Chapter 3
Data Management—Relational Database Systems (RDBMS)

Hemanth Kumar Dasararaju and Peeyush Taori

1 Introduction

Storage and management of data is a key aspect of data science. Data, simply speaking, is nothing but a collection of facts—a snapshot of the world—that can be stored and processed by computers. In order to process and manipulate data efficiently, it is very important that data is stored in an appropriate form. Data comes in many shapes and forms, and some of the most commonly known forms of data are numbers, text, images, and videos. Depending on the type of data, there exist multiple ways of storage and processing. In this chapter, we focus on one of the most commonly known and pervasive means of data storage—relational database management systems. We provide an introduction using which a reader can perform the essential operations. References for a deeper understanding are given at the end of the chapter.

2 Motivating Example

Consider an online store that sells stationery to customers across a country. The owner of this store would like to set up a system that keeps track of inventory, sales, operations, and potential pitfalls. While she is currently able to do so on her own, she knows that as her store scales up and starts to serve more and more people, she

H. K. Dasararaju
Indian School of Business, Hyderabad, Telangana, India

P. Taori (✉)
London Business School, London, UK
e-mail: taori.peeyush@gmail.com

B. Pochiraju, S. Seshadri (eds.), *Essentials of Business Analytics*, International Series in Operations Research & Management Science 264,
https://doi.org/10.1007/978-3-319-68837-4_3

will no longer have the capacity to manually record transactions and create records for new occurrences. Therefore, she turns to relational database systems to run her business more efficiently.

A database is a collection of organized data in the form of rows, columns, tables and indexes. In a database, even a small piece of information becomes data. We tend to aggregate related information together and put them under one gathered name called a Table. For example, all student-related data (student ID, student name, date of birth, etc.) would be put in one table called STUDENT table. It decreases the effort necessary to scan for a specific information in an entire database. Since a database is very flexible, data gets updated and extended when new data is added and the database shrinks when data is deleted from the database.

3 Database Systems—What and Why?

As data grows in size, there arises a need for a means of storing it efficiently such that it can be found and processed quickly. In the "olden days" (which was not too far back), this was achieved via systematic filing systems where individual files were catalogued and stored neatly according to a well-developed data cataloging system (similar to the ones you will find in libraries or data storage facilities in organizations). With the advent of computer systems, this role has now been assumed by database systems. Plainly speaking, a database system is a digital record-keeping system or an electronic filing cabinet. Database systems can be used to store large amounts of data, and data can then be queried and manipulated later using a querying mechanism/language. Some of the common operations that can be performed in a database system are adding new files, updating old data files, creating new databases, querying of data, deleting data files/individual records, and adding more data to existing data files. Often pre processing and post-processing of data happen using database languages. For example, one can selectively read data, verify its correctness, and connect it to data structures within applications. Then, after processing, write it back into the database for storage and further processing.

With the advent of computers, the usage of database systems has become ubiquitous in our personal and work lives. Whether we are storing information about personal expenditures using an Excel file or making use of MySQL database to store product catalogues for a retail organization, databases are pervasive and in use everywhere. We also discuss the difference between the techniques discussed in this chapter compared to methods for managing big data in the next chapter.

3.1 Database Management System

A database management system (DBMS) is the system software that enables users to create, organize, and manage databases. As Fig. 3.1 illustrates, The DBMS serves as an interface between the database and the end user, guaranteeing that information is reliably organized and remains accessible.

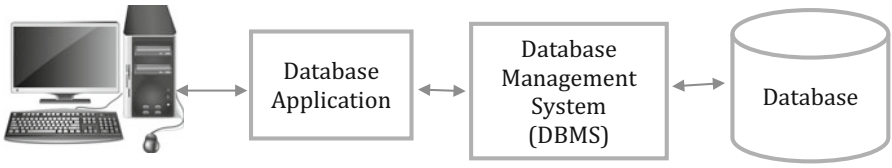

Fig. 3.1 Relating databases to end users

The main objectives of DBMS are mass storage; removal of duplicity—DBMS makes sure that same data has not been stored earlier; providing multiple user access—two or more users can work concurrently; data integrity—ensuring the privacy of the data and preventing unauthorized access; data backup and recovery; nondependence on a particular platform; and so on. There are dozens of DBMS products available. Popular products include Microsoft Access, MYSQL, Oracle from Oracle Corporation, SQL Server from Microsoft, and DB2 from IBM.

3.2 Relational Database Management System

Relational database management system (RDBMS) is a database management system (DBMS) that is based on the relational model of data. DBMS tells us about the tables but Relational DBMS specifies about relations between different entities in the database. The two main principles of the RDBMS are entity integrity and referential integrity.

- *Entity integrity*: Here, all the data should be organized by having a unique value (primary key), so it cannot accept null values.
- *Referential integrity*: Referential integrity must have constraints specified between two relations and the relationship must always be consistent (e.g., foreign key column must be equal to the primary key column).
  - *Primary key*: Primary key is a column in a table that uniquely identifies the rows in that relation (table).
  - *Foreign key*: Foreign keys are columns that point to primary key columns of another table.

Normalization:

Normalization is the database design technique that is used to efficiently organize the data, optimize the table structures, and remove duplicate data entries. It separates the larger tables into smaller tables and links them using the relationships. Normalization is used to improve the speed, for efficient usage of space, and to increase the data integrity. The important normalizations that are used to organize the database are as follows:

- *First normal form (1NF)*: The table must contain "atomic" values only (should not contain any duplicate values, and cannot hold multiple values).

  Example: Suppose the university wants to store the details of students who are finalists of a competition. Table 3.1 shows the data.

Three students (Jon, Robb, and Ken) have two different parents numbers so the university put two numbers in the same field as you see in Table 3.1. This table is not in 1NF as it does not follow the rule "Only atomic values in the field" as there are multiple values in parents_number field. To make the table into 1NF we should store the information as shown in Table 3.2.

- *Second normal form (2NF)*: Must follow first normal form and no non-key attributes are dependent on the proper subset of any candidate key of the table.

Example: Assume a university needs to store the information of the instructors and the topics they teach. They make a table that resembles the one given below (Table 3.3) since an instructor can teach more than one topic.

