12 Simple Queries in SQL

In This Chapter

12.1 INTRODUCTION

SQL, Structured Query Language, was developed in 1970s in an IBM Laboratory. SQL, sometimes also referred to as *SEQUEL* is a 4th generation non-procedural language. The non-procedural languages just need to be specified 'WHAT' of the problem. That is, in non-procedural languages 'WHAT is required' is specified in contrast to 3GL's 'HOW it is to be done'. Since SQL is non-procedural and that is what makes it simple, it makes an RDBMS possible.

SQL, being non-procedural, describes WHAT all data is to be retrieved or inserted or modified or deleted, rather than specifying code describing HOW to perform the entire operation.

SQL, enables the following : (*i*) Creating/modifying a database's structure (*ii*) Changing security settings for system (*iii*) Permitting users for working on databases or tables (*iv*) Querying database (*v*) Inserting/Modifying/Deleting the database contents.

This chapter is dedicated to making simple queries. But before that let us talk about some important elements of SQL.

12.2 SOME MYSQL SQL ELEMENTS

The MySQL implementation of SQL has certain elements that play an important role in defining/querying a database. In this section, we shall talk about some basic elements of MySQL SQL that you must be aware of.

These basic elements are :

(*i*) Literals (*ii*) Datatypes (*iii*) Nulls (*iv*) Comments

12.2.1 Literals

Literals, in general, refer to a fixed data value. This fixed data value may be of *character* type or *numeric* literal. For example, *'Synthia'*, *'Ekagra'*, *'Raunak Raj Singh'*, *'8'* and *'305'* are all *character text literals*. Notice that all character literals are enclosed in single quotation marks or double quotation marks. Characters that are not enclosed in quotation marks refer to the schema object names *e.g., value* refers to a schema object whereas *'value'* refers to a character literal.

NOTE

A numeric literal can store a maximum of 53 digits of precision.

To store an apostrophe in a text literal, you should use \' (backslash followed by apostrophe *e.g.,* to store *Kush's* in a text literal, you should write it as 'Kush\'s').

Numbers that are not enclosed in quotation marks are numeric literals *e.g.,* 22, 18, 1997, 2003 are all *numeric literals*.

NOTE

A text literal can have maximum length of 4000 bytes in MySQL 5.1.

Numeric literals can either be **integer literals** *i.e.,* without any decimal or be **real literals** *i.e.,* with a decimal point *e.g., 17* is an integer literal but *17.0* and *17.5* are *real literals*.

12.2.2 Data Types

Data types are means to identify the type of data and associated operations for handling it. A value's *datatype* associates a fixed set of properties with the value. These properties cause different treatment of datatypes. For example, you can add values of NUMBER datatypes, but you cannot add values of CHAR or VARCHAR types. Internal datatypes supported by MySQL include the ones given here.

MySQL uses many different data types, divided into *three* categories :

- Numeric
- Date and time, and
- String types.

Numeric Data Types

MySQL uses all the standard ANSI SQL numeric data types. The following list shows the common numeric data types and their descriptions.

INT

A normal-sized integer that can be signed or unsigned. If signed, the allowable range is from –2147483648 to 2147483647. If unsigned, the allowable range is from 0 to 4294967295. You can specify a width of up to 11 digits.

TINYINT

A very small integer that can be signed or unsigned. If signed, the allowable range is from –128 to 127. If unsigned, the allowable range is from 0 to 255. You can specify a width of up to 4 digits.

SMALLINT

A small integer that can be signed or unsigned. If signed, the allowable range is from –32768 to 32767. If unsigned, the allowable range is from 0 to 65535. You can specify a width of up to 5 digits.

MEDIUMINT	*A medium-sized integer that can be signed or unsigned.* If signed, the allowable range is from -8388608 to 8388607. If unsigned, the allowable range is from 0 to 16777215. You can specify a width of up to 9 digits.
BIGINT	*A large integer that can be signed or unsigned.* If signed, the allowable range is from -9223372036854775808 to 9223372036854775807. If unsigned, the allowable range is from 0 to 18446744073709551615. You can specify a width of up to 11 digits.
FLOAT(M,D)	*A floating-point number that cannot be unsigned.* You can define the display length (M) and the number of decimals (D). This is not required and will default to 10,2, where 2 is the number of decimals and 10 is the total number of digits (including decimals). Decimal precision can go to 24 places for a FLOAT.
DOUBLE(M,D)	*A double precision floating-point number that cannot be unsigned.* You can define the display length (M) and the number of decimals (D). This is not required and will default to 16,4, where 4 is the number of decimals. Decimal precision can go to 53 places for a DOUBLE. REAL is a synonym for DOUBLE.
DECIMAL(M,D)	*An unpacked floating-point number that cannot be unsigned.* In unpacked decimals, each decimal corresponds to one byte. Defining the display length (M) and the number of decimals (D) is required. NUMERIC is a synonym for DECIMAL.

Date and Time Types

The MySQL date and time datatypes are :

DATE	*A date in YYYY-MM-DD format, between 1000-01-01 and 9999-12-31.* For example, December 30th, 1973 would be stored as 1973-12-30.
DATETIME	*A date and time combination in YYYY-MM-DD HH:MM:SS format, between 1000-01-01 00:00:00 and 9999-12-31 23:59:59.* For example, 3:30 in the afternoon on December 30th, 1973 would be stored as 1973-12-30 15:30:00.
TIMESTAMP	*A timestamp between midnight, January 1, 1970 and sometime in 2038.* This looks like the previous DATETIME format, only without the hyphens between numbers; 3:30 in the afternoon on December 30th, 1973 would be stored as 19731230153000 (YYYYMMDDHHMMSS).
TIME	*Stores the time in HH:MM:SS format.*
YEAR(M)	*Stores a year in 2-digit or 4-digit format. If the length is specified as 2 (for example YEAR(2)), YEAR can be 1970 to 2069 (70 to 69). If the length is specified as 4, YEAR can be 1901 to 2155. The default length is 4.*

String/Text Types

Most data that you use in a database is in string format. This list describes the common string datatypes in MySQL.

CHAR(M)	*A fixed-length string between 1 and 255 characters in length* (for example CHAR(5)), right-padded with spaces to the specified length when stored. Defining a length is not required, but the default is 1.
VARCHAR(M)	*A variable-length string between 1 and 255 characters in length;* for example VARCHAR(25). You must define a length when creating a VARCHAR field.
BLOB or TEXT	*A field with a maximum length of 65535 characters.* BLOBs are "Binary Large Objects" and are used to store large amounts of binary data, such as images or other types of files. Fields defined as TEXT also hold large amounts of data; the difference between the two is that sorts and comparisons on stored data are case sensitive on BLOBs and are not case sensitive in TEXT fields. You do not specify a length with BLOB or TEXT.

TINYBLOB or TINYTEXT	*A BLOB or TEXT column with a maximum length of 255 characters.* You do not specify a length with TINYBLOB or TINYTEXT.
MEDIUMBLOB or MEDIUMTEXT	*A BLOB or TEXT column with a maximum length of 16777215 characters.* You do not specify a length with MEDIUMBLOB or MEDIUMTEXT.
LONGBLOB or LONGTEXT	*A BLOB or TEXT column with a maximum length of 4294967295 characters.* You do not specify a length with LONGBLOB or LONGTEXT.
ENUM	*An enumeration is a fancy term for list.* When defining an ENUM, you are creating a list of items from which the value must be selected (or it can be NULL). For example, if you wanted your field to contain "A" or "B" or "C", you would define your ENUM as ENUM ('A', 'B', 'C') and only those values (or NULL) could ever populate that field.

Difference between Char and Varchar Datatypes

In order to understand the difference between CHAR and VARCHAR datatypes, one must be clear about fixed length and variable length fields.

A field is a data element which can store one type of information which is composed of characters. Number of characters that a field can store, is called *field length* or *field width*. As each character requires one byte for its storage, number of characters inside a field determine its field width in bytes (as those many bytes the field requires for its storage).

There can be *fixed length fields* as well as *variable length fields.*

Fixed length fields have fixed lengths *i.e.*, they occupy fixed number of bytes for every data element they store. These number of bytes are determined by maximum number of characters the field can store. Not necessarily, all data elements inside a field are of same length, but the field length is determined by determining the maximum possible characters in a data element. In fixed length fields, some space is always wasted as all data elements do not use all the space reserved but processing is much simpler in case of fixed length fields.

Variable length fields have varied field lengths *i.e.*, field length is determined separately for every data element inside the field. The number of characters in the data element become its field length. Let us consider a field "*Student Name*" is having data elements "SUNILA RAJ" and "SHALU HARBANS". For the first data element "SUNILA RAJ" field length is 10 bytes (1 byte for space also, as it is also a character) and for "SHALU HARBANS" it is 13 bytes. In variable length fields, there is no wastage of space, but processing is complex in case of variable length fields.

The difference between CHAR and VARCHAR is that of *fixed length* and *variable length*. The CHAR datatype specifies a *fixed length* character string. When a column is given datatype as CHAR(*n*), then MySQL ensures that all values stored in that column have this length *i.e.*, *n* bytes. If a value is shorter than this length *n* then blanks are added, but the size of value remains *n* bytes.

