



CHAPTER 1

Introduction

2 Oracle Database 10g SQL

In this chapter, you will

- Learn about relational databases.
- Be introduced to the Structured Query Language (SQL), which is used to access a database.
- Use SQL*Plus, Oracle's interactive text-based tool for running SQL statements.
- Briefly see PL/SQL, Oracle's procedural programming language built around SQL. PL/SQL allows you to develop programs that are stored in the database.

Let's plunge in and consider what a relational database is.

What Is a Relational Database?

The concept of a relational database is not new. It was originally developed back in 1970 by Dr. E.F. Codd. He laid down the theory of relational databases in his seminal paper entitled "A Relational Model of Data for Large Shared Data Banks" published in *Communications of the ACM* (Association for Computing Machinery), Vol. 13, No. 6, June 1970.

The basic concepts of a relational database are fairly easy to understand. A *relational database* is a collection of related information that has been organized into structures known as *tables*. Each table contains *rows* that are further organized into *columns*. These tables are stored in the database in structures known as *schemas*, which are areas where database users may store their tables. Each user may also choose to grant permissions to other users to access their tables.

Most of us are familiar with data being stored in tables—stock prices and train timetables are sometimes organized into tables. An example used in one of the schemas in this book is a table that records customer information for a hypothetical store. Part of this table consists of columns containing the customer's first name, last name, date of birth (dob), and phone number:

first_name	last_name	dob	phone
John	Brown	01-JAN-1965	800-555-1211
Cynthia	Green	05-FEB-1968	800-555-1212
Steve	White	16-MAR-1971	800-555-1213
Gail	Black		800-555-1214
Doreen	Blue	20-MAY-1970	

This table could be stored in a variety of forms: a piece of paper in a filing cabinet or ledger or in the file system of a computer, for example. An important point to note is that the *information* that makes up a database (in the form of tables) is different from the system used to access that information. The system used to access a database is known as a *database management system*.

In the case of a database consisting of pieces of paper, the database management system might be a set of alphabetically indexed cards in a filing cabinet. For a database accessed using a computer, the database management system is the software that manages the files stored in the file system of the computer. The Oracle database is one such piece of software; other examples include SQL Server, DB2, and 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. Today's database management systems implement a standard language known as *Structured Query Language*, or SQL. Among other things, SQL allows you to retrieve, add, update, and delete information in a database.

Introducing the Structured Query Language (SQL)

Structured Query Language (SQL) is the standard language designed to access relational databases. SQL is pronounced either as the word “sequel” or as the letters “S-Q-L.” (I prefer “sequel” as it’s quicker to say.)

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 is now fully standardized and recognized by the American National Standards Institute (ANSI). You can use SQL to access an Oracle, SQL Server, DB2, or MySQL database.

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** Allow you to retrieve rows stored in database tables. You write a query using the SQL `SELECT` statement.
- **Data Manipulation Language (DML) statements** Allow you to modify the contents of tables. There are three DML statements:
 - **INSERT** Allows you to add rows to a table.
 - **UPDATE** Allows you to change a row.
 - **DELETE** Allows you to remove rows.
- **Data Definition Language (DDL) statements** Allow you to define the data structures, such as tables, that make up a database. There are five basic types of DDL statements:
 - **CREATE** Allows you to create 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** Allows you to modify a database structure. For example, `ALTER TABLE` is used to modify a table.
 - **DROP** Allows you to remove a database structure. For example, `DROP TABLE` is used to remove a table.
 - **RENAME** Allows you to change the name of a table.
 - **TRUNCATE** Allows you to delete the entire contents of a table.

4 Oracle Database 10g SQL

- **Transaction Control (TC) statements** Allow you to permanently record the changes made to the rows stored in a table or undo those changes. There are three TC statements:
 - **COMMIT** Allows you to permanently record changes made to rows.
 - **ROLLBACK** Allows you to undo changes made to rows.
 - **SAVEPOINT** Allows you to set a “savepoint” to which you can roll back changes made to rows.
- **Data Control Language (DCL) statements** Allow you to change the permissions on database structures. There are two DCL statements:
 - **GRANT** Allows you to give another user access to your database structures, such as tables.
 - **REVOKE** Allows you to prevent another user from accessing to your database structures, such as tables.

There are many ways to run SQL statements and get results back from the database, some of which include programs written using Oracle Forms and Reports. SQL statements may also be embedded within programs written in other languages, such as Oracle’s Pro*C, which allows you to add SQL statements to a C program. You can also add SQL statements to a Java program though JDBC; for more details see my book *Oracle9i JDBC Programming* (Oracle Press, 2002).

