record, consisting of columns from both "t1" and "t2".

Note that the records in both "t1" and "t2" need to be merged according to the "join condition". If no two records from the two tables are identified meeting the "join condition", an empty "result set" is returned by the query. The same logic applies when more than two tables are joined.

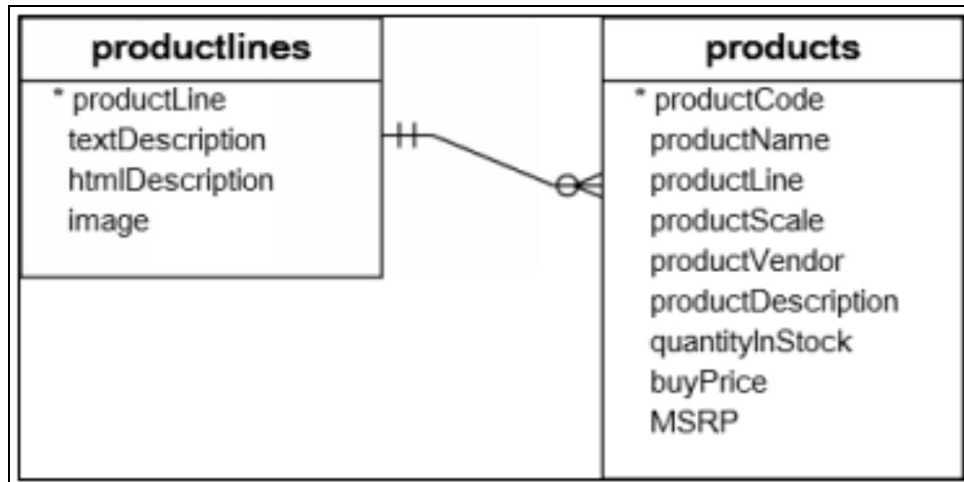
The Venn diagram in the picture below shows the working of the "INNER JOIN" clause. The records in the "result set" are required to be present in both "t1" and "t2" as represented by the section overlapping the 2 circles, depicting each table respectively.



Example

Consider the "products" and "productlines" table in the "MySQL sample database", shown in the picture below. The "products" table is connected to the "productlines" table by referencing the "productline" column. Therefore, the "productline" column serves as "foreign key" in the "products" table.

Conventionally, tables that have "foreign key" relationships are queried using the "JOIN" clause.



Let's assume that you would like to view the "productCode" and "productName" columns from the "products" table as well as "textDescription" of the product lined from the "productlines" table. You can accomplish this by using the syntax below and matching the rows of both the tables on the basis of "productline" columns in the two tables.

"SELECT

productCode,

productName,

textDescription

FROM

products t1

INNER JOIN

productlines t2 ON t1.productline = t2.productline;"

The "result set" is shown in the picture below:

productCode	productName	textDescription
S10_1949	1952 Alpine Renault 1300	Attention car enthusiasts: Make your wildest car ownership dreams come true.
S10_4757	1972 Alfa Romeo GTA	Attention car enthusiasts: Make your wildest car ownership dreams come true.
S10_4962	1962 LanciaA Delta 16V	Attention car enthusiasts: Make your wildest car ownership dreams come true.
S12_1099	1968 Ford Mustang	Attention car enthusiasts: Make your wildest car ownership dreams come true.
S12_1108	2001 Ferrari Enzo	Attention car enthusiasts: Make your wildest car ownership dreams come true.

Here's a tip!

Since both the tables contain identical column called “productline”, you could use the simpler query below (without the need of using the table aliases) and obtain the same “result set” as shown in the picture above:

"SELECT

productCode,

productName,

textDescription

FROM

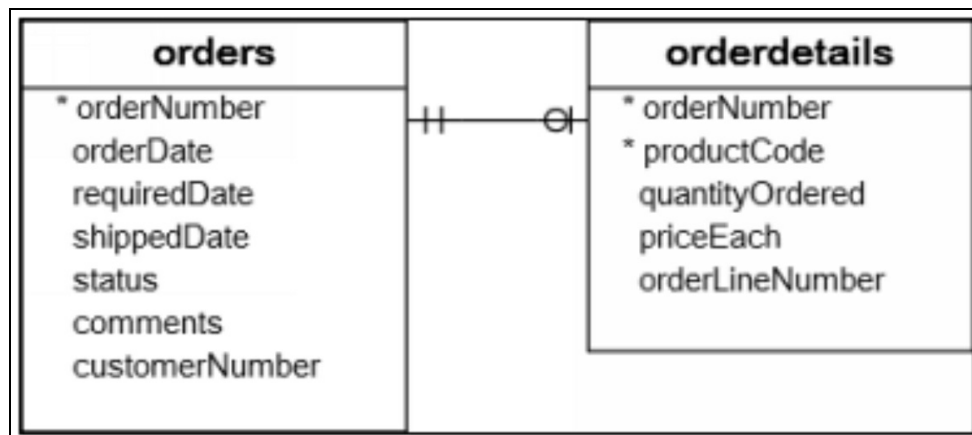
products

INNER JOIN

productlines USING (productline);"

“INNER JOIN” with “GROUP BY”

The picture below contains the “orders” and “orderdetails” tables:



You could utilize "INNER JOIN" with "GROUP BY" clause to obtain desired information from the "orders" and "orderdetails" tables as shown in the two syntax below:

"SELECT

T1.orderNumber,

status,

*SUM(quantityOrdered * priceEach) total*

FROM

```

orders AS T1
  INNER JOIN
  orderdetails AS T2 ON T1.orderNumber = T2.orderNumber
GROUP BY orderNumber;"
"SELECT
  orderNumber,
  status,
  SUM(quantityOrdered * priceEach) total
FROM
  orders
  INNER JOIN
  orderdetails USING (orderNumber)
GROUP BY orderNumber;"

```

The “result set” is shown in the picture below:

	orderNumber	status	total
	10100	Shipped	10223.83
	10101	Shipped	10549.01
	10102	Shipped	5494.78
	10103	Shipped	50218.95
	10104	Shipped	40206.20

MySQL “INNER JOIN” with different operators

We have only explored the join condition using the "equal operator (=)" to match the rows. But we can also use other operators with the join predicate including "greater than (>)", "less than (<)", and "not-equal (< >)" operators.

The syntax below utilizes a "less than (<)" operator to identify the sales price for the item with the code "S10 1678", which are less than the “manufacturer's suggested retail price (MSRP)”.

```

"SELECT

```

```

    orderNumber,
    productName,
    msrp,
    priceEach
FROM
    products p
    INNER JOIN
    orderdetails o ON p.productcode = o.productcode
    AND p.msrp > o.priceEach
WHERE
    p.productcode = 'S10_1678';"

```

The “result set” is shown in the picture below:

	orderNumber	productName	msrp	priceEach
▶	10107	1969 Harley Davidson Ultimate Chopper	95.70	81.35
	10121	1969 Harley Davidson Ultimate Chopper	95.70	86.13
	10134	1969 Harley Davidson Ultimate Chopper	95.70	90.92
	10145	1969 Harley Davidson Ultimate Chopper	95.70	76.56
	10159	1969 Harley Davidson Ultimate Chopper	95.70	81.35
	10168	1969 Harley Davidson Ultimate Chopper	95.70	94.74
	10180	1969 Harley Davidson Ultimate Chopper	95.70	76.56
	10201	1969 Harley Davidson Ultimate Chopper	95.70	82.30

“LEFT JOIN”

The “MySQL LEFT JOIN” enables querying of data from 2 or multiple tables on a database. The "SELECT" statement is not required to contain the "LEFT JOIN" clause for execution, which appears right next to the "FROM" clause. The principles of "left table" and "right table" are implemented when the 2 tables are linked, using the "LEFT JOIN" clause.

Unlike an "INNER JOIN", the "LEFT JOIN" or “LEFT OUTER JOIN” will return all records in the left table, the records meeting the “join condition” and even the records that don’t. “NULL” value appears in the "result set"

of the “columns from the right table for rows that do not match the join condition”.

Now, to further simplify this query we will assume that we are interested in linking only tables “t1” and “t2” with the "LEFT JOIN" per the syntax below:

"SELECT

t1.c1, t1.c2, t2.c1, t2.c2

FROM

t1

LEFT JOIN

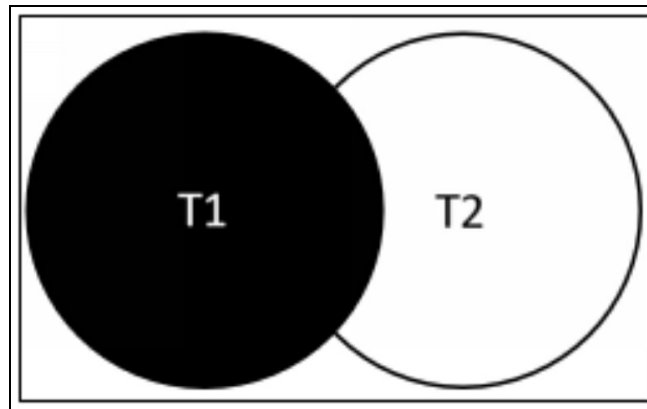
t2 ON t1.c1 = t2.c1;"

In the syntax above, you are using the "LEFT JOIN" clause to connect the "t1" to the "t2", on the basis of a record from the "left table t1" aligning a record from the "right table t2" as defined by the “join predicate (t1.c1= t2.c1)", which are then included in the "result set".

If the record in the “left table” has no matching record that aligns with it in the “right table”, then the record in the “left table” will still be selected and joined with a "virtual record" containing "NULL" values from the “right table”.

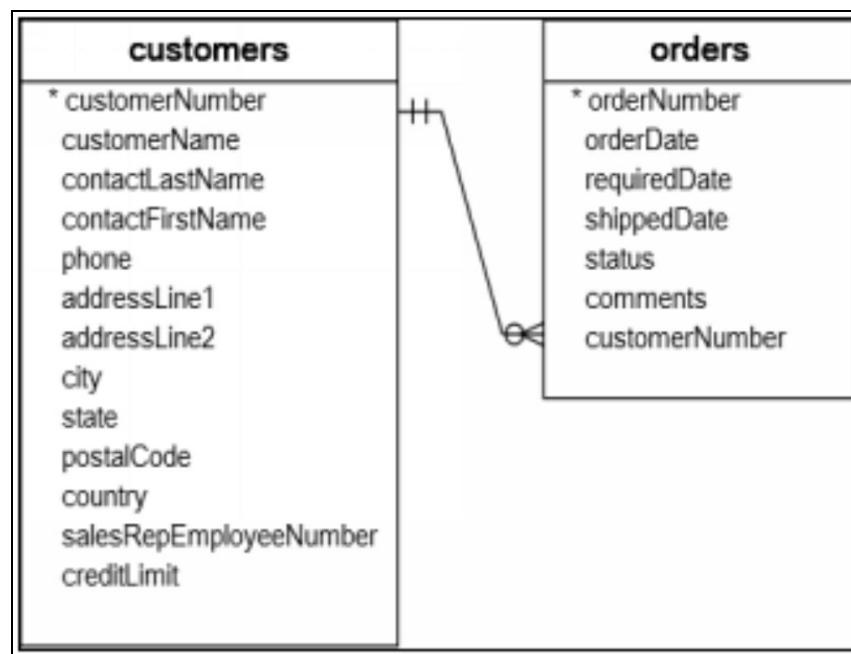
To put it simply, the "LEFT JOIN" clause enables selection of matching records from both the “left and right tables”, as well as all records from the "left table (t1)", even without aligning records from the "right table (t2)".

The “Venn diagram” shown in the picture below will help you understand how the working of the "LEFT JOIN". The overlapping section of the two circles includes records that matched from the two tables, and the rest of the portion of the “left circle” includes records in the "t1" table with “no corresponding record in the t2 table”. Therefore, the "result set" will include all records in the “left table”.



Example

Consider the “customers” and “orders” table in the “MySQL sample database”, shown in the picture below, wherein every “order value” in the “orders” table corresponds to a “customer value” in the “customers” table but every customer value in the “customers” table may not have a corresponding order value in the “orders” table.



All the “orders corresponding to a customer” can be queried using the “LEFT JOIN” clause using the syntax below:

"SELECT

c.customerNumber,

```

c.customerName,
orderNumber,
o.status
FROM
customers c
LEFT JOIN orders o ON c.customerNumber = o.customerNumber;"

```

The “result set” is shown in the picture below, where the left table refers to the "customers" table, so all the rows with value in the "customers" table will be added to the "result set"; although it contains rows with customer data that have no order data such as "168" and "169". For such rows, the order data value is reflected as "NULL" implying that there is no order in the "orders" table for those customers.

	customerNumber	customerName	orderNumber	status
	166	Handji Gifts& Co	10288	Shipped
	166	Handji Gifts& Co	10409	Shipped
	167	Heriku Gifts	10181	Shipped
	167	Heriku Gifts	10188	Shipped
	167	Heriku Gifts	10289	Shipped
	168	American Souvenirs Inc	NULL	NULL
	169	Porto Imports Co.	NULL	NULL
	171	Daedalus Designs Imports	10180	Shipped
	171	Daedalus Designs Imports	10224	Shipped
	172	La Corne D'abondance, ...	10114	Shipped

Here's a tip!

