CHAPTER 1

Introduction
In this chapter, you will learn about the following:

- Relational databases
- Structured Query Language (SQL), which is used to access a database
- SQL*Plus, Oracle’s interactive text-based tool for running SQL statements
- Oracle SQL Developer, which is a graphical tool for database development
- PL/SQL, Oracle’s procedural language that contains programming statements

What Is a Relational Database?

The concept of a relational database was originally developed back in 1970 by Dr. E.F. Codd. He developed the theory of relational databases in a paper entitled “A Relational Model of Data for Large Shared Data Banks,” published in Communications of the Association for Computing Machinery, Vol. 13, No. 6, June 1970.

The basic concepts of a relational database are easy to understand. A relational database is a collection of related information that has been organized into tables. Each table stores data in rows, with the data arranged into columns. The tables are stored in database schemas, which are areas where users can store their own tables. A user can grant permissions to other users so they can access the tables.

Most of us are familiar with data being stored in tables. For example, stock prices and train timetables are sometimes organized into tables. One example table used in this book records the customer information for an imaginary store. The table stores each customer’s first name, last name, date of birth (dob), and phone number:

<table>
<thead>
<tr>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>DOB</th>
<th>PHONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>Brown</td>
<td>01-JAN-1965</td>
<td>800-555-1211</td>
</tr>
<tr>
<td>Cynthia</td>
<td>Green</td>
<td>05-FEB-1968</td>
<td>800-555-1212</td>
</tr>
<tr>
<td>Steve</td>
<td>White</td>
<td>16-MAR-1971</td>
<td>800-555-1213</td>
</tr>
<tr>
<td>Gail</td>
<td>Black</td>
<td></td>
<td>800-555-1214</td>
</tr>
<tr>
<td>Doreen</td>
<td>Blue</td>
<td>20-MAY-1970</td>
<td></td>
</tr>
</tbody>
</table>

This table could be stored in a variety of forms:

- A table in a database
- An HTML file on a web page
- A piece of paper stored in a filing cabinet

An important point to remember is that the information that makes up a database is different from the system used to access that information. The software used to access a database is known as a database management system. Example software includes Oracle Database 12c, Microsoft SQL Server, IBM DB2, and open-source MySQL.

Of course, every database must have some way to get data in and out of it, preferably using a common language understood by all databases. Database management systems implement a
standard language known as Structured Query Language, or SQL. SQL allows you to retrieve, add, modify, and delete information in a database.

**Introducing Structured Query Language (SQL)**

Structured Query Language (SQL) is the standard language designed to access relational databases. SQL should be pronounced as the letters “S-Q-L.”

> **NOTE**
> “S-Q-L” is the correct way to pronounce SQL according to the American National Standards Institute. However, the single word “sequel” is frequently used instead.

SQL is based on the groundbreaking work of Dr. E.F. Codd, with the first implementation of SQL being developed by IBM in the mid-1970s. IBM was conducting a research project known as System R, and SQL was born from that project. Later, in 1979, a company then known as Relational Software, Inc. (known today as Oracle Corporation) released the first commercial version of SQL.

SQL became a standard of the American National Standards Institute (ANSI) in 1986, but there are differences between the implementations of SQL from each software company.

SQL uses a simple syntax that is easy to learn and use. You’ll see some simple examples of its use in this chapter. There are five types of SQL statements, outlined in the following list:

- **Query statements** retrieve rows stored in database tables. You write a query using the SQL SELECT statement.
- **Data Manipulation Language (DML) statements** modify the contents of tables. There are three DML statements:
  - **INSERT** adds rows to a table.
  - **UPDATE** changes rows.
  - **DELETE** removes rows.
- **Data Definition Language (DDL) statements** define the data structures, such as tables, that make up a database. There are five basic types of DDL statements:
  - **CREATE** creates a database structure. For example, CREATE TABLE is used to create a table. Another example is CREATE USER, which is used to create a database user.
  - **ALTER** modifies a database structure. For example, ALTER TABLE is used to modify a table.
  - **DROP** removes a database structure. For example, DROP TABLE is used to remove a table.
  - **RENAME** changes the name of a table.
  - **TRUNCATE** deletes all the rows from a table.
Transaction Control (TC) statements permanently record row changes or undo row changes. There are three TC statements:

- **COMMIT** permanently records the row changes.
- **ROLLBACK** undoes the row changes.
- **SAVEPOINT** sets a “savepoint” to which you can roll back changes.

Data Control Language (DCL) statements change the permissions on database structures. There are two DCL statements:

- **GRANT** gives a user access to a specified database structure.
- **REVOKE** prevents a user from accessing a specified database structure.

Oracle has a program called SQL*Plus that allows you to enter SQL statements and get results back from the database. SQL*Plus also allows you to run scripts containing SQL statements and SQL*Plus commands.

