**CHAPTER 1  
Introduction**

In this chapter, you will learn about the following:

image Relational databases.

image The Structured Query Language (SQL), which is used to access a database.

image SQL\*Plus, Oracle’s interactive text-based tool for running SQL statements.

image SQL Developer, which is a graphical tool for database development.

image PL/SQL, Oracle’s procedural programming language. PL/SQL allows you to develop programs that are stored in the database.

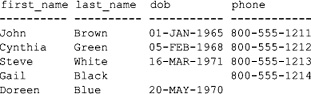
Let’s plunge in and consider what a relational database is.

**WHAT IS A RELATIONAL DATABASE?**

The concept of a relational database 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 *tables*. Each table stores data in *rows*; the data is arranged into *columns*. The tables are stored in database *schemas*, which are areas where users may store their own tables. A user may grant *permissions* to other users so they can access their tables.**

Most of us are familiar with data being stored in tables—stock prices and train timetables are sometimes organized into tables. One example table used in this book records customer information for an imaginary store; the table stores the customer first names, last names, dates of birth (dobs), and phone numbers:



This table could be stored in a variety of forms:

image A card in a box

image An HTML file on a web page

image A table in a database

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*. 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. Database management systems implement a standard language known as *Structured Query Language*, or SQL. Among other things, SQL allows you to retrieve, add, modify, 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 should be pronounced as the letters "S-Q-L."

image

**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 is now fully standardized and recognized by the American National Standards Institute.

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:

image **Query statements** retrieve rows stored in database tables. You write a query using the SQL SELECTstatement.

image **Data Manipulation Language (DML) statements** modify the contents of tables. There are three DML statements:

image **INSERT** adds rows to a table.

image **UPDATE** changes rows.

image **DELETE** removes rows.

image **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:

image **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.

image **ALTER** modifies a database structure. For example, ALTER TABLE is used to modify a table.

image **DROP** removes a database structure. For example, DROP TABLE is used to remove a table.

image **RENAME** changes the name of a table.

image **TRUNCATE** deletes all the rows from a table.

image **Transaction Control (TC) statements** either permanently record any changes made to rows, or undo those changes. There are three TC statements:

image **COMMIT** permanently records changes made to rows.

image **ROLLBACK** undoes changes made to rows.

image **SAVEPOINT** sets a "save point" to which you can roll back changes.

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

image **GRANT** gives another user access to your database structures.

image **REVOKE** prevents another user from accessing your database structures.

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 using JDBC; for more details, see my book *Oracle9*i *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 run a script containing SQL statements. SQL\*Plus enables you to conduct a "conversation" with the database; you enter SQL statements and view the results returned by the database. You’ll be introduced to SQL\*Plus next.

**USING SQL\*PLUS**

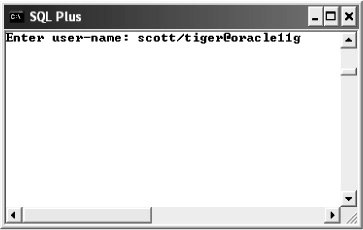
If you’re at all familiar with the Oracle database, chances are that you’re already familiar with SQL\*Plus. If you’re not, don’t worry: you’ll learn how to use SQL\*Plus in this book.

In the following sections, you’ll learn how to start SQL\*Plus and run a query.

**Starting SQL\*Plus**

If you’re using Windows XP Professional Edition and Oracle Database 11*g*, you can start SQL\*Plus by clicking START and selecting All Programs | Oracle | Application Development | SQL Plus.

[Figure 1-1](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch01.html#fig_1-1) shows SQL\*Plus running on Windows XP. SQL\*Plus asks you for a username. [Figure 1-1](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch01.html#fig_1-1) shows the scott user connecting to the database (scott is an example user that is contained in many Oracle databases; scott has a default password of tiger). The host string after the @ character tells SQL\*Plus where the database is running. If you are running the database on your own computer, you’ll typically omit the host string (that is, you enter scott/tiger)—doing 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) to get the host string. If the scott user doesn’t exist 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; you don’t absolutely have to use the scott user).

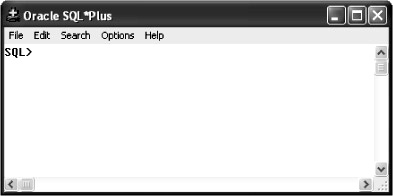


**FIGURE 1-1** *Oracle Database 11*g *SQL\*Plus Running on Windows XP*

If you’re using Windows XP and Oracle Database 10*g* or below, you can run a special Windows-only version of SQL\*Plus. You start this version of SQL\*Plus by clicking Start and selecting All Programs | Oracle | Application Development | SQL Plus. The Windows-only version of SQL\*Plus is deprecated in Oracle Database 11*g* (that is, it doesn’t ship with 11*g*), but it will still connect to an 11*g* database. [Figure 1-2](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch01.html#fig_1-2) shows the Windows-only version of Oracle Database 10*g* SQL\*Plus running on Windows XP.

image

**NOTE**  
*The Oracle Database 11*g *version of SQL\*Plus is slightly nicer than the Windows-only version. In the 11*g *version, you can scroll through previous commands you’ve run by pressing the UP and DOWN ARROW keys on the keyboard*.



**FIGURE 1-2** *Oracle Database 10*g *SQL\*Plus Running on Windows XP*

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

You can also start SQL\*Plus from the command line. To do this, you use the sqlplus command. The full syntax for the sqlplus command is

sqlplus [*user\_name*[/*password*[@*host\_string*]]]

where

image *user\_name* is the name of the database user.

image *password* is the password for the database user.

image *host\_string* is the database you want to connect to.

The following examples show sqlplus commands:

sqlplus scott/tiger

sqlplus scott/tiger@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 a non-Windows operating system (for example, Unix or Linux), either you must be in the same directory as the SQL\*Plus program to run it or, better still, you should add the directory to your path. If you need help with that, talk to your system administrator.

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

sqlplus scott@orcl

SQL\*Plus then prompts you to enter the password. As you type in the password, it is hidden from prying eyes. This also works when starting SQL\*Plus in Windows.

You can also just enter

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, go ahead and run the following SELECT statement (it returns the current date):

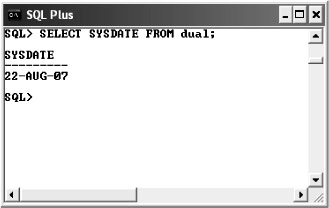
**SELECT SYSDATE FROM dual;**

SYSDATE is a built-in database function that returns the current date, and the dual table is a table that contains a single row. The dual table is useful when you need the database to evaluate an expression (e.g., 2 \* 15/5), or when you want to get the current date.

image

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

This illustration shows the results of this SELECT statement in SQL\*Plus running on Windows. As you can see, the query displays the current date from the database.



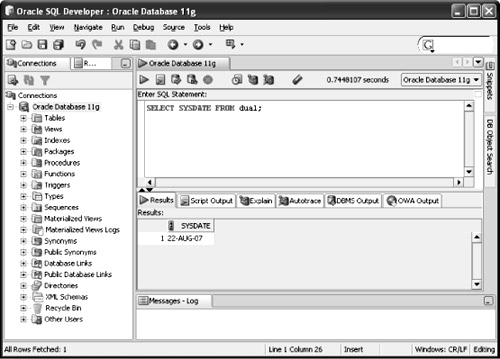
You can edit your last SQL statement in SQL\*Plus by entering EDIT. Doing this is useful when you make a mistake or you want to make a change to your SQL statement. On Windows, when you enter EDIT you are taken to the Notepad application; you then use Notepad to edit your SQL statement. When you exit Notepad and save your statement, the new statement is passed back to SQL\*Plus, where you can re-execute it by entering a forward slash (/). On Linux or Unix, the default editor is typically set to vi or emacs.

image

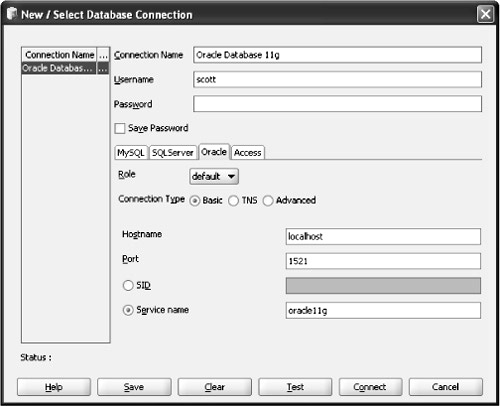
**NOTE**  
*You’ll learn more about editing SQL statements using SQL\*Plus in*[*Chapter 3*](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch03.html#ch03).

**SQL DEVELOPER**

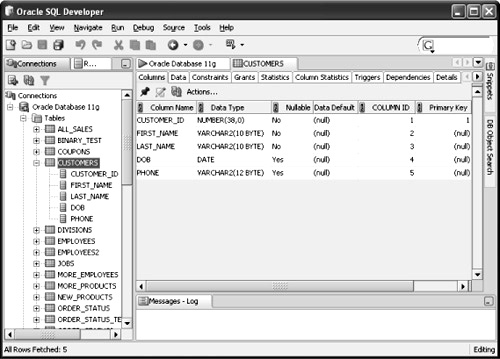
You can also enter SQL statements using SQL Developer. SQL Developer uses a very nice graphical user interface through which you can enter SQL statements, examine database tables, run scripts, edit and debug PL/SQL code, and much more. SQL Developer can connect to any Oracle Database, version 9.2.0.1 and higher, and runs on Windows, Linux, and Mac OSX. The following illustration shows SQL Developer running.



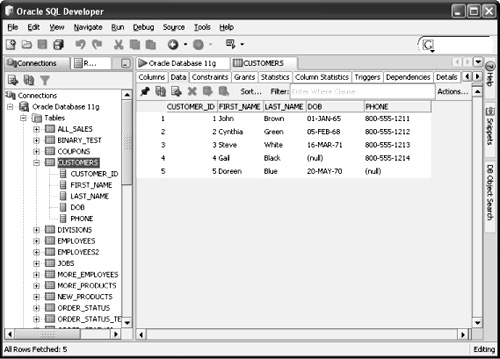
You need to have Java installed on your computer before you can run SQL Developer. If you’re using Windows XP Professional Edition and Oracle Database 11*g*, you start SQL Developer by clicking Start and selecting All Programs | Oracle | Application Development | SQL Developer. SQL Developer will prompt you to select the Java executable. You then browse to the location where you have installed it and select the executable. Next, you need to create a connection by right-clicking Connections and selecting New Connection, as shown in the following illustration.



Once you’ve created a connection and tested it, you can use it to connect to the database and run queries, examine database tables, and so on. The following illustration shows the details for a database table named customers.



You can also view the data stored in a table, as shown in the following illustration.



You can see full details on using SQL Developer by selecting Help | Table of Contents from the menu bar in SQL Developer.

In the next section, you’ll learn how to create the imaginary store schema used throughout 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 used to create the storeschema. You’ll now learn how to run the store\_schema.sql script.

**Running the SQL\*Plus Script to Create the Store Schema**

You perform the following steps to create the store schema:

**1**. Start SQL\*Plus.

**2**. Log into the database as a user with 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. You may need to speak with your database administrator about setting up a user for you with the required privileges (they might also run the store\_schema.sql script for you).

**3**. Run the store\_schema.sql script from within SQL\*Plus using the @ command.

The @ command has the following syntax:

@ *directory*\store\_schema.sql

where *directory* is the directory where your store\_schema.sql script is located.

For example, if the script is stored in E:\sql\_book\SQL, then you enter

@ E:\sql\_book\SQL\store\_schema.sql

If you have placed the store\_schema.sql script in a directory that contains spaces, then you must place the directory and script in quotes after the @ command. For example:

@ "E:\Oracle SQL book\sql\_book\SQL\store\_schema.sql"

If you’re using Unix or Linux and you saved the script in a directory named SQL in the tmp file system, then you enter

@ /tmp/SQL/store\_schema.sql

image

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

The first executable line in the store\_schema.sql script attempts to drop the store user, generating an error because the user doesn’t exist yet. Don’t worry about the error: the line is there so you don’t have to manually drop the store user when recreating the schema later in the book.

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 about the details of the statements contained in the script—you’ll learn the details as you progress through this book.

image

**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 see the DDL statements used to create the storeuser and some of the tables.

image

**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: you just run the*store\_schema.sql *script*.

The next sections describe the following:

image How to create a database user

image The commonly used data types used in an Oracle database

image 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

image *user\_name* is the user name

image *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 connectpermission) 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:

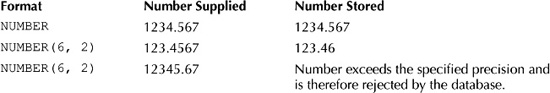
GRANT connect, resource TO store;

Once a user has been created, the database tables and other database objects can be created in the associated schema for that user. Many of the examples in this book use the store schema. Before I get into the details of the store tables, you need to know about the commonly used Oracle database types.

**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](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch01.html#table_1-1).

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



**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

image Customer details

image Types of products sold

image Product details

image A history of the products purchased by the customers

image Employees of the store

image Salary grades

The following tables are used to hold the information:

image **customers** holds the customer details.

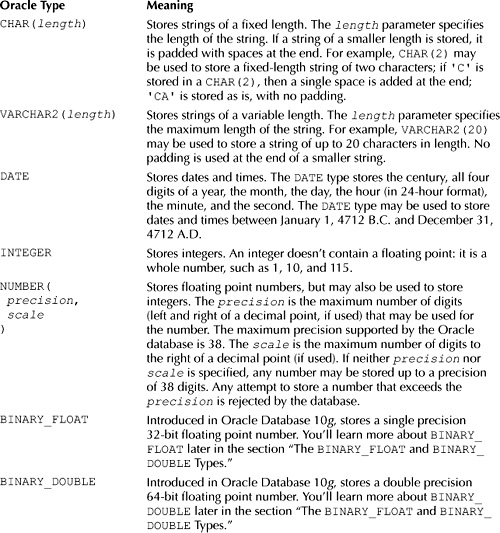
image **product\_types** holds the types of products sold by the store.

image **products** holds the product details.

image **purchases** holds which products were purchased by which customers.

image **employees** holds the employee details.

image **salary\_grades** holds the salary grade details.