Oracle also has a tool called SQL*Plus that allows you to enter SQL statements using the keyboard or to supply a file that contains SQL statements and run those statements. SQL*Plus enables you to conduct a “conversation” with the database because you can enter SQL statements and view the results returned by the database. You’ll be introduced to SQL*Plus in the next section.

Using SQL*Plus

There are two versions of SQL*Plus: the Windows version and the command-line version. You may use the command-line version of SQL*Plus with any operating system on which the Oracle database runs. If you’re at all familiar with the Oracle database, chances are that you’re already familiar with SQL*Plus. If you’re not, don’t worry: you’ll learn how to use SQL*Plus in this book.

In the next two sections, you’ll learn how to start each version of SQL*Plus, beginning with the Windows version. After you’ve learned how to start SQL*Plus, you’ll see how to run a query against the database.

Starting the Windows Version of SQL*Plus

If you are using Windows, you may start SQL*Plus by clicking Start and selecting Programs | Oracle | Application Development | SQL*Plus. Figure 1-1 shows the Log On dialog box for SQL*Plus running on Windows. Enter **scott** for the user name and **tiger** for the password (scott is an example user that is contained in most Oracle databases). The host string is used to tell SQL*Plus where the database is running. If you are running the database on your own computer, you’ll typically leave the host string blank—this causes SQL*Plus to attempt to connect to a database on the same machine on which SQL*Plus is running. If the database isn’t running on your machine, you should speak with your database administrator (DBA). Click OK to continue.

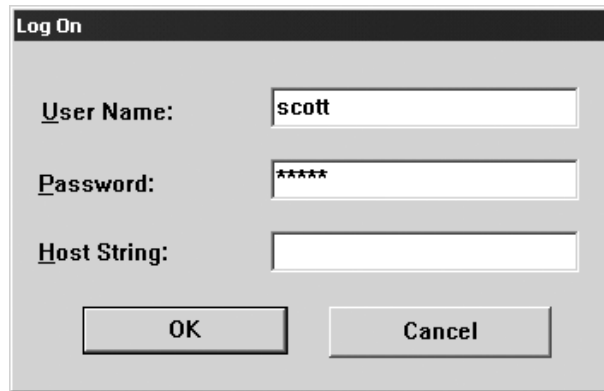


FIGURE 1-1. The SQL*Plus Log On dialog box

NOTE

If you can't log on using "scott" and "tiger," speak with your DBA. They'll be able to provide you with a `user_name`, `password`, and `host_string` for the purposes of this example.

After you've clicked OK and successfully logged on to the database, you'll see the SQL*Plus window through which you can interact with the database. Figure 1-2 shows the SQL*Plus window.

Starting the Command-Line Version of SQL*Plus

To start the command-line version of SQL*Plus, you may use the `sqlplus` command. The full syntax for the `sqlplus` command is

```
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 you want to connect to

6 Oracle Database 10g SQL

The following are examples of issuing the `sqlplus` command:

```
sqlplus scott/tiger  
sqlplus scott/tiger@orcl
```

NOTE

*If you are using SQL*Plus with the Windows operating system, the Oracle installer automatically adds SQL*Plus to your path. If you are using a non-Windows operating system, you must either be in the same directory as the SQL*Plus program to run it or, better still, have added the program to your path. If you need help with that, talk to your system administrator.*

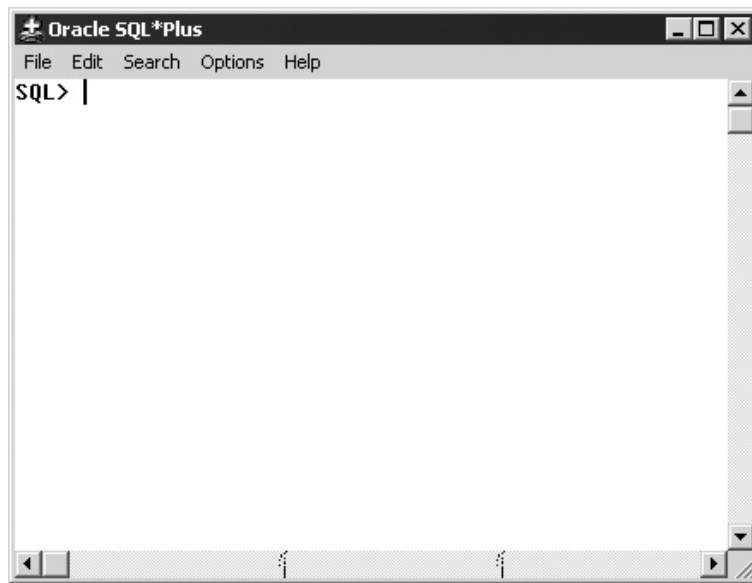


FIGURE 1-2. *The SQL*Plus window*

Performing a SELECT Statement Using SQL*Plus

Once you're logged on to the database using SQL*Plus, try entering the following `SELECT` statement that returns the current date from the database:

```
SELECT SYSDATE FROM dual;
```

`SYSDATE` is a built-in Oracle function that returns the current date, and the `dual` table is a built-in table that contains a single row. You can use the `dual` table to perform simple queries whose results are not retrieved from a specific table.

NOTE

*SQL statements directly entered into SQL*Plus are terminated using a semicolon character (;).*

Figure 1-3 shows the results of this `SELECT` statement in SQL*Plus running on Windows.

As you can see from the previous figure, the result of the query displays the current date from the database.

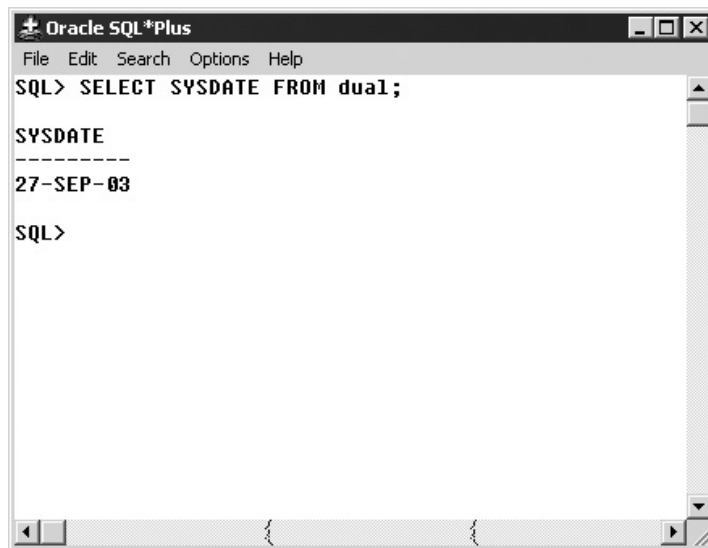


FIGURE 1-3. Executing a SQL `SELECT` statement using SQL*Plus

8 Oracle Database 10g SQL

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 make a change to your SQL statement. In Windows, when you enter `EDIT` you are taken to the Notepad application; you then use Notepad to edit your SQL statement. When you exit Notepad and save your statement, the statement is passed to SQL*Plus where you can re-execute it.

NOTE

*You'll learn more about editing SQL statements using SQL*Plus in Chapter 5.*

The SQL*Plus Worksheet

You can also enter SQL statements using the SQL*Plus worksheet, which has an improved user interface. If you are using Windows, you can start SQL*Plus by clicking Start and selecting Programs | Oracle | Application Development | SQL*Plus Worksheet. Figure 1-4 shows the SQL*Plus Worksheet window once you've logged on to the database. If you have SQL*Plus Worksheet installed, go ahead and log on to the database as the `scott` user, enter the `SELECT SYSDATE FROM dual`, query, and select Execute from the Worksheet menu.

TIP

You can also execute a statement by clicking the Execute button (it has a lightning bolt on it). You can also press F5 on your keyboard to execute a statement.

Figure 1-4 shows the result of running the query that retrieves the current date. Notice that the top part of the window shows the SQL statement executed and the lower part shows the result of the executed statement.

In the next section, you'll learn how to create a fictional store database schema.

Creating the Store Schema

Most of the examples in this book will use an example database schema that will be used to hold information about the customers, inventory, and sales of a simple store. This example store sells items such as books, videos, DVDs, and CDs. This schema will be named `store`, the definition of which is contained in the SQL*Plus script `store_schema.sql`, which is contained in the Zip file you can download from this book's web site. The `store_schema.sql` script contains the DDL and DML statements to create the `store` schema. Once you've obtained the script, you may run it using SQL*Plus or have your DBA run it for you. You'll now learn how to run the `store_schema.sql` script.

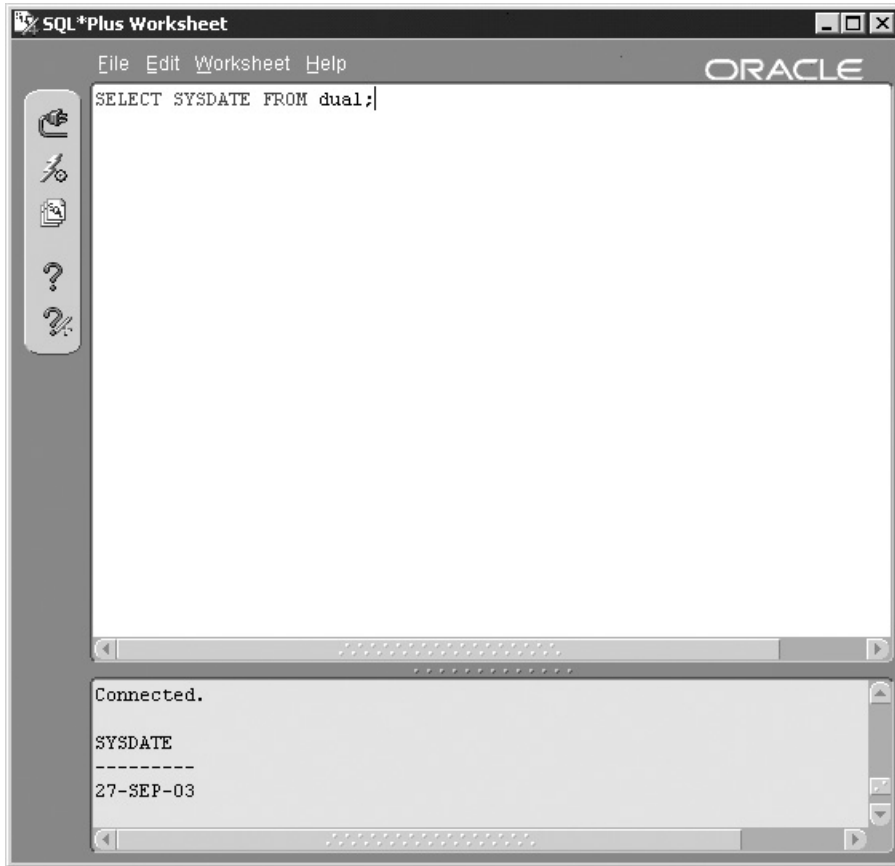


FIGURE 1-4. Executing a SQL *SELECT* statement using the SQL*Plus worksheet

Running the SQL*Plus Script to Create the Store Schema


Perform the following steps:

1. Open the `store_schema.sql` script using an editor and change the password for the `system` user if necessary. The `system` user has privileges to create new users and tables, among other items, and has a default password of `manager`. If that is not the correct password for the `system` user, ask your DBA for the correct password (or just have your DBA run the script for you).

10 Oracle Database 10g SQL

2. Start the SQL*Plus tool.
3. Run the `store_schema.sql` script from within SQL*Plus using the `@` command.

The `@` command has the following syntax:


```
 directory_path\store_schema.sql
```

where *directory_path* is the directory and path where your `store_schema.sql` script is stored.

For example, if the script is stored in a directory named SQL on the C partition of your Windows file system, then you would enter

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

If you're using Unix (or Linux), and you saved the script in a directory named SQL on your tmp file system, for example, you would enter

```
 @/tmp/SQL/store_schema.sql
```

NOTE

Windows uses backslash characters (\) in directory paths, whereas Unix and Linux use forward slash characters (/).

When the `store_schema.sql` script has finished running, you'll be connected as the `store` user. If you want to, open the `store_schema.sql` script using a text editor like Windows Notepad and examine the statements contained in it. Don't worry too much about the details of the statements contained in this file—you'll learn the details as you progress through this book.

NOTE

*To end SQL*Plus, you enter **EXIT**. To reconnect to the `store` schema in SQL*Plus, you enter **store** as the user name with a password of **store_password**. 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**. You can then reconnect to a database by entering **CONNECT**.*

Data Definition Language (DDL) Statements Used to Create the Store Schema

As mentioned earlier, *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 learn how to use DDL statements to create the database user and tables for the `store` schema.

**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 the statements in yourself, just run the `store_schema.sql` script as described earlier.

The following sections describe how to create a database user, followed by the commonly used data types used in the Oracle database, and finally the various tables used for the hypothetical 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` specifies the name you assign to your database user
- `password` specifies the password for your database 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;
```

Next, 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`, the user must be able to log on to 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 (the `DBA`, for example) using the `GRANT` statement.