There are other ways to run SQL statements and get results back from the database. For example, Oracle Forms and Oracle Reports allow you to run SQL statements. SQL statements can also be embedded within programs written in programming languages like Java and C#. For details on how to add SQL statements to a Java program, see my book *Oracle9i JDBC Programming* (Oracle Press, 2002). For details on how to add SQL statements to a C# program, see my book *Mastering C# Database Programming* (Sybex, 2003).

**Using SQL*Plus**

In this section, you’ll learn how to start SQL*Plus and run a query.

**Starting SQL*Plus**

If you’re using Windows 7, you can start SQL*Plus by selecting All Programs | Oracle | Application Development | SQL Plus. If you’re using Unix or Linux, you start SQL*Plus by running `sqlplus` from the command prompt.

This illustration shows SQL*Plus running on Windows 7.

The illustration shows the scott user connecting to the database. The scott user is contained in many Oracle database installations. The password for scott in my database is oracle.

The host string after the `@` character tells SQL*Plus which database system to connect to. If you are running the database on your own computer, you’ll typically omit the host string. For example, I could enter `scott/oracle` and omit the `@` character and the `orcl` string. If the host string is omitted, SQL*Plus attempts to connect to a database on the same computer on which SQL*Plus is running. If the database isn’t running on your computer, you should speak with your database administrator (DBA) to get the host string.
Chapter 1: Introduction

If the `scott` user doesn’t exist in your database or is locked, ask your DBA for an alternative user and password. For the examples in the first part of this chapter, you can use any user to connect to the database.

**Starting SQL*Plus from the Command Line**

You can start SQL*Plus from the command line. To do this, you use the `sqlplus` command. The full syntax for the `sqlplus` command is as follows:

```
sqlplus [user_name[/password[@host_string]]]
```

where

- `user_name` specifies the name of the database user.
- `password` specifies the password for the database user.
- `host_string` specifies the database to connect to.

The following examples show `sqlplus` commands:

```
sqlplus scott/oracle
sqlplus scott/oracle@orcl
```

If you’re using SQL*Plus with a Windows operating system, the Oracle installer automatically adds the directory for SQL*Plus to your path. If you’re using Unix or Linux, you can do one of the following to run SQL*Plus:

- Change directories using the `cd` command into the same directory as the `sqlplus` executable, and then run `sqlplus` from that directory.
- Add the directory where `sqlplus` is located to your path, and then run `sqlplus`. If you need help with setting up a directory path, you should speak with your system administrator.

For security, you can hide the password when connecting to the database. For example, you can enter the following command:

```
sqlplus scott@orcl
```

SQL*Plus then prompts you to enter the password. As you type in the password, it is hidden. You can also enter the following command:

```
sqlplus
```

SQL*Plus then prompts you for the user name and password. You can specify the host string by adding it to the user name (for example, `scott@orcl`).

**Performing a SELECT Statement Using SQL*Plus**

Once you’re logged onto the database using SQL*Plus, run the following `SELECT` statement that returns the current date:

```
SELECT SYSDATE FROM dual;
```
In this book, SQL statements shown in bold are statements that you should type in and run if you want to follow along with the examples. Non-bold statements are statements you don’t need to type in.

SYSDATE is a built-in database function that returns the current date. The dual table contains a single dummy row. You’ll learn more about the dual table in the next chapter.

SQL statements are terminated using a semicolon character (;).

The following illustration shows the date returned by the previous SELECT statement.

![SQL Plus output showing the current date](image)

You can edit your last SQL statement in SQL*Plus by entering EDIT. This is useful when you make a mistake or you want to change your SQL statement. On Windows, when you enter EDIT you are taken to the Notepad application. When you exit Notepad and save your statement, the new statement is passed back to SQL*Plus. You can re-execute the statement by entering a forward slash (/). On Linux and Unix, the default editor is ed. To save the statement changes and exit ed, you enter wq.

Resolving the Error When Attempting to Edit Statements

If you encounter error SP2-0110 when trying to edit a statement on Windows, you can run SQL*Plus as an administrator. On Windows 7, you can do that by right-clicking the SQL*Plus shortcut and selecting “Run as administrator.” You can permanently set this in Windows 7 by right-clicking the SQL*Plus shortcut and selecting the “Run this program as an administrator” option in the Compatibility tab.

You can also set the directory that SQL*Plus starts in by right-clicking the SQL*Plus shortcut and changing the “Start in” directory in the Shortcut tab. SQL*Plus will use that default directory when saving and retrieving files. For example, you could set the directory to C:\My_SQL_files, and SQL*Plus will store and retrieve files from that directory by default.
On the Windows version of SQL*Plus, you can scroll through previous commands you’ve run by pressing the UP and DOWN ARROW keys on the keyboard. You’ll learn more about SQL*Plus in Chapter 3.