Table 3.1 Students in a university competition

Student_ID	Student_Name	Address	Parents_number
71121	Jon	New York	75430105417540
71122	Janet	Chicago	1915417
71123	Robb	Boston	63648014889636
71124	Zent	Los Angeles	7545413
71125	Ken	Atlanta	40136924016371

Table 3.2 Students in a university competition sorted efficiently

Student_ID	Student_Name	Address	Parents_number
71121	Jon	New York	7543010
71121	Jon	New York	5417540
71122	Janet	Chicago	1915417
71123	Robb	Boston	6364801
71123	Robb	Boston	4889636
71124	Zent	Los Angeles	7545413
71125	Ken	Atlanta	4013692
71125	Ken	Atlanta	4016371

Table 3.3 Instructors in a university

Instructor_ID	Topic	Instructor_Age
56121	Neural Network	37
56121	IoT	37
56132	Statistics	51
56133	Optimization	43
56133	Simulation	43

Table 3.4 Breaking tables into two in order to agree with 2NF

Instructor_ID	Instructor_Age
56121	37
56132	51
56133	43

Instructor_ID	Topic
56121	Neural Network
56121	IoT
56132	Statistics
56133	Optimization
56133	Simulation

Table 3.5 Students in a university competition

Student_ID	Student_Name	Student_ZIP	Student_State	Student_city	Student_Area
71121	Jon	10001	New York	New York	Queens Manhattan
71122	Janet	60201	Illinois	Chicago	Evanston
71123	Robb	02238	Massachusetts	Boston	Cambridge
71124	Zent	90089	California	Los Angeles	Trousdale

Here Instructor_ID and Topic are key attributes and Instructor_Age is a non-key attribute. The table is in 1NF but not in 2NF because the non-key attribute Instructor_Age is dependent on Instructor_ID. To make the table agree to 2NF, we can break the table into two tables like the ones given in Table 3.4.

- *Third normal form (3NF)*: Must follow second normal form and none of the non-key attributes are determined by another non-key attributes.

  Example: Suppose the university wants to store the details of students who are finalists of a competition. The table is shown in Table 3.5.

Here, student_ID is the key attribute and all other attributes are non-key attributes. Student_State, Student_city, and Student_Area depend on Student_ZIP and Student_ZIP is dependent on Student_ID that makes the non-key attribute transitively dependent on the key attribute. This violates the 3NF rules. To make the table agree to 3NF we can break into two tables like the ones given in Table 3.6.

3NF is the form that is practiced and advocated across most organizational environments. It is because tables in 3NF are immune to most of the anomalies associated with insertion, updation, and deletion of data. However, there could be specific instances when organizations might want to opt for alternate forms of table normalization such as 4NF and 5NF. While 2NF and 3NF normalizations focus on functional aspects, 4NF and 5NF are more concerned with addressing multivalued dependencies. A detailed discussion of 4NF and 5NF forms is beyond the scope

Table 3.6 Breaking tables into two in order to agree with 3NF

Student table:		
Student_ID	Student_Name	Student_ZIP
71121	Jon	10001
71122	Janet	60201
71123	Robb	02238
71124	Zent	90089

Student_zip table:			
Student_ZIP	Student_State	Student_city	Student_Area
10001	New York	New York	Queens Manhattan
60201	Illinois	Chicago	Evanston
02238	Massachusetts	Boston	Cambridge
90089	California	Los Angeles	Trousdale

of discussion for this chapter, but interested reader can learn more online from various sources.[1] It should be noted that in many organizational scenarios, the focus is mainly on achieving 3NF.

3.3 Advantages of RDBMS over EXCEL

Most businesses today need to record and store information. Sometimes this may be only for record keeping and sometimes data is stored for later use. We can store the data in Microsoft Excel. But why is RDBMS the most widely used method to store data?

Using Excel we can perform various functions like adding the data in rows and columns, sorting of data by various metrics, etc. But Excel is a two-dimensional spreadsheet and thus it is extremely hard to make connections between information in various spreadsheets. It is easy to view the data or find the particular data from Excel when the size of the information is small. It becomes very hard to read the information once it crosses a certain size. The data might scroll many pages when endeavoring to locate a specific record.

Unlike Excel, in RDBMS, the information is stored independently from the user interface. This separation of storage and access makes the framework considerably more scalable and versatile. In RDBMS, data can be easily cross-referenced between multiple databases using relationships between them but there are no such options in Excel. RDBMS utilizes centralized data storage systems that makes backup and maintenance much easier. Database frameworks have a tendency to be significantly faster as they are built to store and manipulate large datasets unlike Excel.

[1] (http://www.bkent.net/Doc/simple5.htm (accessed on Feb 6, 2019))

4 Structured Query Language (SQL)

SQL (structured query language) is a computer language exclusive to a particular application domain in contrast to some other general-purpose language (GPL) such as C, Java, or Python that is broadly applicable across domains. SQL is text oriented, and designed for managing (access and manipulate) data. SQL was authorized as a national standard by the ANSI (American National Standards Institute) in 1992. It is the standard language for relational database management systems. Some common relational database management systems that operate using SQL are Microsoft Access, MySQL, Oracle, SQL Server, and IBM DB2. Even though many database systems make use of SQL, they also have their unique extensions that are specific to their systems.

SQL statements are used to select the particular part of the data, retrieve data from a database, and update data on the database using CREATE, SELECT, INSERT, UPDATE, DELETE, and DROP commands. SQL commands can be sliced into four categories: DDL (data definition language), which is used to define the database structures; DML (data manipulation language), which is used to access and modify database data; DCL (data control language); and TCL (transaction control language).

DDL (Data Definition Language):

DDL deals with the database schemas and structure. The following statements are used to take care of the design and storage of database objects.

1. CREATE: Creates the database, table, index, views, store, procedure, functions, and triggers.
2. ALTER: Alters the attributes, constraints, and structure of the existing database.
3. DROP: Deletes the objects (table, view, functions, etc.) from the database.
4. TRUNCATE: Removes all records from a table, including the space allocated to the records.
5. COMMENT: Associates comments about the table or about any objects to the data dictionary.
6. RENAME: Renames the objects.

DML (Data Manipulation Language):

DML deals with tasks like storing, modifying, retrieving, deleting, and updating the data in/from the database.

1. SELECT: The only data retrieval statement in SQL, used to select the record(s) from the database.
2. INSERT: Inserts a new data/observation into the database.
3. UPDATE: Modifies the existing data within the database.
4. DELETE: Removes one or more records from the table.

Note: There is an important difference between the DROP, TRUNCATE, and DELETE commands. DELETE (Data alone deleted) operations can be recalled back (undo), while DROP (Table structure + Data are deleted) and TRUNCATE operations cannot be recalled back.

DCL (Data Control Language):

Data control languages are used to uphold the database security during multiple user data environment. The database administrator (DBA) is responsible for "grant/revoke" privileges on database objects.