VARCHAR, on the other hand, specifies a variable length string. When a column is given datatype as VARCHAR(*n*), then the maximum size a value in this column can have is *n* bytes. Each value that is stored in this column stores exactly as you specify it *i.e.*, no blanks are added if the length is shorter than maximum length *n*. However, if you exceed the maximum length *n*, then an error message is displayed.

12.2.3 Null Values

If a column in a row has no value, then column is said to be *null*, or to contain a null. Nulls can appear in columns of any data type provided they are not restricted by NOT NULL or PRIMARY KEY integrity constraints[1]. You should use a *null* value when the actual value is not known or when a value would not be meaningful.

One thing that you should make sure is, not to use null to represent a value of zero, because they are not equivalent. Any arithmetic expression containing a null always evaluates to null. For example, null added to 10 is null. In fact, all operators (except concatenation) return null when given a null operand.

NOTE

Any arithmetic expression containing a null, always evaluates to null.

12.2.4 Comments

A comment is a text that is not executed ; it is only for documentation purpose. A comment generally describes the statement's purpose within an application.

A comment can appear between any keywords, parameters or punctuation marks in a statement. You can include a comment in a statement using either of these means :

- *Begin the comment with* /*. Proceed with the text of the comment. This text can span multiple lines. End the comment with */. The opening and terminating characters need not be separated from the text by a space or a line break.
- *Begin the comment with* -- (followed by a space). Proceed with the text of the Comment. This text cannot extend to a new line. End the comment with a line break.
- *Begin the comment with* #. Proceed with the text of the comment. This text cannot extend to a new line. End the comment with a line break.

A SQL statement can contain multiple comments of both styles. The text of a comment can contain any printable characters in your database character set. For example,

```
SELECT ename, sal, job, loc
    /* Select all employees whose compensation is
    greater than 300. */
FROM empl, dept
WHERE empl.deptno = dept.deptno          #joining two tables
AND sal > 300 ;        -- this condition is testing whether salary is more than 300 or not.
```

12.3 SQL COMMAND SYNTAX

The *SQL* provides a predefined set of commands that help us work on relational databases. But before discussing them, we must be familiar with the conventions and terminology used in *SQL* commands.

Keywords are words that have a special meaning in *SQL*. They are understood to be instructions. Here, the *SQL* keywords have been printed in capital letters. ***Commands***, or ***statements***, are instructions given by you to a *SQL* database. Commands consist of one or more logically distinct parts called ***clauses***. Clauses begin with a keyword for which they are generally named, and consist of keywords and arguments. Examples of clauses are "*FROM sales*" and "*WHERE value = 1500.00*", ***Arguments*** complete or modify the meaning of a clause. In the above examples, '*sales*' is the argument, and FROM is the keyword of FROM clause.

1. A constraint is a condition applicable on column(s) of a table. You'll learn about constraints later.

Likewise '*value = 1500.00*' is the argument of the WHERE clause. ***Objects*** are structures in the database that are given names and stored in memory. They include *base tables, views,* and *indexes*[2].

Now that we are about to start *SQL* commands, following table (Table 12.1) summarizes the symbols used in syntax statements :

Table 12.1 *Symbols Used in Syntax Statements*

Symbol	Meaning
|	This is a symbolic way of saying "or". That is, whatever precedes this symbol may optionally be replaced by whatever follows it.
{ }	Everything enclosed in it, is treated as a unit for the purposes of evaluating |, , , , . or other symbols.
[ ]	This means, everything enclosed in it is optional.
...	This means whatever precedes it may be repeated any number of times.
,...	Whatever precedes this, may be repeated any number of times with the individual occurrences separated by commas.
<>	*SQL* and other special terms are in angle brackets.

A statement in MySQL SQL is completed with a semicolon(;) *i.e.*, at the end of the SQL statement, you have to give a semicolon(;). Also, you can write SQL statement in any case, whether lower-case or upper-case, as the commands in SQL are not case-sensitive. However the text literals or character literals that you specify in quotation marks are case-sensitive.

NOTE

Commands in SQL are not case-sensitive.

12.4 SAMPLE DATABASE

In the coming sections, you'll be learning about how to use SQL commands on a MySQL database. But before you start using commands, one more thing you need to do and that is to load a sample database. For this purpose, we downloaded *menagerie* example database from the MySQL site (dev/mysql/com/doc/) and loaded it on our MySQL server as per given instructions.

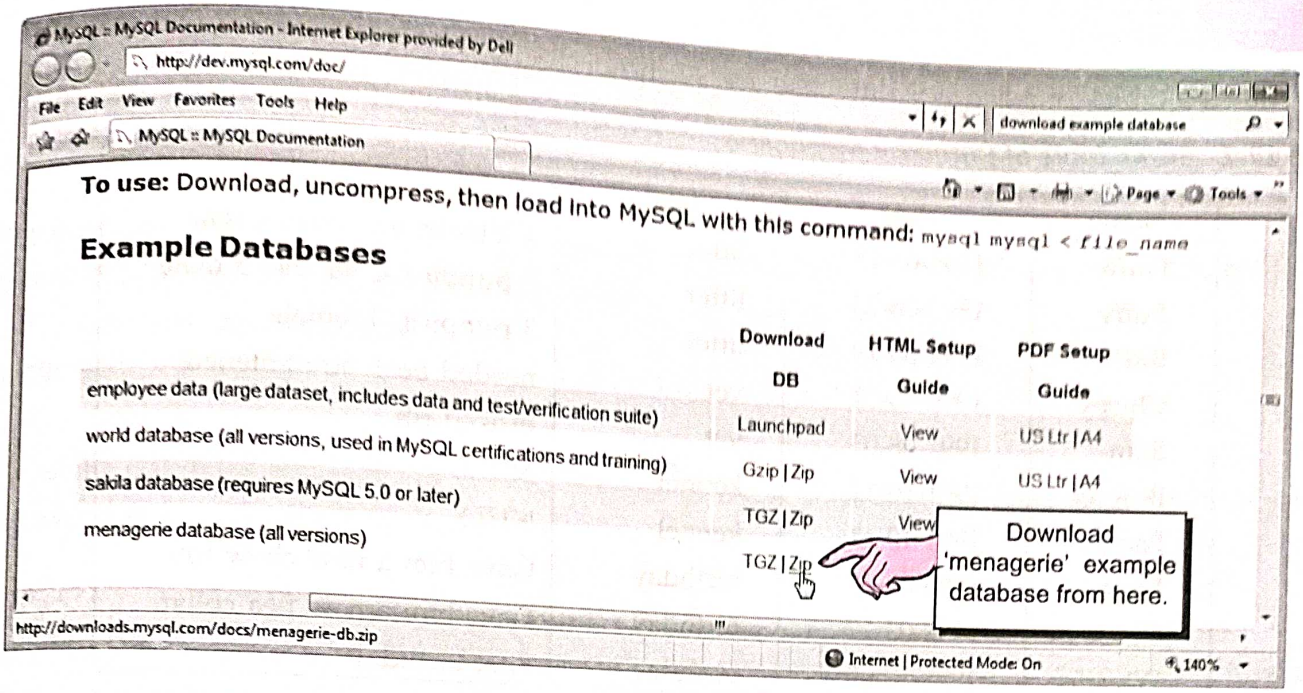
(Create a folder namely *SampleDB* on hard disk of your machine. Say its path is **D:\SampleDB**. You can create this folder on any other drive too. If you intend to do so, simply use that path in place of the "D:\SampleDB" in the coming lines.)

1. To load this database, first you need to uncompress the **menagerie.zip** file in the folder SampleDB that you created earlier.
2. Now, create a textfile "mena.sql" for editing (notice, it must have **.sql** extension). Enter following text in it :

```
Drop database if exists menagerie ;
CREATE DATABASE menagerie ;
USE menagerie ;
SOURCE D:/SampleDB/cr_pet_tbl.sql
LOAD DATA LOCAL INFILE 'D:/SampleDB/pet.txt' INTO TABLE pet ;
SOURCE D:/SampleDB/ins_puff_rec.sql
SOURCE D:/SampleDB/cr_event_tbl.sql
LOAD DATA LOCAL INFILE 'D:/SampleDB/event.txt' INTO TABLE event ;
```

Replace path D:/SampleDB with the path of the folder where you downloaded and uncompressed file menagerie.zip.