**Using SQL Developer**

You can also enter SQL statements using SQL Developer. SQL Developer has a graphical user interface (GUI) through which you can enter SQL statements, examine database tables, run scripts, edit and debug PL/SQL code, and perform other tasks. SQL Developer can connect to Oracle Database version 9.2.0.1 and higher. SQL Developer is available for many operating systems. The following illustration shows SQL Developer running.

You must either download the version of SQL Developer that contains the Java Software Development Kit (SDK), or already have the correct version of the Java SDK installed on your computer. The Java version required varies depending on the version of SQL Developer, and you should examine the SQL Developer web page on www.oracle.com for details.

After successfully starting SQL Developer, you will need to create a database connection by right-clicking Connections and selecting New Connection. SQL Developer will display a dialog
8 Oracle Database 12c SQL

in which you specify the database connection details. The following illustration shows an example dialog with the connection details completed.

Once you’ve created a connection and tested it, you can use SQL Developer to examine database tables and run queries. The following illustration shows the columns in a table named customers, which is one of the tables used in this book.
You can also view the data stored in a table by selecting the Data tab, as shown in the following illustration, which shows the rows in the customers table.

In the next section, you’ll learn how to create the store schema used in this book.

Creating the Store Schema
The imaginary store sells items such as books, videos, DVDs, and CDs. The database for the store will hold information about the customers, employees, products, and sales. The SQL*Plus script to create the database is named store_schema.sql, which is located in the SQL directory where you extracted the Zip file for this book. The store_schema.sql script contains the DDL and DML statements that create the schema.

Examining the Script
Open the script in an editor and examine the statements in the script. This section introduces the script statements and guides you through any script changes you might need to make. You’ll learn more about the script statements later in this chapter.
Dropping and Creating the User

The first executable statement in the `store_schema.sql` script is as follows:

```
DROP USER store CASCADE;
```

The `DROP USER` statement is there so that you don’t have to manually drop the store user when re-creating the schema later in this book.

The next statement creates the store user with a password of `store_password`:

```
CREATE USER store IDENTIFIED BY store_password;
```

The next statement allows the store user to connect to the database and create database items:

```
GRANT connect, resource TO store;
```

Allocating Tablespace Storage

The following statement allocates the store user 10 megabytes of space in the `users` tablespace:

```
ALTER USER store QUOTA 10M ON users;
```

Tablespaces are used by the database to store tables and other database objects. You’ll learn more about tablespaces in Chapter 10. Most databases have a `users` tablespace to store user data. To check this, first connect to the database as a privileged user (for example, the `system` user) and run the following statement:

```
SELECT property_value
FROM database_properties
WHERE property_name = 'DEFAULT_PERMANENT_TABLESPACE';
```

```
PROPERTY_VALUE
-------------
USERS
```

This query returns the name of the tablespace to use in the `ALTER USER` statement. In my database, the tablespace is `users`.

If the tablespace returned by the query is different from `users`, then you must replace `users` in the script’s `ALTER USER` statement with the one returned by the previous query. For example, if the name of your tablespace is `another_ts`, then change the script statement to:

```
ALTER USER store QUOTA 10M ON another_ts;
```

Setting the Connection

The following statement in the script connects as the store user:

```
CONNECT store/store_password;
```

You’ll need to modify the `CONNECT` statement in the script if you’re connecting to a database on a different computer. For example, if you’re connecting to a database named `orcl1`, then change the `CONNECT` statement in the script to:

```
CONNECT store/store_password@orcl1;
```
Chapter 1: Introduction

Using the Pluggable Database Feature

Pluggable databases are a new feature in Oracle Database 12c. Pluggable databases are created within an outer container database. Pluggable databases save system resources, simplify system administration, and are typically implemented by database administrators. Therefore, a full discussion of pluggable databases is beyond the scope of this book.

If you’re using the pluggable database feature, then you’ll need to modify the CONNECT statement in the script to include the name of the pluggable database. For example, if your pluggable database name is `pdborcl`, then change the statement to:

```sql
CONNECT store/store_password@pdborcl;
```

If you made any changes to the `store_schema.sql` script, save your modified script. The remaining statements in the script create the tables and other items required for the example store. You’ll learn about those statements later in this chapter.

Running the Script

To create the `store` schema, you perform the following steps:

1. Start SQL*Plus.
2. Log onto the database as a user with the privileges to create new users, tables, and PL/SQL packages. I run scripts in my database using the `system` user. This user has all the required privileges.
3. If you’re using the pluggable database feature, then you must set the session database container to your pluggable database name. For example, if your pluggable database name is `pdborcl`, then run the following command:

   ```sql
   ALTER SESSION SET CONTAINER=pdborcl;
   ```

4. Run the `store_schema.sql` script using the `@` command. The `@` command has the following syntax:

   ```sql
   @ directory\store_schema.sql
   ```

   where `directory` is the directory where your `store_schema.sql` script is located.