1. GRANT: Provide access or privilege on database objects to the group of users or particular user.
2. REVOKE: Remove user access rights or privilege to the database objects.

TCL (Traction Control Language):

Transaction control language statements enable you to control and handle transactions to keep up the trustworthiness of the information within SQL statements.

1. BEGIN: Opens a transaction.
2. COMMIT: Saves the transaction on the database.
3. ROLLBACK: Rollback (undo the insert, delete, or update) the transaction in the database in case of any errors.
4. SAVEPOINT: Rollback to the particular point (till the savepoint marked) of transaction. The progression done until the savepoint will be unaltered and all transaction after that will be rolled back.

4.1 Introduction to MySQL

In this section, we will walk through the basics of creating a database using MySQL[2] and query the database using the MySQL querying language. As described earlier in the chapter, a MySQL database server is capable of hosting many databases. In databases parlance, a database is often also called a schema. Thus, a MySQL server can contain a number of schemas. Each of those schemas (database) is made up of a number of tables, and every table contains rows and columns. Each row represents an individual record or observation, and each column represents a particular attribute such as age and salary.

When you launch the MySQL command prompt, you see a command line like the one below (Fig. 3.2).

[2]MySQL Workbench or Windows version can be downloaded from https://dev.mysql.com/downloads/windows/ (accessed on Feb 15, 2018) for practice purpose.

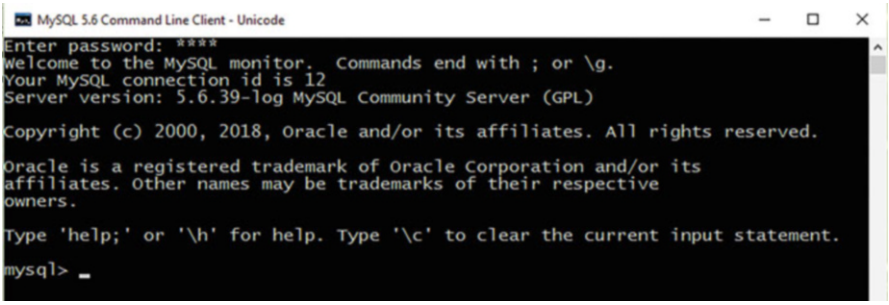

Fig. 3.2 MySQL command prompt Interface

The command line starts with "mysql>" and you can run SQL scripts by closing commands with semicolon (;).

4.2 How to Check the List of Databases Available in MySQL?

In order to get started we will first check the databases that are already present in a MySQL server. To do so, type "**show databases**" in the command line. Once you run this command, it will list all the available databases in the MySQL server installation. The above-mentioned command is the first SQL query that we have run. Please note that keywords and commands are case-insensitive in MySQL as compared to R and Python where commands are case-sensitive in nature.

```
mysql> SHOW DATABASES;

Output:
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| test               |
+--------------------+
4 rows in set (0.00 sec)
```

You would notice that there are already four schemas listed though we have not yet created any one of them. Out of the four databases, "information_schema", "mysql", and "performance_schema" are created by MySQL server for its internal monitoring and performance optimization purposes and should not be used when we are creating our own database. Another schema "test" is created by MySQL during the installation phase and it is provided for testing purposes. You can remove the "test" schema or can use it to create your own tables.

4.3 Creating and Deleting a Database

Now let us create our own database. The syntax for creating a database in MySQL is:

```
CREATE DATABASE databasename;
```

Let us create a simple inventory database. We shall create a number of tables about products and their sales information such as customers, products, orders, shipments, and employee. We will call the database "product_sales". In order to create a database, type the following SQL query:

```
mysql> CREATE DATABASE product_sales;

Output:
Query OK, 1 row affected (0.00 sec)

mysql> SHOW DATABASES;

Output:
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| product_sales      |
| test               |
+--------------------+
5 rows in set (0.00 sec)
```

In the above-mentioned query, we are creating a database called "product_sales." Once the query is executed, if you issue the "show databases" command again, then it will now show five databases (with "product_sales" as the new database). As of now, "product_sales" will be an empty database, meaning there would be no tables inside it. We will start creating tables and populating them with data in a while.

In order to delete a database, you need to follow the following syntax:

```
DROP DATABASE databasename;
```

In our case, if we need to delete "product_sales", we will issue the command:

```
mysql> DROP DATABASE product_sales;

Output:
Query OK, 0 rows affected (0.14 sec)

mysql> SHOW DATABASES;

Output:
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
```

```
| performance_schema  |
| test                |
+---------------------+
4 rows in set (0.00 sec)
```

Oftentimes, when you have to create a database, you might not be sure if a database of the same name exists already in the system. In such cases, conditions such as "IF EXISTS" and "IF NOT EXISTS" come in handy. When we execute such query, then the database is created if there is no other database of the same name. This helps us in avoiding overwriting of the existing database with the new one.

```
mysql> CREATE DATABASE IF NOT EXISTS product_sales;
```

```
Output:
Query OK, 1 row affected (0.00 sec)
```

One important point to keep in mind is the use of SQL DROP commands with extreme care, because once you delete an entity or an entry, then there is no way to recover the data.

4.4 Selecting a Database

There can be multiple databases available in the MySQL server. In order to work on a specific database, we have to select the database first. The basic syntax to select a database is:

```
USE databasename;
```

In our case, if we have to select "product_sales" database, we will issue the command:

```
mysql> USE product_sales;
```

```
Output:
Database changed
```

When we run the above query, the default database now is "product_sales". Whatever operations we will now perform will be performed on this database. This implies that if you have to use a specific table in the database, then you can simply do so by calling the table name. If at any point of time you want to check which your selected database is then issue the command:

```
mysql> SELECT DATABASE();
```

```
Output:
+---------------+
| DATABASE()    |
+---------------+
| product_sales |
+---------------+
1 row in set (0.00 sec)
```

If you want to check all tables in a database, then issue the following command:

```
mysql> SHOW TABLES;

Output:
Empty set (0.00 sec)
```

As of now it is empty since we have not yet created any table. Let us now go ahead and create a table in the database.