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

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

Once a user has been created, the database tables and other database objects can be created in the associated schema for that user. For most of the examples in this book, I've chosen to implement a simple store; these tables will be created in the schema of `store`. Before I get into the details of the tables required for the store, you need to understand a little bit about the commonly used Oracle database types that are used to define the database columns.

Understanding the Common Oracle Database Types

There are many types that may be used to handle data in an Oracle database. Some of the commonly used types are shown in Table 1-1.

Oracle Type	Meaning
CHAR (<i>length</i>)	Stores strings of a fixed length. The <i>length</i> parameter specifies the length of the string. If a string of a smaller length is stored, it is padded with spaces at the end. For example, CHAR (2) may be used to store a fixed length string of two characters; if C is stored using this definition, then a single space is added at the end. CA would be stored as is with no padding.
VARCHAR2 (<i>length</i>)	Stores strings of a variable length. The <i>length</i> parameter specifies the maximum length of the string. For example, VARCHAR2 (20) may be used to store a string of up to 20 characters in length. No padding is used at the end of a smaller string.
DATE	Stores dates and times. The DATE type stores the century, all four digits of a year, the month, the day, the hour (in 24-hour format), the minute, and the second. The DATE type may be used to store dates and times between January 1, 4712 B.C. and December 31, 4712 A.D.
INTEGER	Stores integer numbers. An integer number doesn't contain a floating point: it is a whole number, such as 1, 10, and 115, for example.
NUMBER (<i>precision</i> , <i>scale</i>)	Stores floating point numbers, but may also be used to store integer numbers. <i>precision</i> is the maximum number of digits (in front of and behind a decimal point, if used) that may be used for the number. The maximum precision supported by the Oracle database is 38. <i>scale</i> is the maximum number of digits to the right of a decimal point (if used). If neither <i>precision</i> nor <i>scale</i> is specified, any number may be stored up to a precision of 38 digits. Numbers that exceed the <i>precision</i> are rejected by the database.
BINARY_FLOAT	New for Oracle10g. Stores a single precision 32-bit floating point number. You'll learn more about BINARY_FLOAT later in the section "The New Oracle10g BINARY_FLOAT and BINARY_DOUBLE Types."
BINARY_DOUBLE	New for Oracle10g. Stores a double precision 64-bit floating point number. You'll learn more about BINARY_DOUBLE later in the section "The New Oracle10g BINARY_FLOAT and BINARY_DOUBLE Types."

TABLE 1-1. *Commonly Used Oracle Data Types*

You can see all the data types in Appendix A. The following table illustrates a few examples of how numbers of type `NUMBER` are stored in the database:

Format	Number Supplied	Number Stored
<code>NUMBER</code>	1234.567	1234.567
<code>NUMBER (6, 2)</code>	123.4567	123.46
<code>NUMBER (6, 2)</code>	12345.67	Number exceeds the specified precision and is rejected by the database.

Examining the Store Tables

In this section, you'll learn how the tables for the `store` schema are created. The `store` schema will hold the details of the hypothetical store. Some of the information held in the store schema includes

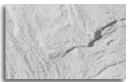
- Customer details
- Types of products sold
- Product details
- A history of the products purchased by the customers
- Employees of the store
- Salary grades

The following tables will be used to store this information:

- `customers` Stores customer details
- `product_types` Stores the types of products stocked by the store
- `products` Stores product details
- `purchases` Stores which products were purchased by which customers
- `employees` Stores the employee details
- `salary_grades` Stores the salary grade details

NOTE

The `store_schema.sql` script creates other tables and database items not mentioned in the previous list. You'll learn about these items in later chapters.



14 Oracle Database 10g SQL

In the next sections, you'll see the details of some of the `store` tables, and you'll see the `CREATE TABLE` statements included in the `store_schema.sql` script that creates these tables.

The customers Table The `customers` table is used to store the details of the customers of the hypothetical store. The following items are to be stored in this table for each one of the store's customers:

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

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

```
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 following list contains the details of each of these columns:

- **customer_id** Stores a unique integer for each row in the table. Each table should have one or more columns that uniquely identifies each row in the table and is known as that table's *primary key*. The `CONSTRAINT` clause for the `customer_id` column indicates that this is the table's primary key. A `CONSTRAINT` clause is used to restrict the values stored in a table or column and, for the `customer_id` column, the `PRIMARY KEY` keywords indicate that the `customer_id` column must contain a unique number for each row. You can also attach an optional name to a constraint, which must immediately follow the `CONSTRAINT` keyword—in this case, the name of the constraint is `customers_pk`. When a row is added to the `customers` table, a unique value for the `customer_id` column must be given, and the Oracle database will prevent you from adding a row with the same primary key value. If you try to do so, you will get an error from the database.
- **first_name** Stores the first name of the customer. You'll notice the use of the `NOT NULL` constraint for the `first_name` column—this means that a value must be supplied for `first_name`. If no constraint is specified, a column uses the default constraint of `NULL` and allows the column to remain empty.
- **last_name** Stores the last name of the customer. This column is `NOT NULL`, and therefore you must supply a value.

- **dob** Stores the date of birth for the customer. Notice that a `NOT NULL` constraint is not specified for this column, therefore the default `NULL` is assumed, and a value is optional.
- **phone** Stores the phone number of the customer. This is an optional value.

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

```
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
```

Notice that customer #4's date of birth is null, as 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 you want to retrieve all the columns from the `customers` table.

The product_types Table The `product_types` table is used to store the names of the product types that may be stocked 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.

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

```
product_type_id name
-----
1 Book
2 Video
3 DVD
4 CD
5 Magazine
```

16 Oracle Database 10g SQL

This defines the product types for the store. Each product stocked by the store may be of one of these types.

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

```
SELECT * FROM product_types;
```

The products Table The `products` table is used to store detailed information about the products sold. The following pieces of information are to be stored 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 this 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 and 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. When you add a new product, you should also associate that product with a type by supplying the product type ID number in the `product_type_id` column. This type of relationship is known as a *master-detail* or *parent-child* relationship.
- **name** Stores the product name, which must be specified as the `name` column is `NOT NULL`.
- **description** Stores an optional description of the product.

- **price** Stores an optional price for a product. This column is defined as `NUMBER(5, 2)`—the precision is 5, and therefore a maximum of 5 digits may be supplied for this number. The scale is 2, and so 2 of those maximum 5 digits may be to the right of the decimal point.

The following is a subset of the rows that are stored in the `products` table, populated by the `store_schema.sql` script:

<code>product_id</code>	<code>product_type_id</code>	<code>name</code>	<code>description</code>	<code>price</code>
1	1	Modern Science	A description of modern science	19.95
2	1	Chemistry	Introduction to Chemistry	30
3	2	Supernova	A star explodes	25.99
4	2	Tank War	Action movie about a future war	13.95

The first row in the `products` table has a `product_type_id` of 1, which means that this product represents a book. The `product_type_id` value comes from the `product_types` table, which uses a `product_type_id` value of 1 to represent books. The second row also represents a book, but the third and fourth rows represent videos.

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

```
SELECT * FROM products;
```

The purchases Table The `purchases` table stores the purchases made by a customer. For each purchase made by a customer, the following information is to be stored:

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

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

```
CREATE TABLE purchases (
  product_id INTEGER
  CONSTRAINT purchases_fk_products
  REFERENCES products(product_id),
```

18 Oracle Database 10g SQL

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

The `purchases` table has a constraint named `purchases_pk` that spans multiple columns in the table. The `purchases_pk` constraint is also a `PRIMARY KEY` constraint and specifies that the table's primary key consists of two columns: `product_id` and `customer_id`. The combination of the two values in these columns must be unique for each row in the table.

The following is a subset of the rows that are stored in the `purchases` table, populated by the `store_schema.sql` script:

```
product_id customer_id  quantity
-----
          1             1           1
          2             1           3
          1             4           1
          2             2           1
          1             3           1
```

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

The employees Table The `employees` table stores the details of the employees of the store. The following information is to be stored:

- Employee ID
- If applicable, the employee ID of the employee's manager
- 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 this table with the following rows:

employee_id	manager_id	first_name	last_name	title	salary
1		James	Smith	CEO	800000
2	1	Ron	Johnson	Sales Manager	600000
3	2	Fred	Hobbs	Salesperson	150000
4	2	Susan	Jones	Salesperson	500000

The salary_grades Table The `salary_grades` table stores the different grades of salaries available to employees. The following information is to be stored:

- 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:

```
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 this 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. You do that using the SQL `INSERT`, `UPDATE`, and `DELETE` statements, respectively. This section doesn't exhaustively cover all the details of using these statements; you'll learn more about them in Chapter 8.

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 inserting 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 if you don't want to—and those columns will be automatically set to null.