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

image

**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*.

In the following sections, you’ll see the details of some of the tables, and you’ll see the CREATE TABLEstatements included 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:

image First name

image Last name

image Date of birth (dob)

image 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:

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

image **customer\_id** Contains a unique integer for each row in the table. Each table should have one or more columns that uniquely identifies each row; the column(s) are known as the *primary key*. The CONSTRAINTclause 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 when a constraint error occurs it is easy to spot where it happened.

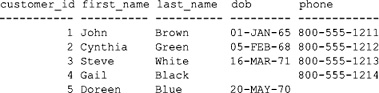
image **first\_name** Contains the first name of the customer. You’ll notice the use of the NOT NULL constraint for this column—this means that a value must be supplied for first\_ name when adding or modifying a row. If a NOT NULL constraint is omitted, a user doesn’t need to supply a value and the column can remain empty.

image **last\_name** Contains the last name of the customer. This column is NOT NULL, and therefore a value must be supplied when adding or modifying a row.

image **dob** Contains the date of birth for the customer. Notice that no NOT NULL constraint is specified for this column; therefore, the default NULL is assumed, and a value is optional when adding or modifying a row.

image **phone** Contains the phone number of the customer. This is an optional value.

The store\_schema.sql script populates the customers table with the following rows:



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 that you want to retrieve all the columns from the customers table.

image

**NOTE**  
*In this book, SQL statements shown in* **bold** *are statements 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*.

**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:

image **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.

image **name** contains the product type name. It is a NOT NULL column, and therefore a value must be supplied when adding or modifying a row.

The store\_schema.sql script populates the product\_types table with the following rows:

product\_type\_id name

--------------- ----------

1 Book

2 Video

3 DVD

4 CD

5 Magazine

The product\_types table contains the product types for the store. Each product sold by the store must be 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 holds the products sold by the store. The following pieces of information are held for each product:

image Product type

image Name

image Description

image 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:

image **product\_id** uniquely identifies each row in the table. This column is the primary key of the table.

image **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; it 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.

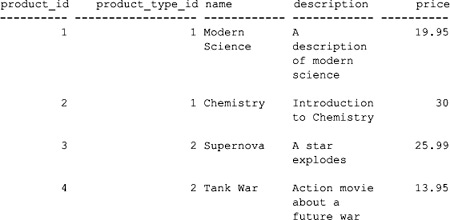
This type of relationship is known as a *master-detail* or *parent-child* relationship. When you add a new product, you associate that product with a type by supplying a matching product\_types.product\_type\_id value in the products.product\_type\_id column (you’ll see an example later).

image **name** contains the product name, which must be specified, as the name column is NOT NULL.

image **description** contains an optional description of the product.

image **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 5 digits may be supplied for this number. The scale is 2; therefore 2 of those maximum 5 digits may be to the right of the decimal point.

The following is a subset of the rows stored in the products 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, but 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 for yourself by executing the following SELECT statement using SQL\*Plus:

**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:

image Product ID

image Customer ID

image 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 purchasestable:

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:

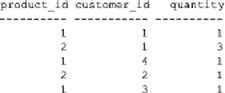
image **product\_id** contains the ID of the product that was purchased. This must match a product\_id column value in the products table.

image **customer\_id** contains the ID of a customer who made the purchase. This must match a customer\_idcolumn value in the customers table.

image **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 spans 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 is a subset of the rows that are stored in the purchases 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:

image Employee ID

image The ID of the employee’s manager (if applicable)

image First name

image Last name

image Title

image Salary

The store\_schema.sql script uses the following CREATE TABLE statement to create the employeestable:

CREATE TABLE employees (

employee\_id INTEGER CONSTRAINT employees\_pk PRIMARY KEY,

manager\_id INTEGER,

first\_name VARCHAR2(10) NOT NULL,

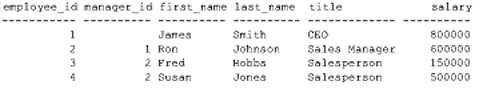
last\_name VARCHAR2(10) NOT NULL,

title VARCHAR2(20),

salary NUMBER(6, 0)

);

The store\_schema.sql script populates the employees table with the following rows:



As you can see, James Smith doesn’t have a manager. That’s because he is the CEO of the store.

**The salary\_grades Table** The salary\_grades table holds the different salary grades available to employees. The following information is held:

image Salary grade ID

image Low salary boundary for the grade

image 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 (

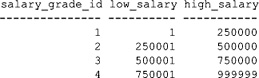
salary\_grade\_id INTEGER CONSTRAINT salary\_grade\_pk PRIMARY KEY,

low\_salary NUMBER(6, 0),

high\_salary NUMBER(6, 0)

);

The store\_schema.sql script populates the salary\_grades table with the following rows:



**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 using the COMMIT statement, or you can undo them 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 8](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch08.html#ch08).

**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:

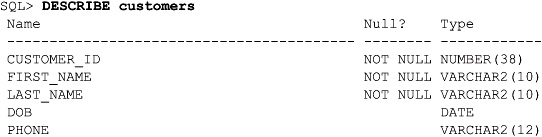
image The table into which the row is to be inserted

image A list of columns for which you want to specify column values

image 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; 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. The following example DESCRIBEs the customers table:



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.

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 (**

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.

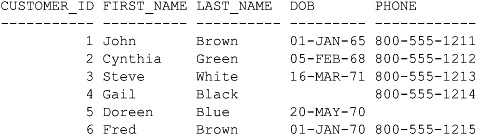
image

**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 running the following SELECT statement:

**SELECT \***

**FROM customers;**



Notice the new row that has been added to the end of the table.

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 causes an error because a row with a customer\_id of 1 already exists:

SQL> **INSERT INTO customers (**

2 **customer\_id, first\_name, last\_name, dob, phone**

3 **) VALUES (**

4 **1, 'Lisa', 'Jones', '02-JAN-1971', '800-555-1225'**

5 **);**

INSERT INTO customers (

\*

ERROR at line 1:

ORA-00001: unique constraint (STORE.CUSTOMERS\_PK) violated

Notice that the name of the constraint is shown in the error (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 (for example, SYS\_C0011277).

**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:

image The table containing the rows that are to be changed

image A WHERE clause that specifies the rows that are to be changed

image 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 made for all the rows. The following example updates customer #2’s last\_name to Orange:

**UPDATE customers**

**SET last\_name = 'Orange'**

**WHERE customer\_id = 2;**

1 row updated.

SQL\*Plus confirms that one row was updated.

image

**CAUTION**  
*If you forget to add a* WHERE *clause, then all the rows will be updated*.

The following query confirms the update worked:

**SELECT \***

**FROM customers**

**WHERE customer\_id = 2;**

image

**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, *all* the rows will be deleted from the table.

The following DELETE statement removes customer #2:

**DELETE FROM customers**

**WHERE customer\_id = 2;**

1 row deleted.

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

**ROLLBACK;**

Rollback complete.

Go ahead and run the ROLLBACK to undo any changes you’ve made so far. That way, your results will match those shown in subsequent chapters.

image

**NOTE**  
*You can make changes to rows permanent using* COMMIT. *You’ll see how to do that in*[*Chapter 8*](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch08.html#ch08).

**THE BINARY\_FLOAT AND BINARY\_DOUBLE TYPES**

Oracle Database 10*g* introduced 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 of Electrical and Electronics Engineers) standard for binary floating-point arithmetic.

**Benefits of BINARY\_FLOAT and BINARY\_DOUBLE**

BINARY\_FLOAT and BINARY\_DOUBLE are intended to complement the existing NUMBER type. BINARY\_FLOAT and BINARY\_DOUBLE offer the following benefits over NUMBER:

image **Smaller storage required** BINARY\_FLOAT and BINARY\_DOUBLE require 5 and 9 bytes of storage space, whereas NUMBER might use up to 22 bytes.

image **Greater range of numbers represented** BINARY\_FLOAT and BINARY\_DOUBLE support numbers much larger and smaller than can be stored in a NUMBER.

image **Faster performance of operations** Operations involving BINARY\_FLOAT and BINARY\_DOUBLE are typically performed faster than NUMBER operations. This is because BINARY\_FLOAT and BINARY\_DOUBLEoperations are typically performed in the hardware, whereas NUMBERs must first be converted using software before operations can be performed.

image **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\_FLOATby another BINARY\_FLOAT, a BINARY\_FLOAT is returned.

image **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.

image

**TIP**  
*If you are developing a system that involves a lot of numerical computations, you should use*BINARY\_FLOAT *and* BINARY\_DOUBLE *to represent numbers. Of course, you must be using Oracle Database 10*g *or higher*.

**Using BINARY\_FLOAT and BINARY\_DOUBLE in a Table**

The following statement creates a table named binary\_test that contains a BINARY\_FLOAT and a BINARY\_DOUBLE column:

CREATE TABLE binary\_test (

bin\_float BINARY\_FLOAT,

bin\_double BINARY\_DOUBLE

);

image

**NOTE**  
*You’ll find a script named* oracle\_10g\_examples.sql *in the* SQL *directory that 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 are using Oracle Database 10*g *or higher*.

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 that f indicates a number is a BINARY\_FLOAT, and d indicates a number is a BINARY\_DOUBLE.

**Special Values**

You can also use the special values shown in [Table 1-2](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch01.html#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:

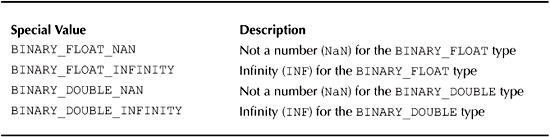
INSERT INTO binary\_test (

bin\_float, bin\_double

) VALUES (

BINARY\_FLOAT\_INFINITY, BINARY\_DOUBLE\_INFINITY

);



**TABLE 1-2** *Special Values*

The following query retrieves the rows from binary\_test:

**SELECT \***

**FROM binary\_test;**

image

**QUITTING SQL\*PLUS**

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

**EXIT**

image

**NOTE**  
*When you exit SQL\*Plus in this way, it automatically performs a* COMMIT *for you. If SQL\*Plus terminates abnormally—for example, if the computer on which SQL\*Plus is running crashes—a*ROLLBACK *is automatically performed. You’ll learn more about this in*[*Chapter 8*](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch08.html#ch08).

**INTRODUCING ORACLE PL/SQL**

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

image Variable declarations

image Conditional logic (if-then-else, and so on)

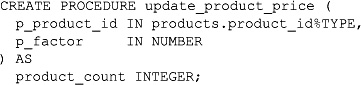
image Loops

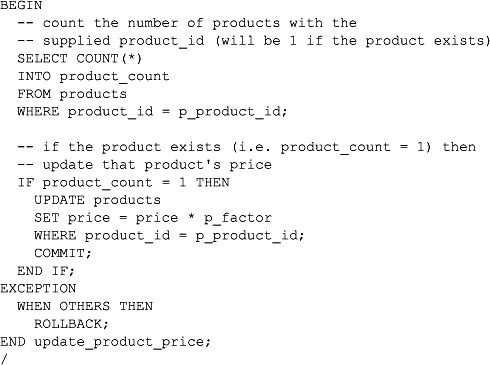
image Procedures and functions

The following CREATE PROCEDURE statement creates 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.

image

**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 11*](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch11.html#ch11)*. I just want you to get a feel for PL/SQL at this stage*.





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:

image 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 PL/SQL procedures).

image Structured Query Language (SQL) is the standard language designed to access relational databases.

image SQL\*Plus allows you to run SQL statements and SQL\*Plus commands.

image SQL Developer is a graphical tool for database development.

image How to run SELECT, INSERT, UPDATE, and DELETE statements.

image PL/SQL is Oracle’s procedural language that contains programming statements.

**HAPTER 2  
Retrieving Information from Database Tables**

In this chapter, you will see how to

image Retrieve information from one or more database tables using SELECT statements

image Use arithmetic expressions to perform calculations

image Limit the retrieval of rows to just those you are interested in using a WHERE clause

image Sort the rows retrieved from a table

The examples in this section use the store schema. If you want to follow along with the examples, you should start SQL\*Plus and log in as the store user.

**PERFORMING SINGLE TABLE SELECT STATEMENTS**

You use the SELECT statement to retrieve information from database tables. In the statement’s simplest form, you specify the table and columns from which you want to retrieve data. The following SELECT statement retrieves the customer\_id, first\_name, last\_name, dob, and phone columns from the customerstable:

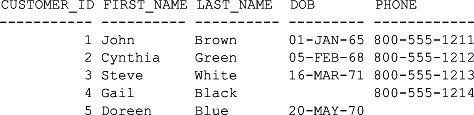
SELECT customer\_id, first\_name, last\_name, dob, phone

FROM customers;

Immediately after the SELECT keyword, you supply the column names that you want to retrieve; after the FROMkeyword, you supply the table name. The SQL statement is ended using a semicolon (;). SELECT statements are also known as *queries*.

You don’t tell the database management system software exactly how to access the information you want. You just tell it what you want and let the software worry about how to actually get it. The items that immediately follow the SELECT keyword needn’t always be columns from a table: They can be any valid expression. You’ll see examples of expressions later in this chapter.

After you press ENTER at the end of the SQL statement, the statement is executed and the results are returned to SQL\*Plus for display on the screen:



The rows returned by the database are known as a *result set*. As you can see from the example, the Oracle database converts the column names into their uppercase equivalents. Character and date columns are left-justified; number columns are right-justified. 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 displays only the last two digits.

image

**NOTE**  
*A database administrator can change the default display format for dates by setting an Oracle database parameter called* NLS\_DATE\_FORMAT. *You’ll learn more about dates in*[*Chapter 5*](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch05.html#ch05).

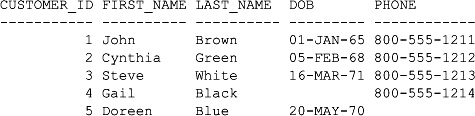
Although you can specify column names and table names using either lowercase or uppercase text, it is better to stick with one style. The examples in this book use uppercase for SQL and Oracle keywords, and lowercase for everything else.

**RETRIEVING ALL COLUMNS FROM A TABLE**

If you want to retrieve all columns in a table, you can use the asterisk character (\*) in place of a list of columns. In the following query, the asterisk is used to retrieve all columns from the customers table:

**SELECT \***

**FROM customers;**



As you can see, all the columns in the customers table are retrieved.