4.5 Table Creation and Deletion

The syntax for creating a new table is:

```
CREATE TABLE tablename (IF EXISTS);
```

The above command will create the table with table name as specified by the user. You can also specify the optional condition IF EXISTS/IF NOT EXISTS similar to the way you can specify them while creating a database. Since a table is nothing but a collection of rows and columns, in addition to specifying the table name, you would also want to specify the column names in the table and the type of data that each column can contain. For example, let us go ahead and create a table named "products." We will then later inspect it in greater detail.

```
mysql> CREATE TABLE products (productID INT 10 UNSIGNED NOT NULL
  AUTO_INCREMENT, code CHAR(6) NOT NULL DEFAULT '', productname
  VARCHAR(30) NOT NULL DEFAULT '', quantity INT UNSIGNED NOT NULL
  DEFAULT 0, price DECIMAL(5,2) NOT NULL DEFAULT 0.00, PRIMARY
  KEY (productID) );

Output:
Query OK, 0 rows affected (0.41 sec)
```

In the above-mentioned command, we have created a table named "*products*." Along with table name, we have also specified the columns and the type of data that each column contains within the parenthesis. For example, "*products*" table contains five columns—productID, code, productname, quantity, and price. Each of those columns can contain certain types of data. Let us look at them one by one:

- *productID* is INT 10 UNSIGNED (INT means integer, it accepts only integer values for productID). And the number 10 after INT represents the size of the integer; here in this case productID accepts an integer of maximum size 10. And the attribute UNSIGNED means nonnegative integers, which means the productID will accept only positive integers. Thus, if you enter any non-integer value, negative value, or number great than that of size 10, it will throw you an error. If you do not specify the attribute UNSIGNED in the command, by default it will take SIGNED attribute, which accepts both positive and negative integers.
- *code* is CHAR(6)—CHAR(6) means a fixed-length alphanumeric string that can contain exactly six characters. It accepts only six characters.

- *productname* is VARCHAR(30). Similar to CHAR, VARCHAR stands for a variable length string that can contain a maximum of 30 characters. The contrast between CHAR and VARCHAR is that whereas CHAR is a fixed length string, VARCHAR can vary in length. In practice, it is always better to use VARCHAR unless you suspect that the string in a column is always going to be of a fixed length.
- *quantity* INT. This means that quantity column can contain integer values.
- *price* DECIMAL(5,2). Price column can contain floating point numbers (decimal numbers) of length 5 and the length of decimal digits can be a maximum of 2. Whenever you are working with floating point numbers, it is advisable to use DECIMAL field.

There are a number of additional points to be noted with regard to the above statement.

For a number of columns such as productID, productname you would notice the presence of NOT NULL. NOT NULL is an attribute that essentially tells MySQL that the column cannot have null values. NULL in MySQL is not a string and is instead a special character to signify absence of values in the field. Each column also contains the attribute DEFAULT. This essentially implies that if no value is provided by the user then use default value for the column. For example, default value for column quantity will be 0 in case no values are provided when inputting data to the table.

The column productID has an additional attribute called AUTO_INCREMENT, and its default value is set to 1. This implies that whenever there is a null value specified for this column, a default value would instead be inserted but this default value will be incremented by 1 with a starting value of 1. Thus, if there are two missing productID entries, then the default values of 1 and 2 would be provided.

Finally, the last line of table creation statement query is PRIMARY KEY (productID). Primary key for a table is a column or set of columns where each observation in that column would have a unique value. Thus, if we have to look up any observation in the table, then we can do so using the primary key for the table. Although it is not mandatory to have primary keys for a table, it is a standard practice to have one for every table. This also helps during indexing the table and makes query execution faster.

If you would now run the command SHOW TABLES, then the table would be reflected in your database.

```
mysql> SHOW TABLES;

Output:
+-------------------------+
| Tables_in_product_sales |
+-------------------------+
| products                |
+-------------------------+
1 row in set (0.00 sec)
```

You can always look up the schema of a table by issuing the "DESCRIBE" command:

```
mysql> DESCRIBE products;
```

```
Output:
+------------+------------------+------+-----+---------+----------------+
| Field      | Type             | Null | Key | Default | Extra          |
+------------+------------------+------+-----+---------+----------------+
| productID  | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
| code       | char(6)          | NO   |     |         |                |
| productname| varchar(30)      | NO   |     |         |                |
| quantity   | int(10) unsigned | NO   |     | 0       |                |
| price      | decimal(5,2)     | NO   |     | 0.00    |                |
+------------+------------------+------+-----+---------+----------------+
5 rows in set (0.01 sec)
```

4.6 Inserting the Data

Once we have created the table, it is now time to insert data into the table. For now we will look at how to insert data manually in the table. Later on we will see how we can import data from an external file (such as CSV or text file) in the database. Let us now imagine that we have to insert data into the products table we just created. To do so, we make use of the following command:

```
mysql> INSERT INTO products VALUES (1, 'IPH', 'Iphone 5S Gold',
  300, 625);
```

```
Output:
Query OK, 1 row affected (0.13 sec)
```

When we issue the above command, it will insert a single row of data into the table "*products*." The parenthesis after VALUES specified the actual values that are to be inserted. An important point to note is that values should be specified in the same order as that of columns when we created the table "*products.*" All numeric data (integers and decimal values) are specified without quotes, whereas character data must be specified within quotes.

Now let us go ahead and insert some more data into the "*products.*" table:

```
mysql> INSERT INTO products VALUES(NULL, 'IPH',
  'Iphone 5S Black', 8000, 655.25),(NULL, 'IPH',
  'Iphone 5S Blue', 2000, 625.50);
```

```
Output:
Query OK, 2 rows affected (0.13 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

In the above case, we inserted multiple rows of data at the same time. Each row of data was specified within parenthesis and each row was separated by a comma (,). Another point to note is that we kept the productID fields as null when inserting the data. This is to demonstrate that even if we provide null values, MySQL will make use of AUTO_INCREMENT operator to assign values to each row.

Sometimes there might be a need where you want to provide data only for some columns or you want to provide data in a different order as compared to the original one when we created the table. This can be done using the following command:

```
mysql> INSERT INTO products (code, productname, quantity, price)
  VALUES ('SNY', 'Xperia Z1', 10000, 555.48),('SNY', 'Xperia S',
  8000, 400.49);

Output:
Query OK, 2 rows affected (0.13 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

Notice here that we did not specify the productID column for values to be inserted in, but rather explicitly specified the columns and their order in which we want to insert the data. The productID column will be automatically populated using AUTO_INCREMENT operator.