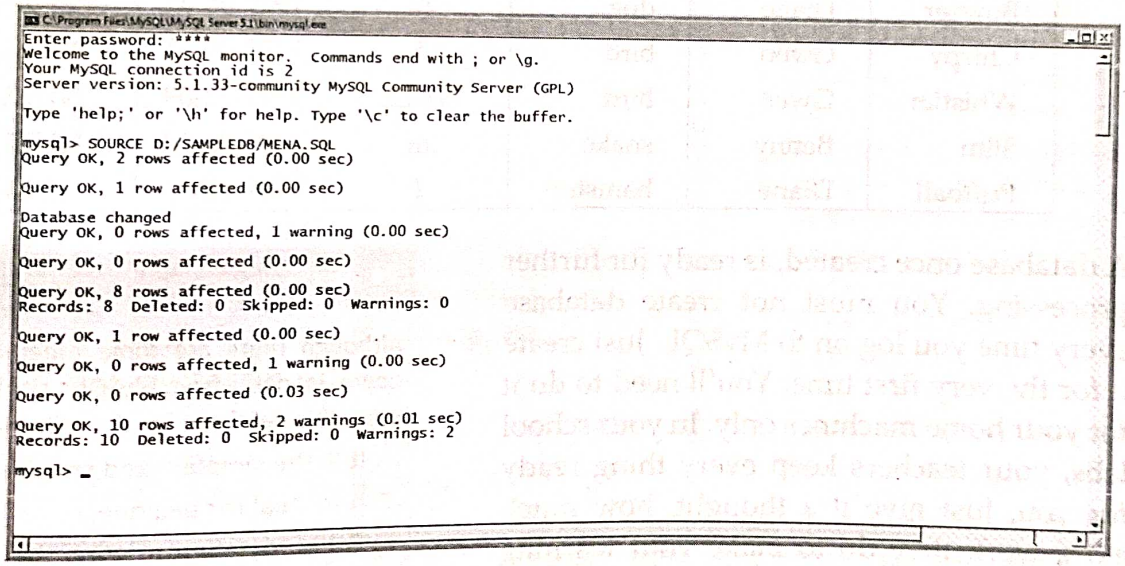
2. **An index is an ordered list of single or grouped column values that stores the disk locations of the rows containing those values.**



3. Replace **D:/SampleDB/** with the path of your folder you created earlier, if the path of that folder is different from *D:/SampleDB*. (Make sure to use front slash (/) in place of backslash (\) in paths and pathnames.)
4. Save the file.
5. Now start *MySQL*. In front of *MySQL* prompt, type :

```
mysql> SOURCE D:/SAMPLEDB/MENA.SQL
```

6. It will create the example database "menagerie" for you. (See fig below)



7. Now type Show tables; in front of mysql>

```
mysql> show tables;
```

It will display the list of tables in your newly created database *menagerie*.

```
+------------------------+
| Tables_in_menagerie    |
+------------------------+
| event                  |
| pet                    |
+------------------------+
2 rows in set (0.00 sec)
```

The two tables of *menagerie* database are as shown below :

Table 12.2 *Event Table of Database Menagerie*

name	date	type	remark
Fluffy	1995-05-15	litter	4 kittens, 3 female, 1 male
Buffy	1993-06-23	litter	5 puppies, 2 female, 3 male
Buffy	1994-06-19	litter	3 puppies, 3 female
Chirpy	1999-03-21	vet	needed beak straightened
Slim	1997-08-03	vet	broken rib
Bowser	1991-10-12	kennel	NULL
Fang	1991-10-12	kennel	NULL
Fang	1998-08-28	birthday	Gave him a new chew toy
Claws	1998-03-17	birthday	Gave him a new flea collar
Whistler	1998-12-09	birthday	First birthday

Table 12.3 *Pet Table of Database Menagerie*

name	owner	species	sex	birth	death
Fluffy	Harold	cat	f	1993-02-04	NULL
Claws	Gwen	cat	m	1994-03-17	NULL
Buffy	Harold	dog	f	1989-05-13	NULL
Fang	Benny	dog	m	1990-08-27	NULL
Bowser	Diane	dog	m	1979-08-31	1995-07-29
Chirpy	Gwen	bird	f	1998-09-11	NULL
Whistler	Gwen	bird	NULL	1997-12-09	NULL
Slim	Benny	snake	m	1996-04-29	NULL
Puffball	Diane	hamster	f	1999-03-30	NULL

A database once created, is ready for further processing. You must not create database every time you log on to MySQL. Just create it for the very first time. You'll need to do it for your home machines only. In your school labs, your teachers keep every thing ready for you. Just give it a thought, how much extra work they do to make your learning effective and fruitful. Don't you think, they deserve heartfelt thanks for that ?

NOTE

Although there are *three* other example databases down- loadable from MySQL's site but we have zeroed upon this database for the following reasons :

- It is the simplest and smallest.
- It is ideal for beginners.
- Other databases store thousands of records and consume a lot of disk space, which is a useless thing at the beginning stage of learning.

12.5 MAKING SIMPLE QUERIES

To fully use the power of an RDBMS, you need to communicate with it. A powerful way of communicating with it is making queries[3]. That is, asking the RDBMS to show your desired data in desired format. This thing can be achieved through SELECT command. Coming sections are going to describe the usage of SQL SELECT command to make simple queries.

3. Query in context of SQL is a misnomer. A query statement in SQL cannot only make queries, but also insert, delete, modify data in RDBMS.

12.5.1 Accessing Database

Before you start making queries upon the data in tables of a database, you need to open the database for use. For this, after logging into MySQL, you need to issue a command :

```
Use <databasename> ;
```

For example, we want to work on our sample database namely *menagerie*, so we shall write the following command after logging in *MySQL*.

```
mysql> USE menagerie ;
Database changed
mysql>
```

Once the database *menagerie* is opened for use, you can start issuing the commands that you'll be learning in a short while from now.

12.5.2 The SELECT Command

The SELECT command of *SQL* lets you make queries on the database. A query is a command that is given to produce certain specified information from the database table(s). There are various ways and combinations, a SELECT statement can be used into. It can be used to retrieve a subset of rows or columns from one or more tables. In its simplest form, SELECT statement is used as given below :

```
SELECT  <column name> [, <column name>, ... ]
FROM <table name> ;
```

Selecting columns

For example, if you want to view only the information of two columns *name* and *owner* of table *pet*, you may write your query as

```
mysql> SELECT name, owner
    -> FROM pet ;
```

Then MySQL will produce the adjacent result which is based on the **Pet** table of **Menagerie** database, shown in Table 12.2

```
+----------+----------+
| name     | owner    |
+----------+----------+
| Fluffy   | Harold   |
| Claws    | Gwen     |
| Buffy    | Harold   |
| Fang     | Benny    |
| Bowser   | Diane    |
| Chirpy   | Gwen     |
| Whistler | Gwen     |
| Slim     | Benny    |
| Puffball | Diane    |
+----------+----------+
9 rows in set (0.00 sec)
```

```
+----------+----------+---------+
| name     | owner    | species |
+----------+----------+---------+
| Fluffy   | Harold   | cat     |
| Claws    | Gwen     | cat     |
| Buffy    | Harold   | dog     |
| Fang     | Benny    | dog     |
| Bowser   | Diane    | dog     |
| Chirpy   | Gwen     | bird    |
| Whistler | Gwen     | bird    |
| Slim     | Benny    | snake   |
| Puffball | Diane    | hamster |
+----------+----------+---------+
9 rows in set (0.00 sec)
```

If you want to see the information of columns *name*, *owner*, and *species* from the table *pet*, you will write

```
SELECT name, owner, species
FROM pet ;
```

Output of above command will be as shown on left side.

```
mysql > SELECT name, owner, species
    -> FROM pet ;
```

12.5.3 Selecting all Columns

If you want to see the entire table *i.e.*, every column of a table, you need not give a complete list of columns. The asterisk (*) can be substituted for a complete list of columns as follows :

```
SELECT * FROM pet ;
```

This will display all the rows present in the *pet* table.

12.5.4 Reordering Columns in Query Results

While giving a querying, the result can be obtained in any order. For example, if you give

```
SELECT species, name, sex, FROM pet ;
```

the result will be having *species* as first column, *name* as second column, and *sex* as third column, will be displayed as shown below. You can write the column names in any order and the output will be having information in exactly the same order.

```
mysql> SELECT species, name, sex
    -> FROM pet ;
```

NOTE

The order of selection determines the order of display.

```
+-----------+-----------+-----------+
| species   | name      | sex       |
+-----------+-----------+-----------+
| cat       | Fluffy    | f         |
| cat       | Claws     | m         |
| dog       | Buffy     | f         |
| dog       | Fang      | m         |
| dog       | Bowser    | m         |
| bird      | Chirpy    | f         |
| bird      | Whistler  | NULL      |
| snake     | Slim      | m         |
| hamster   | Puffball  | f         |
+-----------+-----------+-----------+
9 rows in set (0.02 sec)
```

12.5.5 Eliminating Redundant Data (with keyword DISTINCT)

By default, data is selected from all the rows of the table, even if the data appearing in the result gets duplicated. The DISTINCT keyword eliminates duplicate rows from the results of a SELECT statement. For example, if the *Suppliers* table stores the names and cities of the *suppliers* and we want to see the cities where the Suppliers belong to. The table may consist of more than one supplier belonging to the same city but the result of the query should not contain duplicated city names.

To do so, we shall write

```
SELECT DISTINCT city
FROM Suppliers ;
```

In the output, there would be no duplicate rows. Whenever DISTINCT is used, only one NULL value is returned in the results, no matter how many NULL values are encountered.

```
+-----------+
| City      |
+-----------+
| Delhi     |
| Mumbai    |
| Jaipur    |
| Bangalore |
+-----------+
```

See the duplicate entry Delhi has not reappeared. This is the property of DISTINCT.

If we consider *Suppliers* table of chapter 15, shown in Fig. 15.1, then the above query will produce the output in the right.

DISTINCT, in effect, applies to the entire output row, not a specific field. The DISTINCT keyword can be specified only once in a given SELECT clause. If the clause selects multiple fields, DISTINCT eliminates rows where all of the selected fields are identical. Rows in which some values are the same and some different will be retained.