**SPECIFYING ROWS TO RETRIEVE USING THE WHERE CLAUSE**

You use the WHERE clause in a query to specify the rows you want to retrieve. This is very important, as Oracle has the capacity to store large numbers of rows in a table, and you may be interested in only a very small subset of those rows. You place the WHERE clause after the FROM clause:

SELECT *list of items*

FROM *list of tables*

WHERE *list of conditions*;

In the following query, the WHERE clause is used to retrieve the row from the customers table where the customer\_id column is equal to 2:

**SELECT \***

**FROM customers**

**WHERE customer\_id = 2;**

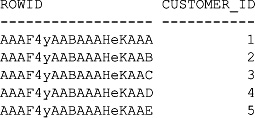
image

**ROW IDENTIFIERS**

Each row in an Oracle database has a unique row identifier, or *rowid*, which is used internally by the Oracle database to store the physical location of the row. A rowid is an 18-digit number that is represented as a base-64 number. You can view the rowid for rows in a table by retrieving the ROWID column in a query. For example, the following query retrieves the ROWID and customer\_id columns from the customers table; notice the base-64 number in the output:

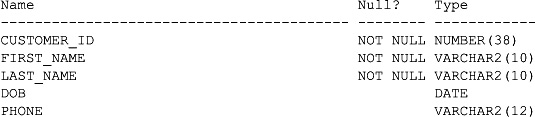
**SELECT ROWID, customer\_id**

**FROM customers;**



When you describe a table using the SQL\*Plus DESCRIBE command, ROWID doesn’t appear in the output from the command because it is only used internally by the database. ROWID is known as a *pseudo* column. The following example describes the customers table; notice ROWID doesn’t appear in the output:

**DESCRIBE customers**

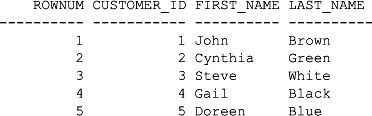


**ROW NUMBERS**

Another pseudo column is ROWNUM, which returns the row number in a result set. The first row returned by a query has a row number of 1, the second has a row number of 2, and so on. For example, the following query includes ROWNUM when retrieving the rows from the customers table:

**SELECT ROWNUM, customer\_id, first\_name, last\_name**

**FROM customers;**



Here’s another example:

**SELECT ROWNUM, customer\_id, first\_name, last\_name**

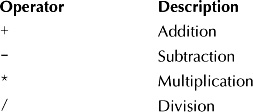
**FROM customers**

**WHERE customer\_id = 3;**

image

**PERFORMING ARITHMETIC**

Oracle allows you to perform arithmetic in SQL statements using arithmetic expressions, consisting of addition, subtraction, multiplication, and division. Arithmetic expressions consist of two *operands*—numbers or dates—and an arithmetic *operator*. The four arithmetic operators are shown in the following table:



The following query shows how to use the multiplication operator (\*) to calculate 2 multiplied by 6 (the numbers 2 and 6 are the operands):

**SELECT 2\*6**

**FROM dual;**

2\*6

----------

12

As you can see from this query, the correct result of 12 is displayed. The use of 2\*6 in the query is an example of an *expression*. An expression may contain a combination of columns, literal values, and operators.

**Performing Date Arithmetic**

You can use the addition and subtraction operators with dates. You can add a number—representing a number of days—to a date. The following example adds two days to July 25, 2007, and displays the resulting date:

**SELECT TO\_DATE('25-JUL-2007') + 2**

**FROM dual;**

TO\_DATE(

---------

27-JUL-07

|  |
| --- |
| **The dual Table** |
| You’ll notice the use of the dual table in the previous example. I mentioned the dual table in the previous chapter—dual is a table that contains a single row. The following output from the DESCRIBEcommand shows the structure of the dual table, along with a query that retrieves the row from the dualtable: |
| **DESCRIBE dual**  Name                                     Null?    Type ----------------------------------------- -------- -----------  DUMMY                                             VARCHAR2(1)  **SELECT \* FROM dual;**  D - X |
| Notice the dual table has one VARCHAR2 column named dummy and contains a single row with the value X. |

image

**NOTE**  
TO\_DATE() *is a function that converts a string to a date. You’ll learn more about* TO\_DATE() *in*[*Chapter 5*](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch05.html#ch05).

The next example subtracts three days from August 2, 2007:

**SELECT TO\_DATE('02-AUG-2007') - 3**

**FROM dual;**

TO\_DATE('

---------

30-JUL-07

You can also subtract one date from another, yielding the number of days between the two dates. The following example subtracts July 25, 2007, from August 2, 2007:

**SELECT TO\_DATE('02-AUG-2007') - TO\_DATE('25-JUL-2007')**

**FROM dual;**

TO\_DATE('02-AUG-2007')-TO\_DATE('25-JUL-2007')

---------------------------------------------

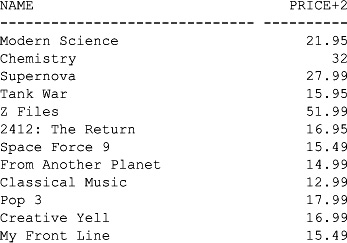
8

**Using Columns in Arithmetic**

Operands do not have to be literal numbers or dates; they may also be columns from a table. In the following query, the name and price columns are retrieved from the products table; notice that 2 is added to the value in the price column using the addition operator (+) to form the expression price + 2:

**SELECT name, price + 2**

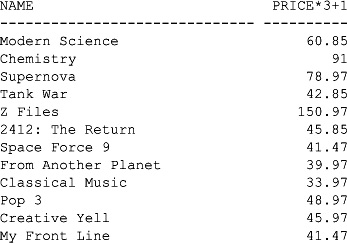
**FROM products;**



You can also combine more than one operator in an expression. In the following query, the price column is multiplied by 3, and then 1 is added to the resulting value:

**SELECT name, price \* 3 + 1**

**FROM products;**



The normal rules of arithmetic operator precedence apply in SQL: multiplication and division are performed first, followed by addition and subtraction. If operators of the same precedence are used, they are performed from left to right. For example, in the expression 10\*12/3–1, the first calculation would be 10 multiplied by 12, yielding a result of 120; then 120 would be divided by 3, yielding 40; finally, 1 would be subtracted from 40, yielding 39:

**SELECT 10 \* 12 / 3 – 1**

**FROM dual;**

10\*12/3-1

----------

39

You can also use parentheses () to specify the order of execution for the operators, as in the following:

**SELECT 10 \* (12 / 3−1)**

**FROM dual;**

10\*(12/3-1)

-----------

30

In this example, the parentheses are used to force calculation of 12/3-1 first, the result of which is then multiplied by 10, yielding 30 as the final answer.

**USING COLUMN ALIASES**

As you’ve seen, when you select a column from a table, Oracle uses the uppercase version of the column name as the header for the column in the output. For example, when you select the price column, the header in the resulting output is PRICE. When you use an expression, Oracle strips out the spaces and uses the expression as the header. You aren’t limited to using the header generated by Oracle; you can provide your own using an *alias*. In the following query, the expression price \* 2 is given the alias DOUBLE\_PRICE:

**SELECT price \* 2 DOUBLE\_PRICE**

**FROM products;**

DOUBLE\_PRICE

------------

39.9

60

51.98

27.9

99.98

29.9

26.98

25.98

21.98

31.98

29.98

26.98

If you want to use spaces and preserve the case of your alias text, you must place the text within double quotation marks (""):

**SELECT price \* 2 "Double Price"**

**FROM products;**

Double Price

------------

39.9

…

You can also use the optional AS keyword before the alias, as shown in the following query:

**SELECT 10 \* (12 / 3 - 1) AS "Computation"**

**FROM dual;**

Computation

-----------

30

**COMBINING COLUMN OUTPUT USING CONCATENATION**

You can combine the column values retrieved by a query using concatenation, which allows you to create more friendly and meaningful output. For example, in the customers table, the first\_name and last\_namecolumns contain the customer name, and in the previous queries the column values were displayed independently. But wouldn’t it be nice to combine the first\_name and last\_name columns? You can do this using the concatenation operator (||), as shown in the following query; notice that a space character is added after the first\_name column, and then the last\_name column is added:

**SELECT first\_name || ' ' || last\_name AS "Customer Name"**

**FROM customers;**

Customer Name

--------------------

John Brown

Cynthia Green

Steve White

Gail Black

Doreen Blue

The first\_name and last\_name column values are combined together in the output under the "Customer Name" alias.

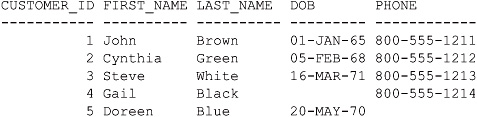
**NULL VALUES**

How does a database represent a value that is unknown? It uses a special value called a *null value*. A null value is not a blank string—it is a distinct value. A null value means the value for the column is unknown.

When you retrieve a column that contains a null value, you see nothing in the output for that column. You saw this (or rather, didn’t see it!) in the earlier examples that retrieved rows from the customers table: customer #4 has a null value in the dob column, and customer #5 has a null value in the phone column. In case you missed it, here’s the query again:

**SELECT \***

**FROM customers;**



You can also check for null values using IS NULL in a query. In the following example, customer #4 is retrieved because its dob value is null:

**SELECT customer\_id, first\_name, last\_name, dob**

**FROM customers**

**WHERE dob IS NULL;**

CUSTOMER\_ID FIRST\_NAME LAST\_NAME DOB

----------- ---------- ---------- ---------

4 Gail Black

In the next example, customer #5 is retrieved because its phone value is null:

**SELECT customer\_id, first\_name, last\_name, phone**

**FROM customers**

**WHERE phone IS NULL;**

CUSTOMER\_ID FIRST\_NAME LAST\_NAME PHONE

----------- ---------- ---------- ------------

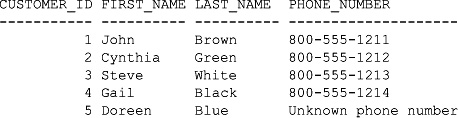
5 Doreen Blue

Since null values don’t display anything, how do you tell the difference between a null value and a blank string? The answer is to use the Oracle NVL() built-in function. NVL() returns another value in place of a null. NVL()accepts two parameters: a column (or, more generally, any expression that results in a value) and the value to be returned if the first parameter is null. In the following query, NVL() returns string 'Unknown phone number' when the phone column contains a null value:

**SELECT customer\_id, first\_name, last\_name**,

**NVL(phone, 'Unknown phone number') AS PHONE\_NUMBER**

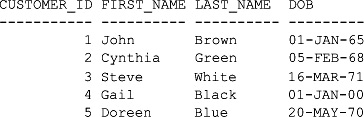
**FROM customers;**



You can also use NVL() to convert null numbers and dates. In the following query, NVL() returns the date 01-JAN-2000 when the dob column contains a null value:

**SELECT customer\_id, first\_name, last\_name, NVL(dob, '01-JAN-2000') AS DOB**

**FROM customers;**



Notice that customer #4’s dob is now displayed as 01-JAN-00.

**DISPLAYING DISTINCT ROWS**

Suppose you wanted to get the list of customers who purchased products from our imaginary store. You can get that list using the following query, which retrieves the customer\_id column from the purchases table:

**SELECT customer\_id**

**FROM purchases;**

CUSTOMER\_ID

-----------

1

2

3

4

1

2

3

4

3

The customer\_id column contains the IDs of customers who purchased a product. As you can see from the output returned by the query, some customers made more than one purchase and therefore appear twice. Wouldn’t it be great if you could throw out the duplicate rows that contain the same customer ID? You do this using the DISTINCT keyword. In the following query, DISTINCT is used to suppress the duplicate rows:

**SELECT DISTINCT customer\_id**

**FROM purchases;**

CUSTOMER\_ID

-----------

1

2

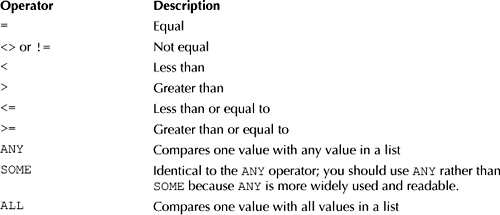
4

3

From this list, it’s easy to see that customers #1, #2, #3, and #4 made purchases; the duplicate rows are suppressed.

**COMPARING VALUES**

The following table lists the operators you can use to compare values:

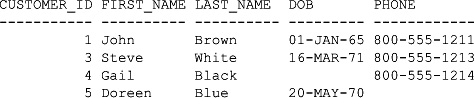


The following query uses the not equal (<>) operator in the WHERE clause to retrieve the rows from the customers table whose customer\_id is not equal to 2:

**SELECT \***

**FROM customers**

**WHERE customer\_id <> 2;**



The next query uses the > operator to retrieve the product\_id and name columns from the products table where the product\_id column is greater than 8:

**SELECT product\_id, name**

**FROM products**

**WHERE product\_id > 8;**

PRODUCT\_ID NAME

----------- ----------------

9 Classical Music

10 Pop 3

11 Creative Yell

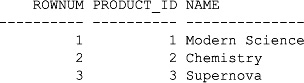
12 My Front Line

The following query uses the ROWNUM pseudo column and the <= operator to retrieve the first 3 rows from the products table:

**SELECT ROWNUM, product\_id, name**

**FROM products**

**WHERE ROWNUM <= 3;**

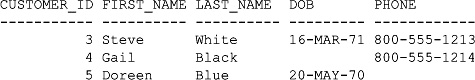


You use the ANY operator in a WHERE clause to compare a value with *any* of the values in a list. You must place an =, <>, <, >, <=, or >= operator before ANY. The following query uses ANY to retrieve rows from the customers table where the value in the customer\_id column is greater than any of the values 2, 3, or 4:

**SELECT \***

**FROM customers**

**WHERE customer\_id > ANY (2, 3, 4);**



You use the ALL operator in a WHERE clause to compare a value with *all* of the values in a list. You must place an =, <>, <, >, <=, or >= operator before ALL. The following query uses ALL to retrieve rows from the customers table where the value in the customer\_id column is greater than all of the values 2, 3, and 4:

**SELECT \***

**FROM customers**

**WHERE customer\_id > ALL (2, 3, 4);**

CUSTOMER\_ID FIRST\_NAME LAST\_NAME DOB PHONE

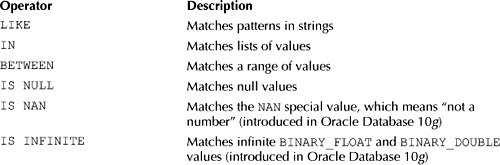
----------- ---------- ---------- --------- ------------

5 Doreen Blue 20-MAY-70

Only customer #5 is returned because 5 is greater than 2, 3, and 4.