4.7 Querying the Database

Now that we have inserted some values into the products table, let us go ahead and see how we can query the data. If you want to see all observations in a database table, then make use of the SELECT * FROM tablename query:

```
mysql> SELECT * FROM products;

Output:
+----------+------+-----------------+----------+--------+
| productID | code | productname     | quantity | price  |
+----------+------+-----------------+----------+--------+
|         1 | IPH  | Iphone 5S Gold  |      300 | 625.00 |
|         2 | IPH  | Iphone 5S Black |     8000 | 655.25 |
|         3 | IPH  | Iphone 5S Blue  |     2000 | 625.50 |
|         4 | SNY  | Xperia Z1       |    10000 | 555.48 |
|         5 | SNY  | Xperia S        |     8000 | 400.49 |
+----------+------+-----------------+----------+--------+
5 rows in set (0.00 sec)
```

SELECT query is perhaps the most widely known query of SQL. It allows you to query a database and get the observations matching your criteria. SELECT * is the most generic query, which will simply return all observations in a table. The general syntax of SELECT query is as follows:

```
SELECT column1Name, column2Name, ... FROM tableName
```

This will return selected columns from a particular table name. Another variation of SELECT query can be the following:

```
SELECT column1Name, column2Name ....from tableName where
 somecondition;
```

In the above version, only those observations would be returned that match the criteria specified by the user. Let us understand them with the help of a few examples:

```
mysql> SELECT productname, quantity FROM products;

Output:
+-----------------+----------+
| productname     | quantity |
+-----------------+----------+
| Iphone 5S Gold  |      300 |
| Iphone 5S Black |     8000 |
| Iphone 5S Blue  |     2000 |
| Xperia Z1       |    10000 |
| Xperia S        |     8000 |
+-----------------+----------+
5 rows in set (0.00 sec)

mysql> SELECT productname, price FROM products WHERE price < 600;

Output:
+-------------+--------+
| productname | price  |
+-------------+--------+
| Xperia Z1   | 555.48 |
| Xperia S    | 400.49 |
+-------------+--------+
2 rows in set (0.00 sec)
```

The above query will only give name and price columns for those records whose price <600.

```
mysql> SELECT productname, price FROM products
        WHERE price >= 600;

Output:
+-----------------+--------+
| productname     | price  |
+-----------------+--------+
| Iphone 5S Gold  | 625.00 |
| Iphone 5S Black | 655.25 |
| Iphone 5S Blue  | 625.50 |
+-----------------+--------+
3 rows in set (0.00 sec)
```

The above query will only give name and price columns for those records whose price >= 600.

In order to select observations based on string comparisons, enclose the string within quotes. For example:

```
mysql> SELECT productname, price FROM products
  WHERE code = 'IPH';

Output:
+-----------------+--------+
| productname     | price  |
+-----------------+--------+
| Iphone 5S Gold  | 625.00 |
| Iphone 5S Black | 655.25 |
| Iphone 5S Blue  | 625.50 |
+-----------------+--------+
3 rows in set (0.00 sec)
```

The above command gives you the name and price of the products whose code is "IPH."

In addition to this, you can also perform a number of string pattern matching operations, and wildcard characters. For example, you can make use of operators LIKE and NOT LIKE to search if a particular string contains a specific pattern. In order to do wildcard matches, you can make use of underscore character "_" for a single-character match, and percentage sign "%" for multiple-character match. Here are a few examples:

- "phone%" will match strings that start with phone and can contain any characters after.
- "%phone" will match strings that end with phone and can contain any characters before.
- "%phone%" will match strings that contain phone anywhere in the string.
- "c_a" will match strings that start with "c" and end with "a" and contain any single character in-between.

```
mysql> SELECT productname, price FROM products WHERE productname
  LIKE 'Iphone%';

Output:
+-----------------+--------+
| productname     | price  |
+-----------------+--------+
| Iphone 5S Gold  | 625.00 |
| Iphone 5S Black | 655.25 |
| Iphone 5S Blue  | 625.50 |
+-----------------+--------+
3 rows in set (0.00 sec)

mysql> SELECT productname, price FROM products WHERE productname
   LIKE '%Blue%';

Output:
+----------------+--------+
| productname    | price  |
+----------------+--------+
| Iphone 5S Blue | 625.50 |
+----------------+--------+
1 row in set (0.00 sec)
```

Additionally, you can also make use of Boolean operators such as AND, OR in SQL queries to create multiple conditions.

```
mysql> SELECT * FROM products WHERE quantity >= 5000 AND
  productname LIKE 'Iphone%';

Output:
+-----------+------+-----------------+----------+--------+
| productID | code | productname     | quantity | price  |
+-----------+------+-----------------+----------+--------+
|         2 | IPH  | Iphone 5S Black |     8000 | 655.25 |
+-----------+------+-----------------+----------+--------+
1 row in set (0.00 sec)
```

This gives you all the details of products whose quantity is >=5000 and the name like 'Iphone'.

```
mysql> SELECT * FROM products WHERE quantity >= 5000 AND price >
  650 AND productname LIKE 'Iphone%';

Output:
+-----------+------+-----------------+----------+--------+
| productID | code | productname     | quantity | price  |
+-----------+------+-----------------+----------+--------+
|         2 | IPH  | Iphone 5S Black |     8000 | 655.25 |
+-----------+------+-----------------+----------+--------+
1 row in set (0.00 sec)
```

If you want to find whether the condition matches any elements from within a set, then you can make use of IN operator. For example:

```
mysql> SELECT * FROM products WHERE productname IN ('Iphone 5S
  Blue', 'Iphone 5S Black');

Output:
+-----------+------+-----------------+----------+--------+
| productID | code | productname     | quantity | price  |
+-----------+------+-----------------+----------+--------+
|         2 | IPH  | Iphone 5S Black |     8000 | 655.25 |
|         3 | IPH  | Iphone 5S Blue  |     2000 | 625.50 |
+-----------+------+-----------------+----------+--------+
2 rows in set (0.00 sec)
```

This gives the product details for the names provided in the list specified in the command (i.e., "Iphone 5S Blue", "Iphone 5S Black").

Similarly, if you want to find out if the condition looks for values within a specific range then you can make use of BETWEEN operator. For example:

```
mysql> SELECT * FROM products WHERE (price BETWEEN 400 AND 600)
  AND (quantity BETWEEN 5000 AND 10000);
```

```
Output:
+-----------+------+-------------+----------+--------+
| productID | code | productname | quantity | price  |
+-----------+------+-------------+----------+--------+
|         4 | SNY  | Xperia Z1   |    10000 | 555.48 |
|         5 | SNY  | Xperia S    |     8000 | 400.49 |
+-----------+------+-------------+----------+--------+
2 rows in set (0.00 sec)
```

This command gives you the product details whose price is between 400 and 600 and quantity is between 5000 and 10000, both inclusive.