You can tell which columns are defined as `NOT NULL` using the SQL*Plus `DESCRIBE` command. The following example describes the `customers` table:

```
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)
```

As you can see, the `customer_id`, `first_name`, and `last_name` columns are `NOT NULL`, meaning that you must supply a value for these columns. The `dob` and `phone` columns don't require a value—you could omit the values if you wanted, and they would be automatically set to null.

The following `INSERT` statement 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. Also notice that the statement has two parts: the column list and the values to be added.

```
SQL> INSERT INTO customers (
2   customer_id, first_name, last_name, dob, phone
3 ) VALUES (
4   6, 'Fred', 'Brown', '01-JAN-1970', '800-555-1215'
5 );
```

1 row created.

**NOTE**

*SQL*Plus automatically numbers lines after you hit 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 issuing the following `SELECT` statement:

```
SQL> SELECT *
```

```
2 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 that has been added to the table.

By default, the Oracle database displays dates in the format `DD-MON-YY`, where `DD` is the day number, `MON` are the first three characters of the month (in uppercase), and `YY` are 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.

Modifying an Existing Row in a Table

You use the `UPDATE` statement to change rows in a table. Normally, 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, specified using the `SET` clause

You can change one or more rows using the same `UPDATE` statement. If more than one row is specified, the same change will be implemented for all of those rows. The following statement updates the `last_name` column to `Orange` for the row in the `customers` table whose `customer_id` column is 2:

```
SQL> UPDATE customers
```

```
2 SET last_name = 'Orange'
```

```
3 WHERE customer_id = 2;
```

```
1 row updated.
```

SQL*Plus confirms that one row was updated.

22 Oracle Database 10g SQL



CAUTION

If you forget to add a `WHERE` clause, all the rows will be updated. This is typically not the result you want.

Notice that the `SET` clause is used in the previous `UPDATE` statement to specify the column and the new value for that column. You can confirm the previous `UPDATE` statement did indeed change customer #2's last name using the following query:

```
SQL> SELECT *
      2 FROM customers
      3 WHERE customer_id = 2;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
2	Cynthia	Orange	05-FEB-68	800-555-1212

Removing a Row from a Table

You use the `DELETE` statement to remove rows from a table. As with the `UPDATE` statement, you typically use a `WHERE` clause to limit the rows you wish to delete—if you don't, *all* the rows will be deleted from the table.

The following example uses a `DELETE` statement to remove the row from the `customers` table whose `customer_id` is 2:

```
SQL> DELETE FROM customers
      2 WHERE customer_id = 2;
```

1 row deleted.

SQL*Plus confirms that one row has been deleted.

To undo any changes you make to the database, you use `ROLLBACK`:

```
SQL> ROLLBACK;
```

Rollback complete.

Go ahead and issue a `ROLLBACK` to undo any changes you've made so far.

The New Oracle10g `BINARY_FLOAT` and `BINARY_DOUBLE` Types

Oracle10g introduces two new data types: `BINARY_FLOAT` and `BINARY_DOUBLE`. `BINARY_FLOAT` stores a single precision 32-bit floating point number; `BINARY_DOUBLE` stores a double precision 64-bit floating point number. These new data types are based on the IEEE (Institute for Electrical and Electronic Engineering) standard for binary floating-point arithmetic.

Benefits of BINARY_FLOAT and BINARY_DOUBLE

BINARY_FLOAT and BINARY_DOUBLE are intended to be complementary to the existing NUMBER type. BINARY_FLOAT and BINARY_DOUBLE offer the following benefits over NUMBER:

- **Smaller storage required** BINARY_FLOAT and BINARY_DOUBLE require 5 and 9 bytes of storage space, whereas NUMBER may use up to 22 bytes.
- **Can represent a greater range of numbers** BINARY_FLOAT and BINARY_DOUBLE support numbers much larger and smaller than can be stored in a NUMBER.
- **Operations are typically performed faster** Operations involving BINARY_FLOAT and BINARY_DOUBLE are typically performed faster than on NUMBER. This is because BINARY_FLOAT and BINARY_DOUBLE operations are typically performed in the hardware, whereas NUMBERS must first be converted using software before operations can be performed.
- **Closed operations** Arithmetic operations involving BINARY_FLOAT and BINARY_DOUBLE are closed, which means that either a number or a special value is returned. For example, if you divide a BINARY_FLOAT by another BINARY_FLOAT, a BINARY_FLOAT is returned.
- **Transparent rounding** BINARY_FLOAT and BINARY_DOUBLE use binary base-2 to represent a number, whereas NUMBER uses decimal base-10. The base used to represent a number affects how rounding occurs for that number. For example, a decimal floating-point number is rounded to the nearest decimal place, but a binary floating-point number is rounded to the nearest binary place.