**USING THE SQL OPERATORS**

The SQL operators allow you to limit rows based on pattern matching of strings, lists of values, ranges of values, and null values. The SQL operators are listed in the following table:



You can also use NOT to reverse the meaning of an operator:

image NOT LIKE

image NOT IN

image NOT BETWEEN

image IS NOT NULL

image IS NOT NAN

image IS NOT INFINITE

You’ll learn about the LIKE, IN, and BETWEEN operators in the following sections.

**Using the LIKE Operator**

You use the LIKE operator in a WHERE clause to search a string for a pattern. You specify patterns using a combination of normal characters and the following two wildcard characters:

image **Underscore (\_)** Matches one character in a specified position

image **Percent (%)** Matches any number of characters beginning at the specified position

For example, consider the following pattern:

'\_o%'

The underscore (\_) matches any one character in the first position, the o matches an *o* character in the second position, and the percent (%) matches any characters following the *o* character. The following query uses the LIKE operator to search the first\_name column of the customers table for the pattern '\_o%':

**SELECT \***

**FROM customers**

**WHERE first\_name LIKE '\_o%';**

image

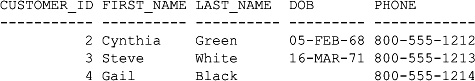
As you can see from the results, two rows are returned, because the strings John and Doreen both have o as the second character.

The next query uses NOT LIKE to get the rows not retrieved by the previous query:

**SELECT \***

**FROM customers**

**WHERE first\_name NOT LIKE '\_o%';**



If you need to search for actual underscore or percent characters in a string, you can use the ESCAPE option to identify those characters. For example, consider the following pattern:

'%\%%' ESCAPE '\'

The character after the ESCAPE tells the database how to differentiate between characters to search for and wildcards, and in the example the backslash character (\) is used. The first % is treated as a wildcard and matches any number of characters; the second % is treated as an actual character to search for; the third % is treated as a wildcard and matches any number of characters. The following query uses the promotions table, which contains the details for products being discounted by the store (you’ll learn more about this table later in this book). The query uses the LIKE operator to search the name column of the promotions table for the pattern '%\%%' ESCAPE '\':

**SELECT name**

**FROM promotions**

**WHERE name LIKE '%\%%' ESCAPE '\';**

NAME

------------------------------

10% off Z Files

20% off Pop 3

30% off Modern Science

20% off Tank War

10% off Chemistry

20% off Creative Yell

15% off My Front Line

As you can see from the results, the query returns the rows whose names contain a percentage character.

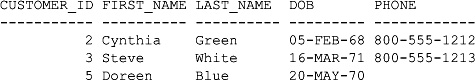
**Using the IN Operator**

You may use the IN operator in a WHERE clause to retrieve the rows whose column value is in a list. The following query uses IN to retrieve rows from the customers table where the customer\_id is 2, 3, or 5:

**SELECT \***

**FROM customers**

**WHERE customer\_id IN (2, 3, 5);**



NOT IN retrieves the rows not retrieved by IN:

**SELECT \***

**FROM customers**

**WHERE customer\_id NOT IN (2, 3, 5);**

image

One important point to be aware of is that NOT IN returns false if a value in the list is null. This is illustrated by the following query, which doesn’t return any rows because null is included in the list:

**SELECT \***

**FROM customers**

**WHERE customer\_id NOT IN (2, 3, 5, NULL);**

no rows selected

image

**CAUTION**  
NOT IN *returns false if a value in the list is null. This is important because, since you can use any expression in the list and not just literal values, it can be difficult to spot when a null value occurs. Consider using* NVL() *with expressions that might return a null value*.

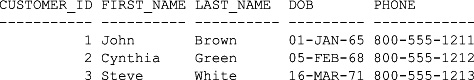
**Using the BETWEEN Operator**

You may use the BETWEEN operator in a WHERE clause to retrieve the rows whose column value is in a specified range. The range is *inclusive*, which means the values at both ends of the range are included. The following query uses BETWEEN to retrieve the rows from the customers table where the customer\_id is between 1 and 3:

**SELECT \***

**FROM customers**

**WHERE customer\_id BETWEEN 1 AND 3;**



NOT BETWEEN retrieves the rows not retrieved by BETWEEN:

**SELECT \***

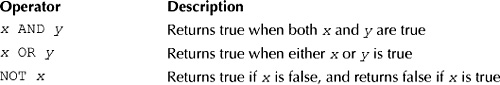
**FROM customers**

**WHERE customer\_id NOT BETWEEN 1 AND 3;**

image

**USING THE LOGICAL OPERATORS**

The logical operators allow you to limit rows based on logical conditions. The logical operators are listed in the following table:



The following query illustrates the use of the AND operator to retrieve the rows from the customers table where *both* of the following conditions are true:

image The dob column is greater than January 1, 1970.

image The customer\_id column is greater than 3.

**SELECT \***

**FROM customers**

**WHERE dob > '01-JAN-1970'**

**AND customer\_id > 3;**

CUSTOMER\_ID FIRST\_NAME LAST\_NAME DOB PHONE

----------- ---------- ---------- --------- ------------

5 Doreen Blue 20-MAY-70

The next query illustrates the use of the OR operator to retrieve rows from the customers table where *either* of the following conditions is true:

image The dob column is greater than January 1, 1970.

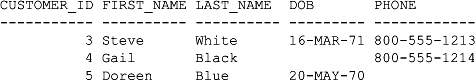
image The customer\_id column is greater than 3.

**SELECT \***

**FROM customers**

**WHERE dob > '01-JAN-1970'**

**OR customer\_id > 3;**



You can also use AND and OR to combine expressions in a WHERE clause, as you’ll see in the following section.

**OPERATOR PRECEDENCE**

If you combine AND and OR in the same expression, the AND operator takes precedence over the OR operator ("takes precedence over" means it’s executed first). The comparison operators take precedence over AND. Of course, you can override the default precedence by using parentheses to indicate the order in which you want to execute the expressions.

The following example retrieves the rows from the customers table where *either* of the following two conditions is true:

image The dob column is greater than January 1, 1970.

image The customer\_id column is less than 2 *and* the phone column has 1211 at the end.

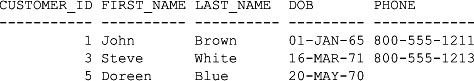
**SELECT \***

**FROM customers**

**WHERE dob > '01-JAN-1970'**

**OR customer\_id < 2**

**AND phone LIKE '%1211';**



As mentioned earlier, AND takes precedence over OR, so you can think of the WHERE clause in the previous query as follows:

dob > '01-JAN-1970' OR (customer\_id < 2 AND phone LIKE '%1211')

Therefore, customers #1, #3, and #5 are returned by the query.

**SORTING ROWS USING THE ORDER BY CLAUSE**

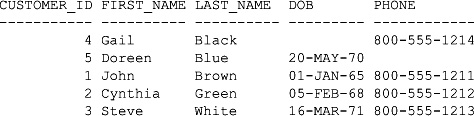
You use the ORDER BY clause to sort the rows retrieved by a query. The ORDER BY clause may specify one or more columns on which to sort the data; also, the ORDER BY clause must follow the FROM clause or the WHEREclause (if a WHERE clause is supplied).

The following query uses ORDER BY to sort the rows retrieved from the customers table by the last\_name:

**SELECT \***

**FROM customers**

**ORDER BY last\_name;**



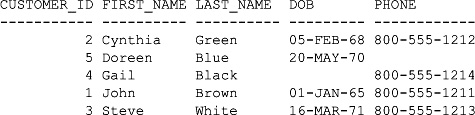
By default, ORDER BY sorts the columns in ascending order (lower values appear first). You can use the DESCkeyword to sort the columns in descending order (higher values appear first). You can also use the ASC keyword to explicitly specify an ascending sort—as I mentioned, ascending order is the default, but you can still specify it if you want to make it clear what the order is for the sort.

The next query uses ORDER BY to sort the rows retrieved from the customers table by ascending first\_name and descending last\_name:

**SELECT \***

**FROM customers**

**ORDER BY first\_name ASC, last\_name DESC;**

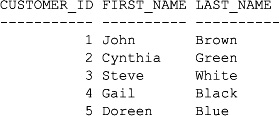


You can also use a column position number in the ORDER BY clause to indicate which column to sort: Use 1 to sort by the first column selected, 2 to sort by the second column selected, and so on. In the following query, column 1 (the customer\_id column) is used to sort the rows:

**SELECT customer\_id, first\_name, last\_name**

**FROM customers**

**ORDER BY 1;**



Because the customer\_id column is in position 1 after the SELECT keyword, it is the column used in the sort.

**PERFORMING SELECT STATEMENTS THAT USE TWO TABLES**

Database schemas have more than one table, with those tables storing different data. For example, the storeschema has tables that store information on customers, products, employees, and so on. Up to now, all the queries shown in this book retrieve rows from only one table. In the real world, you will often need to get information from multiple tables; for example, you might need to get the name of a product and the name of its product type. In this section, you’ll learn how to perform queries that use two tables; later, you’ll see queries that use more than two tables.

Let’s return to the example where you want to get the name of product #3 and its product type. The name of the product is stored in the name column of the products table, and the name of the product type is stored in the name column of the product\_types table. The products and product\_types tables are related to each other via the foreign key column product\_type\_id; the product\_type\_id column (the foreign key) of the products table points to the product\_type\_id column (the primary key) of the product\_typestable.

The following query retrieves the name and product\_type\_id columns from the products table for product #3:

**SELECT name, product\_type\_id**

**FROM products**

**WHERE product\_id = 3;**

NAME PRODUCT\_TYPE\_ID

------------------------------ ---------------

Supernova 2

The next query retrieves the name column from the product\_types table for the product\_type\_id of 2:

**SELECT name**

**FROM product\_types**

**WHERE product\_type\_id = 2;**

NAME

----------

Video

From this, you know that product #3 is a video. Nothing new so far, but what you really want is to retrieve the product name and its product type name using one query. You do this using a *table join* in the query. To join two tables in a query, you include both tables in the query’s FROM clause and include the related columns from each table in the WHERE clause.

For our example query, the FROM clause becomes

FROM products, product\_types

And the WHERE clause is

WHERE products.product\_type\_id = product\_types.product\_type\_id

AND products.product\_id = 3;

The join is the first condition in the WHERE clause (products.product\_type\_id = product\_types.product\_type\_id); typically, the columns used in the join are a primary key from one table and a foreign key from the other table. Because the equality operator (=) is used in the join condition, the join is known as an *equijoin*. The second condition in the WHERE clause (products.product\_id = 3) gets product #3, the product we are interested in viewing.

You’ll notice the tables as well as their columns are included in the WHERE clause. This is because there is a product\_type\_id column in both the products and product\_types tables, and you need to tell the database which table the column you want to use is in. If the columns had different names you could omit the table names, but you should always include them to make it clear where the columns come from.

The SELECT clause for the query is

SELECT products.name, product\_types.name

Notice the tables and their columns are specified again.

Putting everything together, the completed query is

**SELECT products.name, product\_types.name**

**FROM products, product\_types**

**WHERE products.product\_type\_id = product\_types.product\_type\_id**

**AND products.product\_id = 3;**

NAME NAME

------------------------------ ----------

Supernova Video

Perfect! This is exactly what we wanted: the name of the product and the name of the product type.

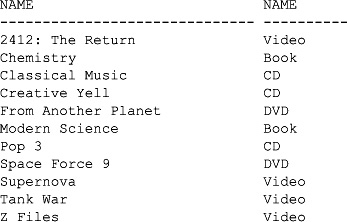
The next query gets all the products and orders them by the products.name column:

**SELECT products.name, product\_types.name**

**FROM products, product\_types**

**WHERE products.product\_type\_id = product\_types.product\_type\_id**

**ORDER BY products.name;**



Notice the product with the name "My Front Line" is missing from the results. The product\_type\_id for this product row is null, and the join condition does not return the row. You’ll see how to include this row later in the section "Outer Joins."

The join syntax you’ve seen in this section uses Oracle’s syntax for joins, which is based on the American National Standards Institute (ANSI) SQL/86 standard. With the introduction of Oracle Database 9*i*, the database also implements the ANSI SQL/92 standard syntax for joins; you’ll see this new syntax later in the section "Performing Joins Using the SQL/92 Syntax." You should use the SQL/92 standard in your queries when working with Oracle Database 9*i* and above, and you should use SQL/86 queries only when you’re using Oracle Database 8*i* and below.

**USING TABLE ALIASES**

In the previous section you saw the following query:

SELECT products.name, product\_types.name

FROM products, product\_types

WHERE products.product\_type\_id = product\_types.product\_type\_id

ORDER BY products.name;

Notice that the products and product\_types table names are used in the SELECT and WHERE clauses. Repeating the table names is redundant typing. A better way is to define table aliases in the FROM clause and then use the aliases when referencing the tables elsewhere in the query. For example, the following query uses the alias p for the products table and pt for the product\_types table; notice the table aliases are specified in the FROM clause, and the aliases are placed before the columns in the rest of the query:

SELECT p.name, pt.name

FROM products p, product\_types pt

WHERE p.product\_type\_id = pt.product\_type\_id

ORDER BY p.name;

image

**TIP**  
*Table aliases also make your queries more readable, especially when you start writing longer queries that use many tables*.

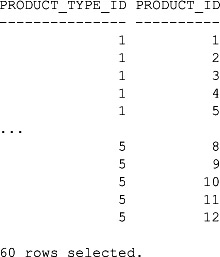
**CARTESIAN PRODUCTS**

If a join condition is missing, you will end up joining all rows from one table with all the rows from the other table; this result set is known as a *Cartesian product*. When this occurs, you will end up with a lot of rows being returned by the query. For example, assume you had one table containing 50 rows and a second table containing 100 rows. If you select columns from those two tables without a join, you would get 5,000 rows returned. This result happens because each row from table 1 would be joined to each row in table 2, which would yield a total of 50 rows multiplied by 100 rows, or 5,000 rows.