4.8 ORDER BY Clause

Many a times when we retrieve a large number of results, we might want to sort them in a specific order. In order to do so, we make use of ORDER BY in SQL. The general syntax for this is:

```
SELECT ... FROM tableName
WHERE criteria
ORDER BY columnA ASC|DESC, columnB ASC|DESC
```

```
mysql> SELECT * FROM
  products WHERE productname LIKE 'Iphone%' ORDER BY price DESC;
```

```
Output:
+-----------+------+-----------------+----------+--------+
| productID | code | productname     | quantity | price  |
+-----------+------+-----------------+----------+--------+
|         2 | IPH  | Iphone 5S Black |     8000 | 655.25 |
|         3 | IPH  | Iphone 5S Blue  |     2000 | 625.50 |
|         1 | IPH  | Iphone 5S Gold  |      300 | 625.00 |
+-----------+------+-----------------+----------+--------+
3 rows in set (0.00 sec)
```

If you are getting a large number of results but want the output to be limited only to a specific number of observations, then you can make use of LIMIT clause. LIMIT followed by a number will limit the number of output results that will be displayed.

```
mysql> SELECT * FROM products ORDER BY price LIMIT 2;
```

```
Output:
+-----------+------+-------------+----------+--------+
| productID | code | productname | quantity | price  |
+-----------+------+-------------+----------+--------+
|         5 | SNY  | Xperia S    |     8000 | 400.49 |
|         4 | SNY  | Xperia Z1   |    10000 | 555.48 |
+-----------+------+-------------+----------+--------+
2 rows in set (0.00 sec)
```

Oftentimes, we might want to display the columns or tables by an intuitive name that is different from the original name. To be able to do so, we make use of AS alias.

```
mysql> SELECT productID AS ID, code AS productCode , productname
  AS Description, price AS Unit_Price FROM products ORDER
  BY ID;

Output:
+----+-------------+-----------------+------------+
| ID  | productCode  | Description      | Unit_Price  |
+----+-------------+-----------------+------------+
|  1  | IPH          | Iphone 5S Gold   |     625.00  |
|  2  | IPH          | Iphone 5S Black  |     655.25  |
|  3  | IPH          | Iphone 5S Blue   |     625.50  |
|  4  | SNY          | Xperia Z1        |     555.48  |
|  5  | SNY          | Xperia S         |     400.49  |
+----+-------------+-----------------+------------+
5 set (0.00 sec)
```

4.9 Producing Summary Reports

A key part of SQL queries is to be able to provide summary reports from large amounts of data. This summarization process involves data manipulation and grouping activities. In order to enable users to provide such summary reports, SQL has a wide range of operators such as DISTINCT, GROUP BY that allow quick summarization and production of data. Let us look at these operators one by one.

4.9.1 DISTINCT

A column may have duplicate values. We could use the keyword DISTINCT to select only distinct values. We can also apply DISTINCT to several columns to select distinct combinations of these columns. For example:

```
mysql> SELECT DISTINCT code FROM products;

Output:
+-----+
| Code |
+-----+
| IPH |
| SNY |
+-----+
2 rows in set (0.00 sec)
```

4.9.2 GROUP BY Clause

The GROUP BY clause allows you to *collapse* multiple records with a common value into groups. For example,

```
mysql> SELECT * FROM products ORDER BY code, productID;

Output:
+-----------+------+-----------------+----------+--------+
| productID | code | productname     | quantity | price  |
+-----------+------+-----------------+----------+--------+
|         1 | IPH  | Iphone 5S Gold  |      300 | 625.00 |
|         2 | IPH  | Iphone 5S Black |     8000 | 655.25 |
|         3 | IPH  | Iphone 5S Blue  |     2000 | 625.50 |
|         4 | SNY  | Xperia Z1       |    10000 | 555.48 |
|         5 | SNY  | Xperia S        |     8000 | 400.49 |
+-----------+------+-----------------+----------+--------+
5 rows in set (0.00 sec)mysql> SELECT * FROM products GROUP BY
  code; #-- Only first record in each group is shown

Output:
+-----------+------+----------------+----------+--------+
| productID | code | productname    | quantity | price  |
+-----------+------+----------------+----------+--------+
|         1 | IPH  | Iphone 5S Gold |      300 | 625.00 |
|         4 | SNY  | Xperia Z1      |    10000 | 555.48 |
+-----------+------+----------------+----------+--------+
2 rows in set (0.00 sec)
```

We can apply GROUP BY clause with aggregate functions to produce group summary report for each group.

The function COUNT(*) returns the rows selected; COUNT(*columnName*) counts only the non-NULL values of the given column. For example,

```
mysql> SELECT COUNT(*) AS 'Count' FROM products;

Output:
+-------+
| Count |
+-------+
|     5 |
+-------+
1 row in set (0.00 sec)

mysql> SELECT code, COUNT(*) FROM products
  GROUP BY code;

Output:
+------+----------+
| code | COUNT(*) |
+------+----------+
| IPH  |        3 |
| SNY  |        2 |
+------+----------+
2 rows in set (0.00 sec)
```

We got "IPH" count as 3 because we have three entries in our table with the product code "IPH" and similarly two entries for the product code "SNY." Besides

COUNT(), there are many other aggregate functions such as AVG(), MAX(), MIN(), and SUM(). For example,

```
mysql> SELECT MAX(price), MIN(price), AVG(price), SUM(quantity)
  FROM products;
Output:
+------------+------------+------------+---------------+
| MAX(price) | MIN(price) | AVG(price) | SUM(quantity) |
+------------+------------+------------+---------------+
|     655.25 |     400.49 | 572.344000 |         28300 |
+------------+------------+------------+---------------+
1 row in set (0.00 sec)
```

This gives you MAX price, MIN price, AVG price, and total quantities of all the products available in our products table. Now let us use GROUP BY clause:

```
mysql> SELECT code, MAX(price) AS 'Highest Price', MIN(price) AS
  'Lowest Price' FROM products GROUP BY code;

Output:
+------+---------------+--------------+
| code | Highest Price | Lowest Price |
+------+---------------+--------------+
| IPH  |        655.25 |       625.00 |
| SNY  |        555.48 |       400.49 |
+------+---------------+--------------+
2 rows in set (0.00 sec)
```

This means, the highest price of an IPhone available in our database is 655.25 and the lowest price is 625.00. Similarly, the highest price of a Sony is 555.48 and the lowest price is 400.49.

4.10 Modifying Data

To modify the existing data, use UPDATE, SET command, with the following syntax:

```
UPDATE tableName SET columnName = {value|NULL|DEFAULT}, ... WHERE
  criteria

mysql> UPDATE products SET quantity = quantity + 50,
  price = 600.5 WHERE productname = 'Xperia Z1';

Output:
Query OK, 1 row affected (0.14 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Let us check the modification in the products table.