Since both the tables share the column name “customersNumber”, you could utilize the simpler query below (without the need of using the table aliases) and obtain the same “result set” as shown in the picture above:

```

"SELECT
c.customerNumber,
customerName,

```

```
orderNumber,  
status  
FROM  
customers c  
LEFT JOIN orders USING (customerNumber);"
```

MySQL “LEFT JOIN” with “WHERE” clause

Consider the syntax below with “WHERE” clause is used after the “LEFT JOIN” to retrieve desired information from the “orders” and “orderDetails” tables.

```
"SELECT  
    o.orderNumber,  
    customerNumber,  
    productCode  
FROM  
    orders o  
    LEFT JOIN  
    orderDetails USING (orderNumber)  
WHERE  
    orderNumber = 10123;"
```

The “result set” is shown in the picture below for all the order number listed as “10123”:

	orderNumber	customerNumber	productCode
▶	10123	103	S18_1589
	10123	103	S18_2870
	10123	103	S18_3685
	10123	103	S24_1628

Note that if the “join condition” was changed from “WHERE” clause to the “ON” clause as shown in the syntax below, then the query will produce a different “result set” containing all orders but only the details associated with

the order number “10123” will be displayed, as shown in the picture below:

```
"SELECT
    o.orderNumber,
    customerNumber,
    productCode
FROM
    orders o
    LEFT JOIN
    orderDetails d ON o.orderNumber = d.orderNumber
    AND o.orderNumber = 10123;"
```

	orderNumber	customerNumber	productCode
▶	10123	103	S18_1589
	10123	103	S18_2870
	10123	103	S18_3685
	10123	103	S24_1628
	10298	103	NULL
	10345	103	NULL
	10124	112	NULL
	10278	112	NULL
	10346	112	NULL
	10120	114	NULL

“RIGHT JOIN”

The “RIGHT JOIN” or “RIGHT OUTER JOIN” is similar to the “LEFT JOIN” with the only difference being the treatment of the tables, which has been reversed from left to right. Each record from the “right table (t2)” appears in the “result set” with a “RIGHT JOIN”. The “NULL” value will be displayed for columns in the “left table (t1)” for the records in the “right table without any corresponding rows in the left table (t1)”.

You could utilize the “RIGHT JOIN” clause to link tables “t1” and “t2” as illustrated in the syntax below:

```
"SELECT
```

```
*
```

```
FROM t1
```

```
    RIGHT JOIN t2 ON join_predicate;"
```

In the syntax above, the right table is “t2” and the left table is “t1” and “join_predicate” indicated the matching criteria with which records from “t1” will be aligned with records from “t2”.

With the execution of the query above, all the rows from the “right table t2” will be displayed in the “result set” and on the basis of the join condition, if no match are found for “t2” with the rows of “t1” then “NULL” value will be added to those columns from “t1” table.

Example

Let’s consider the tables “t1” and “t2” as described in the syntax below:

```
"CREATE TABLE t1 (
```

```
    id INT PRIMARY KEY,
```

```
    pattern VARCHAR(50) NOT NULL
```

```
);
```

```
CREATE TABLE t2 (
```

```
    id VARCHAR(50) PRIMARY KEY,
```

```
    pattern VARCHAR(50) NOT NULL
```

```
);
```

```
INSERT INTO t1(id, pattern)
```

```
VALUES(1,'Divot'),
```

```
    (2,'Brick'),
```

```
    (3,'Grid');
```

```
INSERT INTO t2(id, pattern)
```

```
VALUES('A','Brick'),
```

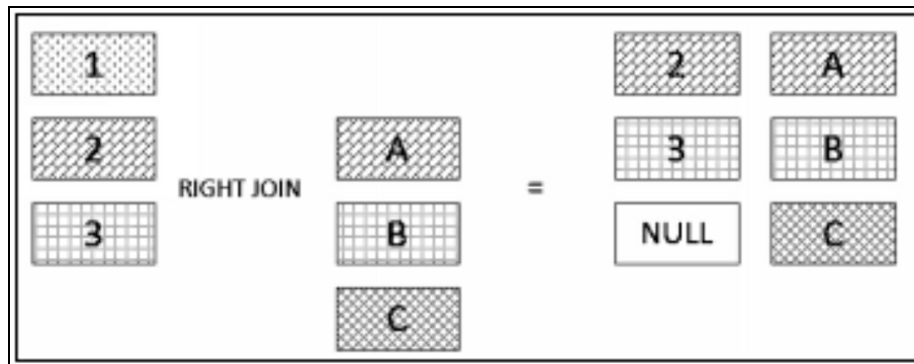
```
    ('B','Grid'),
```

```
    ('C','Diamond');"
```

The tables “t1” and “t2” can be joined with the “pattern” columns, as illustrated in the query below:

```
"SELECT
    t1.id, t2.id
FROM
    t1
    RIGHT JOIN t2 USING (pattern)
ORDER BY t2.id;"
```

The “result set” is shown in the illustrated below:



Or look at the syntax below to view the “sales rep” and their “customers” from the “employees” and “customers” table in the “MySQL sample database”:

```
"SELECT
    concat(e.firstName, ' ', e.lastName) salesman,
    e.jobTitle,
    customerName
FROM
    employees e
    RIGHT JOIN
    customers c ON e.employeeNumber = c.salesRepEmployeeNumber
    AND e.jobTitle = 'Sales Rep'
ORDER BY customerName;"
```

The “result set” is shown in the picture below:

salesman	jobTitle	customerName
Gerard Hernandez	Sales Rep	Alpha Cognac
Foon Yue Tseng	Sales Rep	American Souvenirs Inc
Pamela Castillo	Sales Rep	Amica Models & Co.
NULL	NULL	ANG Resellers
Andy Fixter	Sales Rep	Anna's Decorations, Ltd
NULL	NULL	Anton Designs, Ltd.
NULL	NULL	Asian Shopping Network, Co
NULL	NULL	Asian Treasures, Inc.

“CROSS JOIN”

The “CROSS JOIN” clause is used to obtain a “Cartesian product”, which is defined as a “join of every row of one table to every row of another table”. When the “CROSS JOIN” clause is utilized, the "result set" contains all rows from the two tables, which are a composite of the record from the “first table” and record from the second table. This case exists only when a “common link or column between the two tables” that are joined cannot be found.

The biggest drawback is sheer volume of data, assume each table contains 1,000 rows, then the "result set" will contain 1,000x 1,000= 1,000,000 rows.

You could utilize the "CROSS JOIN" to join “t1” and “t2” as displayed in the syntax below:

"SELECT

*

FROM

T1

CROSS JOIN

T2;"

Unlike the “RIGHT JOIN” and “LEFT JOIN”, the “CROSS JOIN” can be operated without using a “join predicate”. Moreover, if the “WHERE” clause is added to the “CROSS JOIN” and the two tables “t1” and “t2” have a connection then the “CROSS JOIN” will operate as “INNER JOIN”, as

displayed in the syntax below:

```
"SELECT
    *
FROM
    T1
    CROSS JOIN
    T2
WHERE
    T1.id = T2.id;"
```

Example

Let's use "testdb" database as described in the syntax below to better understand how the "CROSS JOIN" operates:

```
"CREATE DATABASE IF NOT EXISTS testdb;
```

```
USE testdb;
```

```
CREATE TABLE products (
```

```
    id INT PRIMARY KEY AUTO_INCREMENT,
```

```
    product_name VARCHAR (100),
```

```
    price DECIMAL (13 , 2 )
```

```
);
```

```
CREATE TABLE stores (
```

```
    id INT PRIMARY KEY AUTO_INCREMENT,
```

```
    store_name VARCHAR (100)
```

```
);
```

```
CREATE TABLE sales (
```

```
    product_id INT,
```

```
    store_id INT,
```

```
    quantity DECIMAL (13 , 2 ) NOT NULL,
```

```
    sales_date DATE NOT NULL,
```

```
PRIMARY KEY (product_id , store_id),
FOREIGN KEY (product_id)
REFERENCES products (id)
ON DELETE CASCADE ON UPDATE CASCADE,
FOREIGN KEY (store_id)
REFERENCES stores (id)
ON DELETE CASCADE ON UPDATE CASCADE
);"
```

In the syntax above, we have 3 tables:

- The “products” table, which holds the fundamental data of the products including “product id”, “product name”, and “sales price”.
- The “stores” table, which holds the data pertaining to the stores where the products are available for purchase.
- The “sales” table, which holds the data pertaining to the products sold in specific stores by date and quantity.

Let’s assume there are 3 items “iPhone”, “iPad” and “MacBook Pro” available for sell in 2 stores named “North” and “South”. We can populate the tables to hold this data using the syntax below:

```
"INSERT INTO products(product_name, price)
VALUES('iPhone', 699),
      ('iPad',599),
      ('Macbook Pro',1299);
INSERT INTO stores(store_name)
VALUES('North'),
      ('South');
INSERT INTO sales(store_id,product_id,quantity,sales_date)
VALUES(1,1,20,'2017-01-02'),
      (1,2,15,'2017-01-05'),
```

```
(1,3,25,'2017-01-05'),  
(2,1,30,'2017-01-02'),  
(2,2,35,'2017-01-05');"
```

Now, to view the “total sales for every store and for every product”, the sales must be calculated first and then “grouped” by the store and product using the syntax below:

```
"SELECT  
    store_name,  
    product_name,  
    SUM (quantity * price) AS revenue  
FROM  
    sales  
    INNER JOIN  
    products ON products.id = sales.product_id  
    INNER JOIN  
    stores ON stores.id = sales.store_id  
GROUP BY store_name , product_name;"
```

The “result set” is shown in the picture below:

	store_name	product_name	revenue
▶	North	iPad	8985.0000
	North	iPhone	13980.0000
	North	Macbook Pro	32475.0000
	South	iPad	20965.0000
	South	iPhone	20970.0000

Now, if you wanted to view the stores that had zero sale for a particular product, the syntax above will not provide you that information. This is where you can utilize the “CROSS JOIN” to first view a composite of all products and stores, as displayed in the query below:

```
"SELECT
    store_name, product_name
FROM
    stores AS a
    CROSS JOIN
    products AS b;"
```

The “result set” is shown in the picture below:

	store_name	product_name
►	North	iPhone
	South	iPhone
	North	iPad
	South	iPad
	North	Macbook Pro
	South	Macbook Pro

And then, you can join the query result received earlier with this query that will result in the “total of the sales by store and product”, using the syntax below:

```
"SELECT
    b.store_name,
    a.product_name,
    IFNULL (c.revenue, 0) AS revenue
FROM
    products AS a
    CROSS JOIN
    stores AS b
    LEFT JOIN
    (SELECT
```



```

    stores.id AS store_id,
    products.id AS product_id,
    store_name,
    product_name,
    ROUND (SUM(quantity * price), 0) AS revenue
FROM
    sales
INNER JOIN products ON products.id = sales.product_id
INNER JOIN stores ON stores.id = sales.store_id
GROUP BY store_name , product_name) AS c ON c.store_id = b.id
AND c.product_id= a.id
ORDER BY b.store_name;"

```

The “result set” is shown in the picture below:

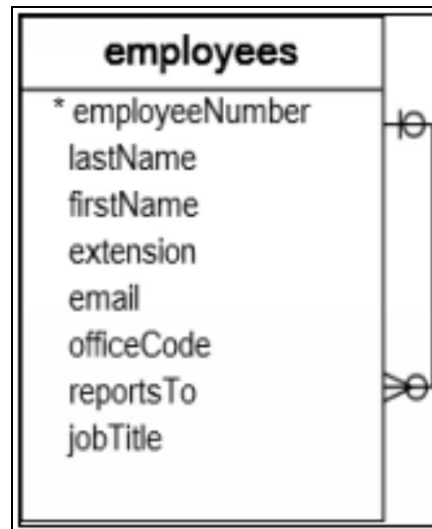
	store_name	product_name	revenue
►	North	Macbook Pro	32475
	North	iPad	8985
	North	iPhone	13980
	South	iPhone	20970
	South	Macbook Pro	0
	South	iPad	20965

The “IFNULL” function was used in the syntax above to return “0” in case the revenue was reported as “NULL”.

“SELF JOIN”

As the name indicates, the “SELF JOIN” statements are utilized to link rows of a table to rows within the same table instead of another table. This requires you to use a “table alias” to aid the server in differentiating between the “left table” from the “right table” within the same query. For instance, take the “employees” table in the “MySQL sample database” where structural data for the organization is stored along with the employee data, as shown in the picture below. The “reportsTo” column can be utilized to view the manager

Id of any employee.



If you would like to view the complete “organization structure”, you could self-join the “employees” table using “employeeNumber” and “reportsTo” columns, using the syntax below. The 2 roles in the “employees” table are “Manager” and “Direct Reports”.