The following example shows a subset of the rows for a Cartesian product between the product\_types and products tables:

**SELECT pt.product\_type\_id, p.product\_id**

**FROM product\_types pt, products p;**



A total of 60 rows are selected because the product\_types and products tables contain 5 and 12 rows, respectively, and 5\*12 = 60.

**PERFORMING SELECT STATEMENTS THAT USE MORE THAN TWO TABLES**

Joins can be used to connect any number of tables. You use the following formula to calculate the number of joins you will need in your WHERE clause:

*Number of joins = the number of tables used in the query − 1*

For example, the following query uses two tables and one join:

SELECT p.name, pt.name

FROM products p, product\_types pt

WHERE p.product\_type\_id = pt.product\_type\_id

ORDER BY p.name;

Let’s consider a more complicated example that uses four tables—and therefore requires three joins. Let’s say you want to see the following information:

image The purchases each customer has made (comes from the purchases table)

image The customer’s first and last name (comes from the customers table)

image The name of the product they purchased (comes from the products table)

image The name of the product type (comes from the product\_types table)

In order to view this information, you need to query the customers, purchases, products, and product\_types tables, and the joins use the foreign key relationships between these tables. The following list shows the required joins:

**1**. To get the customer who made the purchase, join the customers and purchases tables using the customer\_id columns (customers.customer\_id = purchases.customer\_id).

**2**. To get the product purchased, join the products and purchases tables using the product\_id columns (products.product\_id = purchases.product\_id).

**3**. To get the product type name for the product, join the products and product\_types tables using the product\_type\_id columns (products.product\_type\_id = product\_types.product\_type\_id).

The following query uses these joins; notice I’ve used table aliases and renamed the headings for the product name to PRODUCT and the product type name to TYPE:

**SELECT c.first\_name, c.last\_name, p.name AS PRODUCT, pt.name AS TYPE**

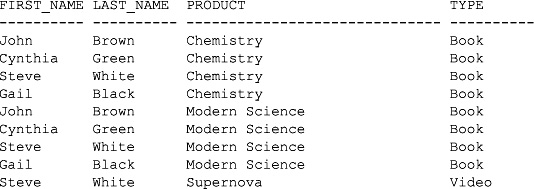
**FROM customers c, purchases pr, products p, product\_types pt**

**WHERE c.customer\_id = pr.customer\_id**

**AND p.product\_id = pr.product\_id**

**AND p.product\_type\_id = pt.product\_type\_id**

**ORDER BY p.name;**



The multi-table queries you’ve seen so far use the equality operator (=) in the join conditions; these joins are known as *equijoins*. As you’ll see in the next section, the equijoin is not the only join you can use.

**JOIN CONDITIONS AND JOIN TYPES**

In this section, you’ll explore join conditions and join types that allow you to create more advanced queries.

There are two types of join conditions, which are based on the operator you use in your join:

image **Equijoins** use the equality operator (=). You’ve already seen examples of equijoins.

image **Non-equijoins** use an operator other than the equality operator, such as <, >, BETWEEN, and so on. You’ll see an example of a non-equijoin shortly.

There are also three different types of joins:

image **Inner joins** return a row *only* when the columns in the join contain values that satisfy the join condition. This means that if a row has a null value in one of the columns in the join condition, that row isn’t returned. The examples you’ve seen so far have been inner joins.

image **Outer joins** return a row *even when* one of the columns in the join condition contains a null value.

image **Self joins** return rows joined on the same table.

You’ll learn about non-equijoins, outer joins, and self joins next.

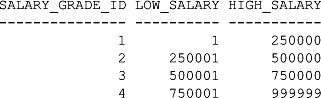
**Non-equijoins**

A non-equijoin uses an operator other than the equality operator (=) in the join. These operators are not-equal (<>), less than (<), greater than (>), less than or equal to (<=), greater than or equal to (>=), LIKE, IN, and BETWEEN. It’s pretty rare to encounter situations where you need to use a non-equijoin, but I have come across a few occasions when I’ve needed to use one; on those occasions I’ve had to use the BETWEEN operator.

For example, let’s say you want to get the salary grades for the employees. First, the following query retrieves the salary grades from the salary\_grades table:

**SELECT \***

**FROM salary\_grades;**



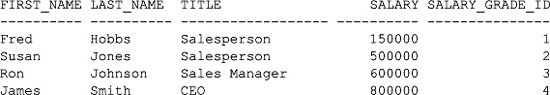
The next query uses a non-equijoin to retrieve the salary and salary grades for the employees; the salary grade is determined using the BETWEEN operator:

**SELECT e.first\_name, e.last\_name, e.title, e.salary, sg.salary\_grade\_id**

**FROM employees e, salary\_grades sg**

**WHERE e.salary BETWEEN sg.low\_salary AND sg.high\_salary**

**ORDER BY salary\_grade\_id;**



In this query, the BETWEEN operator returns true if the employee’s salary is between the low salary and high salary for the salary grade; when true is returned, the salary grade found is the salary grade for the employee. For example, Fred Hobbs' salary is $150,000; this is between the low salary of $1 and the high salary of $250,000 in the salary\_grades table for the salary\_grade\_id of 1; therefore, Fred Hobbs' salary grade is 1. Similarly, Susan Jones' salary is $500,000; this is between the low salary of $250,001 and the high salary of $500,000 for the salary grade ID of 2; therefore, Susan Jones' salary grade is 2. Ron Johnson and James Smith have salary grades of 3 and 4 respectively.

**Outer Joins**

An outer join retrieves a row even when one of the columns in the join contains a null value. You perform an outer join by supplying the outer join operator in the join condition; the Oracle proprietary outer join operator is a plus character in parentheses (+).

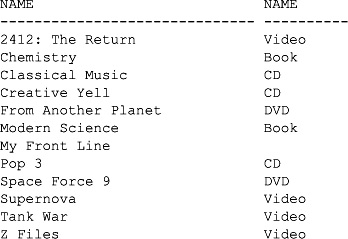
Let’s take a look at an example. Remember the query earlier that didn’t show the "My Front Line" product because its product\_type\_id is null? You can use an outer join to get that row. In the following query, notice that the Oracle outer join operator (+) is on the opposite side of the product\_type\_id column in the product table (this is the column that contains the null value):

**SELECT p.name, pt.name**

**FROM products p, product\_types pt**

**WHERE p.product\_type\_id = pt.product\_type\_id (+)**

**ORDER BY p.name;**



Notice that "My Front Line"—the product with the null product\_type\_id—is now retrieved, even though its product\_type\_id is null.

image

**NOTE**  
*You can place the outer join operator on either side of the join operator, but you always place it on the opposite side of the column that contains the null value*.

The following query returns the same results as the previous one, but notice that the column with the null value (pt.product\_type\_id) and the Oracle outer join operator are on the left of the equality operator:

SELECT p.name, pt.name

FROM products p, product\_types pt

WHERE pt.product\_type\_id (+) = p.product\_type\_id

ORDER BY p.name;

**Left and Right Outer Joins**

Outer joins can be split into two types:

image Left outer joins

image Right outer joins

To understand the difference between left and right outer joins, consider the following syntax:

SELECT …

FROM table1, table2

…

Assume the tables are to be joined on table1.column1 and table2.column2. Also, assume table1contains a row with a null value in column1. To perform a left outer join, the WHERE clause is

WHERE table1.column1 = table2.column2 (+);

image

**NOTE**  
*In a left outer join, the outer join operator is actually on the right of the equality operator*.

Next, assume table2 contains a row with a null value in column2. To perform a right outer join, you switch the position of the Oracle outer join operator to the *left* of the equality operator, and the WHERE clause becomes

WHERE table1.column1 (+) = table2.column2;

image

**NOTE**  
*As you’ll see, if* table1 *and* table2 *both contain rows with null values, you get different results depending on whether you use a left or right outer join*.

Let’s take a look at some concrete examples to make left and right outer joins clearer.

**An Example of a Left Outer Join** The following query uses a left outer join; notice that the Oracle outer join operator appears on the *right* of the equality operator:

**SELECT p.name, pt.name**

**FROM products p, product\_types pt**

**WHERE p.product\_type\_id = pt.product\_type\_id (+)**

**ORDER BY p.name;**



Notice all the rows from the products table are retrieved, including the "My Front Line" row, which has a null value in the product\_type\_id column.

**An Example of a Right Outer Join** The product\_types table contains a type of product not referenced in the products table (there are no magazines in the products table); the magazine product type appears at the end of the following example:

**SELECT \***

**FROM product\_types;**

PRODUCT\_TYPE\_ID NAME

--------------- ----------

1 Book

2 Video

3 DVD

4 CD

5 Magazine

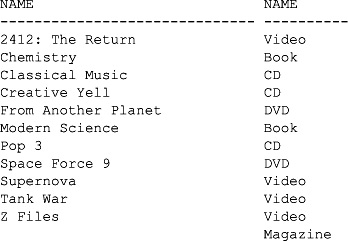
You can retrieve the magazine in a join on the products and product\_types tables by using a right outer join, as shown in the following query; notice that the Oracle outer join operator appears on the *left* of the equality operator:

**SELECT p.name, pt.name**

**FROM products p, product\_types pt**

**WHERE p.product\_type\_id (+) = pt.product\_type\_id**

**ORDER BY p.name;**



**Limitations on Outer Joins**

There are limitations when using outer joins, and you will learn about some of them in this section.

You may only place the outer join operator on one side of the join (not both). If you try to place the Oracle outer join operator on both sides, you get an error, as shown in the following example:

SQL> **SELECT p.name, pt.name**

2 **FROM products p, product\_types pt**

3 **WHERE p.product\_type\_id (+) = pt.product\_type\_id (+);**

WHERE p.product\_type\_id (+) = pt.product\_type\_id (+)

\*

ERROR at line 3:

ORA-01468: a predicate may reference only one outer-joined table

You cannot use an outer join condition with the IN operator:

SQL> **SELECT p.name, pt.name**

2 **FROM products p, product\_types pt**

3 **WHERE p.product\_type\_id (+) IN (1, 2, 3, 4);**

WHERE p.product\_type\_id (+) IN (1, 2, 3, 4)

\*

ERROR at line 3:

ORA-01719: outer join operator (+) not allowed in operand of OR or IN

You cannot use an outer join condition with another join using the OR operator:

SQL> **SELECT p.name, pt.name**

2 **FROM products p, product\_types pt**

3 **WHERE p.product\_type\_id (+) = pt.product\_type\_id**

4 **OR p.product\_type\_id = 1;**

WHERE p.product\_type\_id (+) = pt.product\_type\_id

\*

ERROR at line 3:

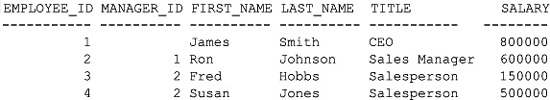
ORA-01719: outer join operator (+) not allowed in operand of OR or IN

image

**NOTE**  
*These are the commonly encountered limitations when using the outer join operator. For all the limitations, read the Oracle Database SQL Reference manual from Oracle Corporation*.

**Self Joins**

A self join is a join made on the same table. To perform a self join, you must use a different table alias to identify each reference to the table in the query. Let’s consider an example. The employees table has a manager\_idcolumn that contains the employee\_id of the manager for each employee; if the employee has no manager, then the manager\_id is null. The employees table contains the following rows:



James Smith—the CEO—has a null value for the manager\_id, meaning that he doesn’t have a manager. Susan Jones and Fred Hobbs are managed by Ron Johnson, and Ron Johnson is managed by James Smith.

You can use a self join to display the names of each employee and their manager. In the following query, the employees table is referenced twice, using two aliases w and m. The w alias is used to get the worker name, and the m alias is used to get the manager name. The self join is made between w.manager\_id and m.employee\_id:

**SELECT w.first\_name || ' ' || w.last\_name || ' works for '||**

**m.first\_name || ' ' || m.last\_name**

**FROM employees w, employees m**

**WHERE w.manager\_id = m.employee\_id**

**ORDER BY w.first\_name;**

W.FIRST\_NAME||''||W.LAST\_NAME||'WORKSFOR'||M.FIRST\_NA

-----------------------------------------------------

Fred Hobbs works for Ron Johnson

Ron Johnson works for James Smith

Susan Jones works for Ron Johnson

Because James Smith’s manager\_id is null, the join condition does not return the row.

You can perform outer joins in combination with self joins. In the following query, an outer join is used with the self join shown in the previous example to retrieve the row for James Smith. You’ll notice the use of the NVL()function to provide a string indicating that James Smith works for the shareholders (he’s the CEO, so he reports to the store’s shareholders):

**SELECT w.last\_name || ' works for ' ||**

**NVL(m.last\_name, 'the shareholders')**

**FROM employees w, employees m**

**WHERE w.manager\_id = m.employee\_id (+)**

**ORDER BY w.last\_name;**

W.LAST\_NAME||'WORKSFOR'||NVL(M.LAST\_N

-------------------------------------

Hobbs works for Johnson

Johnson works for Smith

Jones works for Johnson

Smith works for the shareholders

**PERFORMING JOINS USING THE SQL/92 SYNTAX**

The joins you’ve seen so far use Oracle’s syntax for joins, which is based on the ANSI SQL/86 standard. With the introduction of Oracle Database 9*i*, the database implements the ANSI SQL/92 standard syntax for joins, and you should use SQL/92 in your queries. You’ll see how to use SQL/92 in this section, including its use in avoiding unwanted Cartesian products.

image

**NOTE**  
*You can visit the ANSI website at*[*www.ansi.org*](http://www.ansi.org/).

**Performing Inner Joins on Two Tables Using SQL/92**

Earlier, you saw the following query, which uses the SQL/86 standard for performing an inner join:

SELECT p.name, pt.name

FROM products p, product\_types pt

WHERE p.product\_type\_id = pt.product\_type\_id

ORDER BY p.name;

SQL/92 introduces the INNER JOIN and ON clauses for performing an inner join. The following example rewrites the previous query using the INNER JOIN and ON clauses:

SELECT p.name, pt.name

FROM products p INNER JOIN product\_types pt

ON p.product\_type\_id = pt.product\_type\_id

ORDER BY p.name;

You can also use non-equijoin operators with the ON clause. Earlier, you saw the following query, which uses the SQL/86 standard for performing a non-equijoin:

SELECT e.first\_name, e.last\_name, e.title, e.salary, sg.salary\_grade\_id

FROM employees e, salary\_grades sg

WHERE e.salary BETWEEN sg.low\_salary AND sg.high\_salary

ORDER BY salary\_grade\_id;

The following example rewrites this query to use the SQL/92 standard:

SELECT e.first\_name, e.last\_name, e.title, e.salary, sg.salary\_grade\_id

FROM employees e INNER JOIN salary\_grades sg

ON e.salary BETWEEN sg.low\_salary AND sg.high\_salary

ORDER BY salary\_grade\_id;

**Simplifying Joins with the USING Keyword**

SQL/92 allows you to further simplify the join condition with the USING clause, but with the following limitations:

image The query must use an equijoin.

image The columns in the equijoin must have the same name.