   For example, if your script is located in `C:\sql_book\SQL`, then you enter the following command:

   ```sql
   @ C:\sql_book\SQL\store_schema.sql
   ```

   If your script is located in a directory that contains spaces, then you must place the directory and script in quotes after the `@` command. For example:

   ```sql
   @ "C:\Oracle SQL book\sql_book\SQL\store_schema.sql"
   ```

   If you’re using Unix or Linux and the script is in a directory named `SQL` located in `tmp`, then you enter the following command:

   ```sql
   @ /tmp/SQL/store_schema.sql
   ```
NOTE
Windows uses backslash characters (\) in the directory path. Unix and Linux use forward slash characters (/).

When the script has finished running, you’ll be connected as the store user. The user has a password of store_password.
You’ll be asked to run other scripts later in this book. You’ll need to perform the steps described in this section before running each script:

■ If your database does not have a users tablespace, then you’ll need to edit the ALTER USER statement in the script.
■ If you need to set a host string to connect to a database, then you’ll need to edit the CONNECT statement in the script.
■ If you’re using the pluggable database feature, then you’ll need to edit the CONNECT statement in the script and run the ALTER SESSION SET CONTAINER command prior to running the script.

You don’t have to edit all of the scripts now. Just remember that you might have to edit each script before running it.

Examining the Store Data Definition Language Statements
Data Definition Language (DDL) statements are used to create users and tables, plus many other types of structures in the database. In this section, you’ll see the DDL statements used to create the store user and some of the tables.

NOTE
The SQL statements you’ll see in the rest of this chapter are the same as those contained in the store_schema.sql script. You don’t have to type in the statements yourself.

The next sections describe the following:

■ How to create a database user
■ The data types commonly used in an Oracle database
■ Some of the tables in the imaginary store

Creating a Database User
To create a user in the database, you use the CREATE USER statement. The simplified syntax for the CREATE USER statement is as follows:

```
CREATE USER user_name IDENTIFIED BY password;
```

where

■ user_name is the user name
■ password is the password for the user
For example, the following `CREATE USER` statement creates the `store` user with a password of `store_password`:

```
CREATE USER store IDENTIFIED BY store_password;
```

If you want the user to be able to work in the database, the user must be granted the necessary permissions to do that work. In the case of `store`, this user must be able to log onto the database (which requires the `connect` permission) and create items like database tables (which requires the `resource` permission). Permissions are granted by a privileged user (for example, the `system` user) using the `GRANT` statement.

The following example grants the `connect` and `resource` permissions to `store`:

```
GRANT connect, resource TO store;
```

Many of the examples in this book use the `store` schema. Before I get into the details of the tables in the `store` schema, you need to know about the commonly used Oracle database types.

### The Common Oracle Database Types

There are many types that can be used to handle data in an Oracle database. Some of the commonly used types are shown in Table 1-1. You can see all of the data types in the appendix of this book.

<table>
<thead>
<tr>
<th>Oracle Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR(<code>length</code>)</td>
<td><code>CHAR</code> stores a fixed-length string. The <code>length</code> parameter specifies the length of the string. If a string of a smaller length is stored, then the string is padded with spaces at the end. For example, <code>CHAR(2)</code> can be used to store a fixed-length string of two characters. If 'C' is stored in a <code>CHAR(2)</code>, then a single space is added at the end. 'CA' is stored as is, with no padding necessary.</td>
</tr>
<tr>
<td>VARCHAR2(<code>length</code>)</td>
<td><code>VARCHAR2</code> stores a variable-length string. The <code>length</code> parameter specifies the maximum length of the string. For example, <code>VARCHAR2(20)</code> can be used to store a string of up to 20 characters in length. No padding is added at the end of a smaller string.</td>
</tr>
<tr>
<td>DATE</td>
<td><code>DATE</code> stores a date and time. A <code>DATE</code> stores the century, all four digits of a year, the month, the day, the hour (in 24-hour format), the minute, and the second. A <code>DATE</code> can store a date between January 1, 4712 B.C. and December 31, 9999 A.D.</td>
</tr>
<tr>
<td>INTEGER</td>
<td><code>INTEGER</code> stores an integer. An integer is a whole number, such as 1, 10, and 115. An integer doesn't contain a floating point.</td>
</tr>
</tbody>
</table>
The following table shows some NUMBER examples.