"SELECT

CONCAT (m.lastname, ' ', m.firstname) AS 'Manager',

CONCAT (e.lastname, ' ', e.firstname) AS 'Direct report'

FROM

employees e

INNER JOIN

employees m ON m.employeeNumber = e.reportsto

ORDER BY manager;"

The “result set” is shown in the picture below:

	Manager	Direct report
	Bondur, Gerard	Jones, Barry
	Bondur, Gerard	Bott, Larry
	Bondur, Gerard	Castillo, Pamela
	Bondur, Gerard	Hernandez, Gerard
	Bondur, Gerard	Bondur, Loui
	Bondur, Gerard	Gerard, Martin
	Bow, Anthony	Tseng, Foon Yue
	Bow, Anthony	Patterson, Steve
	Bow, Anthony	Firrelli, Julie
	Bow, Anthony	Thompson, Leslie
	Bow, Anthony	Jennings, Leslie
	Bow, Anthony	VanauF, George

Only employees who are reporting to a manager can be seen in the "result set" above. But the "top manager" is not visible as his name has been filtered out with the "INNER JOIN" clause. The "top manager" is the manager who is not reporting to any other manager or their "manager number" is holding a "NULL" value.

We can alter the "INNER JOIN" clause in the syntax above to the "LEFT JOIN" clause so it would include the "top manager" in the "result set". If the name of the "manager" is holding a "NULL" value, you can utilize the "IFNULL" clause to include the "top manager" in the output, as shown in the query below:

```
"SELECT
    IFNULL (CONCAT (m.lastname, ' ', m.firstname),
        'Top Manager') AS 'Manager',
    CONCAT (e.lastname, ' ', e.firstname) AS 'Direct report'
FROM
    employees e
    LEFT JOIN
    employees m ON m.employeeNumber = e.reportsto
```

ORDER BY manager DESC;"

The “result set” is shown in the picture below:

Manager	Direct report
Top Manager	Murphy, Diane
Patterson, William	King, Tom
Patterson, William	Marsh, Peter
Patterson, William	Fixter, Andy
Patterson, Mary	Bondur, Gerard
Patterson, Mary	Nishi, Mami
Patterson, Mary	Patterson, William
Patterson, Mary	Bow, Anthony
Nishi, Mami	Kato, Yoshimi
Murphy, Diane	Firrelli, Jeff
Murphy, Diane	Patterson, Mary

You will be able to show a “list of customers that are in the same location” by linking the "customers" table to itself with the use of the MySQL self join, as shown in the query below:

"SELECT

c1.city, c1.customerName, c2.customerName

FROM

customers c1

INNER JOIN

customers c2 ON c1.city = c2.city

AND c1.customername > c2.customerName

ORDER BY c1.city;"

The “result set” is shown in the picture below:

	city	customerName	customerName
	Auckland	Kelly's Gift Shop	Down Under Souvenirs, Inc
	Auckland	GiftsForHim.com	Down Under Souvenirs, Inc
	Auckland	Kelly's Gift Shop	GiftsForHim.com
	Boston	Gifts4AllAges.com	Diecast Collectables
	Brickhaven	Online Mini Collectables	Auto-Moto Classics Inc.
	Brickhaven	Collectables For Less Inc.	Auto-Moto Classics Inc.
	Brickhaven	Online Mini Collectables	Collectables For Less Inc.
	Cambridge	Marta's Replicas Co.	Cambridge Collectables Co.
	Frankfurt	Messner Shopping Network	Blauer See Auto, Co.
	Glendale	Gift Ideas Corp.	Boards & Toys Co.
	Lisboa	Porto Imports Co.	Lisboa Souvenirs, Inc
	London	Stylish Desk Decors, Co.	Double Decker Gift Stores, Ltd

MySQL UNION

The “UNION” function in MySQL is used to merge 2 or more “result sets” into one comprehensive “result set”. The syntax for the “UNION” operator is given below:

```
"SELECT column_list
UNION [DISTINCT | ALL]
SELECT column_list
UNION [DISTINCT | ALL]
SELECT column_list
..."
```

The fundamental rules to follow with the use of the “UNION” operator are:

- It is very critical that the number and the sequence of columns included in the “SELECT” statement is the same.
- The “data types” of the columns are required to be identical or “convertible” to the same.
- The redundant or duplicate rows will be removed without explicitly using the “DISTINCT” function in the query.

Example

Consider the sample tables “t1” and “t2” as described in the syntax below:

```

"DROP TABLE IF EXISTS t1;
DROP TABLE IF EXISTS t2;
CREATE TABLE t1 (
    id INT PRIMARY KEY
);
CREATE TABLE t2 (
    id INT PRIMARY KEY
);
INSERT INTO t1 VALUES (1),(2),(3);
INSERT INTO t2 VALUES (2),(3),(4);"

```

The query below can be used to combine the “result sets” from “t1” and “t2” tables using the “UNION” operator:

```

"SELECT id
FROM t1
UNION
SELECT id
FROM t2;"

```

The combined “result set” generated will contain varying values from both the “result sets”, as shown below:

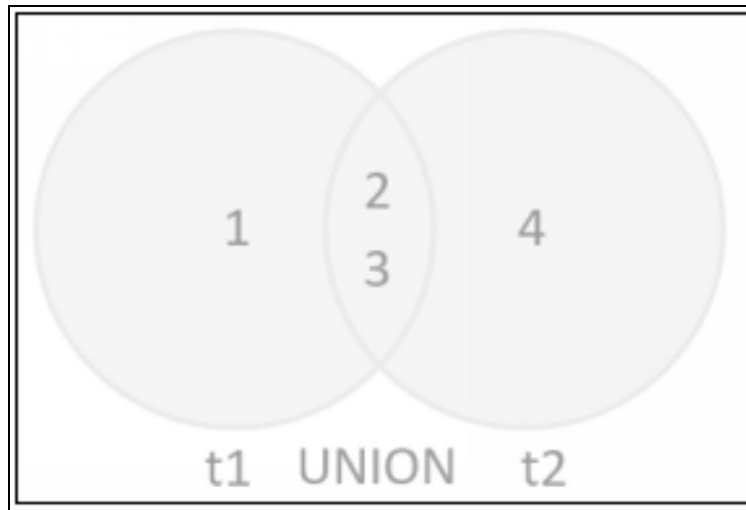
```

"+----+
| id |
+----+
| 1 |
| 2 |
| 3 |
| 4 |
+----+

```

4 rows in set (0.00 sec)"

One can notice that the rows containing values “2” and “3” are redundant, so the “UNION” function dropped the duplicate and retain the distinct values only. The picture below of the Venn diagram, represents the combination of the “result sets” from “t1” and “t2” tables:



MySQL UNION ALL

The “UNION ALL” operator is utilized when you want to retain the duplicate rows (if any) in the “final result set”. The speed with which the query is executed is much higher for the “UNION ALL” operator than the “UNION” or “UNION DISTINCT” operator, as it does not need to deal with the redundancy of the data. The syntax for the “UNION ALL” operator is shown in the query below:

```
"SELECT id
FROM t1
UNION ALL
SELECT id
FROM t2;"
```

The “result set” is given below, which contains duplicate rows:

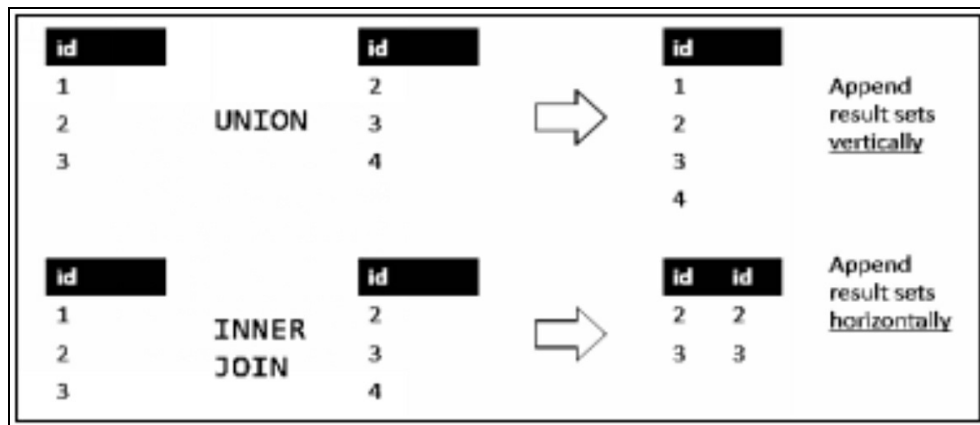
```
"+----+
| id |
+----+
```

```
| 1 |
| 2 |
| 3 |
| 2 |
| 3 |
| 4 |
+----+
```

6 rows in set (0.00 sec)"

MySQL JOIN vs UNION

The “JOIN” clause is utilized to merge the “result sets” horizontally or on the basis of the rows or records. On the other hand, the “UNION” clause is utilized to merge the “result sets” vertically or on the basis of the columns of the tables. The picture below will help you understand the distinction between “UNION” and “JOIN” operators:



MySQL UNION and ORDER BY

The “ORDER BY” clause can be utilized to sort the results of a “UNION” operator as shown the query below:

"SELECT

concat (firstName, ' ', lastName) fullname

FROM

employees

UNION SELECT


```

concat (contactFirstName, ' ', contactLastName)
FROM
customers
ORDER BY fullname;"

```

The “result set” is shown in the picture below:

	fullname
▶	Adrian Huxley
	Akiko Shimamura
	Alejandra Camino
	Alexander Feuer
	Alexander Semenov
	Allen Nelson
	Andy Fixter
	Ann Brown
	Anna O'Hara
	Annette Roulet

If you would like to sort the result on the basis of a column position, you could utilize the “ORDER BY” as shown in the syntax below:

```

"SELECT
concat (firstName, ' ', lastName) fullname
FROM
employees
UNION SELECT
concat (contactFirstName, ' ', contactLastName)
FROM
customers
ORDER BY 1;"

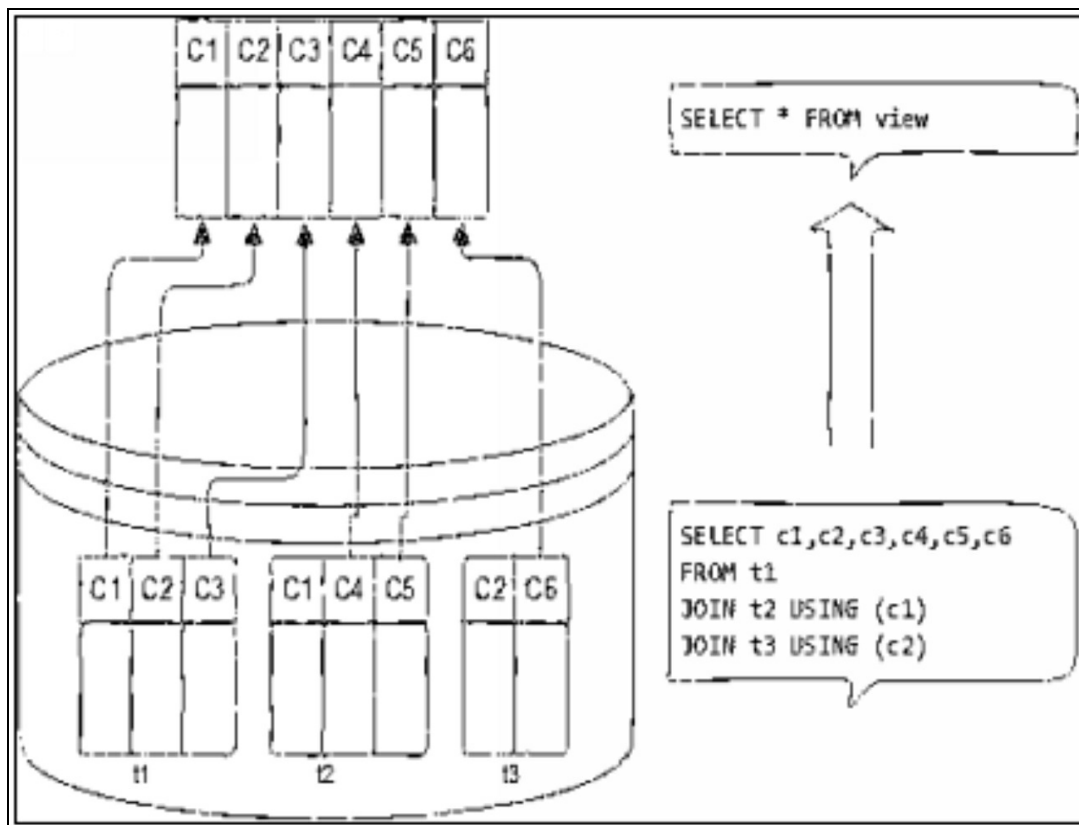
```

Chapter 4

SQL Views and Transactions

A "Database View" in SQL can be defined as a “virtual table” or “logical table” described as "SQL SELECT" statement containing join function. As a "Database View" is like a table in the database consisting of rows and columns, you will be able to easily run queries on it. Many DBMSs, including “MySQL”, enable users to modify information in the existing tables using "Database View" by meeting certain prerequisites, as shown in the picture below.