TIP

If you are developing a system that involves a lot of numerical computations, you should consider using BINARY_FLOAT and BINARY_DOUBLE to represent your numbers.

Using BINARY_FLOAT and BINARY_DOUBLE in a Table

The following statement creates a table named `binary_test` that contains a BINARY_FLOAT and BINARY_DOUBLE column:

```
CREATE TABLE binary_test (
  bin_float BINARY_FLOAT,
  bin_double BINARY_DOUBLE
);
```



NOTE

You'll find a script named `oracle_10g_examples.sql` in the SQL directory, which creates the `binary_test` table in the `store` schema. The script also performs the `INSERT` statements you'll see in this section. You can run this script if you have access to an Oracle10g database.

The following example adds a row to the `binary_test` table:

```
INSERT INTO binary_test (  
    bin_float, bin_double  
) VALUES (  
    39.5f, 15.7d  
);
```

Notice you use “f” and “d” to indicate a literal number is a `BINARY_FLOAT` or a `BINARY_DOUBLE`.

Special Values

In addition to literal values, you can also use the special values shown in Table 1-2 with a `BINARY_FLOAT` or `BINARY_DOUBLE`.

The following example inserts `BINARY_FLOAT_INFINITY` and `BINARY_DOUBLE_INFINITY` into the `binary_test` table:

```
INSERT INTO binary_test (  
    bin_float, bin_double  
) VALUES (  
    BINARY_FLOAT_INFINITY, BINARY_DOUBLE_INFINITY  
);
```

Special Value	Description
<code>BINARY_FLOAT_NAN</code>	Not a number (NaN) for <code>BINARY_FLOAT</code> type
<code>BINARY_FLOAT_INFINITY</code>	Infinity (INF) for <code>BINARY_FLOAT</code> type
<code>BINARY_DOUBLE_NAN</code>	Not a number (NaN) for <code>BINARY_DOUBLE</code> type
<code>BINARY_DOUBLE_INFINITY</code>	Infinity (INF) for <code>BINARY_DOUBLE</code> type

TABLE 1-2. *Special Values*

Quitting SQL*Plus

You use the `EXIT` command to quit from SQL*Plus. On Windows this will terminate SQL*Plus; on Unix and Linux it will terminate SQL*Plus and take you back to the command-line prompt from which you started SQL*Plus. The following example quits SQL*Plus using the `EXIT` command:

```
SQL> EXIT
```

Introducing Oracle PL/SQL

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

- Blocks
- Variable declarations
- Conditionals
- Loops
- Cursors
- The ability to define procedures and functions

The following `CREATE PROCEDURE` statement defines a procedure named `update_product_price()`. The procedure multiplies the price of a product by a factor—the product ID and the factor are passed as parameters to the procedure. If the specified product doesn't exist, the procedure takes no action; otherwise, it updates the product price by the factor.

NOTE

Don't worry too much about the details of the PL/SQL shown in the following listing for now—you'll learn the details as you progress through this book. I just want you to get a feel for PL/SQL at this stage.

```
CREATE OR REPLACE PROCEDURE update_product_price (
  p_product_id IN products.product_id%TYPE,
  p_factor      IN NUMBER
) AS
  product_count INTEGER;
BEGIN
  -- count the number of products with the
  -- supplied product_id (should be 1 if the product exists)
```

26 Oracle Database 10g SQL

```
SELECT COUNT(*)
INTO product_count
FROM products
WHERE product_id = p_product_id;

-- if the product exists (product_count = 1) then
-- update that product's price
IF 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 any exception is thrown in the code.

You'll learn more about PL/SQL in Chapter 11.

Summary

In this chapter, you learned

- That a *relational database* is a collection of related information that has been organized into structures known as *tables*. Each table contains *rows* that are further organized into *columns*. These tables are stored in the database in structures known as *schemas*, which are areas where database users may store their objects (such as tables and procedures).
- That *Structured Query Language (SQL)* is the standard language designed to access relational databases.
- That *SQL*Plus* allows you to enter SQL statements using the keyboard or to supply a file that contains SQL statements and run those statements.
- How to run a script in *SQL*Plus* that creates the example `store` schema.
- How to execute simple SQL `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements.
- That *PL/SQL* is Oracle's procedural language that allows you to add programming constructs around SQL. PL/SQL is primarily used for adding procedures and functions to a database to implement business logic.

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