<table>
<thead>
<tr>
<th>Format</th>
<th>Number Supplied</th>
<th>Number Stored</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMBER</td>
<td>1234.567</td>
<td>1234.567</td>
</tr>
<tr>
<td>NUMBER(6, 2)</td>
<td>123.4567</td>
<td>123.46</td>
</tr>
<tr>
<td>NUMBER(6, 2)</td>
<td>12345.67</td>
<td>Number exceeds the specified precision and is therefore rejected by the database</td>
</tr>
</tbody>
</table>

### Examining the Store Tables

In this section, you’ll learn how the tables for the store schema are created. Some of the information held in the store schema includes the following:

- Customer details
- Types of products sold
- Product details
- A history of the products purchased by the customers
- Employees of the store
- Salary grades

The following tables are used to hold the information:

- **customers** holds the customer details
- **product_types** holds the types of products sold by the store
- **products** holds the product details
- **purchases** holds which products were purchased by which customers
- **employees** holds the employee details
- **salary_grades** holds the employee salary grades
In the following sections, you’ll see the CREATE TABLE statements in the store_schema.sql script that create the tables.

The customers Table  The customers table holds the details of the customers. The following items are held in this table:

- First name
- Last name
- Date of birth (dob)
- Phone number

Each of these items requires a column in the customers table. The customers table is created by the store_schema.sql script using the following CREATE TABLE statement:

```sql
CREATE TABLE customers (
  customer_id INTEGER CONSTRAINT customers_pk PRIMARY KEY,
  first_name VARCHAR2(10) NOT NULL,
  last_name VARCHAR2(10) NOT NULL,
  dob DATE,
  phone VARCHAR2(12)
);  
```

As you can see, the customers table contains five columns, one for each item in the previous list, and an extra column named customer_id. The columns are as follows:

- **customer_id**  Contains a unique integer for each row in the table. Each table should have a column that uniquely identifies each row. The column is known as the primary key. A primary key can consist of multiple columns. The CONSTRAINT clause in the CREATE TABLE statement indicates that the customer_id column is the primary key. A CONSTRAINT clause restricts the values stored in a column, and, for the customer_id column, the PRIMARY KEY keywords indicate that the customer_id column must contain a unique value for each row. You can also attach an optional name to a constraint, which must immediately follow the CONSTRAINT keyword—for example, customers_pk. You should always name your primary key constraints so that if a constraint error occurs, you can easily find where it happened.

- **first_name**  Contains the first name of the customer. This column is marked as NOT NULL, which means that a value must be supplied for first_name when adding a row. If a NOT NULL constraint is omitted, a user doesn’t need to supply a value.

- **last_name**  Contains the last name of the customer. This column is NOT NULL, and therefore a value must be supplied when adding a row.

- **dob**  Contains the date of birth for the customer. No NOT NULL constraint is specified for this column. Therefore, the default NULL is assumed, and a value is optional when adding a row.

- **phone**  Contains the phone number of the customer. No NOT NULL constraint is specified for this column.
The store_schema.sql script populates the customers table with the following rows:

<table>
<thead>
<tr>
<th>CUSTOMER_ID</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>DOB</th>
<th>PHONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John</td>
<td>Brown</td>
<td>01-JAN-65</td>
<td>800-555-1211</td>
</tr>
<tr>
<td>2</td>
<td>Cynthia</td>
<td>Green</td>
<td>05-FEB-68</td>
<td>800-555-1212</td>
</tr>
<tr>
<td>3</td>
<td>Steve</td>
<td>White</td>
<td>16-MAR-71</td>
<td>800-555-1213</td>
</tr>
<tr>
<td>4</td>
<td>Gail</td>
<td>Black</td>
<td></td>
<td>800-555-1214</td>
</tr>
<tr>
<td>5</td>
<td>Doreen</td>
<td>Blue</td>
<td>20-MAY-70</td>
<td></td>
</tr>
</tbody>
</table>

Notice that customer #4’s date of birth is null, and so is customer #5’s phone number.

You can see the rows in the customers table for yourself by executing the following SELECT statement using SQL*Plus:

```
SELECT * FROM customers;
```

The asterisk (*) indicates that you want to retrieve all the columns from the customers table.

**The product_types Table**

The product_types table holds the names of the product types sold by the store. This table is created by the store_schema.sql script using the following CREATE TABLE statement:

```
CREATE TABLE product_types (
    product_type_id INTEGER CONSTRAINT product_types_pk PRIMARY KEY,
    name VARCHAR2(10) NOT NULL
);
```

The product_types table contains the following two columns:

- **product_type_id** uniquely identifies each row in the table. The product_type_id column is the primary key for this table. Each row in the product_types table must have a unique integer value for the product_type_id column.
- **name** contains the product type name. It is a NOT NULL column, and therefore a value must be supplied when adding a row.

The store_schema.sql script populates the product_types table with the following rows:

<table>
<thead>
<tr>
<th>PRODUCT_TYPE_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Book</td>
</tr>
<tr>
<td>2</td>
<td>Video</td>
</tr>
<tr>
<td>3</td>
<td>DVD</td>
</tr>
<tr>
<td>4</td>
<td>CD</td>
</tr>
<tr>
<td>5</td>
<td>Magazine</td>
</tr>
</tbody>
</table>

The product_types table holds the names of the product types for the store. Each product sold by the store must be a valid product type.