A "SQL database View" can be deemed dynamic since there is no connection between the "SQL View" to the physical system. The database system will store "SQL Views" in the form on "SELECT" statements using "JOIN" clause. When the information in the table is modified, the "SQL View" will also reflect that modification.



Pros of using SQL View

- A "SQL View" enables simplification of complicated queries: a "SQL View" is characterized by a SQL statement which is associated with multiple tables. To conceal the complexity of the underlying tables from the end users and external apps, "SQL View" is extremely helpful. You only need to use straightforward "SQL" statements instead of complicated ones with multiple "JOIN" clauses using the "SQL View".
- A "SQL View" enables restricted access to information depending on the user requirements. Perhaps you would not like all users to be able to query a subset of confidential information. In such cases "SQL View" can be used to selectively expose non-sensitive information to a targeted set of users.
- The "SQL View" offers an additional layer of safety. Security is a key component of every "relational database management system". The "SQL View" ensures “extra security” for the DBMS. The "SQL View" enables generation of a “read-only” view to display “read-only” information for targeted users. In "read-only view", users are able to only retrieve data and are not allowed to update any information.
- The "SQL View" is used to enable computed columns. The table in the database is not capable of containing computed columns but a "SQL View" can easily contain computed column. Assume in the "OrderDetails" table there is "quantityOrder" column for the amount of products ordered and "priceEach" column for the price of each item. But the "orderDetails" table cannot contain a calculated column storing total sales for every product from the order. If it could, the database schema may have never been a good design. In such a situation, to display the calculated outcome, you could generate a computed column called "total", which would be a product of "quantityOrder" and "priceEach" columns. When querying information from the "SQL View", the calculated column information will be calculated on the fly.
- A "SQL View" allows for backward compatibility. Assume that we have a central database that is being used by multiple applications. Out of the blue you have been tasked to redesign the database accommodating the new business needs. As you delete some tables and create new ones, you would not

want other applications to be affected by these modifications. You could generate "SQL Views" in such situations, using the identical schematic of the “legacy tables” that you are planning to delete.

Cons of using SQL View

In addition to the pros listed above, the use of "SQL View" may have certain disadvantages such as:

- Performance: Executing queries against "SQL View" could be slow, particularly if it is generated from another "SQL View".
- Table dependencies: Since a "SQL View" is created from the underlying tables of the database. Anytime the tables structure connected with "SQL View" is modified, you also need to modify the "SQL View".

Views in MySQL server

As of the release of MySQL version 5+, it has been supporting database views. In MySQL, nearly all characteristics of a "View" conform to the "standard SQL: 2003".

MySQL can run queries against the views in couple of ways:

1. MySQL can produce a “temporary table” on the basis of the “view definition statement” and then execute all following queried on this "temporary table".
2. MySQL can combine the new queries with the query that defines the “SQL View” into a single comprehensive query and then this merged query can be executed.

MySQL offers versioning capability for all "SQL Views". Whenever a “SQL View” is modified or substituted, a clone of the view is backed up in the "arc (archive) directory" residing in a particular database folder. The “backup file” is named as "view name.frm-00001". If you modify your view again, then MySQL will generate a new “backup file” called "view name.frm-00002."

You can also generate a view based on other views through MySQL, by creating references for other views in the "SELECT" statement defining the

target "SQL View".

“CREATE VIEW” in MySQL

The “CREATE VIEW” query can be utilized to generate new “SQL Views” in MySQL, as shown in the syntax below:

"CREATE

[ALGORITHM = {MERGE | TEMPTABLE | UNDEFINED}]

VIEW view_name [(column_list)]

AS

select-statement;"

“View processing algorithms”

The attribute of the algorithm enables you to regulate which mechanism is utilized by MySQL to create the "SQL View". Three algorithms are provided by MySQL, namely: "MERGE", "TEMPTABLE" and "UNDEFINED".

- To use the "MERGE" algorithm, server will start by combining the incoming query with the "SELECT" statement, that will define the "view" into a merged query. Subsequently, the merged query is executed by MySQL to retrieve the "result set". The "MERGE" algorithm cannot be used if the "SELECT" statement consists of “aggregate functions” such as "MIN, MAX, SUM, COUNT, AVG or DISTINCT, GROUP BY, HAVING, LIMIT, UNION, UNION ALL, subquery". If the "SELECT" statement does not refer to a table, then the “MERGE” algorithm cannot be utilized. In the cases, where the "MERGE" algorithm is not permitted, server will instead use the "UNDEFINED" algorithm. Please remember that the view resolution is defined as "the combination of input query and the query in the view definition into one query".
- Using the "TEMPTABLE" algorithm, server will start with production of a "temporary table" that describes the "SQL View" on the basis of the "SELECT" statement. Subsequently any incoming query will be executed against this "temporary table". The "TEMPTABLE" algorithm is lower efficiency than the "MERGE" algorithm because MySQL

requires generation of a "temporary table" to store the "result set" and will transfer the data from the "permanent tables" to the "temporary table". Moreover, a "SQL View" using the "TEMPTABLE" algorithm can not be updated.

- If you generate a "view" without explicitly stating an algorithm that needs to be used, then the "UNDEFINED" algorithm will be used by default. The "UNDEFINED" algorithm allows MySQL to choose from the MERGE or TEMPTABLE algorithm to be used. Since the "MERGE" algorithm is much more effective, MySQL prefers "MERGE" algorithm over "TEMPTABLE" algorithm.

View Name

Views and tables are stored in the same space within the database, so it is not possible to give the same name to a view and a table. Furthermore, the name of a view has to be in accordance with the naming conventions of the table.

“SELECT” statement

It is easy to query desired data from any table or view contained within a database by utilizing the "SELECT" statement. The "SELECT" statement is required to meet various guidelines, such as:

- It may comprise a “subquery” in the "WHERE" clause but cannot have one in the "FROM" clause.
- No variables, including "local variables", "user variables", and "session variables", can be referred to in the "SELECT" statement.
- The "SELECT" statement can also not have a reference to the parameters of prepared statements and is not required to refer to any table in the database.

Example

To create a “SQL View” of the “orderDetails” table that contains the total “sales per order”, you can use the query below:

```
"CREATE VIEW SalePerOrder AS  
SELECT
```

```

        orderNumber, SUM (quantityOrdered * priceEach) total
FROM
    orderDetails
GROUP by orderNumber
ORDER BY total DESC;"

```

By using the "SHOW TABLE" command you can check all the tables in the "classicmodels" database, it is easily visible that the "SalesPerOrder" view is on the displayed list (shown in the picture below):

	Tables_in_classicmodels
▶	customers
	employees
	offices
	orderdetails
	orders
	payments
	productlines
	products
	saleperorder

This proves that the view names and table names are stored in the same space within the database. So, now if you would like to know which object in the picture above is a “view” and which one is a “table”, use the “SHOW FULL TABLE” query and the “result set” will be displayed as shown in the picture below:

	Tables_in_classicmodels	Table_type
▶	customers	BASE TABLE
	employees	BASE TABLE
	offices	BASE TABLE
	orderdetails	BASE TABLE
	orders	BASE TABLE
	payments	BASE TABLE
	productlines	BASE TABLE
	products	BASE TABLE
	saleperorder	VIEW

You can check the “table_type” column in the picture above to confirm which object is a view or a table.

Now, you can simply use the “SELECT” statement below to view “total sales

for each sales order”:

```
"SELECT
```

```
    *
```

```
FROM
```

```
    salePerOrder;"
```

The “result set” is displayed in the picture below:

	orderNumber	total
▶	10165	67392.84
	10287	61402
	10310	61234.66
	10212	59830.54
	10207	59265.14
	10127	58841.35
	10204	58793.53
	10126	57131.92

Creating a “SQL View” from another “SQL view”

The MySQL server permits creation of a new view on the basis of another view. For instance, you could produce a view named "BigSalesOrder" based on the "SalesPerOrder" view, that we created earlier to show every sales order for which the total adds up to more than 60,000 using the syntax below:

```
"CREATE VIEW BigSalesOrder AS
```

```
    SELECT
```

```
        orderNumber, ROUND(total,2) as total
```

```
FROM
```

```
    saleperorder
```

```
WHERE
```

```
    total > 60000;"
```

Now, you could easily retrieve desired data from the "BigSalesOrder" view as shown below:

```
"SELECT
```



```
    orderNumber, total
FROM
    BigSalesOrder;"
```

	orderNumber	total
▶	10165	67392.85
	10287	61402.00
	10310	61234.67

Create “SQL View” with “JOIN” clause

The MySQL database allows you to create “SQL View” using “JOIN” clause. For instance, the query below with the “INNER JOIN” clause can be used to create a view containing "order number", "customer name", and "total sales per order":

```
"CREATE VIEW customerOrders AS
SELECT
    d.orderNumber,
    customerName,
    SUM (quantityOrdered * priceEach) total
FROM
    orderDetails d
    INNER JOIN
    orders o ON o.orderNumber = d.orderNumber
    INNER JOIN
    customers c ON c.customerNumber = d.customerNumber
GROUP BY d.orderNumber
ORDER BY total DESC;"
```

You can use the syntax below to retrieve desired data from the “customerOrders” view:

```
"SELECT
    *
```

FROM

customerOrders;"

The “result set” is displayed in the picture below:

	orderNumber	customerName	total
►	10165	Dragon Souvenirs, Ltd.	67392.84
	10287	Wda Sport, Ltd	61402
	10310	Terra Spezialitäten, Ltd	61234.66
	10212	Euro+ Shopping Channel	59830.54
	10207	Diecast Collectables	59265.14
	10127	Muscle Machine Inc	58841.38
	10204	Muscle Machine Inc	58793.53
	10126	Comida Auto Replicas, Ltd	57131.92

Create “SQL View” with a subquery

The query below can be used to generate a “SQL View” with a “subquery”, containing products with purchase prices greater than the average product prices.

"CREATE VIEW aboveAvgProducts AS

SELECT

productCode, productName, buyPrice

FROM

products

WHERE

buyPrice >

(SELECT

AVG(buyPrice)

FROM

products)

ORDER BY buyPrice DESC;"

The query to extract data from the “aboveAvgProducts” view is even more straightforward, as shown in the picture below:

"SELECT

*

FROM

aboveAvgProducts;"

The “result set” is displayed in the picture below:

	productCode	productName	buyPrice
▶	S10_4962	1962 LanciaA Delta 16V	103.42
	S18_2238	1998 Chrysler Plymouth Prowler	101.51
	S10_1949	1952 Alpine Renault 1300	98.58
	S24_3856	1966 Porsche 356A Coupe	98.3
	S12_1108	2001 Ferrari Enzo	96.59
	S12_1099	1968 Ford Mustang	95.34
	S18_1984	1995 Honda Civic	93.89
	S18_4027	1970 Triumph Spitfire	91.92

Updatable SQL Views

The MySQL server offers the capability to not only query the view but also to update them. The “INSERT” or “UPDATE” statements can be used add and edit the records of the underlying table through the updatable “MySQL View”. Moreover, the “DELETE” statement can be used to drop records from the underlying table through the updatable “MySQL View”.

Furthermore, the "SELECT" statement used to generate and define an updatable view can not include any of the functions listed below:

- “Aggregate functions” such as “MIN, MAX, SUM, AVG, and COUNT”.
- “DISTINCT”.
- “GROUP BY”.
- “HAVING”.
- “UNION” or “UNION ALL”.
- “LEFT JOIN” or “OUTER JOIN”.
- “Subquery in the SELECT statement or in the WHERE clause referring to the table indicated in the FROM clause”.
- “Reference to non-updatable view in the FROM clause”.
- “Reference only to literal values”.
- “Multiple references to any column of the underlying table”.

Remember, the “TEMPTABLE” algorithm cannot be used to generate an updatable “MySQL View”.

Example

You can use the syntax below to generate a view called “officeInfo” on the basis of the “offices” table in the “MySQL sample database”. This “updatable view” will refer to 3 columns of the "offices" table: "officeCode", "phone", and "city".

```
"CREATE VIEW officeInfo
```

```
AS
```

```
    SELECT officeCode, phone, city
```

```
    FROM offices;"
```

Now, if you would like to query the data from the “officeInfo” view, you can easily do that by using the syntax below:

```
"SELECT
```

```
    *
```

```
FROM
```

```
    officeInfo;"
```

The “result set” is displayed in the picture below:

	officeCode	phone	city
►	1	+1 650 219 4782	San Francisco
	2	+1 215 837 0825	Boston
	3	+1 212 555 3000	NYC
	4	+33 14 723 4404	Paris
	5	+81 33 224 5000	Tokyo
	6	+61 2 9264 2451	Sydney
	7	+44 20 7877 2041	London

You could also update the office contact number with "officeCode" 4 using the "UPDATE" statement as given below, on the "officeInfo" view:

```
"UPDATE officeInfo
```

```
SET
```

```
    phone = '+33 14 723 5555'
```

```
WHERE
```

```
    officeCode = 4;"
```

Lastly, you can confirm this modification, using the syntax below to retrieve desired data from the "officeInfo" view:

```
"SELECT
    *
FROM
    officeInfo
WHERE
    officeCode = 4;"
```

The “result set” is displayed in the picture below:

	officeCode	phone	city
►	4	+33 14 723 5555	Paris

Check if an existing view is updatable

By running a query against the "is_updatable" column from the view in the "information_schema" database, you should verify whether a view in the database is updatable or not.