```
mysql> SELECT * FROM products WHERE productname = 'Xperia Z1';
```

```
Output:
+-----------+------+-------------+----------+--------+
| productID | code | productname | quantity | price  |
+-----------+------+-------------+----------+--------+
|         4 | SNY  | Xperia Z1   |    10050 | 600.50 |
+-----------+------+-------------+----------+--------+
1 row in set (0.00 sec)
```

You can see that the quantity of Xperia Z1 is increased by 50.

4.11 Deleting Rows

Use the DELETE FROM command to delete row(s) from a table; the syntax is:

`DELETE FROM tableName #` to delete all rows from the table.

`DELETE FROM tableName WHERE criteria #` to delete only the row(s) that meets the *criteria.* For example,

```
mysql> DELETE FROM products WHERE productname LIKE 'Xperia%';

Output:
Query OK, 2 rows affected (0.03 sec)

mysql> SELECT * FROM products;

Output:
+-----------+------+-----------------+----------+--------+
| productID | code | productname     | quantity | price  |
+-----------+------+-----------------+----------+--------+
|         1 | IPH  | Iphone 5S Gold  |      300 | 625.00 |
|         2 | IPH  | Iphone 5S Black |     8000 | 655.25 |
|         3 | IPH  | Iphone 5S Blue  |     2000 | 625.50 |
+-----------+------+-----------------+----------+--------+
3 rows in set (0.00 sec)

mysql> DELETE FROM products;

Output:
Query OK, 3 rows affected (0.14 sec)

mysql> SELECT * FROM products;

Output:
Empty set (0.00 sec)
```

Beware that "DELETE FROM *tableName*" without a WHERE clause deletes ALL records from the table. Even with a WHERE clause, you might have deleted some records unintentionally. It is always advisable to issue a SELECT command with the same WHERE clause to check the result set before issuing the DELETE (and UPDATE).

4.12 Create Relationship: One-To-Many

4.12.1 PRIMARY KEY

Suppose that each product has one supplier, and each supplier supplies one or more products. We could create a table called "*suppliers*" to store suppliers' data (e.g., name, address, and phone number). We create a column with unique value called supplierID to identify every supplier. We set supplierID as the *primary key* for the table suppliers (to ensure uniqueness and facilitate fast search).

In order to relate the suppliers table to the products table, we add a new column into the "*products*" table—the supplierID.

We then set the supplierID column of the products table as a *foreign key* which references the supplierID column of the "*suppliers*" table to ensure the so-called *referential integrity*. We need to first create the "*suppliers*" table, because the "*products*" table references the "*suppliers*" table.

```
mysql> CREATE TABLE suppliers (supplierID INT UNSIGNED NOT NULL
  AUTO_INCREMENT, name VARCHAR(30) NOT NULL DEFAULT '', phone
  CHAR(8) NOT NULL DEFAULT '', PRIMARY KEY (supplierID));

Output:
Query OK, 0 rows affected (0.33 sec)

mysql> DESCRIBE suppliers;

Output:
+------------+------------------+------+-----+---------+----------------+
| Field      | Type             | Null | Key | Default | Extra          |
+------------+------------------+------+-----+---------+----------------+
| supplierID | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
| name       | varchar(30)      | NO   |     |         |                |
| phone      | char(8)          | NO   |     |         |                |
+------------+------------------+------+-----+---------+----------------+
3 rows in set (0.01 sec)
```

Let us insert some data into the suppliers table.

```
mysql> INSERT INTO suppliers VALUE (501, 'ABC Traders',
  '88881111'), (502, 'XYZ Company', '88882222'), (503, 'QQ Corp',
  '88883333');

Output:
Query OK, 3 rows affected (0.13 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM suppliers;
```

```
Output:
+------------+-------------+----------+
| supplierID | name        | phone    |
+------------+-------------+----------+
|        501 | ABC Traders | 88881111 |
|        502 | XYZ Company | 88882222 |
|        503 | QQ Corp     | 88883333 |
+------------+-------------+----------+
3 rows in set (0.00 sec)
```

4.12.2 ALTER TABLE

The syntax for ALTER TABLE is as follows:

```
ALTER TABLE tableName
{ADD [COLUMN] columnName columnDefinition}
{ALTER|MODIFY [COLUMN] columnName columnDefinition
{SET DEFAULT columnDefaultValue} | {DROP DEFAULT}}
{DROP [COLUMN] columnName [RESTRICT|CASCADE]}
{ADD tableConstraint}
{DROP tableConstraint [RESTRICT|CASCADE]}
```

Instead of deleting and re-creating the products table, we shall use the statement "ALTER TABLE" to add a new column supplierID into the products table. As we have deleted all the records from products in recent few queries, let us rerun the three INSERT queries referred in the Sect. 4.6 before running "ALTER TABLE."

```
mysql> ALTER TABLE products ADD COLUMN supplierID INT UNSIGNED
  NOT NULL;

Output:
Query OK, 0 rows affected (0.43 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> DESCRIBE products;

Output:
+-------------+-----------------+-----+-----+---------+----------------+
| Field       | Type            | Null | Key | Default | Extra          |
+-------------+-----------------+-----+-----+---------+----------------+
| productID   | int(10) unsigned | NO  | PRI | NULL    | auto_increment |
| code        | char(6)          | NO  |     |         |                |
| productname | varchar(30)      | NO  |     |         |                |
| quantity    | int(10) unsigned | NO  |     | 0       |                |
| price       | decimal(5,2)     | NO  |     | 0.00    |                |
| supplierID  | int(10) unsigned | NO  |     | NULL    |                |
+-------------+-----------------+-----+-----+---------+----------------+
6 rows in set (0.00 sec)
```

4.12.3 FOREIGN KEY

Now, we shall add a *foreign key constraint* on the supplierID columns of the "*products*" child table to the "*suppliers*" parent table, to ensure that every supplierID in the "*products*" table always refers to a *valid* supplierID in the "*suppliers*" table. This is called *referential integrity*.

Before we add the foreign key, we need to set the supplierID of the existing records in the "*products*" table to a valid supplierID in the "*suppliers*" table (say supplierID = 501).

Now let us set the supplierID of the existing records to a valid supplierID of "*supplier*" table. As we have deleted the records from "*products*" table, we can add or update using UPDATE command.