EXAMPLE 12.1 *Display distinct species of pets from table pet.*

Solution.

```
mysql> SELECT DISTINCT(species) FROM pet ;
```

```
+-----------+
| species   |
+-----------+
| cat       |
| dog       |
| bird      |
| snake     |
| hamster   |
+-----------+
5 rows in set (0.03 sec)
```

12.5.6 Selecting from all the Rows – ALL Keyword

If in place of DISTINCT, you give ALL then the result retains the duplicate output rows. It is just the same as when you specify neither DISTINCT nor ALL ; ALL is essentially a clarifier rather than a functional argument. Thus if you give

```
SELECT ALL city FROM suppliers ;
```

it will give values of *city* column from every row of the table without considering the duplicate entries. Considering the same table *Suppliers*, now the output will be as shown here.

```
+-----------+
| City      |
+-----------+
| Delhi     |
| Mumbai    |
| Delhi     |
| Banglore  |
+-----------+
```

EXAMPLE 12.2 *Display species of all pets from table pet.*

Solution.

```
mysql> SELECT ALL species FROM pet ;
```

```
+-----------+
| species   |
+-----------+
| cat       |
| cat       |
| dog       |
| dog       |
| dog       |
| bird      |
| bird      |
| snake     |
| hamster   |
+-----------+
9 rows in set (0.00 sec)
```

12.5.7 Viewing Structure of Table

You have learnt just back that Use <databasename> ; opens a database for use. To view the tables in a database, you may write following command after opening the database.

```
SHOW TABLES
```

That is , if you give following set of statements, the output will be as shown here :

```
mysql> USE menagerie
       Database changed
mysql> SHOW TABLES ;
```

```
+-----------------------+
| Tables_in_menagerie   |
+-----------------------+
| event                 |
| pet                   |
+-----------------------+
2 rows in set (0.01 sec)
```

If you want to know the structure of a table, you can use *Describe* or *Desc* command as per following syntax :

```
DESCRIBE | DESC <table name> ;
```

For instance, the commands : DESCRIBE pet; or DESC pet ; will display the structure of table *pet*.

```
mysql> DESC pet ;
+---------+-------------+------+-----+---------+-------+
| Field   | Type        | Null | Key | Default | Extra |
+---------+-------------+------+-----+---------+-------+
| name    | varchar(20) | YES  |     | NULL    |       |
| owner   | varchar(20) | YES  |     | NULL    |       |
| species | varchar(20) | YES  |     | NULL    |       |
| sex     | char(1)     | YES  |     | NULL    |       |
| birth   | date        | YES  |     | NULL    |       |
| death   | date        | YES  |     | NULL    |       |
+---------+-------------+------+-----+---------+-------+
6 rows in set (0.01 sec)
```

12.5.8 How to Perform Simple Calculations ?

Often a simple calculation needs to be done, for example, 4 * 3. The only SQL verb to cause an output to be written to monitor is SELECT. For instance, when a calculation is to be performed such as 3 * 4 or 8 * 3 etc., there really is no table being referenced, only numeric literals are being used.

For this you can use SELECT statement to retrieve rows computed without reference to any table. For example, consider the following query that computes the expression 1+ 6 and outputs the result. Notice that *no table* or *FROM clause* is used in the following query.

```
mysql > SELECT 1 + 6 ;
+----------+
| 1+6      |
+----------+
|   7      |
+----------+
1 row in set (0.00 sec)
```

The result of expression given with SELECT

MySQL also provides a dummy table called ***Dual*** to provide compatibility with SQL of other DBMSs. **Dual** table is a small worktable, which has just one row and one column. It can be used for obtaining calculation results and also system-date.

The following query :

```
SELECT 4 * 3 FROM dual ;
```

will produce the result as :

```
+----------+
| 4 * 3    |
+----------+
| 12       |
+----------+
```

The current date can be obtained, using function **curdate( )**, as shown below :

```
mysql> SELECT curdate( ) ;
+------------+
| curdate()  |
+------------+
| 2009-05-03 |
+------------+
1 row in set (0.05 sec)
```

Current date of the system

NOTE

To perform calculations, you may either write expression with SELECT or use table **Dual** (in FROM clause), which is a one row, one column dummy table, provided by MySQL.

12.5.9 Scalar Expressions with Selected Fields

If you want to perform simple numeric computations on the data to put it in a form more appropriate to your needs, *SQL* allows you to place scalar expressions and constants among the selected fields. For example, you might consider it desirable to present your salespeople's commissions as percentage rather than decimal numbers. You may do it as :

```
SELECT salesman_name, comm*100
FROM salesman ;
```

```
Ajay        13.00
Amit        11.00
Shally      07.00
Isha        15.00
```

The output will be somewhat like :

Please note that if a value in the expression is *Null* then result of the expression will be *Null* only.

The operators that you can use in arithmetic expressions are *plus (+), minus (–), divide (/), multiply (*)* and *modulo (%)* (for remainder).

12.5.10 Using Column Aliases

The columns that you select in a query can be given a different name *i.e.*, **column alias name** for output purposes. *For instance,* if you want to extract information (date and type) from **Event** table and want the name of **Type** column should appear as Event Type, then you should give query as follows :

```
SELECT date, type As "Event Type"
FROM Event ;
```

The output produced by above query will be :

```
mysql> SELECT date, type AS "Event Type"
    -> FROM event ;
```

```
+---------------+---------------+
| date          | Event Type    |
+---------------+---------------+
| 1995-05-15    | litter        |
| 1993-06-23    | litter        |
| 1994-06-19    | litter        |
| 1999-03-21    | vet           |
| 1997-08-03    | vet           |
| 1991-10-12    | kennel        |
| 1991-10-12    | kennel        |
| 1998-08-28    | birthday      |
| 1998-03-17    | birthday      |
| 1998-12-09    | birthday      |
+---------------+---------------+
10 rows in set (0.00 sec)
```

Notice that the column **Comm** has been given column alias name **Commission**. This has been done as per following syntax :

```
Select <columnname> AS [columnalias] [, <columnname> AS [columnalias]]
    .....
From <tablename> ;
```

If you want to provide alias to an expression, you may do it as given below :

```
mysql> SELECT 22/7 as PI ;
+-----------+
| PI        |
+-----------+
| 3.1429    |
+-----------+
1 row in set (0.03 sec)
```

NOTE

The column alias name having more than one word, should be enclosed in quotation marks.

See the expression 22/7 has been given alias as PI. When the alias name contains more than one word, you have to enclose them in quotes as we did in previous query where we enclosed the column alias **Event Type** in quotes as "**Event Type**".

Table : *Student*

name	age	sex	matric	aggregate
Abu Bakar	15	M	9531185	456
Aanya	16	F	9531186	340
Gurvinder	14	F	9531187	480
Ali	16	M	9531188	260
Michelle	15	F	9531188	321
Zubin	15	M	9531184	412
Simran	15	F	9531183	378
Fatimah	14	F	9531182	400
Anup	15	M	9531181	302
Mita	16	F	9531180	150

EXAMPLE 12.3. *Given a table student as shown here. Write a query to display name, sex and aggregate/5. Label this computation as percentage.*

Solution.

```
mysql> SELECT name, sex, aggregate/5 AS Percentage
    -> FROM student ;
+-----------+-----------+------------+
| name      | sex       | Percentage |
+-----------+-----------+------------+
| Abu Bakar | M         | 91.2000    |
| Aanya     | F         | 68.0000    |
| Gurvinder | F         | 96.0000    |
| Ali       | M         | 52.0000    |
| Michelle  | F         | 64.2000    |
| Zubin     | M         | 82.4000    |
| Simran    | F         | 75.6000    |
| Fatimah   | F         | 80.0000    |
| Anup      | M         | 60.4000    |
| Mita      | F         | 30.0000    |
+-----------+-----------+------------+
10 rows in set (0.00 sec)
```

12.5.11 Handling Nulls

As already mentioned, the empty values are represented as Nulls in a table. When you list a column having null values, only non-null values are displayed. For instance, if you write a query as :

```
SELECT name, birth, death
FROM pet ;
```

It will produce the result as shown below :

```
mysql> SELECT name, birth, death FROM pet;
+------------+------------+------------+
| name       | birth      | death      |
+------------+------------+------------+
| Fluffy     | 1993-02-04 | NULL       |
| Claws      | 1994-03-17 | NULL       |
| Buffy      | 1989-05-13 | NULL       |
| Fang       | 1990-08-27 | NULL       |
| Bowser     | 1979-08-31 | 1995-07-29 |
| Chirpy     | 1998-09-11 | NULL       |
| Whistler   | 1997-12-09 | NULL       |
| Slim       | 1996-04-29 | NULL       |
| Puffball   | 1999-03-30 | NULL       |
+------------+------------+------------+
9 rows in set (0.00 sec)
```

See the null values appear as NULL. If you want to substitute null with a value in the output, you can use **IFNULL( )** function. IFNULL( ) function may be used as per following syntax :

SYNTAX

```
IFNULL (<columnname>, value-to-be-substituted)
```

That is, if you want to substitute value alive for null in **death**, the query should look like :

```
SELECT name, birth, IFNULL (death, "alive")
FROM pet ;
```

And the output will be like :

```
mysql> SELECT name, birth, IFNULL(death, "Alive") FROM pet;
+------------+------------+-------------------------+
| name       | birth      | IFNULL(death,"Alive")   |
+------------+------------+-------------------------+
| Fluffy     | 1993-02-04 | Alive                   |
| Claws      | 1994-03-17 | Alive                   |
| Buffy      | 1989-05-13 | Alive                   |
| Fang       | 1990-08-27 | Alive                   |
| Bowser     | 1979-08-31 | 1995-07-29              |
| Chirpy     | 1998-09-11 | Alive                   |
| Whistler   | 1997-12-09 | Alive                   |
| Slim       | 1996-04-29 | Alive                   |
| Puffball   | 1999-03-30 | Alive                   |
+------------+------------+-------------------------+
9 rows in set (0.03 sec)
```

You may also use column alias to replace output heading. This can be done as follows :

```
mysql> SELECT name, birth, IFNULL(death, "Alive") AS "Died On"
       FROM pet ;
```

Now the output produced will be :

```
+------------+------------+------------+
| name       | birth      | Died On    |
+------------+------------+------------+
| Fluffy     | 1993-02-04 | Alive      |
| Claws      | 1994-03-17 | Alive      |
| Buffy      | 1989-05-13 | Alive      |
| Fang       | 1990-08-27 | Alive      |
| Bowser     | 1979-08-31 | 1995-07-29 |
| Chirpy     | 1998-09-11 | Alive      |
| Whistler   | 1997-12-09 | Alive      |
| Slim       | 1996-04-29 | Alive      |
| Puffball   | 1999-03-30 | Alive      |
+------------+------------+------------+
9 rows in set (0.00 sec)
```

NOTE

The value being substituted for Null values through **IFNULL( )**, should be of the same type as that of the column.