You can use the query below to display all the views from the "classicmodels" database and check which of them are updatable:

```
"SELECT
    table_name,
    is_updatable
FROM
    information_schema.views
WHERE
    table_schema = 'classicmodels';"
```

The “result set” is displayed in the picture below:

	table_name	is_updatable
▶	aboveavgproducts	YES
	customerorders	NO
	officeinfo	YES
	saleperorder	NO

Dropping rows using “SQL View”

To understand this concept, execute the syntax below to first create a table called “items”, use the “INSERT” statement to add records into this table and then use the “CREATE” clause to generate a view containing items with prices higher than “700”.

```
-- create a new table named items
```

```
CREATE TABLE items (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    price DECIMAL(11 , 2 ) NOT NULL
);
```

```
-- insert data into the items table
```

```
INSERT INTO items(name,price)
VALUES('Laptop',700.56),('Desktop',699.99),('iPad',700.50) ;
```

```
-- create a view based on items table
```

```
CREATE VIEW LuxuryItems AS
SELECT
    *
FROM
    items
WHERE
    price > 700;
```

```
-- query data from the LuxuryItems view
```

```
SELECT
    *
FROM
    LuxuryItems;"
```

The “result set” is displayed in the picture below:

	id	name	price
▶	1	Laptop	700.56
	3	iPad	700.50

Now, use the “DELETE” clause to drop a record with “id value 3”.

```
"DELETE FROM LuxuryItems
WHERE
    id = 3;"
```

After you run the query above, you will receive a message stating “1 row(s) affected”.

Now, you can verify the data with the view, using the query below:

```
"SELECT
    *
FROM
    LuxuryItems;"
```

The “result set” is displayed in the picture below:

	id	name	price
▶	1	Laptop	700.56

Finally, use the syntax below to retrieve desired data from the underlying table “items” to confirm if the “DELET” statement in fact removed the record:

```
"SELECT
    *
```

FROM

items;"

The “result set” is displayed in the picture below, which confirms that the row with “id value 3” was deleted from the “items” table:

	id	name	price
▶	1	Laptop	700.56
	2	Desktop	699.99

Modification of “SQL View”

In MySQL, you can use “ALTER VIEW” and “CREATE OR REPLACE VIEW” statements to make changes to an existing view.

Using “ALTER VIEW” statement

The syntax for “ALTER VIEW” is very similar to the “CREATE VIEW” statement that you learned earlier, the only difference being that the “ALTER” keyword is used instead of the “CREATE” keyword, as shown below:

```
"ALTER
```

```
[ALGORITHM = {MERGE | TEMPTABLE | UNDEFINED}]
```

```
VIEW [database_name]. [view_name]
```

```
AS
```

```
[SELECT statement]"
```

The query below will change the “organization” view by incorporating the “email” column in the table:

```
"ALTER VIEW organization
```

```
AS
```

```
SELECT CONCAT(E.lastname,E.firstname) AS Employee,
```

```
        E.email AS employeeEmail,
```

```
        CONCAT(M.lastname,M.firstname) AS Manager
```

```
FROM employees AS E
```



```

INNER JOIN employees AS M
  ON M.employeeNumber = E.ReportsTo
ORDER BY Manager;"

```

You can run the query below against the “organization” view to verify the modification:

```

"SELECT
  *
FROM
  Organization;"

```

The “result set” is displayed in the picture below:

	Employee	employeeEmail	Manager
▶	JonesBarry	bjones@classicmodelcars.com	BondurGerard
	HernandezGerard	ghernande@classicmodelcars.com	BondurGerard
	BottLarry	lbott@classicmodelcars.com	BondurGerard
	GerardMartin	mgerard@classicmodelcars.com	BondurGerard
	BondurLoui	lbondur@classicmodelcars.com	BondurGerard
	CastilloPamela	pcastillo@classicmodelcars.com	BondurGerard
	VanaufGeorge	gvanauf@classicmodelcars.com	BowAnthony

Using “CREATE OR REPLACE VIEW” statement

The “CREATE OR REPLACE VIEW” can be used to replace or generate a “SQL View” that already exists in the database. For all existing views, MySQL will easily modify the view but if the view is non-existent, it will create a new view based on the query.

The syntax below can be used to generate the “contacts view” on the basis of the “employees” table:

```

"CREATE OR REPLACE VIEW contacts AS
  SELECT
    firstName, lastName, extension, email
  FROM

```

employees;"

The “result set” is displayed in the picture below:

	firstName	lastName	extension	email
►	Diane	Murphy	x5800	dmurphy@classicmodelcars.com
	Mary	Patterson	x4611	mpatterso@classicmodelcars.com
	Jeff	Firrelli	x9273	jfirrelli@classicmodelcars.com
	William	Patterson	x4871	wpatterson@classicmodelcars.com
	Gerard	Bondur	x5408	gbondur@classicmodelcars.com
	Anthony	Bow	x5428	abow@classicmodelcars.com
	Leslie	Jennings	x3291	ljennings@classicmodelcars.com

Now, assume that you would like to insert the “jobtitle” column to the “contacts view”. You can accomplish this with the syntax below:

"CREATE OR REPLACE VIEW contacts AS

SELECT

firstName,

lastName,

extension,

email,

jobtitle

FROM

employees;"

The “result set” is displayed in the picture below:

	firstName	lastName	extension	email	jobtitle
►	Diane	Murphy	x5800	dmurphy@classicmodelcars.com	President
	Mary	Patterson	x4611	mpatterso@classicmodelcars.com	VP Sales
	Jeff	Firrelli	x9273	jfirrelli@classicmodelcars.com	VP Marketing
	William	Patterson	x4871	wpatterson@classicmodelcars.com	Sales Manager (APAC)
	Gerard	Bondur	x5408	gbondur@classicmodelcars.com	Sale Manager (EMEA)
	Anthony	Bow	x5428	abow@classicmodelcars.com	Sales Manager (NA)
	Leslie	Jennings	x3291	ljennings@classicmodelcars.com	Sales Rep

Dropping a “SQL View”

The “DROP VIEW” statement can be utilized to delete an existing view from the database, using the syntax below:

"DROP VIEW [IF EXISTS] [database_name].[view_name]"

The "IF EXISTS" clause is not mandatory in the statement above and is used to determine if the view already exists in the database. It prevents you from mistakenly removing a view that does not exist in the database.

You may, for instance, use the "DROP VIEW" statement as shown in the syntax below to delete the "organization" view:

"DROP VIEW IF EXISTS organization;"

SQL TRANSACTIONS

A transaction can be defined as "a unit of work that is performed against a database". They are “units or work sequences” that are performed in a “logical order”, either manually by a user or automatically using by the database program.

Or simply put, they are “the spread of one or more database modifications”. For instance, if you create a row, update a row, or delete a row from the table, that means you are executing a transaction on that table. To maintain data integrity and address database errors, it is essential to regulate these transactions.

Basically, to execute a transaction, you must group several SQL queries and run them at the same time.

Properties of Transactions

The fundamental properties of a transaction can be defined using the acronym “**ACID**” for the properties listed below:

- **Atomicity** – guarantees successful completion of all operations grouped in the work unit. Or else, at the point of failure, the transaction will be aborted and all prior operations will be rolled back to their original state.
- **Consistency** – makes sure that when a transaction is properly committed, the database states are also correctly updated.
- **Isolation** – allows independent and transparent execution of the transactions.
- **Durability** – makes sure that in case of a system malfunction, the outcome or impact of a committed transaction continues to exist.

To explain this concept in greater detail, consider the steps below for addition of a new sales order in the “MySQL sample database”:

- Start by querying the most recent “sales order number” from the "orders" table and utilize the next “sales order number” as the new “order number”.
- Then use the "INSERT" clause to add a new “sales order” into the "orders" table.
- Next, retrieve the “sales order number” that was inserted in the previous step.
- Now, "INSERT" the new “sales order items” into the "orderdetails" table containing the “sales order numbers”.
- At last, to verify the modifications, select data from both "orders" and "orderdetails" tables.

Try to think, how would the sales order data be modified, if even a single steps listed here were to fail, for whatever reason. For instance, if the step for inserting items of an order to the "orderdetails" table failed, it will result in an “empty sales order”.

This is where the "transaction processing" is used as a safety measure. You can perform "MySQL transactions" to run a set of operations making sure that the database will not be able to contain any partial operations. When working with multiple operations concurrently, if even one of the

operation fails, a "rollback" can be triggered. If there is no failure, all the statements will be “committed” to the database.

“MySQL Transaction” statements

MySQL offers statements listed below for controlling the transactions:

- For initiating a transaction, utilize the "START TRANSACTION" statement. The "BEGIN" or "BEGIN WORK" statement are same as the "START TRANSACTION".
- For committing the “current transaction” and making the modifications permanent, utilize the "COMMIT" statement.
- By using the "ROLLBACK" declaration, you can simply undo the current transaction and void its modifications.
- By using the "SET auto-commit" statement, you can deactivate or activate the auto-commit mode for the current transaction.

By default, MySQL is designed to commit the modifications to the database permanently. By using the statement below, you can force MySQL not to commit the modifications by default:

```
"SET autocommit = 0;
```

Or

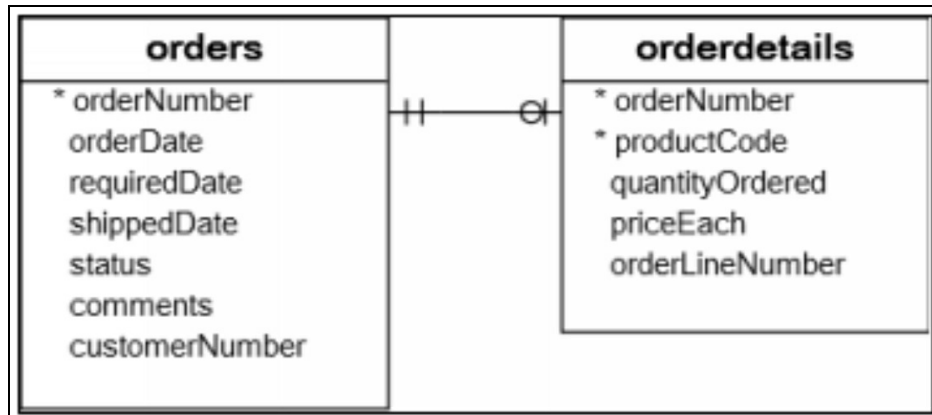
```
SET autocommit = OFF;"
```

To reactivate the default mode for auto-commit, you can use the syntax below:

```
"SET autocommit = ON;"
```

Example

Let’s utilize the “orders” and “orderDetails” tables, shown in the picture below, from the “MySQL sample database” to understand this concept further.



“COMMIT” transaction

You must first split the SQL statement into logical parts to effectively use a transaction and assess when the transaction needs to be committed or rolled back.

The steps below show how to generate a new “sales order”:

1. Utilize the "START TRANSACTION" statement to begin a transaction.
2. Select the most recent “sales order number” from the "orders" table and utilize the subsequent “order number” as the new “order number”.
3. “Insert” a new sales order in the "orders" table.
4. “Insert” sales order items into the "orderdetails" table.
5. Lastly, use the "COMMIT" statement to commit the transaction.

You could potentially use data from “orders” and “orderdetails” table to verify the new “sales order”, as shown in the syntax below:

“ start a new transaction

START TRANSACTION;

Get the latest order number

SELECT

@orderNumber:=MAX(orderNUmber)+1

FROM

orders;

insert a new order for customer 145

```
INSERT INTO orders (orderNumber,
                    orderDate,
                    requiredDate,
                    shippedDate,
                    status,
                    customerNumber)
```

```
VALUES (@orderNumber,
        '2005-05-31',
        '2005-06-10',
        '2005-06-11',
        'In Process',
        145);
```

Insert order line items

```
INSERT INTO orderdetails(orderNumber,
                          productCode,
                          quantityOrdered,
                          priceEach,
                          orderLineNumber)
```

```
VALUES (@orderNumber,'S18_1749', 30, '136', 1),
        (@orderNumber,'S18_2248', 50, '55.09', 2);
```

commit changes

```
COMMIT;"
```

The “result set” is displayed in the picture below:

	@orderNumber:=IFNULL(MAX(orderNUmber),0)+1
▶	10426

Using the query below, you can retrieve the new sales order that you just created:

```

"SELECT
    a.orderNumber,
    orderDate,
    requiredDate,
    shippedDate,
    status,
    comments,
    customerNumber,
    orderLineNumber,
    productCode,
    quantityOrdered,
    priceEach
FROM
    orders a
    INNER JOIN
    orderdetails b USING (orderNumber)
WHERE
    a.ordernumber = 10426;"

```

The “result set” is displayed in the picture below:

	orderNumber	orderDate	requiredDate	shippedDate	status	comments	customerNumber	orderLineNumber	productCode	quantityOrdered	priceEach
▶	10426	2005-05-31	2005-06-10	2005-06-11	In Process	NULL	145	1	S18_1749	50	136.00
	10426	2005-05-31	2005-06-10	2005-06-11	In Process	NULL	145	2	S18_2248	50	55.09

“ROLLBACK” transaction

The first step here is deletion of the data in the “orders” table, using the statement below:

```

"mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)
mysql> DELETE FROM orders;
Query OK, 327 rows affected (0.03 sec)"

```


It's obvious from the output of the queries above that all records from the “orders” table have been removed.