You can see the rows in the product_types table by executing the following SELECT statement using SQL*Plus:

```
SELECT * FROM product_types;
```
The products Table  The products table holds the products sold by the store. The following information is held for each product:

- Product type
- Name
- Description
- Price

The store_schema.sql script creates the products table using the following CREATE TABLE statement:

```
CREATE TABLE products (  
  product_id INTEGER CONSTRAINT products_pk PRIMARY KEY,  
  product_type_id INTEGER  
    CONSTRAINT products_fk_product_types  
    REFERENCES product_types(product_type_id),  
  name VARCHAR2(30) NOT NULL,  
  description VARCHAR2(50),  
  price NUMBER(5, 2)  
);  
```

The columns in the products table are as follows:

- **product_id** uniquely identifies each row in the table. This column is the primary key of the table.
- **product_type_id** associates each product with a product type. This column is a reference to the product_type_id column in the product_types table. This column is known as a foreign key because it references a column in another table. The table containing the foreign key (the products table) is known as the detail or child table, and the table that is referenced (the product_types table) is known as the master or parent table. This type of relationship is known as a master-detail or parent-child relationship. When you add a new product, you associate that product with a type by supplying a matching product_types.product_type_id value in the products.product_type_id column (you’ll see an example later).
- **name** contains the product name. The column is NOT NULL.
- **description** contains an optional description of the product.
- **price** contains an optional price for a product. This column is defined as NUMBER (5, 2). The precision is 5, and therefore a maximum of five digits can be supplied for this number. The scale is 2; therefore, two of those maximum five digits can be to the right of a decimal point.

The following shows the first four rows stored in the products table:

<table>
<thead>
<tr>
<th>PRODUCT_ID</th>
<th>PRODUCT_TYPE_ID</th>
<th>NAME</th>
<th>DESCRIPTION</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 Modern</td>
<td>Science</td>
<td>A description of modern science</td>
<td>19.95</td>
</tr>
</tbody>
</table>
The first row in the products table has a product_type_id of 1, which means the product is a book (this product_type_id matches the “book” product type in the product_types table). The second product is also a book. The third and fourth products are videos (their product_type_id is 2, which matches the “video” product type in the product_types table).

You can see all the rows in the products table by executing the following SELECT statement using SQL*Plus:

```sql
SELECT * FROM products;
```

**The purchases Table** The purchases table holds the purchases made by a customer. For each purchase made by a customer, the following information is held:

- Product ID
- Customer ID
- Number of units of the product that were purchased by the customer

The store_schema.sql script uses the following CREATE TABLE statement to create the purchases table:

```sql
CREATE TABLE purchases (  
  product_id INTEGER  
    CONSTRAINT purchases_fk_products  
    REFERENCES products(product_id),  
  customer_id INTEGER  
    CONSTRAINT purchases_fk_customers  
    REFERENCES customers(customer_id),  
  quantity INTEGER NOT NULL,  
  CONSTRAINT purchases_pk PRIMARY KEY (product_id, customer_id)  
);
```

The columns in this table are as follows:

- **product_id** contains the ID of the product that was purchased. This must match a product_id column value in the products table.
- **customer_id** contains the ID of a customer who made the purchase. This must match a customer_id column value in the customers table.
- **quantity** contains the number of units of the product that were purchased by the customer.

The purchases table has a primary key constraint named purchases_pk that consists of two columns: product_id and customer_id. The combination of the two column values must
be unique for each row. When a primary key consists of multiple columns, it is known as a composite primary key.

The following shows the first five rows that are stored in the purchases table:

<table>
<thead>
<tr>
<th>PRODUCT_ID</th>
<th>CUSTOMER_ID</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

As you can see, the combination of the values in the product_id and customer_id columns is unique for each row.

You can see all the rows in the purchases table for yourself by executing the following SELECT statement using SQL*Plus:

```
SELECT * FROM purchases;
```

The employees Table  The employees table holds the details of the employees. The following information is held in the table:

- Employee ID
- The ID of the employee's manager (if applicable)
- First name
- Last name
- Title
- Salary

The store_schema.sql script uses the following CREATE TABLE statement to create the employees table:

```
CREATE TABLE employees ( 
  employee_id INTEGER CONSTRAINT employees_pk PRIMARY KEY, 
  manager_id INTEGER, 
  first_name VARCHAR2(10) NOT NULL, 
  last_name VARCHAR2(10) NOT NULL, 
  title VARCHAR2(20), 
  salary NUMBER(6, 0) 
);
```

The store_schema.sql script populates the employees table with the following rows:

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>MANAGER_ID</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>TITLE</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>James</td>
<td>Smith</td>
<td>CEO</td>
<td>800000</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Ron</td>
<td>Johnson</td>
<td>Sales Manager</td>
<td>600000</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>Fred</td>
<td>Hobbs</td>
<td>Salesperson</td>
<td>150000</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>Susan</td>
<td>Jones</td>
<td>Salesperson</td>
<td>500000</td>
</tr>
</tbody>
</table>

James Smith is the CEO and he doesn't have a manager.
The salary_grades Table  The salary_grades table holds the different salary grades available to employees. The following information is held:

- Salary grade ID
- Low salary boundary for the grade
- High salary boundary for the grade

The store_schema.sql script uses the following CREATE TABLE statement to create the salary_grades table:

```sql
CREATE TABLE salary_grades (  
salary_grade_id INTEGER CONSTRAINT salary_grade_pk PRIMARY KEY,  
low_salary  NUMBER(6, 0),  
high_salary NUMBER(6, 0)  
);
```

The store_schema.sql script populates the salary_grades table with the following rows:

```
SALARY_GRADE_ID LOW_SALARY HIGH_SALARY  
--------------- ---------- -----------  
1 1 250000  
2 250001 500000  
3 500001 750000  
4 750001 999999  
```

Adding, Modifying, and Removing Rows
In this section, you’ll learn how to add, modify, and remove rows in database tables by using the SQL INSERT, UPDATE, and DELETE statements. You can make your row changes permanent in the database by using the COMMIT statement. You can undo your row changes by using the ROLLBACK statement. This section doesn’t exhaustively cover all the details of using these statements (you’ll learn more about them in Chapter 9).

Adding a Row to a Table
You use the INSERT statement to add new rows to a table. You can specify the following information in an INSERT statement:

- The table into which the row is to be inserted
- A list of columns for which you want to specify column values
- A list of values to store in the specified columns

When adding a row, you need to supply a value for the primary key and all other columns that are defined as NOT NULL. You don’t have to specify values for the other columns. Those columns will be automatically set to null if you omit values for them.
You can tell which columns are defined as NOT NULL using the SQL*Plus DESCRIBE command. For example:

```
SQL> DESCRIBE customers
Name                    Null?    Type
------------------------ -------- ------------
CUSTOMER_ID              NOT NULL NUMBER(38)
FIRST_NAME               NOT NULL VARCHAR2(10)
LAST_NAME                NOT NULL VARCHAR2(10)
DOB                      DATE
PHONE                    VARCHAR2(12)
```

The `customer_id`, `first_name`, and `last_name` columns are NOT NULL, meaning that you must supply a value for these columns when adding a row. The `dob` and `phone` columns don't require a value. You could omit those values if you wanted, and they would be automatically set to null.

Go ahead and run the following INSERT statement, which adds a row to the `customers` table. Notice that the order of values in the VALUES list matches the order in which the columns are specified in the column list.

```
SQL> INSERT INTO customers (customer_id, first_name, last_name, dob, phone)
VALUES (6, 'Fred', 'Brown', '01-JAN-1970', '800-555-1215');
1 row created.
```

**NOTE**

SQL*Plus automatically numbers lines after you press enter at the end of each line.

In the previous example, SQL*Plus responds that one row has been created after the INSERT statement is executed. You can verify this by running the following SELECT statement:

```
SELECT *
FROM customers;
```

```
CUSTOMER_ID FIRST_NAME LAST_NAME  DOB       PHONE
----------- ---------- ---------- --------- ------------
1 John       Brown      01-JAN-65 800-555-1211
2 Cynthia    Green      05-FEB-68 800-555-1212
3 Steve      White      16-MAR-71 800-555-1213
4 Gail       Black                800-555-1214
5 Doreen     Blue       20-MAY-70
6 Fred       Brown      01-JAN-70 800-555-1215
```

Notice the new row shown at the end.
Oracle Database 12c SQL

By default, the Oracle database displays dates in the format DD-MON-YY, where DD is the day number, MON is the first three characters of the month (in uppercase), and YY is the last two digits of the year. The database actually stores all four digits for the year, but by default, it only displays the last two digits.

When a row is added to the customers table, a unique value for the customer_id column must be given. The Oracle database will prevent you from adding a row with a primary key value that already exists in the table. For example, the following INSERT statement generates an error because a row with a customer_id of 1 already exists:

```sql
SQL> INSERT INTO customers (customer_id, first_name, last_name, dob, phone)
2 VALUES (1, 'Lisa', 'Jones', '02-JAN-1971', '800-555-1225');
```

```
ERROR at line 1:
ORA-00001: unique constraint (STORE.CUSTOMERS_PK) violated
```

Notice that the name of the constraint is shown in the error (STORE.CUSTOMERS_PK). That's why you should always name your primary key constraints. Otherwise, the Oracle database assigns an unfriendly system-generated name to a constraint, which makes it difficult to locate the problem (for example, SYS_C0011277).