Some other examples of IFNULL( ) are :

```
IFNULL (Due_date, '2019-07-03')
IFNULL (Project alloted, 'Not-Alloted-so-far') ;
```

12.5.12 Putting Text in the Query Output

The previous example can be refined by marking the commissions as percentages with the percent sign (%). This enables you to put such items as symbols and comments in the output, as in the following example :

```
SELECT salesman_name, comm * 100, '%'
FROM salesman ;
```

```
+-----------------+-----------+---+
| SALESMAN_NAME   | COMM*100  |   |
+-----------------+-----------+---+
| Ajay            | 13.00     | % |
| Amit            | 11.00     | % |
| Shally          | 07.00     | % |
| Isha            | 15.00     | % |
+-----------------+-----------+---+
```

A sample output produced by above query is shown here.

See, the same comment or symbol gets printed with every row of the output, not simply once for the table.

You could insert text in your query also, making it more presentable. For example,

```
SELECT salesman_name, 'gets the commission', comm*100, '%'
FROM salesman ;
```

The sample output produced by above query is shown below :

```
+-----------------+--------------------------------------+------+---+
| SALESMAN_NAME   | gets the commission | COMM*100 |   |
+-----------------+--------------------------------------+------+---+
| Ajay            | gets the commission | 13.00    | % |
| Amit            | gets the commission | 11.00    | % |
| Shally          | gets the commission | 07.00    | % |
| Isha            | gets the commission | 15.00    | % |
+-----------------+--------------------------------------+------+---+
```

EXAMPLE 12.4. *Create a query that produces display in following format :*

```
<studentname> obtained <aggregate> marks and has <aggregate/5> %.
```

Consider table **Student** *of example 12.3 for this.*

Solution.

```
mysql> SELECT name , "obtained", aggregate, "marks and has", aggregate/5, "%"
    -> FROM student ;
```

```
+-----------+----------+-----------+---------------+---------------+---+
| name      | obtained | aggregate | marks and has | aggregate/5   | % |
+-----------+----------+-----------+---------------+---------------+---+
| Abu Bakar | obtained | 456       | marks and has | 91.2000       | % |
| Aanya     | obtained | 340       | marks and has | 68.0000       | % |
| Gurvinder | obtained | 480       | marks and has | 96.0000       | % |
| Ali       | obtained | 260       | marks and has | 52.0000       | % |
| Michelle  | obtained | 321       | marks and has | 64.2000       | % |
| Zubin     | obtained | 412       | marks and has | 82.4000       | % |
| Simran    | obtained | 378       | marks and has | 75.6000       | % |
| Fatimah   | obtained | 400       | marks and has | 80.0000       | % |
| Anup      | obtained | 302       | marks and has | 60.4000       | % |
| Mita      | obtained | 150       | marks and has | 30.0000       | % |
+-----------+----------+-----------+---------------+---------------+---+
10 rows in set (0.00 sec)
```

12.5.13 Selecting Specific Rows – WHERE clause

In real life, tables can contain unlimited rows. There is no need to view all the rows when only certain rows are needed. *SQL* enables you to define criteria to determine which rows are selected for output. The WHERE clause in SELECT statement specifies the criteria for selection of rows to be returned. The SELECT statement with WHERE clause takes the following general form :

```
SELECT <column name> [, <column name>, ... ]
FROM <table name>
WHERE <condition> ;
```

when a WHERE clause is present, the database program goes through the entire table one row at a time and examines each row to determine if the given condition is true. If it is true for a row, that row is displayed in the output. For example, to display the *name* and *aggregate* for students having their aggregate marks more than 350, the command would be

```
mysql> SELECT name, aggregate
    -> FROM student
    -> WHERE aggregate > 350 ;
```

The above query will produce adjacent output :

Only the records having aggregate > 350 have appeared in the output.

```
+-----------+-----------+
| name      | aggregate |
+-----------+-----------+
| Abu Bakar | 456       |
| Gurvinder | 480       |
| Zubin     | 412       |
| Simran    | 378       |
| Fatimah   | 400       |
+-----------+-----------+
5 rows in set (0.03 sec)
```

EXAMPLE 12.5. *Write a query to display name, age and marks (aggregate) of students whose age is greater than or equal to 16 from table student.*

Solution.

```
mysql> SELECT name,age, aggregate FROM student
    -> WHERE age >= 16 ;
+-----------+----------+-----------+
| name      | age      | aggregate |
+-----------+----------+-----------+
| Aanya     | 16       | 340       |
| Ali       | 16       | 260       |
| Mita      | 16       | 150       |
+-----------+----------+-----------+
3 rows in set (0.02 sec)
```

12.5.14 Relational Operators

To compare two values, a relational operator is used. The result of the comparison is true or false. The *SQL* recognizes following relational operators :

$=, >, <, >=, <=, <>$ (not equal to)

In CHARACTER data type comparisons, < means earlier in the alphabet and > means later in the alphabet. For example, $e < f$ and $g > f$. Apostrophes are necessary around all CHAR, DATE and TIME data. For example, to list all the members not from 'DELHI'

```
SELECT * FROM suppliers
WHERE city < > 'DELHI' ;
```

12.5.15 Logical Operators

The logical operators OR (||), AND (&&) and NOT (!) are used to connect search conditions in the WHERE clause. For example,

1. To list the employees' details having grades 'E2' or 'E3' from table *employee* (not the *EMPL* table), logical operator OR will be used as :

```
SELECT ecode, ename, grade, gross
FROM employee
WHERE (grade = 'E2' OR grade = 'E3') ;
```

Symbol || may also be used as **OR** operator *i.e.*, above query may also be written as :

```
SELECT ecode, ename, grade, gross
FROM employee
WHERE (grade = 'E2' ||grade = 'E3') ;
```

2. To list all the employees' details having grades as 'E4' but with gross< 9000, logical operator AND will be used as :

```
SELECT ecode, ename, grade, gross
FROM employee
WHERE (grade = 'E4' AND gross <9000) ;
```

Symbol *&&* may also be used as **AND** operator.

3. To list all the employees' details whose grades are other than 'G1', logical operator NOT will be used as :

```
SELECT ecode, ename, grade, gross
FROM employee
WHERE (NOT grade = 'G1') ;
```

Symbol ! may also be used as **NOT** operator.

when all the logical operators are used together, the order of precedence is NOT (!), AND (&&), and OR (||). However, parentheses can be used to override the default precedence.

EXAMPLE 12.6. *Write a query to display all the details from pet table for species cat /dog having gender(sex) as male ('m')*

Solution.

```
mysql> SELECT * FROM pet
    -> WHERE (species = 'cat' || species = 'dog') && sex = 'm' ;
```

```
+---------+--------+---------+------+------------+------------+
| name    | owner  | species | sex  | birth      | death      |
+---------+--------+---------+------+------------+------------+
| Claws   | Gwen   | cat     | m    | 1994-03-17 | NULL       |
| Fang    | Benny  | dog     | m    | 1990-08-27 | NULL       |
| Bowser  | Diane  | dog     | m    | 1979-08-31 | 1995-07-29 |
+---------+--------+---------+------+------------+------------+
3 rows in set (0.00 sec)
```

12.5.16 Condition Based on a Range

The BETWEEN operator defines a range of values that the column values must fall into make the condition true. The range includes both lower value and the upper value. For example, to list the items whose *QOH* falls between 30 to 50 (both inclusive), the command would be :

```
SELECT icode, descp, QOH
FROM items
WHERE QOH BETWEEN 30 AND 50 ;
```

If the *Items* Table is as shown below :

Table 12.4 *Items*

Icode	Descp	Price	QOH	ROL	ROQ
I01	Milk	15.00	20	10	20
I02	Cake	5.00	60	20	50
I03	Bread	9.00	40	10	40
I04	Biscuit	10.00	50	40	60
I05	Namkeen	15.00	100	50	70
I06	Cream Roll	7.00	10	20	30

Then the above query will produce the following output :

```
+-----------+-----------+-----------+
| Icode     | Descp     | QOH       |
+-----------+-----------+-----------+
| I03       | Bread     | 40        |
| I04       | Biscuit   | 50        |
+-----------+-----------+-----------+
```

Only the items having QOH between 30 to 50 have been listed

The operator NOT BETWEEN is reverse of BETWEEN operator, that is, the rows not satisfying the BETWEEN condition are retrieved. For example,

```
SELECT icode, descp
FROM items
WHERE ROL NOT BETWEEN 100 AND 1000 ;
```

This query will list the items whose ROL is below 100 or above 1000.