Now, you need to login to the “MySQL database server” in a distinct session and run the query below against the "orders" table.

```
"mysql> SELECT COUNT(*) FROM orders;
```

```
+-----+  
| COUNT(*) |  
+-----+  
|    327 |  
+-----+
```

```
1 row in set (0.00 sec)"
```

The data from the “orders” table can still be viewed from the second session.

The changes made in the first session are not permanent so you need to either commit them or roll them back. Let’s assume you would like to undo the changes from the first session, utilize the query below:

```
"mysql> ROLLBACK;
```

```
Query OK, 0 rows affected (0.04 sec)"
```

The data in the “orders” table from the first session can be verified, using the syntax below:

```
"mysql> SELECT COUNT(*) FROM orders;
```

```
+-----+  
| COUNT(*) |  
+-----+  
|    327 |  
+-----+
```

```
1 row in set (0.00 sec)"
```

It can be seen in the “result set” above that the modifications have been rolled back.

“SAVEPOINT” Transaction

The “SAVEPOINT” is defined as “a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction”, as shown in the syntax below:

"SAVEPOINT SAVEPOINT_NAME;"

The query above is only used to create a “SAVEPOINT” within all the transaction statements and then the “ROLLBACK” query can be utilized to retract desired transactions, as shown in the query below:

"ROLLBACK TO SAVEPOINT_NAME;"

For example, assume you have a “customers” table as shown in the picture below and want to delete 3 rows and create a “SAVEPOINT” prior to every deletion and then subsequent “ROLLBACK” the change to the preceding state as needed.

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

You can accomplish this by running the queries below:

SQL> SAVEPOINT SP1;

Savepoint created.

SQL> DELETE FROM CUSTOMERS WHERE ID=1;

1 row deleted.

SQL> SAVEPOINT SP2;

Savepoint created.

SQL> DELETE FROM CUSTOMERS WHERE ID=2;

1 row deleted.

SQL> SAVEPOINT SP3;

Savepoint created.

```
SQL> DELETE FROM CUSTOMERS WHERE ID=3;
```

1 row deleted."

With the query above, you have successfully deleted 3 records and are now ready to use the query below to "ROLLBACK" the change to the "SAVEPOINT" named "SP2", which was generated post first deletion so the last 2 deletions will be rolled back:

```
"SQL> ROLLBACK TO SP2;
```

Rollback complete."

The "result set" is displayed in the picture below:

ID	NAME	AGE	ADDRESS	SALARY
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

"RELEASE SAVEPOINT" Transaction

The "RELEASE SAVEPOINT" can be utilized to delete a "SAVEPOINT" that might have been generated earlier, as shown in the syntax below:

```
"RELEASE SAVEPOINT SAVEPOINT_NAME;"
```

Remember, once you run the query above there is no function undo the transactions that might have been executed after the last "SAVEPOINT".

"SET TRANSACTION"

The "SET TRANSACTION" can be utilized to start a transaction by specifying features of the subsequent transactions. For instance, using the syntax below, a transaction can be made "read and write only":

```
"SET TRANSACTION [ READ WRITE | READ ONLY ];"
```

Database Recovery Models

A recovery model can be defined as "a database property that controls how transactions are logged, whether the transaction log requires and allows

backing up, and what kinds of restore operations are available". It is a database configuration option that specifies the type of backup that can be performed and enables the data to be restored or recovered in case of a failure. The 3 types of "SQL Server recovery models" are:

SIMPLE

This model has been deemed as the easiest of all the recovery models that exist. It offers "full, differential, and file level" backups. However, backup of the transaction log is not supported. The log space would be reused whenever the checkpoint operation of the SQL Server background process takes place. The log file's inactive part is deleted and made accessible for reuse.

There is no support for point-in-time and page restoration, only the secondary read-only files can be restored.

FULL

All types of transactions "(DDL (Data Definition Language) as well as the DML (Data Manipulation Language))" in the transaction log file can be fully registered in the "FULL" recovery model. The sequence of logs remains untouched and maintained for restoration activities of the database. Unlike the "Simple Recovery Model", during the "CHECKPOINT" transaction, the "transaction log" file would not be truncated automatically.

It supports all restoration activities, such as "point-in-time restore, page restore and file restore".

Bulk Logged

The "Bulk Logged recovery model" is a unique database configuration option, which produces similar outcome as the "FULL recovery model" with the exception of some "bulk operations" that can be logged only minimally. The "transaction log" file utilizes a method called as "minimal logging for bulk operations". The caveat being that specific point in time data can not be restored.

"SQL BACKUP DATABASE" Statement

The "BACKUP DATABASE" statement can be utilized to generate a full back up of a database that already exists on the SQL server, as displayed in the syntax below:

"BACKUP DATABASE databasename

```
TO DISK = 'filepath';"
```

"SQL BACKUP WITH DIFFERENTIAL" Statement

A "differential backup" is used to selectively back up the sections of the database which were altered after the last “full backup of the database”, as displayed in the syntax below:

```
"BACKUP DATABASE databasename
```

```
TO DISK = 'filepath'
```

```
WITH DIFFERENTIAL;"
```

Example

Consider that you have a database called “testDB” and you would like to create a full back up of it. To accomplish this, you can use the query below:

```
"BACKUP DATABASE testDB
```

```
TO DISK = 'D:\backups\testDB.bak';"
```

Now, if you made modifications to the database after running the query above. You can use the query below to create a back up of those modifications:

```
"BACKUP DATABASE testDB
```

```
TO DISK = 'D:\backups\testDB.bak'
```

```
WITH DIFFERENTIAL;"
```

RESTORING Database

The Restoration can be defined as the method by which “data is copied from a backup and logged transactions are applied to that data”. The backup file generated (as discussed earlier) is used to return database to a former state. There are couple of methods to accomplish this:

T-SQL

The syntax for this method is given below:

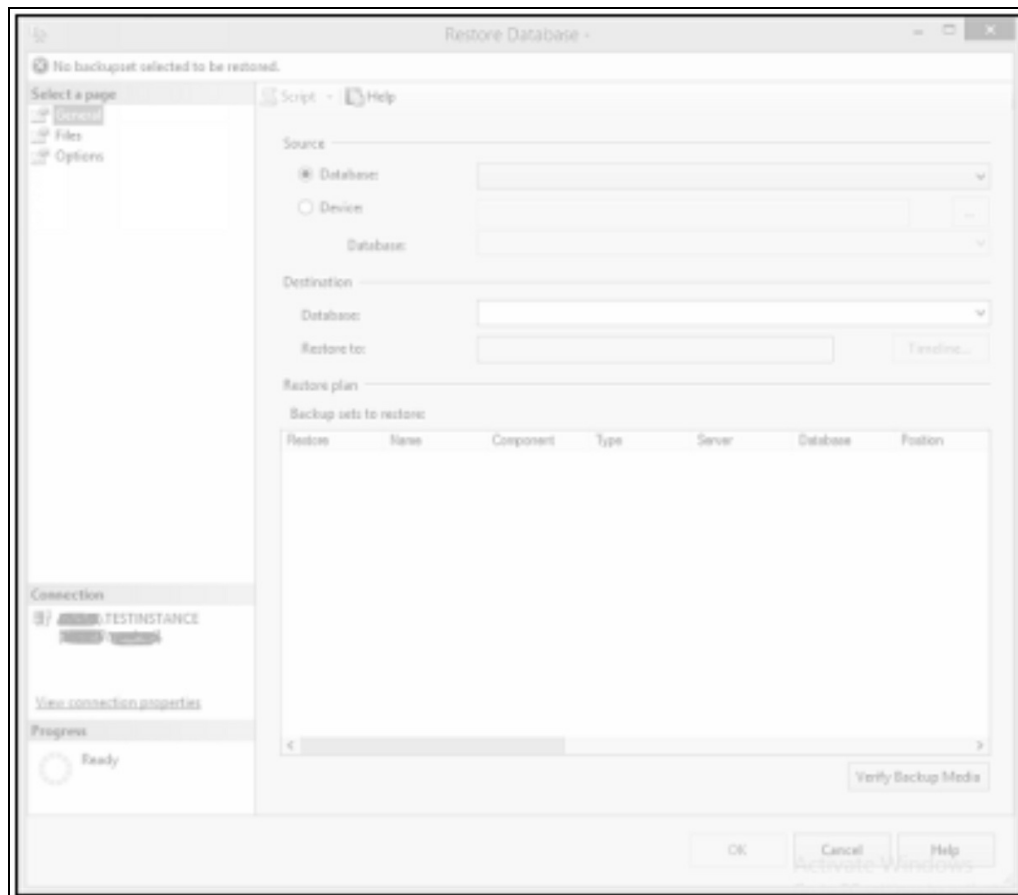
```
"Restore database <Your database name> from disk = '<Backup file location + file name>';"
```

For instance, the query below can be utilized to restore a database named “TestDB” using the back up file named “TestDB_Full.bak”, located in “D:\:

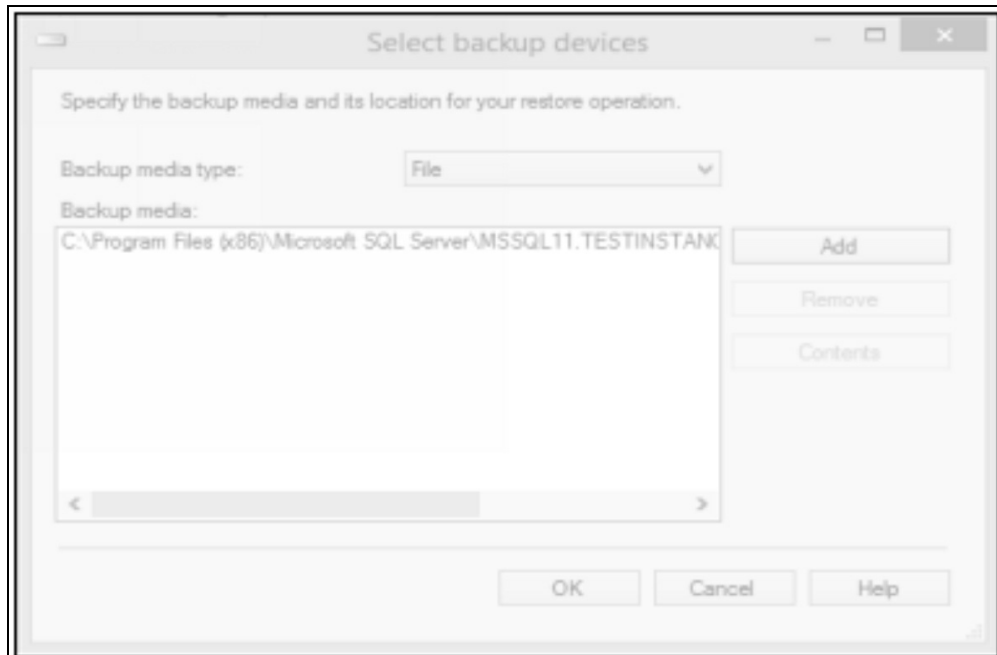
"Restore database TestDB from disk = ' D:\TestDB_Full.bak' with replace;"

“SSMS (SQL SERVER Management Studio)”

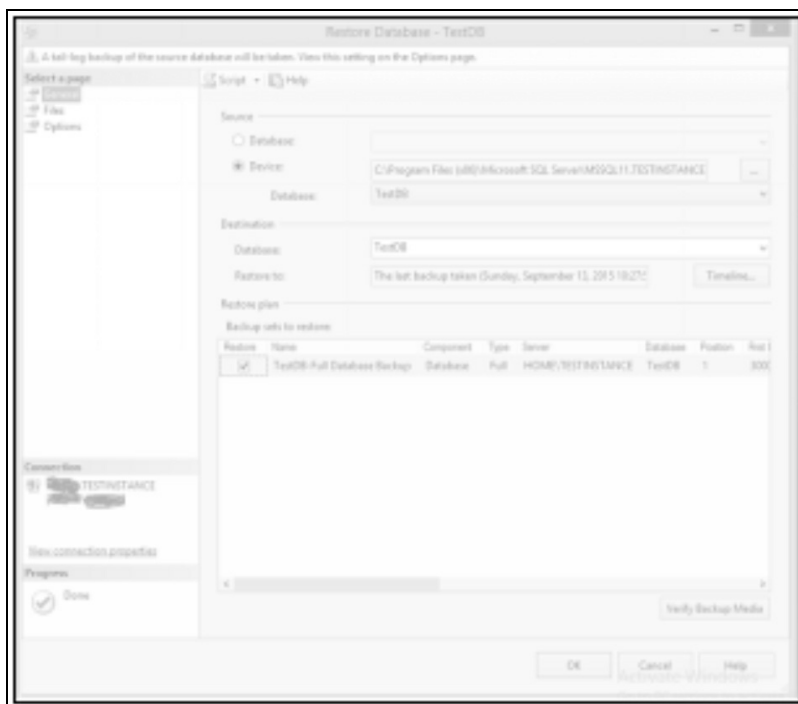
First connect to the "TESTINSTANCE" database then right-click on databases folder and click on "Restore database", as shown in the picture below.