```
mysql> UPDATE products SET supplierID = 501;

Output:
Query OK, 5 rows affected (0.04 sec)
Rows matched: 5  Changed: 5  Warnings: 0
```

Let us add a foreign key constraint.

```
mysql> ALTER TABLE products ADD FOREIGN KEY (supplierID)
  REFERENCES suppliers (supplierID);

Output:
Query OK, 0 rows affected (0.56 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> DESCRIBE products;

Output:
+------------+-----------------+------+-----+---------+---------------+
| Field      | Type            | Null | Key | Default | Extra         |
+------------+-----------------+------+-----+---------+---------------+
| productID  | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
| code       | char(6)         | NO   |     |         |               |
| productname | varchar(30)    | NO   |     |         |               |
| quantity   | int(10) unsigned | NO   |     | 0       |               |
| price      | decimal(5,2)    | NO   |     | 0.00    |               |
| supplierID | int(10) unsigned | NO   | MUL | NULL    |               |
+------------+-----------------+------+-----+---------+---------------+
6 rows in set (0.00 sec)

mysql> SELECT * FROM products;

Output:
+-----------+------+-----------------+----------+--------+------------+
| productID | code | productname     | quantity | price  | supplierID |
+-----------+------+-----------------+----------+--------+------------+
|         1 | IPH  | Iphone 5S Gold  |      300 | 625.00 |        501 |
|         2 | IPH  | Iphone 5S Black |     8000 | 655.25 |        501 |
|         3 | IPH  | Iphone 5S Blue  |     2000 | 625.50 |        501 |
|         4 | SNY  | Xperia Z1       |    10000 | 555.48 |        501 |
|         5 | SNY  | Xperia S        |     8000 | 400.49 |        501 |
+-----------+------+-----------------+----------+--------+------------+
5 rows in set (0.00 sec)
```

```
mysql> UPDATE products SET supplierID = 502 WHERE productID = 1;

Output:
Query OK, 1 row affected (0.13 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT * FROM products;

Output:
+-----------+------+-----------------+----------+--------+------------+
| productID | code | productname     | quantity | price  | supplierID |
+-----------+------+-----------------+----------+--------+------------+
|         1 | IPH  | Iphone 5S Gold  |      300 | 625.00 |        502 |
|         2 | IPH  | Iphone 5S Black |     8000 | 655.25 |        501 |
|         3 | IPH  | Iphone 5S Blue  |     2000 | 625.50 |        501 |
|         4 | SNY  | Xperia Z1       |    10000 | 555.48 |        501 |
|         5 | SNY  | Xperia S        |     8000 | 400.49 |        501 |
+-----------+------+-----------------+----------+--------+------------+
5 rows in set (0.00 sec)
```

4.13 SELECT with JOIN

SELECT command can be used to query and join data from two related tables. For example, to list the product's name (in products table) and supplier's name (in suppliers table), we could join the two tables using the two common supplierID columns:

```
mysql> SELECT products.productname, price, suppliers.name FROM
  products JOIN suppliers ON products.supplierID
  = suppliers.supplierID WHERE price < 650;

Output:
+----------------+--------+-------------+
| productname    | price  | name        |
+----------------+--------+-------------+
| Iphone 5S Gold | 625.00 | XYZ Company |
| Iphone 5S Blue | 625.50 | ABC Traders |
| Xperia Z1      | 555.48 | ABC Traders |
| Xperia S       | 400.49 | ABC Traders |
+----------------+--------+-------------+
4 rows in set (0.00 sec)
```

Here we need to use products.name and suppliers.name to differentiate the two "names."

Join using WHERE clause (legacy method) is not recommended.

```
mysql> SELECT products.productname, price, suppliers.name FROM
  products, suppliers WHERE products.supplierID =
  suppliers.supplierID AND price < 650;
```

```
Output:
+----------------+--------+-------------+
| productname    | price  | name        |
+----------------+--------+-------------+
| Iphone 5S Gold | 625.00 | XYZ Company |
| Iphone 5S Blue | 625.50 | ABC Traders |
| Xperia Z1      | 555.48 | ABC Traders |
| Xperia S       | 400.49 | ABC Traders |
+----------------+--------+-------------+
4 rows in set (0.00 sec)
```

In the above query result, two of the columns have the same heading "name." We could create *aliases* for headings. Let us use aliases for column names for display.

```
mysql> SELECT products.productname AS 'Product Name', price,
  suppliers.name AS 'Supplier Name' FROM products JOIN suppliers
  ON products.supplierID = suppliers.supplierID WHERE price < 650;
```

```
Output:
+----------------+--------+---------------+
| Product Name   | price  | Supplier Name |
+----------------+--------+---------------+
| Iphone 5S Gold | 625.00 | XYZ Company   |
| Iphone 5S Blue | 625.50 | ABC Traders   |
| Xperia Z1      | 555.48 | ABC Traders   |
| Xperia S       | 400.49 | ABC Traders   |
+----------------+--------+---------------+
4 rows in set (0.00 sec)
```

5 Summary

The chapter describes the essential commands for creating, modifying, and querying an RDBMS. Detailed descriptions and examples can be found in the list of books and websites listed in the reference section (Elmasri and Navathe 2014; Hoffer et al. 2011; MySQL using R 2018; MySQL using Python 2018). You can also refer various websites such as w3schools.com/sql, sqlzoo.net (both accessed on Jan 15, 2019), which help you learn SQL in gamified console. The practice would help you learn to query large databases, which is quite a nuisance.

Exercises

Ex. 3.1 Print list of all suppliers who do not keep stock for IPhone 5S Black.
Ex. 3.2 Find out the product that has the biggest inventory by value (i.e., the product that has the highest value in terms of total inventory).
Ex. 3.3 Print the supplier name who maintains the largest inventory of products.

Ex. 3.4 Due to the launch of a newer model, prices of IPhones have gone down and the inventory value has to be written down. Create a new column (new_price) where price is marked down by 20% for all black- and gold-colored phones, whereas it has to be marked down by 30% for the rest of the phones.
Ex. 3.5 Due to this recent markdown in prices (refer to Ex. 3.4), which supplier takes the largest hit in terms of inventory value?

References

Elmasri, R., & Navathe, S. B. (2014). *Database systems: Models, languages, design and application*. England: Pearson.
Hoffer, J. A., Venkataraman, R., & Topi, H. (2011). *Modern database management*. England: Pearson.
MySQL using R. Retrieved February, 2018., from https://cran.r-project.org/web/packages/RMySQL/RMySQL.pdf.
MySQL using Python. Retrieved February, 2018., from http://mysql-python.sourceforge.net/MySQLdb.html.