XAMPLE 12.7. *Write a query to display name and aggregate marks of those students who don't have their aggregate arks in the range of 380 – 425.*

Solution.

```
mysql> SELECT name, aggregate
    -> FROM student
    -> WHERE aggregate NOT BETWEEN 380 AND 425 ;
```

```
+-----------+-----------+
| name      | aggregate |
+-----------+-----------+
| Abu Bakar | 456       |
| Aanya     | 340       |
| Gurvinder | 480       |
| Ali       | 260       |
| Michelle  | 321       |
| Simran    | 378       |
| Anup      | 302       |
| Mita      | 150       |
+-----------+-----------+
8 rows in set (0.00 sec)
```

2.5.17 Condition Based on a List

To specify a list of values, IN operator is used. The IN operator selects values that match any value in a given list of values.

For example, to display a list of members from 'DELHI', 'MUMBAI', 'CHENNAI' or 'BANGALORE' cities, you may give

```
SELECT * FROM members
WHERE city IN ('DELHI', 'MUMBAI', 'CHENNAI', 'BANGALORE') ;
```

The NOT IN operator finds rows that do not match in the list. So if you write

```
SELECT * FROM members
WHERE city NOT IN ('DELHI', 'MUMBAI', 'CHENNAI') ;
```

it will list members not from the cities mentioned in the list.

EXAMPLE 12.8. *Write a query to display all details of pets of species bird, snake or hamster from table pet.*

Solution.

```
mysql> SELECT * FROM pet
    -> WHERE species IN ('bird', 'snake', 'hamster');
+---------+--------+---------+---------+------------+------------+
| name    | owner  | species | sex     | birth      | death      |
+---------+--------+---------+---------+------------+------------+
| Chirpy  | Gwen   | bird    | f       | 1998-09-11 | NULL       |
| Whistler| Gwen   | bird    | NULL    | 1997-12-09 | NULL       |
| Slim    | Benny  | snake   | m       | 1996-04-29 | NULL       |
| Puffball| Diane  | hamster | f       | 1999-03-30 | NULL       |
+---------+--------+---------+---------+------------+------------+
4 rows in set (0.00 sec)
```

12.5.18 Condition Based on Pattern Matches

SQL also includes a string-matching operator, LIKE, for comparisons on character strings using patterns. Patterns are described using two special wildcard characters :

- percent (%). The % character matches any substring.
- underscore (_). The _ character matches any character.

Patterns are case-sensitive, that is, upper-case characters do not match lower-case characters, or vice-versa. To illustrate pattern matching, consider the following examples :

- 'San%' matches any string beginning with 'San'
- '%idge%' matches any string containing 'idge' as a substring, for example, 'Ridge', 'Bridges', 'Cartridge', 'Ridgeway' etc.
- '- - - -' matches any string of exactly 4 characters.
- '- - -%' matches any string of at least 3 characters.

The LIKE keyword is used to select rows containing columns that match a wildcard pattern.

Examples :

1. To list members which are in areas with pin codes starting with 13, the command is :

```
SELECT firstname, lastname, city
FROM members
WHERE pin LIKE '13%' ;
```

2. To list names of pets who have names ending with '*y*', the command would be :

```
SELECT name
FROM emp
WHERE name LIKE '%y' ;
```

Consider the Table 12.2, the above query will produce the adjacent output :

```
+-----------+
| name      |
+-----------+
| Fluffy    |
| Buffy     |
| Chirpy    |
+-----------+
```

3. To list members which are not in areas with pin codes starting with 13, the command is :

```
SELECT firstname, lastname, city
FROM members
WHERE pin NOT LIKE '13%' ;
```

The keyword NOT LIKE is used to select rows that do not match the specified pattern of characters.

In order for patterns to include the special pattern characters (that is, %, _), *SQL* allows the specific of an escape character. The escape character is used immediately before a special pattern character to indicate that the special pattern character is to be treated as a normal character. We define the escape character for a LIKE comparison using the ESCAPE keyword.

To illustrate, consider the following patterns which use a backslash (\) as the escape character.

- LIKE 'wx\%yz%' ESCAPE '\' matches all strings beginning with 'wx%yz'.
- LIKE 'wx\\yz%' ESCAPE '\' matches all string beginning with 'wx\yz'.

The ESCAPE clause can define any character as an escape character. The above two examples use backslash ('\') as the escape character.

EXAMPLE 12.9 *Write query to display the names of pets beginning with 'F'. Use table Pet.*

Solution.

```
mysql> SELECT * FROM pet
    -> WHERE name LIKE "F%";
+--------+--------+---------+---------+------------+------------+
| name   | owner  | species | sex     | birth      | death      |
+--------+--------+---------+---------+------------+------------+
| Fluffy | Harold | cat     | f       | 1993-02-04 | NULL       |
| Fang   | Benny  | dog     | m       | 1990-08-27 | NULL       |
+--------+--------+---------+---------+------------+------------+
2 rows in set (0.00 sec)
```

EXAMPLE 12.10 *Write query to display the names of pets having exactly four letter names. Use table Pet.*

Solution.

```
mysql> SELECT * FROM pet
    -> WHERE name LIKE "____" ;
```

contains 4 underscore symbols

```
+--------+--------+---------+---------+------------+------------+
| name   | owner  | species | sex     | birth      | death      |
+--------+--------+---------+---------+------------+------------+
| Fang   | Benny  | dog     | m       | 1990-08-27 | NULL       |
| Slim   | Benny  | snake   | m       | 1996-04-29 | NULL       |
+--------+--------+---------+---------+------------+------------+
2 rows in set (0.00 sec)
```

12.5.19 Searching for NULL

The NULL value in a column can be searched for in a table using IS NULL in the WHERE clause. (Relational operators like =, <> etc. can't be used with NULL.) For example, to list details of all employees whose departments contain NULL (*i.e.*, novalue), you use the command :

```
SELECT empno, empname, job
FROM emp
WHERE DeptNo IS NULL ;
```

Non-NULL values in a table can be listed using IS NOT NULL.

EXAMPLE 12.11 *Write query to display the names of pets who are no more. (That is the death column stores a non-null value for them). Use table Pet.*

Solution.

```
mysql> SELECT name FROM pet
    -> WHERE death IS NOT NULL;
+-----------+
| name      |
+-----------+
| Bowser    |
+-----------+
1 row in set (0.02 sec)
```

12.5.20 Operator Precedence

When an expression has multiple operators, then the evaluation of expression takes place in the order of operator precedence. The operators with higher precedence are evaluated first.

Operator precedences of MySQL are shown in the following list, from highest precedence to the lowest. Operators that are shown together on a line have the same precedence.

```
!
- (unary minus), ~ (unary bit inversion)
^
*, /, DIV, %, MOD
-, +
<, >
&
|
==, >=, >, <=, <, <>, !=, IS, LIKE, REGEXP, IN BETWEEN, CASE, WHEN, THEN, ELSE
NOT
&&, AND
XOR
||, OR
:=
```

12.5.21 Sorting Results – ORDER BY clause

Whenever a SELECT query is executed, the resulting rows emerge in a predecided order. You can sort the results or a query in a specific order using ORDER BY clause. The ORDER BY clause allows sorting of query results by one or more columns. The sorting can be done either in *ascending* or *descending* order, the default order is *ascending*. The data in the table is not sorted ; only the results that appear on the screen are sorted. The ORDER BY clause is used as :

```
SELECT <column name> [, <column name> , ... ]
FROM <table name>
[WHERE <predicate> ]
[ORDER BY <column name> ] ;
```

For example, to display the list of employees in the alphabetical order of their names, you use the command :

```
SELECT * FROM employee
ORDER BY ename ;
```

To display the list of students having aggregate more than 400 in the alphabetical order of their names, you may give the command :

```
SELECT name, aggregate FROM student
WHERE aggregate > 400
ORDER BY name ;
```

Considering the same *student* table, the above query would produce the output shown here.

```
+-----------+-----------+
| name      | aggregate |
+-----------+-----------+
| Abu Bakar | 456       |
| Gurvinder | 480       |
| Zubin     | 412       |
+-----------+-----------+
3 rows in set (0.03 sec)
```

To display the list of employees in the descending order of employee code, you use the command :

```
SELECT * FROM employee
ORDER BY ecode DESC ;
```

To specify the sort order, we may specify DESC for descending order or ASC for ascending order. Furthermore, ordering can be performed on multiple attributes. Suppose that we wish to list the entire *employee* relation in descending order of *grade*. If several employees have the same grade, we order them in ascending order by their names.

We express this in *SQL* as follows :

```
SELECT * FROM employee
ORDER BY grade DESC, ename ASC ;
```

See, the multiple fields are separated by commas. In order to fulfill an ORDER BY request, *SQL* must perform a sort. Since sorting a large number of tuples may be costly, it is desirable to sort only when necessary.