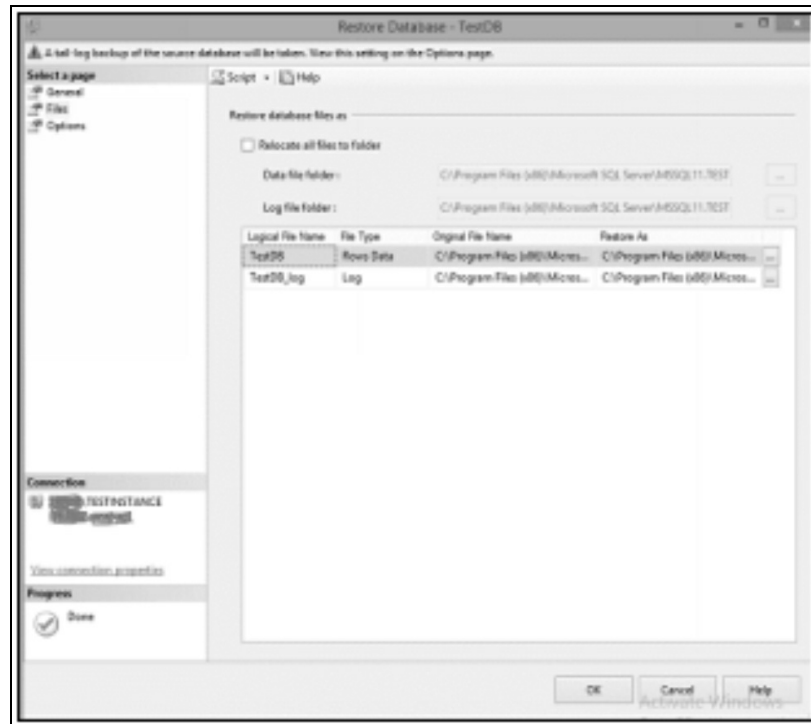
Now, to select the backup file as shown in the picture below, “select the device radio button and click on the ellipse”.



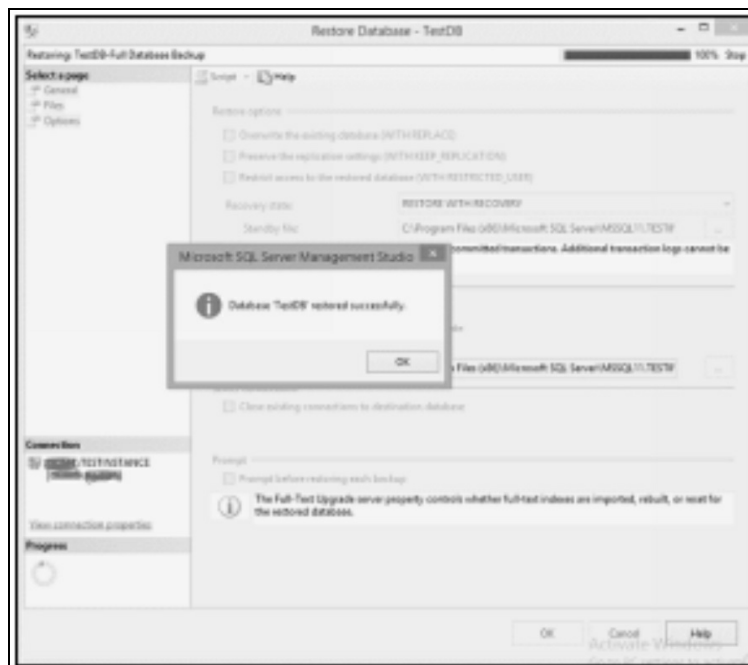
Next, click Ok and the window shown below will be displayed to you.



Now, from the “top-left corner of the screen select the Files option”, as shown in the picture below:



Lastly, select “Options” from the “top-left corner of your screen” and click “OK” to restore the “TestDB” database, as depicted in the picture below:



Chapter 5

Database Security and Administration

MySQL has an integrated advanced access control and privilege system that enables generation of extensive access guidelines for user activities and efficiently prevent unauthorized users from getting access to the database system.

There are 2 phases in MySQL access control system for when a user is connected to the server:

- **Connection verification:** Every user must have a valid username and password, that is connected to the “MySQL database server”. Moreover, the host used by the user to connect must be the same as the host in the "MySQL grant table".
- **Request verification:** After a link has been effectively created for every query executed by the user, MySQL will verify if the user has required privileges to run that specific query. MySQL is capable of checking user privileges at database, table, and field level.

The MySQL installer will automatically generate a database called "mysql". The "mysql database" is comprised of 5 main grant tables. Using "GRANT" and "REVOKE" statements like, these tables can be indirectly manipulated.

- **User:** includes columns for "user accounts" and "global privileges". MySQL either accepts or rejects a connection from the host using these user tables. A privilege given under the "user table" is applicable to all the databases on the “MySQL server”.
- **Database:** comprises “database level” privileges. MySQL utilizes the “db_table” to assess the database that can be used by a user to access and to host the connection. A "database level" privilege in the “db_table” is applicable to the particular database and all the object available in that database, such as “tables, triggers, views, stored procedures”, and many more.
- **“Table_priv” and “columns_priv”:** includes the "table level" and "column level" privileges. A privilege given in the

"table_priv" table is applicable to that particular table and its columns, on the other hand a privilege given in the "columns_priv" table is applicable only to a particular column of the table.

- **“Procs_priv”**: includes privileges for "stored functions" and "stored processes".

MySQL uses the tables listed above to regulate MySQL database server privileges. It is extremely essential to understand these tables before implementing your own dynamic access control system.

Creating User Accounts

You are able to indicate in MySQL not just which user is allowed to connect to the database server, but also which host the user can use to build that connection. As a result, for every user account a "username" and a "host name" in MySQL is generated and divided by the "@" character.

For instance, if an "admin user" is connected from a "localhost" to the "MySQL database server", the user account will be named as "admin@localhost". However, the "admin_user" is only allowed to connect to the "MySQL database server" using a "localhost" or a remote host like "mysqltutorial.org", which ensures that the MySQL database server has higher security.

Moreover, by merging the "username" and "host", various accounts with the same name can be configured and still possess the ability to connect from distinct hosts while being given distinct privileges, as needed.

In the "mysql database" all the user accounts are stored in the "user grant" table.

Using “MySQL CREATE USER” statement

The “CREATE USER” is utilized with the MySQL server to setup new user accounts, as shown in the syntax below:

```
"CREATE USER user_account IDENTIFIED BY password;"
```

In the syntax above, the "CREATE USER" clause is accompanied by the name of the user account in “username @ hostname” format.

In the "IDENTIFIED BY" clause, the user password would be indicated. It is important that the password is specified in plain text. Prior to the user account being saved in the "user" table, MySQL is required to encrypt the

"user password".

For instance, the "CREATE USER" statement can be utilized as given below, for creating a new "user dbadmin", that is connected to the "MySQL database server" from "localhost" using user password as "SECRET".

```
"CREATE USER dbadmin@localhost
IDENTIFIED BY 'SECRET';"
```

If you would like to check the "privileges of a user account", you can run the syntax below:

```
"SHOW GRANTS FOR dbadmin@localhost;"

"+-----+
| Grants for dbadmin@localhost      |
+-----+
| GRANT USAGE ON *.* TO `dbadmin`@`localhost` |
+-----+

1 row in set (0.00 sec)"
```

The *.* in the "result set" above indicates that the "dbadmin user" account is allowed to log into the server only and does not have any other access privilege.

Bear in mind that the portion prior to the "dot (.)" is representing the database and the portion after the "dot (.)" is representing the table, for example, "database.table".

The "percentage (%)" wildcard can be used as shown in the syntax below for allowing a user to create a connection from any host:

```
"CREATE USER superadmin@'%
IDENTIFIED BY 'secret';"
```

The "percentage (%)" wildcard will lead to identical result as when included in the "LIKE" operator, for example, to enable "mysqladmin user" account to link to the server from any "subdomain" of the "mysqldata.org" host, the "percentage (%)" wildcard can be used as shown in the syntax below:

```
"CREATE USER mysqladmin@'%mysqldata.org'
IDENTIFIED by 'secret';"
```

It should also be noted here, that another wildcard “underscore (_)” can be used in the “CREATE USER” statement.

If the host name portion can be omitted from the user account, server will recognize it and enable the user to get connected from any host. For instance, the syntax below will generate a new remote user account that allows creation of a connection to the server from any host:

```
"CREATE USER remote;"
```

To view the privileges given to the “remote” account, you can use the syntax below:

```
"SHOW GRANTS FOR remote;"
```

```
"+-----+
| Grants for remote@%          |
+-----+
| GRANT USAGE ON *.* TO `remote`@`%` |
+-----+
```

```
1 row in set (0.00 sec)"
```

It is necessary to remember that the single quote (' ') in the syntax above is particularly significant, if the user account has special characters like "underscore" or "percentage".

If you inadvertently cite the user account as "username@hostname", the server will create a user with the "username@hostname" name and enables the user to connect from any host that might not be as expected.

The syntax below, for instance, can be used to generate a new account "api@localhost" that can be connected to the “MySQL database server” from any host.

```
"CREATE USER 'api@localhost';"
```

```
"SHOW GRANTS FOR 'api@localhost';"
```

```
"+-----+
```

```
| Grants for api@localhost@%          |
+-----+
| GRANT USAGE ON *.* TO `api@localhost`@`%` |
+-----+
```

1 row in set (0.00 sec)"

If you accidentally generate a user that already exists in the database, then an error will be issued by MySQL. For instance, the syntax below can be used to generate a new user account called "remote":

"CREATE USER remote;"

The error message below will be displayed on your screen:

"ERROR 1396 (HY000): Operation CREATE USER failed for 'remote'@'%"

It can be noted that the "CREATE USER" statement will only create a new user without any privileges. The "GRANT" statement can be utilized to give access privileges to the users.

Updating “USER PASSWORD”

Prior to altering a MySQL user account password, the concerns listed below should be taken into consideration:

- The user account whose password you would like to be modified.
- The applications that are being used with the user account for which you would like to modify the password. If the password is changed without altering the application connection string being used with that user account, then it would be not feasible for those applications to get connected with the database server.

MySQL offers a variety of statements that can be used to alter a user's password, such as "UPDATE", "SET PASSWORD", and "GRANT USAGE" statements.

Let's explore some of these syntaxes!

Using “UPDATE” Statement

The “UPDATE” can be utilized to make updates to the “user” table of the “mysql” database. You must also run the "FLUSH PRIVILEGES" statement to refresh privileges from the "grant" table in the

"mysql" database, by executing the "UPDATE" statement.

Assume that you would like to modify the password for the "dbadmin" user, which links from the "localhost" to the "dolphin". You can accomplish this by executing the query below:

```
"USE mysql;  
UPDATE user  
SET password = PASSWORD('dolphin')  
WHERE user = 'dbadmin' AND  
      host = 'localhost';  
FLUSH PRIVILEGES;"
```

Using “SET PASSWORD” Statement

For updating the user password, the "user@host" format is utilized. INow, imagine that you would like to modify the password for some other user's account, then you will be required to have the "UPDATE" privilege on your user account.

With the use of the "SET PASSOWORD" statement, the "FLUSH PRIVILEGES" statement is not required to be executed, in order to reload privileges to the "mysql" database from the "grant" tables.

The synatx below could be utilized to alter the "dbadmin" user account password, by executing the "SET PASSWORD" statement:

```
"SET PASSWORD FOR 'dbadmin'@'localhost' = PASSWORD('bigshark');"
```

Using “ALTER USER” Statement

Another method to alter the password for a user is by using the “ALTER USER” statement with the “IDENTIFIED BY” clause. For instance, the query below can be executed to change the password of the “dbadmin” user to “littlewhale”.

```
"ALTER USER dbadmin@localhost IDENTIFIED BY 'littlewhale';"
```

USEFUL TIP

If you need to change the password of the MySQL “root account”, then you would need to force quit the “MySQL database server” and restart the server without triggering the “grant table validation”.