### Modifying an Existing Row in a Table

You use the UPDATE statement to change rows in a table. Typically, when you use the UPDATE statement, you specify the following information:

- The table containing the rows that are to be changed
- A WHERE clause that specifies the rows that are to be changed
- A list of column names, along with their new values, which are specified using the SET clause

You can change one or more rows using the same UPDATE statement. If more than one row is specified, then the same change will be made for all the rows. The following example updates customer #2’s last_name to Orange:

```sql
UPDATE customers
SET last_name = 'Orange'
WHERE customer_id = 2;
```

```
1 row updated.
```

SQL*Plus confirms that one row was updated.
CAUTION
If you forget to add a WHERE clause, then all the rows will be updated.

The following query shows the updated row:

```
SELECT *
FROM customers
WHERE customer_id = 2;
```

<table>
<thead>
<tr>
<th>CUSTOMER_ID</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>DOB</th>
<th>PHONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Cynthia</td>
<td>Orange</td>
<td>05-FEB-68</td>
<td>800-555-1212</td>
</tr>
</tbody>
</table>

Removing a Row from a Table

You use the DELETE statement to remove rows from a table. You typically use a WHERE clause to limit the rows you wish to delete. If you don’t, then all of the rows will be deleted from the table.

The following DELETE statement removes customer #6:

```
DELETE FROM customers
WHERE customer_id = 6;
```

1 row deleted.

To undo the changes made to the rows, you use ROLLBACK:

```
ROLLBACK;
```

Rollback complete.

NOTE
You can make changes to rows permanent using COMMIT. You’ll see how to do that in Chapter 9.

Connecting to and Disconnecting from a Database

While you’re connected to the database, SQL*Plus maintains a database session for you. When you disconnect from the database, your session is ended. You can disconnect from the database and keep SQL*Plus running by entering DISCONNECT:

```
DISCONNECT
```

By default, when you disconnect, a COMMIT is automatically performed for you.

You can reconnect to a database by entering CONNECT. To reconnect to the store schema, you enter store as the user name with a password of store_password:

```
CONNECT store/store_password
```
Quitting SQL*Plus

You use the EXIT command to quit SQL*Plus. The following example quits SQL*Plus using the EXIT command:

```sql
EXIT
```

By default, when you quit SQL*Plus using EXIT, a COMMIT is automatically performed. If SQL*Plus terminates abnormally—for example, if the computer on which SQL*Plus is running crashes—then a ROLLBACK is automatically performed. You’ll learn more about COMMIT and ROLLBACK in Chapter 9.

Introducing Oracle PL/SQL

PL/SQL is Oracle’s procedural language. PL/SQL allows you to add programming constructs around SQL statements. PL/SQL is primarily used for creating procedures and functions in a database that contains business logic. PL/SQL contains standard programming constructs such as the following:

- Variable declarations
- Conditional logic (if-then-else)
- Loops
- Procedures and functions

The following CREATE PROCEDURE statement creates a procedure named update_product_price(). The procedure multiplies the price of a specified product by a supplied factor. If the specified product doesn’t exist, then the procedure takes no action. Otherwise, the procedure updates the specified product’s price.

```
CREATE PROCEDURE update_product_price(
    p_product_id IN products.product_id%TYPE,
    p_factor     IN NUMBER
) AS
    v_product_count INTEGER;
BEGIN
    -- count the number of products with the
    -- supplied product_id (will be 1 if the product exists)
    SELECT COUNT(*)
    INTO v_product_count
    FROM products
    WHERE product_id = p_product_id;
```

**NOTE**

Don’t worry about the details of the PL/SQL shown in the following listing. You’ll learn all about PL/SQL in Chapter 12. I just want you to get a feel for PL/SQL at this stage.
-- if the product exists (v_product_count = 1) then
-- update that product's price
IF v_product_count = 1 THEN
    UPDATE products
    SET price = price * p_factor
    WHERE product_id = p_product_id;
    COMMIT;
END IF;
EXCEPTION
    WHEN OTHERS THEN
    ROLLBACK;
END update_product_price;
/

Exceptions are used to handle errors that occur in PL/SQL code. The EXCEPTION block in the previous example performs a ROLLBACK if an exception is thrown in the code.

Summary
In this chapter, you have learned the following:

- A relational database is a collection of related information organized into tables.
- Structured Query Language (SQL) is the standard language for accessing databases.
- SQL*Plus allows you to run SQL statements and SQL*Plus commands.
- SQL Developer is a graphical tool for database development.
- PL/SQL is Oracle's procedural language that contains programming statements.

In the next chapter, you'll learn more about retrieving information from database tables.