Check Point

12.1

1. Maximum how many characters can be stored in a (*i*) text literal (*ii*) numeric literal ?
2. What is a datatype ? Name some datatypes available in MySQL.
3. What are fixed length fields ? What are variable length fields ?
4. Compare Char and Varchar datatypes.
5. What is null value in MySQL database ? Can you use nulls in arithmetic expressions ?
6. Which keyword eliminates the redundant data from a query result ?
7. Which keyword retains duplicate output rows in a query result ?
8. How would you display system date as the result of a query ?
9. How would you calculate 13 * 15 in SQL ?
10. Which function is used to substitute NULL values in a query result ?
11. Which operator concatenates two strings in a query result ?
12. Which comparison operator is used for comparing ?
    (*i*) patterns (*ii*) character values
    (*iii*) null values (*iv*) ranges
    (*v*) list of values.

EXAMPLE 12.12 *Write a query to display name, age, aggregate of students whose aggregate is between 300 and 400. Order the query in descending order of name.*

Solution.

```
mysql> SELECT name, age, aggregate
    -> FROM student
    -> WHERE aggregate BETWEEN 300 AND 400
    -> ORDER BY name DESC;
```

```
+----------+-------+-----------+
| name     | age   | aggregate |
+----------+-------+-----------+
| Simran   | 15    | 378       |
| Michelle | 15    | 321       |
| Fatimah  | 14    | 400       |
| Anup     | 15    | 302       |
| Aanya    | 16    | 340       |
+----------+-------+-----------+
5 rows in set (0.01 sec)
```

12.6 MYSQL FUNCTIONS

A **function** is a special type of predefined command set that performs some operation and returns a single value. Functions operate on zero, one, two or more values that are provided to them. The values that are provided to functions are called **parameters** or **arguments**.

The MySQL functions have been categorised into various categories, such as *String functions*, *Mathematical functions*, *Date and Time functions* and so on. Although MySQL offers a big bouquet of functions, yet in this discussion we shall remain limited to some common functions.

12.6.1 String Functions

The string functions of MySQL can manipulate the text string in many ways. Some commonly used string functions are being discussed below.

	Function	Description	Examples
1.	CHAR( )	Returns the character for each integer passed	1. SELECT CHAR (70, 65, 67, 69) ; 2. SELECT CHAR (65, 67.3, '69.3) ;
2.	CONCAT( )	Returns concatenated string	SELECT CONCAT(name, aggregate) AS "Name Marks" FROM student WHERE age = 14 OR age = 16 ;
3.	LOWER( ) / LCASE( )	Returns the argument in lowercase	SELECT LOWER('MR. OBAMA') AS "LowerName1", LOWER('Ms. Gandhi') AS "LowerName2" ;
4.	SUBSTRING( ), SUBSTR()	Returns the substring as specified	1. SELECT SUBSTR('ABCDEFG', 3, 4) "Subs" ; 2. SELECT SUBSTR ('ABCDEFG', -5, 4) "Subs"
5.	UPPER( )/UCASE( )	Converts to uppercase	SELECT UPPER('Large') "Uppercase" ; or SELECT UCASE('Large') "Uppercase" ;
6.	TRIM( )	Removes leading and trailing spaces	SELECT TRIM(' Bar One ') ;
7.	LENGTH( )	Returns the length of a string in bytes	SELECT LENGTH('CANDIDE') "Length in characters" ;

12.6.2 Numeric Functions

The number functions are those functions that accept numeric values and after performing the required operation, return numeric values. Some useful numeric functions are being discussed below :

	Function	Description	Example
1.	MOD( )	Returns the remainder of one expression by diving by another expression.	SELECT MOD(11, 4) "Modulus" ;
2.	POWER()/POW( )	Returns the value of one expression raised to the power of another expression	SELECT POWER(3, 2) "Raised" ;
3.	ROUND( )	Returns numeric expression rounded to an integer. Can be used to round an expression to a number of decimal points	SELECT ROUND(15.193, 1) "Round" ;
4.	SIGN( )	This function returns sign of a given number	SELECT SIGN(-15) "Sign" ;
5.	SQRT( )	Returns the non-negative square root of numeric expression.	SELECT SQRT(26) "Square root" ;
6.	TRUNCATE( )	Returns numeric exp1 truncated to exp2 decimal places. If exp2 is 0, then the result will have no decimal point.	SELECT TRUNCATE(15.79, 1) "Truncate" ;

12.6.3 Date and Time Functions

Date functions operate on values of the DATE datatype.

	Function	Description	Example
1.	CURDATE( ) / CURRENT_DATE( ) / CURRENT_DATE	Returns the current date	SELECT CURDATE( ) ;
2.	DATE( )	Extracts the date part of a date or date-time expression	SELECT DATE('2020-12-31 01:02:03') ;
3.	MONTH( )	Returns the month from the date passed	SELECT MONTH('2020-02-03') ;
4.	YEAR( )	Returns the year	SELECT YEAR('2020-02-03') ;
5.	NOW( )	Returns the time at which the function executes	SELECT NOW( ) ;
6.	SYSDATE( )	Returns the current date and time	SELECT NOW(), SLEEP(2), NOW();

EXAMPLE 12.13 *Write a query to create a string from the ASCII values 70, 65, 67, 69.*

Solution.

```
mysql> SELECT CHAR (70, 65, 67, 69 ) ;
+------------------------------+
| char (70, 65, 67, 69 )       |
+------------------------------+
| FACE                         |
+------------------------------+
1 row in set (0.00 sec)
```

EXAMPLE 12.14 *Concatenate name and aggregate for students having age as 14 or 16.*

(*Consider table Student of example 12.3*)

Solution.

```
mysql> SELECT CONCAT(name, aggregate) AS "Name Marks" FROM student
    -> WHERE age = 14 OR age = 16 ;
+--------------+
| Name Marks   |
+--------------+
| Aanya340     |
| Gurvinder480 |
| Ali260       |
| Fatimah400   |
| Mita150      |
+--------------+
5 rows in set (0.00 sec)
```

EXAMPLE 12.15 *Display names 'MR. OBAMA' and 'MS. Gandhi' into lowercase.*

Solution.

```
mysql> SELECT LOWER('MR. OBAMA') AS "LowerName1" ,
    -> LOWER('Ms. Gandhi') AS "LowerName2" ;
```

Or

```
mysql> SELECT LCASE('MR. OBAMA') AS "LowerName1" ,
    -> LCASE ('Ms. Gandhi') AS "LowerName2" ;
```

```
+-------------+-------------+
| LowerName1  | LowerName2  |
+-------------+-------------+
| mr. obama   | ms. gandhi  |
+-------------+-------------+
1 row in set (0.00 sec)
```

EXAMPLE 12.16 *Display 4 characters extracted from 3rd left character onwards from string 'ABCDEFG'.*

Solution.

```
mysql> SELECT SUBSTR('ABCDEFG', 3, 4) "Subs" ;
+------------+
| Subs       |
+------------+
| CDEF       |
+------------+
```

EXAMPLE 12.17 *Display first three characters extracted from jobs of employees 8888 and 8900.*

Solution.

```
mysql> SELECT SUBSTR(job, 1, 3)
    -> FROM empl
    -> WHERE empno = 8888 or empno = 8900 ;
+----------------------+
| SUBSTR(job, 1, 3)    |
+----------------------+
| ANA                  |
| CLE                  |
+----------------------+
```

EXAMPLE 12.18 *Write a query to remove leading and trailing spaces from string ' Bar One '.*

Solution.

```
mysql> SELECT TRIM('  Bar One  ') ;
+-------------------------+
| TRIM('  Bar One  ')     |
+-------------------------+
| Bar One                 |
+-------------------------+
1 row in set (0.00 sec)
```

EXAMPLE 12.19 *How many characters are there in string 'CANDIDE'. ?*

Solution.

```
mysql> SELECT LENGTH('CANDIDE') "Length in characters" ;
+------------------------+
| Length in characters   |
+------------------------+
| 7                      |
+------------------------+
```

EXAMPLE 12.20 *Write a query to extract a sub string from string* **'Quadratically'** *which should be 6 characters long and start from 5th character of the given string.*

Solution.

```
mysql> SELECT SUBSTRING('Quadratically',5,6);
+-----------------------------------+
| SUBSTRING('Quadratically',5,6)    |
+-----------------------------------+
| ratica                            |
+-----------------------------------+
1 row in set (0.02 sec)
```

EXAMPLE 12.21 *Find out the remainder of 11 divided by 4 and display 3^2.*

Solution.

```
mysql> SELECT MOD(11, 4) "Modulus", POWER(3, 2) "Raised" ;
+-------------+------------+
| Modulus     | Raised     |
+-------------+------------+
| 3           |          9 |
+-------------+------------+
```

EXAMPLE 12.22 *Truncate value 15.79 to 1 decimal place.*

Solution.

```
mysql> SELECT TRUNCATE(15.79, 1) "Truncate" ;
+-----------+
| Truncate  |
+-----------+
| 15.7      |
+-----------+
```

EXAMPLE 12.23 *Write a query to display current date on your system.*

Solution.

```
mysql> SELECT CURDATE() ;
+------------+
| CURDATE()  |
+------------+
| 2009-06-03 |
+------------+
1 row in set (0.00 sec)
```

EXAMPLE 12.24 *Write a query to display date after 10 days of current date on your system.*

Solution.

```
mysql> SELECT CURDATE() + 10 ;
+-----------------+
| CURDATE() + 10  |
+-----------------+
| 20090613        |
+-----------------+
1 row in set (0.00 sec)
```

when you add a number to a date, then returned value is not formatted as date.