Most of the joins you’ll perform will be equijoins, and if you always use the same name as the primary key for your foreign keys, you’ll satisfy these limitations.

The following query uses the USING clause instead of ON:

SELECT p.name, pt.name

FROM products p INNER JOIN product\_types pt

USING (product\_type\_id);

If you wanted to retrieve the product\_type\_id, you must provide only this column name on its own without a table name or alias in the SELECT clause, as for example:

SELECT p.name, pt.name, product\_type\_id

FROM products p INNER JOIN product\_types pt

USING (product\_type\_id);

If you try to provide a table alias with the column, such as p.product\_type\_id for example, you’ll get an error:

SQL> **SELECT p.name, pt.name, p.product\_type\_id**

2 **FROM products p INNER JOIN product\_types pt**

3 **USING (product\_type\_id);**

SELECT p.name, pt.name, p.product\_type\_id

\*

ERROR at line 1:

ORA-25154: column part of USING clause cannot have qualifier

Also, you only use the column name on its own within the USING clause. For example, if you specify USING(p.product\_type\_id) in the previous query instead of USING (product\_type\_id), you’ll get an error:

SQL> **SELECT p.name, pt.name, p.product\_type\_id**

2 **FROM products p INNER JOIN product\_types pt**

3 **USING (p.product\_type\_id);**

USING (p.product\_type\_id)

\*

ERROR at line 3:

ORA-01748: only simple column names allowed here

image

**CAUTION**  
*Don’t use a table name or alias when referencing columns used in a* USING *clause. You’ll get an error if you do*.

**Performing Inner Joins on More than Two Tables Using SQL/92**

Earlier you saw the following SQL/86 query, which retrieves rows from the customers, purchases, products, and product\_types tables:

SELECT c.first\_name, c.last\_name, p.name AS PRODUCT, pt.name AS TYPE

FROM customers c, purchases pr, products p, product\_types pt

WHERE c.customer\_id = pr.customer\_id

AND p.product\_id = pr.product\_id

AND p.product\_type\_id = pt.product\_type\_id

ORDER BY p.name;

The following example rewrites this query using SQL/92; notice how the foreign key relationships are navigated using multiple INNER JOIN and USING clauses:

SELECT c.first\_name, c.last\_name, p.name AS PRODUCT, pt.name AS TYPE

FROM customers c INNER JOIN purchases pr

USING (customer\_id)

INNER JOIN products p

USING (product\_id)

INNER JOIN product\_types pt

USING (product\_type\_id)

ORDER BY p.name;

**Performing Inner Joins on Multiple Columns Using SQL/92**

If your join uses more than one column from the two tables, you provide those columns in your ON clause and use the AND operator. For example, let’s say you have two tables named table1 and table2 and you want to join these tables using columns named column1 and column2 in both tables. Your query would be

SELECT …

FROM table1 INNER JOIN table2

ON table1.column1 = table2.column1

AND table1.column2 = table2.column2;

You can further simplify your query with the USING clause, but only if you’re performing an equijoin and the column names are identical. For example, the following query rewrites the previous example with the USINGclause:

SELECT …

FROM table1 INNER JOIN table2

USING (column1, column2);

**Performing Outer Joins Using SQL/92**

Earlier you saw how to perform outer joins using the outer join operator (+), which is Oracle proprietary syntax. SQL/92 uses a different syntax for performing outer joins. Instead of using (+), you specify the type of join in the FROM clause using the following syntax:

FROM *table1* { LEFT | RIGHT | FULL } OUTER JOIN *table2*

where

image *table1* and *table2* are the tables you want to join.

image LEFT means you want to perform a left outer join.

image RIGHT means you want to perform a right outer join.

image FULL means you want to perform a full outer join. A full outer join uses all rows in *table1* and *table2*, including those that have null values in the columns used in the join. You cannot directly perform a full outer join using the (+) operator.

You’ll see how to perform left, right, and full outer joins using the SQL/92 syntax in the following sections.

**Performing Left Outer Joins Using SQL/92**

Earlier you saw the following query, which performed a left outer join using the Oracle proprietary (+) operator:

SELECT p.name, pt.name

FROM products p, product\_types pt

WHERE p.product\_type\_id = pt.product\_type\_id (+)

ORDER BY p.name;

The next example rewrites this query using the SQL/92 LEFT OUTER JOIN keywords:

SELECT p.name, pt.name

FROM products p LEFT OUTER JOIN product\_types pt

USING (product\_type\_id)

ORDER BY p.name;

**Performing Right Outer Joins Using SQL/92**

Earlier you saw the following query, which performed a right outer join using the Oracle proprietary (+) operator:

SELECT p.name, pt.name

FROM products p, product\_types pt

WHERE p.product\_type\_id (+) = pt.product\_type\_id

ORDER BY p.name;

The next example rewrites this query using the SQL/92 RIGHT OUTER JOIN keywords:

SELECT p.name, pt.name

FROM products p RIGHT OUTER JOIN product\_types pt

USING (product\_type\_id)

ORDER BY p.name;

**Performing Full Outer Joins Using SQL/92**

A full outer join uses all rows in the joined tables, including those that have null values in either of the columns used in the join. The following example shows a query that uses the SQL/92 FULL OUTER JOIN keywords:

**SELECT p.name, pt.name**

**FROM products p FULL OUTER JOIN product\_types pt**

**USING (product\_type\_id)**

**ORDER BY p.name;**



Notice that both "My Front Line" from the products table and "Magazine" from the product\_types table are returned.

**Performing Self Joins Using SQL/92**

The following example uses SQL/86 to perform a self join on the employees table:

SELECT w.last\_name || ' works for ' || m.last\_name

FROM employees w, employees m

WHERE w.manager\_id = m.employee\_id;

The next example rewrites this query to use the SQL/92 INNER JOIN and ON keywords:

SELECT w.last\_name || ' works for ' || m.last\_name

FROM employees w INNER JOIN employees m

ON w.manager\_id = m.employee\_id;

**Performing Cross Joins Using SQL/92**

Earlier you saw how omitting a join condition between two tables leads to a Cartesian product. By using the SQL/92 join syntax, you avoid inadvertently producing a Cartesian product because you must always provide an ON or USING clause to join the tables—this is a good thing because you usually don’t want a Cartesian product.

If you really want a Cartesian product, the SQL/92 standard requires that you explicitly state this in your query using the CROSS JOIN keywords. In the following query, a Cartesian product between the product\_typesand products tables is generated using the CROSS JOIN keywords:

SELECT \*

FROM product\_types CROSS JOIN products;

**SUMMARY**

In this chapter, you have learned the following:

image How to perform single and multiple table queries

image How to select all columns from a table using an asterisk (\*) in a query

image How a row identifier (rowid) is used internally by the Oracle database to store the location of a row

image How to perform arithmetic in SQL

image How to use addition and subtraction operators with dates

image How to reference tables and columns using aliases

image How to merge column output using the concatenation operator (||)

image How nulls are used to represent unknown values

image How to display distinct rows using the DISTINCT operator

image How to limit the retrieval of rows using the WHERE clause

image How to sort rows using the ORDER BY clause

image How to perform inner, outer, and self joins using the SQL/86 and SQL/92 syntax

**CHAPTER 3  
Using SQL\*Plus**

In this chapter, you will see how to do the following:

Image View the structure of a table.

Image Edit an SQL statement.

Image Save and run scripts containing SQL statements and SQL\*Plus commands.

Image Format the results returned by SQL\*Plus.

Image Use variables in SQL\*Plus.

Image Create simple reports.

Image Get help from SQL\*Plus.

Image Automatically generate SQL statements.

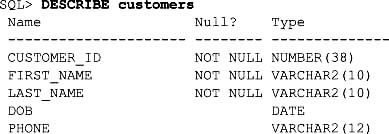
Image Disconnect from a database and exit SQL\*Plus.

Let’s plunge in and examine how you view the structure of a table.

**VIEWING THE STRUCTURE OF A TABLE**

Knowing the structure of a table is useful because you can use the information to formulate an SQL statement. For example, you can figure out the columns you want to retrieve in a query. You use the DESCRIBE command to view the structure of a table.

The following example uses DESCRIBE to view the structure of the customers table; notice that the semicolon character (;) is omitted from the end of the command:



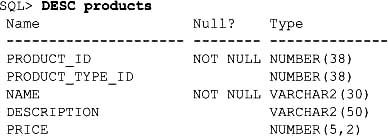
The output from DESCRIBE has three columns that show the structure of the table. These columns are as follows:

Image **Name** lists the names of the columns contained in the table. In the example, you can see the customers table has five columns: customer\_id, first\_name, last\_name, dob, and phone.

Image **Null?** indicates whether the column can store null values. If set to NOT NULL, the column cannot store a null value; if blank, the column can store a null value. In the preceding example, you can see that the customer\_id, first\_name, and last\_ name columns cannot store null values, but the dob and phonecolumns can.

Image **Type** indicates the type of the column. In the preceding example, you can see that the type of the customer\_id column is NUMBER (38) and that the type of the first\_ name is VARCHAR2 (10).

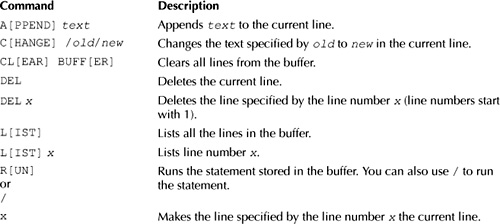
You can save some typing by shortening the DESCRIBE command to DESC (DESC[RIBE]). The following command uses DESC to view the structure of the products table:



**EDITING SQL STATEMENTS**

As you may have noticed, it becomes tedious to have to repeatedly type similar SQL statements into SQL\*Plus. You will be pleased to know SQL\*Plus stores your previous SQL statement in a buffer. You can then edit the lines that make up your SQL statement stored in the buffer.

Some of the editing commands are listed in the following table; notice the optional part of each command in square brackets (for example, you can abbreviate the APPEND command to A).



Let’s take a look at some examples of using the SQL\*Plus editing commands. The following example shows a query in SQL\*Plus:

SQL> **SELECT customer\_id, first\_name, last\_name**

2 **FROM customers**

3 **WHERE customer\_id = 1;**

SQL\*Plus automatically increments the line number when you press ENTER. You can make line 1 the current line by entering 1 at the prompt:

SQL> **1**

1\* SELECT customer\_id, first\_name, last\_name

Notice that SQL\*Plus displays the current line and the line number. The following example uses APPEND to add, "dob" to the end of the line:

SQL> **APPEND, dob**

1\* SELECT customer\_id, first\_name, last\_name, dob

The next example uses LIST to show all the lines in the buffer:

SQL> **LIST**

1 SELECT customer\_id, first\_name, last\_name, dob

2 FROM customers

3\* WHERE customer\_id = 1

Notice that the current line has been changed to the last line, as indicated by the asterisk character (\*). The following example uses CHANGE to replace "customer\_id = 1" with "customer\_id = 2" in this last line:

SQL> **CHANGE/customer\_id = 1/customer\_id = 2**

3\* WHERE customer\_id = 2

The next example uses RUN to execute the query:

SQL> **RUN**

1 SELECT customer\_id, first\_name, last\_name, dob

2 FROM customers

3\* WHERE customer\_id = 2

CUSTOMER\_ID FIRST\_NAME LAST\_NAME DOB

----------- ---------- ---------- ---------

2 Cynthia Green 05-FEB-68

You can also use a forward slash character (/) to run the SQL statement; for example:

SQL> **/**

CUSTOMER\_ID FIRST\_NAME LAST\_NAME DOB

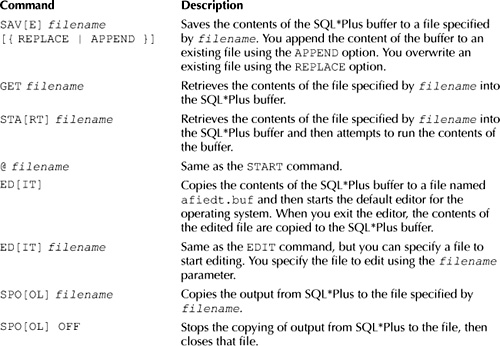
----------- ---------- ---------- ---------

2 Cynthia Green 05-FEB-68

**SAVING, RETRIEVING, AND RUNNING FILES**

SQL\*Plus allows you to save, retrieve, and run scripts containing SQL\*Plus commands and SQL statements. You’ve already seen one example of running an SQL\*Plus script: the store\_schema.sql script file that was run in [Chapter 1](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch01.html#ch01).

Some of the file commands are listed in the following table.



Let’s take a look at some examples of using these SQL\*Plus commands. If you want to follow along with the examples, go ahead and enter the following query into SQL\*Plus:

SQL> **SELECT customer\_id, first\_name, last\_name**

2 **FROM customers**

3 **WHERE customer\_id = 1;**

The following example uses SAVE to save the contents of the SQL\*Plus buffer to a file named cust\_query.sql:

SQL> **SAVE cust\_query.sql**

Created file cust\_query.sql

Image

**NOTE**  
*On my computer, the* cust\_query.sql *file is saved in the* E:\oracle\_11g\product\11.1.0\db\_1\BIN *directory*.