Granting User Privileges

As a new user account is created, there are no access privileges afforded to the user by default. The "GRANT" statement must be used in order "to grant privileges to the user account". The "GRANT" statement syntax is shown below:

```
"GRANT privilege, [privilege],.. ON privilege_level  
TO user [IDENTIFIED BY password]  
[REQUIRE tsl_option]  
[WITH [GRANT_OPTION | resource_option]];"
```

- In the syntax above, we start by specifying one or more privileges following the "GRANT" clause. Every privilege being granted to the user account must be isolated using a "comma", in case you would like to give the user account more than one privilege at the same time. (The list of potential privileges that can be "granted to a user account" is given in the table below).
- After that you must indicate the "privilege_level" that will determine the level at which the privileges are applied. The privilege level supported by MySQL are "global (*. *)", "database (database. *)", "table (database.table)" and "column" level. If you are using the "column" privilege level, single or multiple (a list of comma separated columns after each privilege) must be indicated.
- Next you need to indicate the user that needs to be granted the privileges. If the indicated user can be found on the server, then the "GRANT" statement will modify its privilege. Or else, a new user account will be created by the "GRANT" statement. The "IDENTIFIED BY" clause is not mandatory and enables creation of a new password for the user.
- Thereafter, it's indicated if the user needs to start a connection to the database via a "secure connection" such as "SSL", "X059", etc.
- At last, the "WITH GRANT OPTION" clause is added which is not mandatory but enables granting and revoking the privileges of other user, that have been "granted to your own user account". Moreover, the "WITH" clause can also be used to assign

the resources from the MySQL database server, for example, putting a limit on the number of links or statements that can be used by the user per hour. In shared environments like "MySQL shared hosting", the "WITH" clause is extremely useful.

Note that the "GRANT OPTION" privilege and the privileges you are looking to grant to other users must already be configured to your user account, so that you are able to use the "GRANT" statement. If the read only system variable has been allowed, then execution of the "GRANT" statement requires the "SUPER" privilege.

PRIVILEGE	DESCRIPTION	LEVEL Global	LEVEL Database	LEVEL Table	LEVEL Column	LEVEL Procedure	LEVEL Proxy
"ALL"	"Grant all privileges at specified access level except GRANT OPTION"						
"ALTER"	"Allow user to use of ALTER TABLE statement"	Y	Y	Y			
"ALTER ROUTINE"	"Allow user to alter or drop stored routine"	Y	Y			Y	
"CREATE"	"Allow user to create database and table"	Y	Y	Y			
"CREATE ROUTINE"	"Allow user to create stored routine"	Y	Y				
"CREATE TABLESPACE"	"Allow user to create, alter or drop table spaces and log file groups"	Y					
"CREATE TEMPORARY TABLES"	"Allow user to create temporary table by using CREATE TEMPORARY TABLE"	Y	Y				
"CREATE USER"	"Allow user to use the CREATE USER, DROP USER, RENAME USER, and REVOKE ALL	Y					

	PRIVILEGES statements”						
“CREATE VIEW”	“Allow user to create or modify view”	Y	Y	Y			
“DELETE”	“Allow user to use DELETE”	Y	Y	Y			
“DROP”	“Allow user to drop database, table and view”	Y	Y	Y			
“EVENT”	“Enable use of events for the Event Scheduler”	Y	Y				
“EXECUTE”	“Allow user to execute stored routines”	Y	Y	Y			
“FILE”	“Allow user to read any file in the database directory.”	Y					
“GRANT OPTION”	“Allow user to have privileges to grant or revoke privileges from other accounts.”	Y	Y	Y		Y	Y
“INDEX”	“Allow user to create or remove indexes.”	Y	Y	Y			
“INSERT”	“Allow user to use INSERT statement”	Y	Y	Y	Y		
“LOCK TABLES”	Allow user to use LOCK TABLES on tables for which you have the SELECT privilege”	Y	Y				
“PROCESS”	“Allow user to see all processes with SHOW PROCESSLIST statement”	Y					
“PROXY”	“Enable user proxying”						
REFERENCES	“Allow user to create foreign key”	Y	Y	Y	Y		
“RELOAD”	“Allow user to use FLUSH operations”	Y					

“REPLICATION CLIENT”	“Allow user to query to see where master or slave servers are”	Y					
“REPLICATION SAVE”	“Allow the user to use replicate slaves to read binary log events from the master”	Y					
“SELECT”	“Allow user to use SELECT statement”	Y	Y	Y	Y		
“SHOW DATABASES”	“Allow user to show all databases”	Y					
“SHOW VIEW”	“Allow user to use SHOW CREATE VIEW statement”	Y	Y	Y			
“SHUTDOWN”	“Allow user to use mysqladmin shutdown command”	Y					
“SUPER”	“Allow user to use other administrative operations such as CHANGE MASTER TO, KILL, PURGE BINARY LOGS, SET GLOBAL, and mysqladmin command”	Y					
“TRIGGER”	“Allow user to use TRIGGER operations”	Y	Y	Y			
“UPDATE”	“Allow user to use UPDATE statement”	Y	Y	Y	Y		
“USAGE”	“Equivalent to no privileges”						

Example

More often than not, the "CREATE USER" statement will be used to first create a new user account and then the "GRANT" statement is used to assign the user privileges.

For instance, a new "super user" account can be created by the executing the "CREATE USER" statement given below:

```
"CREATE USER super@localhost IDENTIFIED BY 'dolphin';"
```

In order to check the privileges granted to the “super@localhost” user, the query below with “SHOW GRANTS” statement can be used.

```
"SHOW GRANTS FOR super@localhost;"
```

```
"+-----+
| Grants for super@localhost          |
+-----+
| GRANT USAGE ON *.* TO `super`@`localhost` |
+-----+
1 row in set (0.00 sec)"
```

Now, if you wanted to assign all privileges to the “super@localhost” user, the query below with “GRANT ALL” statement can be used.

```
"GRANT ALL ON *.* TO 'super'@'localhost' WITH GRANT OPTION;"
```

The "ON*. *" clause refers to all databases and items within those databases. The "WITH GRANT OPTION" enables "super@localhost" to assign privileges to other user accounts.

If the "SHOW GRANTS" statement is used again at this point then it can be seen that privileges of the "super@localhost's" user have been modified, as shown in the syntax and the “result set” below:

```
"SHOW GRANTS FOR super@localhost;"
```

```
"+-----+
| Grants for super@localhost          |
+-----+
| GRANT ALL PRIVILEGES ON *.* TO `super`@`localhost` WITH
GRANT OPTION |
+-----+
1 row in set (0.00 sec)"
```

Now, assume that you want to create a new user account with all the server privileges in the “classicmodels sample database”. You can accomplish this by using the query below:

```
"CREATE USER auditor@localhost IDENTIFIED BY 'whale';  
GRANT ALL ON classicmodels.* TO auditor@localhost;"
```

Using only one "GRANT" statement, various privileges can be granted to a user account. For instance, to generate a user account with the privilege of executing "SELECT", "INSERT" and "UPDATE" statements against the database "classicmodels", the query below can be used.

```
"CREATE USER rfc IDENTIFIED BY 'shark';  
GRANT SELECT, UPDATE, DELETE ON classicmodels.* TO rfc;"
```

Revoking User Privileges

You will be using the "MySQL REVOKE" statement to revoke the privileges of any user account(s). MySQL enables withdrawal of one or more privileges or even all of the previously granted privileges of a user account.

The query below can be used to revoke particular privileges from a user account:

```
"REVOKE  privilege_type [(column_list)]  
        [, priv_type [(column_list)]]...  
ON [object_type] privilege_level  
FROM user [, user]..."
```

In the syntax above, we start by specifying a list of privileges that need to be revoked from a user account next to the "REVOKE" keyword. YAs you might recall when listing multiple privileges in a statement they must be separated by commas. Then we indicate the "privilege level" at which the "ON" clause will be revoking these privileges. Lastly we indicate the user account whose privileges will be revoked in the "FROM" clause.

Bear in mind that your own user account must have the "GRANT OPTION" privilege as well as the privileges that you want to revoke from other user accounts.

You will be using the "REVOKE" statement as shown in the syntax below, if you are looking to withdraw all privileges of a user account:

```
"REVOKE ALL PRIVILEGES, GRANT OPTION FROM user [, user]..."
```

It is important to remember that you are required to have the "CREATE USER" or the "UPDATE" privilege at "global level" for the mysql database,

to be able to execute the "REVOKE ALL" statement.

You will be using the "REVOKE PROXY" clause as shown in the query below, in order to revoke proxy users:

```
"REVOKE PROXY ON user FROM user [, user]..."
```

A proxy user can be defined as "a valid user in MySQL who can impersonate another user". As a result, the proxy user is able to attain all the privileges granted to the user that it is impersonating.

The best practice dictates that you should first check what privileges have been assigned to the user by executing the syntax below with "SHOW GRANTS" statement, prior to withdrawing the user's privileges:

```
"SHOW GRANTS FOR user;"
```

Example

Assume that there is a user named "rfc" with "SELECT", "UPDATE" and "DELETE" privileges on the "classicmodels sample database" and you would like to revoke the "UPDATE" and "DELETE" privileges from the "rfc" user. To accomplish this, you can execute the queries below.

To start with we will check the user privileges using the "SHOW GRANTS" statement below:

```
"SHOW GRANTS FOR rfc;"
```

```
"GRANT SELECT, UPDATE, DELETE ON 'classicmodels'.* TO 'rfc'@'%"
```

At this point, the "UPDATE" and "REVOKE" privileges can be revoked from the "rfc" user, using the query below:

```
"REVOKE UPDATE, DELETE ON classicmodels.* FROM rfc;"
```

Next, the privileges of the "rfc" user can be checked with the use of "SHOW GRANTS" statement.

```
"SHOW GRANTS FOR 'rfc'@'localhost';"
```

```
"GRANT SELECT ON 'classicmodels'.* TO 'rfc'@'%"
```

Now, if you wanted to revoke all the privileges from the "rfc" user, you can use the query below:

```
"REVOKE ALL PRIVILEGES, GRANT OPTION FROM rfc;"
```

To verify that all the privileges from the "rfc" user have been revoked, you

will need to use the query below:

```
"SHOW GRANTS FOR rfc;"
```

```
"GRANT USAGE ON *.* TO 'rfc'@'%"
```

Remember, as mentioned in the privileges description table earlier in this book, the “USAGE” privilege simply means that the user has no privileges in the server.

Resulting impact of the “REVOKE” query

The impact of "MySQL REVOKE" statement relies primarily on the level of privilege granted to the user account, as explained below:

- The modifications made to the "global privileges" will only take effect once the user has connected to the MySQL server in a successive session, post the successful execution of the "REVOKE" query. The modifications will not be applicable to all other users connected to the server, while the "REVOKE" statement is being executed.
- The modifications made to the database privileges are only applicable once a "USE" statement has been executed after the execution of the "REVOKE" query.
- The "table and column privilege" modifications will be applicable to all the queries executed, after the modifications have been rendered with the "REVOKE" statement.

Conclusion

Thank you for making it through to the end of SQL Coding For Beginners, let's hope it was informative and able to provide you with all of the tools you need to achieve your goals whatever they may be.

The next step is to make the best use of your new found wisdom and learning of the SQL programming language, that can position you to enter the world of systems and data analysis. It is very important that you follow the instructions in this book and install the free and open MySQL user interface on your operating system, so you can get hands on practice and be able to create not only correct but efficient SQL queries to succeed at work or during job interviews. As is the case with most of the learnings in life repeated practice is the key to achieve perfection. So let this book be your guiding beacon through the journey of learning a programming language for relational database management systems. To make the best use of this book, as you read and understand all the concepts and SQL query structures, try to come up with your own names and labels to be used within the presented examples and verify the result set up obtained with the one given in this book.

Finally, if you found this book useful in any way, a good review on Amazon is always appreciated!

© Copyright 2020 by **Leonard Base** All rights reserved.

This document is geared towards providing exact and reliable information with regards to the topic and issue covered.

In no way is it legal to reproduce, duplicate, or transmit any part of this document in either electronic means or in printed format. Recording of this publication is strictly prohibited and any storage of this document is not

allowed unless with written permission from the publisher. All rights reserved.

The information provided herein is stated to be truthful and consistent, in that any liability, in terms of inattention or otherwise, by any usage or abuse of any policies, processes, or directions contained within is the solitary and utter responsibility of the recipient reader. Under no circumstances will any legal responsibility or blame be held against the publisher for any reparation, damages, or monetary loss due to the information herein, either directly or indirectly.

Respective authors own all copyrights not held by the publisher.

The information herein is offered for informational purposes solely, and is universal as so. The presentation of the information is without contract or any type of guarantee assurance.

The trademarks that are used are without any consent, and the publication of the trademark is without permission or backing by the trademark owner. All trademarks and brands within this book are for clarifying purposes only and are the owned by the owners themselves, not affiliated with this document