EXAMPLE 12.25 *Write a query to extract date from a given datetime value '2020-12-31 01:02:03'.*

Solution.

```
mysql> SELECT DATE('2020-12-31 01:02:03') ;
+-----------------------------+
| DATE('2020-12-31 01:02:03') |
+-----------------------------+
| 2020-12-31                  |
+-----------------------------+
1 row in set (0.03 sec)
```

EXAMPLE 12.26 *Write a query to extract month part from date 3rd Feb 2020.*

SOLUTION.

```
mysql> SELECT MONTH('2020-02-03') ;
+----------------------+
| MONTH('2020-02-03')  |
+----------------------+
| 2                    |
+----------------------+
1 row in set (0.03 sec)
```

EXAMPLE 12.27 *Write a query to display current date and time.*

Solution.

```
mysql> SELECT NOW( ) ;
+---------------------+
| NOW()               |
+---------------------+
| 2009-06-03 15:27:20 |
+---------------------+
1 row in set (0.00 sec)
```

EXAMPLE 12.28 *Write a query to illustrate the difference between now( ) and sysdate( ).*

Solution.

```
mysql> SELECT NOW(), SLEEP(2), NOW();
+---------------------+----------+---------------------+
| NOW()               | SLEEP(2) | NOW()               |
+---------------------+----------+---------------------+
| 2009-06-03 15:32:34 | 0        | 2009-06-03 15:32:34 |
+---------------------+----------+---------------------+
1 row in set (2.01 sec)
```

Notice that in first query the now() gives the same value even after 2 seconds (sleep(2)), but sysdate() values have a difference of 2 seconds because of sleep(2)

```
mysql> SELECT SYSDATE(), SLEEP(2), SYSDATE();
+---------------------+----------+---------------------+
| SYSDATE()           | SLEEP(2) | SYSDATE()           |
+---------------------+----------+---------------------+
| 2009-06-03 15:33:15 | 0        | 2009-06-03 15:33:17 |
+---------------------+----------+---------------------+
1 row in set (2.01 sec)
```

Compare the above *two* results. The *now( )* function will return the same result even if you use it multiple times in same statement, even after creating delays (such as through *sleep( )* that creates delay of specified seconds) because the *now( )* function returns the begin-time of statement, whereas *sysdate( )* function returns the execution time of its own *i.e.*, at what time it started to execute. After reading these lines, compare the above results once again. Now you will clearly understand the difference between the functionality of the two functions *now( )* and *sysdate( )*.

12.7 AGGREGATE FUNCTIONS

Till now you have learnt to work with functions that operate on individual rows in a table *e.g.*, if you use **Round( )** function then it will round off values from each row of the table.

MySQL also supports and provides **group functions** or **aggregate functions**. As you can make out that the group functions or aggregate functions work upon groups of rows, rather than on single rows. That is why, these functions are sometimes also called **multiple row functions**.

Many group functions accept the following options :

DISTINCT This option causes a group function to consider only distinct values of the argument expression.

ALL This option causes a group function to consider all values including all duplicates.

The usage of these options will become clear with the coverage of examples in this section. All the examples that we'll be using here, shall be based upon following table **empl**.

Table 12.5 *Database table empl*

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
8369	SMITH	CLERK	8902	1990-12-18	800.00	NULL	20
8499	ANYA	SALESMAN	8698	1991-02-20	1600.00	300.00	30
8521	SETH	SALESMAN	8698	1991-02-22	1250.00	500.00	30
8566	MAHADEVAN	MANAGER	8839	1991-04-02	2985.00	NULL	20
8654	MOMIN	SALESMAN	8698	1991-09-28	1250.00	1400.00	30
8698	BINA	MANAGER	8839	1991-05-01	2850.00	NULL	30
8839	AMIR	PRESIDENT	NULL	1991-11-18	5000.00	NULL	10
8844	KULDEEP	SALESMAN	8698	1991-09-08	1500.00	0.00	30
8882	SHIAVNSH	MANAGER	8839	1991-06-09	2450.00	NULL	10
8886	ANOOP	CLERK	8888	1993-01-12	1100.00	NULL	20
8888	SCOTT	ANALYST	8566	1992-12-09	3000.00	NULL	20
8900	JATIN	CLERK	8698	1991-12-03	950.00	NULL	30
8902	FAKIR	ANALYST	8566	1991-12-03	3000.00	NULL	20
8934	MITA	CLERK	8882	1992-01-23	1300.00	NULL	10

1. AVG

This function computes the average of given data.

SYNTAX

```
AVG([DISTINCT | ALL] n)
```

- Returns average value of parameter(s) *n*.

Argument type : Numeric *Return value* : Numeric

EXAMPLE 12.29. *Calculate average salary of all employees listed in table empl.*

Solution.

```
mysql> SELECT AVG(sal) "Average"
       FROM empl ;
+--------------+
| Average      |
+--------------+
| 2073.928571  |
+--------------+
1 row in set (0.01 sec)
```

2. COUNT

This function counts the number of rows in a given column or expression.

SYNTAX

```
COUNT({ * [DISTINCT | ALL] expr})
```

- Returns the number of rows in the query.
- If you specify argument *expr*, this function returns rows where *expr* is not null. You can count either all rows, or only distinct values of *expr*.
- If you specify the asterisk (*), this function returns all rows, including duplicates and nulls.

Argument type : Numeric *Return value* : Numeric

EXAMPLE 12.30. *Count number of records in table empl.*

Solution.

```
mysql> SELECT COUNT(*) "Total"
       FROM empl ;
```

```
+----------+
| Total    |
+----------+
|   14     |
+----------+
1 row in set (0.00 sec)
```

EXAMPLE 12.31. *Count number of jobs in table empl.*

Solution.

```
mysql> SELECT COUNT(job) "Job Count"
       FROM empl ;
```

```
+-------------+
| Job Count   |
+-------------+
|     14      |
+-------------+
1 row in set (0.01 sec)
```

EXAMPLE 12.32. *How many distinct jobs are listed in table empl ?*

Solution.

```
mysql> SELECT COUNT(DISTINCT job) "Distinct Jobs"
       FROM empl ;
```

```
+----------------+
| Distinct Jobs  |
+----------------+
|       15       |
+----------------+
1 row in set (0.04 sec)
```

3. MAX

This function returns the maximum value from a given column or expression.

SYNTAX

```
MAX([DISTINCT|ALL] expr)
```

- Returns maximum value of argument *expr*.

Argument type : Numeric *Return value* : Numeric

EXAMPLE 12.33. *Display maximum salary from table empl.*

Solution.

```
mysql>  SELECT MAX(sal) "Maximum Salary"
        FROM empl ;
```

```
+----------------+
| Maximum Salary |
+----------------+
|    5000.00     |
+----------------+
1 row in set (0.01 sec)
```

4. MIN

This function returns the minimum value from a given column or expression.

SYNTAX

```
MIN([DISTINCT | ALL] expr)
```

- Returns minimum value of *expr*.

Argument type : Numeric *Return value* : Numeric

EXAMPLE 12.34. *Display the Joining date of seniormost employee.*

Solution.

```
mysql> SELECT MIN(hiredate) "Minimum Hire Date"
       FROM empl ;
```

```
+-------------------+
| Minimum Hire Date |
+-------------------+
|    1990-12-18     |
+-------------------+
1 row in set (0.06 sec)
```

5. SUM

This function returns the sum of values in given column or expression.

SYNTAX

```
SUM([DISTINCT | ALL] n)
```

▲ Returns sum of values of *n*.

Argument type : Numeric *Return value* : Numeric

EXAMPLE 12.35. *Display total salary of all employees listed in table empl.*

Solution.

```
mysql> SELECT SUM(sal) "Total Salary"
       FROM empl ;
```

```
+----------------+
| Total Salary   |
+----------------+
|    29035.00    |
+----------------+
1 row in set (0.01 sec)
```

Some more examples of group functions are being given below :

Examples :

1. To calculate the total gross for employees of grade 'E2', the command is :

```
SELECT SUM(gross) FROM employee
WHERE grade = 'E2' ;
```

2. To display the average gross of employees with grades 'E1' or 'E2', the command used is :

```
SELECT AVG(gross) FROM employee
WHERE (grade = 'E1' OR grade = 'E2') ;
```

3. To count the number of employees in *employee* table, the *SQL* command is :

```
SELECT COUNT(*)
FROM employee ;
```

4. To count the number of cities, the different members belong to, you use the following command :

```
SELECT COUNT(DISTINCT city)  FROM members ;
```

Here the DISTINCT keyword ensures that multiple entries of the same *city* are ignored. The * is the only argument that includes NULLs when it is used only with COUNT, functions other than COUNT disregard NULLs in any case.

If you want to count the entries including repeats, the keyword ALL is used. The following command will COUNT the number of non NULL *city* fields in the *members* table :

```
SELECT COUNT(ALL city)
FROM members ;
```

In general, GROUP functions

- Return a single value for a set of rows.
- Can be applied to any numeric values, and some Text types and DATE values.

NOTE

These functions are called *aggregate functions* because they operate on aggregates of tuples. The result of an aggregate function is a single value.