The next example uses GET to retrieve the contents of the cust\_query.sql file:

SQL> **GET cust\_query.sql**

1 SELECT customer\_id, first\_name, last\_name

2 FROM customers

3\* WHERE customer\_id = 1

The following example runs the query using /:

SQL> **/**

CUSTOMER\_ID FIRST\_NAME LAST\_NAME

----------- ---------- ----------

1 John Brown

The next example uses START to load and run the contents of the cust\_query.sql file in one step:

SQL> **START cust\_query.sql**

CUSTOMER\_ID FIRST\_NAME LAST\_NAME

----------- ---------- ----------

1 John Brown

You can edit the contents of the SQL\*Plus buffer using the EDIT command:

SQL> **EDIT**

The EDIT command starts the default editor for your operating system. On Windows the default editor is Notepad. On Unix and Linux the default editors are vi or emacs, respectively.

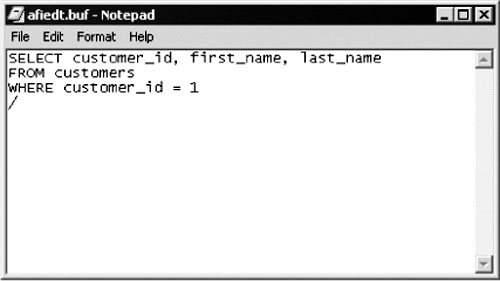
[Figure 3-1](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch03.html#fig_3-1) shows the contents of the SQL\*Plus buffer in Notepad. Notice the SQL statement is terminated using a slash character (/) rather than a semicolon.

In your editor, change the WHERE clause to WHERE customer\_id = 2, then save and exit the editor (in Notepad, you select File | Exit, then click Yes to save your query). SQL\*Plus displays the following output containing your modified query; notice that the WHERE clause has been changed:

1 SELECT customer\_id, first\_name, last\_name

2 FROM customers

3\* WHERE customer\_id = 2



**FIGURE 3-1** *Editing the SQL\*Plus buffer contents using Notepad*

**Changing the Default Editor**

You can change the default editor using the SQL\*Plus DEFINE command:

DEFINE \_EDITOR = '*editor*'

where *editor* is the name of your preferred editor.

For example, the following command sets the default editor to vi:

DEFINE \_EDITOR = 'vi'

You can also change the default editor SQL\*Plus uses by adding the line DEFINE \_EDITOR = '*editor*' to a new file named login.sql, where *editor* is the name of your preferred editor. You can add any SQL\*Plus commands you want to this file. SQL\*Plus will check the current directory for a login.sql file and execute it when SQL\*Plus starts. If there is no login.sql file in the current directory, then SQL\*Plus will check all directories (and their subdirectories) in the SQLPATH environment variable for a login.sql file. On Windows, SQLPATH is defined as a registry entry in HKEY\_LOCAL\_MACHINE\SOFTWARE\ORACLE\*oracle\_home\_key* (where *oracle\_home\_key* is the key for the associated installation of the Oracle database). On my Windows XP computer running Oracle Database 11*g*, SQLPATH is set to E:\oracle\_11g\product\11.1.0\db\_1\dbs. On Unix or Linux, there is no default SQLPATH defined, and you will need to add it as an environment variable. For further details on setting up a login.sql file, you can read the *SQL\*Plus User’s Guide and Reference*, published by Oracle Corporation.

You run your modified query using the slash character (/):

SQL> **/**

CUSTOMER\_ID FIRST\_NAME LAST\_NAME

----------- ---------- ----------

2 Cynthia Green

Image

**TIP**  
*In the Oracle Database 11*g *version of SQL\*Plus, you can also scroll through your previously executed statements using the UP and DOWN ARROW keys on the keyboard. Once you’ve selected a statement, you can use the LEFT and RIGHT ARROW keys to move the cursor to a specific point in the statement*.

You use the SPOOL command to copy the output from SQL\*Plus to a file. The following example spools the output to a file named cust\_results.txt, runs the query again, and then turns spooling off by executing SPOOL OFF:

SQL> **SPOOL cust\_results.txt**

SQL> **/**

CUSTOMER\_ID FIRST\_NAME LAST\_NAME

----------- ---------- ----------

2 Cynthia Green

SQL> **SPOOL OFF**

Feel free to examine the cust\_results.txt file; it will contain the previous output between the slash (/) and SPOOL OFF. On my Windows XP computer, the file is stored in E:\oracle\_11g\product\11.1.0\db\_1\BIN; the directory used is the current directory you are in when you start SQL\*Plus.

You can also specify the full directory path where you want the spool file to be written; for example:

SPOOL C:\my\_files\spools\cust\_results.txt

**FORMATTING COLUMNS**

You use the COLUMN command to format the display of column headings and column data. The simplified syntax for the COLUMN command is as follows:

COL[UMN] {*column* | *alias*} [*options*]

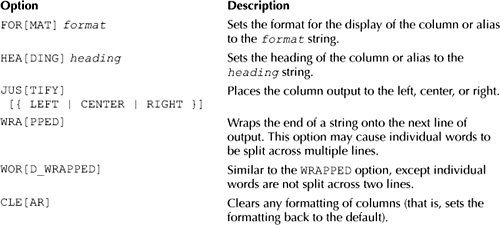
where

Image *column* is the column name.

Image *alias* is the column alias to be formatted. In [Chapter 2](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch02.html#ch02), you saw that you can "rename" a column using a column alias; you can reference an alias in the COLUMN command.

Image *options* are one or more options to be used to format the column or alias.

There are a number of options you can use with the COLUMN command. The following table shows some of these options.



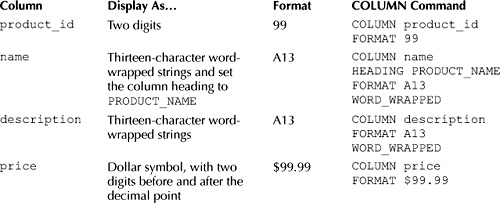
The *format* string in the previous table may take a number of formatting parameters. The parameters you specify depend on the data stored in your column:

Image If your column contains characters, you use A*x* to format the characters, where *x* specifies the width for the characters. For example, A12 sets the width to 12 characters.

Image If your column contains numbers, you can use any of a variety of number formats, which are shown later in [Table 4-4](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch04.html#table_4-4) of [Chapter 4](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch04.html#ch04). For example, $99.99 sets the format to a dollar sign, followed by two digits, the decimal point, plus another two digits.

Image If your column contains a date, you can use any of the date formats shown later in [Table 5-2](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch05.html#table_5-2) of [Chapter 5](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch05.html#ch05). For example, MM-DD-YYYY sets the format to a two-digit month (MM), a two-digit day (DD), and a four-digit year (YYYY).

Let’s take a look at an example. You’ll see how to format the output of a query that retrieves the product\_id, name, description, and price columns from the products table. The display requirements, format strings, and COLUMN commands are shown in the following table:



The following example shows the COLUMN commands in SQL\*Plus:

SQL> **COLUMN product\_id FORMAT 99**

SQL> **COLUMN name HEADING PRODUCT\_NAME FORMAT A13 WORD\_WRAPPED**

SQL> **COLUMN description FORMAT A13 WORD\_WRAPPED**

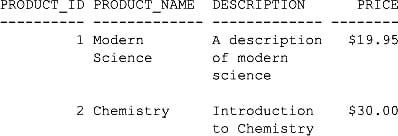
SQL> **COLUMN price FORMAT $99.99**

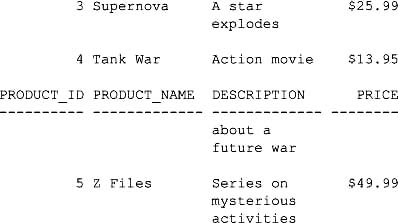
The next example runs a query to retrieve some rows from the products table; notice the formatting of the columns in the output:

SQL> **SELECT product\_id, name, description, price**

2 **FROM products**

3 **WHERE product\_id < 6;**



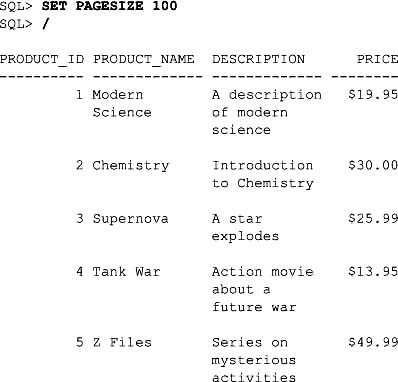


This output is readable, but wouldn’t it be nice if you could display the headings just once at the top? You do that by setting the page size, as you’ll see next.

**SETTING THE PAGE SIZE**

You set the number of lines in a page using the SET PAGESIZE command. This command sets the number of lines that SQL\*Plus considers one "page" of output, after which SQL\*Plus will display the headings again.

The following example sets the page size to 100 lines using the SET PAGESIZE command and runs the query again using /:



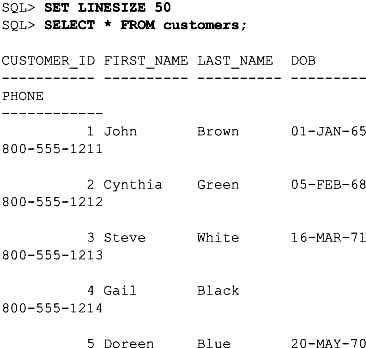
Notice that the headings are shown only once, at the top, and the resulting output looks better.

Image

**NOTE**  
*The maximum number for the page size is 50,000*.

**SETTING THE LINE SIZE**

You set the number of characters in a line using the SET LINESIZE command. The following example sets the line size to 50 lines and runs another query:



The lines don’t span more than 50 characters.

Image

**NOTE**  
*The maximum number for the line size is 32,767*.

**CLEARING COLUMN FORMATTING**

You clear the formatting for a column using the CLEAR option of the COLUMN command. For example, the following COLUMN command clears the formatting for the product\_id column:

SQL> **COLUMN product\_id CLEAR**

You can clear the formatting for all columns using CLEAR COLUMNS. For example:

SQL> **CLEAR COLUMNS**

Once you’ve cleared the columns, the output from the queries will use the default format.

**USING VARIABLES**

In this section, you’ll see how to create variables that may be used in place of actual values in SQL statements. These variables are known as *substitution variables* because they are used as substitutes for values. When you run an SQL statement, you enter values for the variables; the values are then "substituted" into the SQL statement.

There are two types of substitution variables:

Image **Temporary variables** A temporary variable is valid only for the SQL statement in which it is used—it doesn’t persist.

Image **Defined variables** A defined variable persists until you explicitly remove it, redefine it, or exit SQL\*Plus.

You’ll learn how to use these types of variables in this section.

**Temporary Variables**

You define a temporary variable using the ampersand character (&) in an SQL statement, followed by the name you want to call your variable. For example, &v\_product\_id defines a variable named v\_product\_id.

When you run the following query, SQL\*Plus prompts you to enter a value for v\_product\_id and then uses that value in the WHERE clause. If you enter the value 2 for v\_product\_id, the details for product #2 will be displayed.

SQL> **SELECT product\_id, name, price**

2 **FROM products**

3 **WHERE product\_id = &v\_product\_id;**

Enter value for v\_product\_id: **2**

old 3: WHERE product\_id = &v\_product\_id

new 3: WHERE product\_id = 2

PRODUCT\_ID NAME PRICE

---------- ------------------------------ ----------

2 Chemistry 30

Notice that SQL\*Plus does the following:

Image Prompts you to enter a value for v\_product\_id.

Image Substitutes the value you entered for v\_product\_id in the WHERE clause.

SQL\*Plus shows you the substitution in the old and new lines in the output, along with the line number in the query where the substitution was performed. In the previous example, you can see that the old and new lines indicate that v\_product\_id is set to 2 in the WHERE clause.

|  |
| --- |
| **Why Are Variables Useful?** |
| Variables are useful because they allow you to create scripts that a user who doesn’t know SQL can run. Your script would prompt the user to enter the value for a variable and use that value in an SQL statement. Let’s take a look at an example. |
| Suppose you wanted to create a script for a user who doesn’t know SQL, but who wants to see the details of a single specified product in the store. To do this, you could hard code the product\_id value in the WHERE clause of a query and place it in an SQL\*Plus script. For example, the following query retrieves product #1: |
| SELECT product\_id, name, price FROM products WHERE product\_id = 1; |
| This query works, but it only retrieves product #1. What if you wanted to change the product\_idvalue to retrieve a different row? You could modify the script, but this would be tedious. Wouldn’t it be great if you could supply a variable for the product\_id? You can do that using a substitution variable. |

If you rerun the query using the slash character (/), SQL\*Plus will ask you to enter a new value for v\_product\_id. For example:

SQL> **/**

Enter value for v\_product\_id: **3**

old 3: WHERE product\_id = &v\_product\_id

new 3: WHERE product\_id = 3

PRODUCT\_ID NAME PRICE

---------- ------------------------------ ----------

3 Supernova 25.99

Once again, SQL\*Plus echoes the old line of the SQL statement (old 3: WHERE product\_id = &v\_product\_id), followed by the new line containing the variable value you entered (new 3: WHERE product\_id = 3).

**Controlling Output Lines**

You may control the output of the old and new lines using the SET VERIFY command. If you enter SET VERIFY OFF, the old and new lines are suppressed. For example:

SQL> **SET VERIFY OFF**

SQL> **/**

Enter value for v\_product\_id: **4**

PRODUCT\_ID NAME PRICE

---------- ------------------------------ ----------

4 Tank War 13.95

To turn the echoing of the lines back on, you enter SET VERIFY ON. For example:

SQL> **SET VERIFY ON**

**Changing the Variable Definition Character**

You can use the SET DEFINE command to specify a character other than an ampersand (&) for defining a variable. The following example shows how to set the variable character to the pound character (#) and shows a new query:

SQL> **SET DEFINE '#'**

SQL> **SELECT product\_id, name, price**

2 **FROM products**

3 **WHERE product\_id = #v\_product\_id;**

Enter value for v\_product\_id: **5**

old 3: WHERE product\_id = #v\_product\_id

new 3: WHERE product\_id = 5

PRODUCT\_ID NAME PRICE

---------- ------------------------------ ----------

5 Z Files 49.99

The next example uses SET DEFINE to change the character back to an ampersand:

SQL> **SET DEFINE '&'**

**Substituting Table and Column Names Using Variables**

You can also use variables to substitute the names of tables and columns. For example, the following query defines variables for a column name (v\_col), a table name (v\_table), and a column value (v\_val):

SQL> **SELECT name, &v\_col**

2 **FROM &v\_table**

3 **WHERE &v\_col = &v\_val;**

Enter value for v\_col: **product\_type\_id**

old 1: SELECT name, &v\_col

new 1: SELECT name, product\_type\_id

Enter value for v\_table: **products**

old 2: FROM &v\_table

new 2: FROM products

Enter value for v\_col: **product\_type\_id**

Enter value for v\_val: **1**

old 3: WHERE &v\_col = &v\_val

new 3: WHERE product\_type\_id = 1

NAME PRODUCT\_TYPE\_ID

------------------------------ ---------------

Modern Science 1

Chemistry 1

You can avoid having to repeatedly enter a variable by using &&. For example:

SQL> **SELECT name, &&v\_col**

2 **FROM &v\_table**

3 **WHERE &&v\_col = &v\_val;**

Enter value for v\_col: **product\_type\_id**

old 1: SELECT name, &&v\_col

new 1: SELECT name, product\_type\_id

Enter value for v\_table: **products**

old 2: FROM &v\_table

new 2: FROM products

Enter value for v\_val: **1**

old 3: WHERE &&v\_col = &v\_val

new 3: WHERE product\_type\_id = 1

NAME PRODUCT\_TYPE\_ID

------------------------------ ---------------

Modern Science 1

Chemistry 1

Variables give you a lot of flexibility in writing queries that another user may run. You can give the user a script and have them enter the variable values.

**Defined Variables**

You can define a variable prior to using it in an SQL statement. You may use these variables multiple times within an SQL statement. A defined variable persists until you explicitly remove it, redefine it, or exit SQL\*Plus.

You define a variable using the DEFINE command. When you’re finished with your variable, you remove it using UNDEFINE. You’ll learn about these commands in this section. You’ll also learn about the ACCEPTcommand, which allows you to define a variable and set its data type.

You can also define variables in an SQL\*Plus script and pass values to those variables when the script is run. This feature enables you to write generic reports that any user can run—even if they’re unfamiliar with SQL. You’ll learn how to create simple reports later in the section "Creating Simple Reports."

**Defining and Listing Variables Using the DEFINE Command**

You use the DEFINE command to both define a new variable and list the currently defined variables. The following example defines a variable named v\_product\_id and sets its value to 7:

SQL> **DEFINE v\_product\_id = 7**

You can view the definition of a variable using the DEFINE command followed by the name of the variable. The following example displays the definition of v\_product\_id:

SQL> **DEFINE v\_product\_id**

DEFINE V\_PRODUCT\_ID = "7" (CHAR)

Notice that v\_product\_id is defined as a CHAR variable.

You can see all your session variables by entering DEFINE on its own. For example:

SQL> **DEFINE**

DEFINE \_DATE = "12-AUG-07" (CHAR)

DEFINE \_CONNECT\_IDENTIFIER = "Oracle11g" (CHAR)

DEFINE \_USER = "STORE" (CHAR)

DEFINE \_PRIVILEGE = "" (CHAR)

DEFINE \_SQLPLUS\_RELEASE = "1101000400" (CHAR)

DEFINE \_EDITOR = "Notepad" (CHAR)

DEFINE \_O\_VERSION = "Oracle Database 11g …" (CHAR)

DEFINE \_O\_RELEASE = "1101000500" (CHAR)

DEFINE \_RC = "0" (CHAR)

DEFINE V\_PRODUCT\_ID = "7" (CHAR)

You can use a defined variable to specify an element such as a column value in an SQL statement. For example, the following query uses references v\_product\_id in the WHERE clause:

SQL> **SELECT product\_id, name, price**

2 **FROM products**

3 **WHERE product\_id = &v\_product\_id;**

old 3: WHERE product\_id = &v\_product\_id

new 3: WHERE product\_id = 7

PRODUCT\_ID NAME PRICE

---------- ------------------------------ ----------

7 Space Force 9 13.49

You’re not prompted for the value of v\_product\_id; that’s because v\_product\_id was set to 7 when the variable was defined earlier.

**Defining and Setting Variables Using the ACCEPT Command**

The ACCEPT command waits for a user to enter a value for a variable. You can use the ACCEPT command to set an existing variable to a new value or to define a new variable and initialize it with a value. The ACCEPTcommand also allows you to specify the data type for the variable. The simplified syntax for the ACCEPTcommand is as follows:

ACCEPT *variable\_name* [*type*] [FORMAT *format*] [PROMPT *prompt*] [HIDE]

where

Image *variable\_name* is the variable name.

Image *type* is the data type for the variable. You can use the CHAR, NUMBER, and DATE types. By default, variables are defined using the CHAR type. DATE variables are actually stored as CHAR variables.

Image *format* is the format used for the variable. Some examples include A15 (15 characters), 9999 (a four-digit number), and DD-MON-YYYY (a date). You can view the number formats in [Table 4-4](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch04.html#table_4-4) of [Chapter 4](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch04.html#ch04); you can view the date formats in [Table 5-2](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch05.html#table_5-2) of [Chapter 5](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch05.html#ch05).

Image *prompt* is the text displayed by SQL\*Plus as a prompt to the user to enter the variable’s value.

Image HIDE means hide the value as it is entered. For example, you might want to hide passwords or other sensitive information.

Let’s take a look at some examples of the ACCEPT command. The following example defines a variable named v\_customer\_id as a two-digit number:

SQL> **ACCEPT v\_customer\_id NUMBER FORMAT 99 PROMPT 'Customer id: '**

Customer id: **5**

The next example defines a DATE variable named v\_date; the format is DD-MON-YYYY:

SQL> **ACCEPT v\_date DATE FORMAT ’DD-MON-YYYY' PROMPT ’Date: '**

Date: **12-DEC-2006**

The next example defines a CHAR variable named v\_password; the value entered is hidden using HIDE:

SQL> **ACCEPT v\_password CHAR PROMPT 'Password: ' HIDE**

Password:

In Oracle Database 9*i* and below, the value appears as a string of asterisk characters (\*) to hide the value as you enter it. In Oracle Database 10*g* and above, nothing is displayed as you type the value.

You can view your variables using the DEFINE command. For example:

SQL> **DEFINE**

…

DEFINE V\_CUSTOMER\_ID = 5 (NUMBER)

DEFINE V\_DATE = "12-DEC-2006" (CHAR)

DEFINE V\_PASSWORD = "1234567" (CHAR)

DEFINE V\_PRODUCT\_ID = "7" (CHAR)

Notice that v\_date is stored as a CHAR.

**Removing Variables Using the UNDEFINE Command**

You remove variables using the UNDEFINE command. The following example uses UNDEFINE to remove v\_customer\_id, v\_date, v\_password, and v\_product\_id:

SQL> **UNDEFINE v\_customer\_id**

SQL> **UNDEFINE v\_date**

SQL> **UNDEFINE v\_password**

SQL> **UNDEFINE v\_product\_id**

Image

**NOTE**  
*All your variables are removed when you exit SQL\*Plus, even if you don’t explicitly remove them using the*UNDEFINE *command*.

**CREATING SIMPLE REPORTS**

You can use variables in an SQL\*Plus script to create reports that a user can run. You’ll find the SQL\*Plus scripts referenced in this section in the SQL directory.

Image

**TIP**  
*SQL\*Plus was not specifically designed as a full-fledged reporting tool. If you have complex reporting requirements, you should use software like Oracle Reports*.

**Using Temporary Variables in a Script**

The following report1.sql script uses a temporary variable named v\_product\_id in the WHERE clause of a query:

-- suppress display of the statements and verification messages

SET ECHO OFF

SET VERIFY OFF

SELECT product\_id, name, price

FROM products

WHERE product\_id = &v\_product\_id;

The SET ECHO OFF command stops SQL\*Plus from displaying the SQL statements and commands in the script. SET VERIFY OFF suppresses display of the verification messages. I put these two commands in to minimize the number of extra lines displayed by SQL\*Plus when you run the script.

You can run report1.sql in SQL\*Plus using the @ command. For example:

SQL> **@ C:\sql\_book\SQL\report1.sql**

Enter value for v\_product\_id: **2**

PRODUCT\_ID NAME PRICE

---------- ------------------------------ ----------

2 Chemistry 30

You’ll need to replace the directory in the example with the directory where you saved the files for this book. Also, if you have spaces in the directory, you’ll need to put everything after the @ command in quotes; for example:

@ "C:\my directory\sql book\SQL\report1.sql"

**Using Defined Variables in a Script**

The following report2.sql script uses the ACCEPT command to define a variable named v\_product\_id:

SET ECHO OFF

SET VERIFY OFF

ACCEPT v\_product\_id NUMBER FORMAT 99 PROMPT 'Enter product id: '

SELECT product\_id, name, price

FROM products

WHERE product\_id = &v\_product\_id;

-- clean up

UNDEFINE v\_product\_id

Notice that a user-friendly prompt is specified for the entry of v\_product\_id and that v\_product\_id is removed at the end of the script—doing this makes the script cleaner.

You can run the report2.sql script using SQL\*Plus:

SQL> **@ C:\sql\_book\SQL\report2.sql**

Enter product id: **4**

PRODUCT\_ID NAME PRICE

---------- ------------------------------ ----------

4 Tank War 13.95

**Passing a Value to a Variable in a Script**

You can pass a value to a variable when you run your script. When you do this, you reference the variable in the script using a number. The following script report3.sql shows an example of this; notice that the variable is identified using &1:

SET ECHO OFF

SET VERIFY OFF

SELECT product\_id, name, price

FROM products

WHERE product\_id = &1;

When you run report3.sql, you supply the variable’s value after the script name. The following example passes the value 3 to report3.sql:

SQL> **@ C:\sql\_book\SQL\report3.sql 3**

PRODUCT\_ID NAME PRICE

---------- ------------------------------ ----------

3 Supernova 25.99

If you have spaces in the directory where you saved the scripts, you’ll need to put the directory and script name in quotes; for example:

@ "C:\my directory\sql book\SQL\report3.sql" 3

You can pass any number of parameters to a script, with each value corresponding to the matching number in the script. The first parameter corresponds to &1, the second to &2, and so on. The following report4.sql script shows an example with two parameters:

SET ECHO OFF

SET VERIFY OFF

SELECT product\_id, product\_type\_id, name, price

FROM products

WHERE product\_type\_id = &1

AND price > &2;

The following example run of report4.sql shows the addition of two values for &1 and &2, which are set to 1 and 9.99, respectively:

SQL> **@ C:\sql\_book\SQL\report4.sql 1 9.99**

PRODUCT\_ID PRODUCT\_TYPE\_ID NAME PRICE

---------- --------------- ------------------------------ ----------

1 1 Modern Science 19.95

2 1 Chemistry 30

Because &1 is set to 1, the product\_type\_id column in the WHERE clause is set to 1. Also, because &2 is set to 9.99, the price column in the WHERE clause is set to 9.99. Therefore, rows with a product\_type\_id of 1 and a price greater than 9.99 are displayed.

**Adding a Header and Footer**

You add a header and footer to your report using the TTITLE and BTITLE commands. The following is an example TTITLE command:

TTITLE LEFT 'Run date: ' \_DATE CENTER 'Run by the ' SQL.USER ' user'

RIGHT 'Page: ' FORMAT 999 SQL.PNO SKIP 2

The following list explains the contents of this command:

Image \_DATE displays the current date.

Image SQL.USER displays the current user.

Image SQL.PNO displays the current page (FORMAT is used to format the number).

Image LEFT, CENTER, and RIGHT justify the text.

Image SKIP 2 skips two lines.

If the example is run on August 12, 2007 by the store user, it displays

Run date: 12-AUG-07 Run by the STORE user Page: 1

The next example shows a BTITLE command:

BTITLE CENTER 'Thanks for running the report' RIGHT 'Page: ' FORMAT 999 SQL.PNO

This command displays

Thanks for running the report Page: 1

The following report5.sql script contains the TTITLE and BTITLE commands:

TTITLE LEFT 'Run date: ' \_DATE CENTER 'Run by the ' SQL.USER ' user'

RIGHT 'Page: ' FORMAT 999 SQL.PNO SKIP 2

BTITLE CENTER 'Thanks for running the report' RIGHT 'Page: '

FORMAT 999 SQL.PNO

SET ECHO OFF

SET VERIFY OFF

SET PAGESIZE 30

SET LINESIZE 70

CLEAR COLUMNS

COLUMN product\_id HEADING ID FORMAT 99

COLUMN name HEADING 'Product Name' FORMAT A20 WORD\_WRAPPED

COLUMN description HEADING Description FORMAT A30 WORD\_WRAPPED

COLUMN price HEADING Price FORMAT $99.99

SELECT product\_id, name, description, price

FROM products;

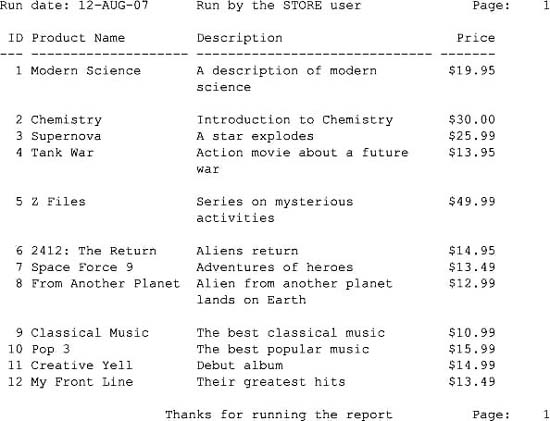
CLEAR COLUMNS

TTITLE OFF

BTITLE OFF

The last two lines switch off the header and footer set by the TTITLE and BTITLE commands. The following example shows a run of report5.sql:

SQL> **@ C:\sql\_book\SQL\report5.sql**



**Computing Subtotals**

You can add a subtotal for a column using a combination of the BREAK ON and COMPUTE commands. BREAK ON causes SQL\*Plus to break up output based on a change in a column value, and COMPUTE causes SQL\*Plus to compute a value for a column.

The following report6.sql script shows how to compute a subtotal for products of the same type:

BREAK ON product\_type\_id

COMPUTE SUM OF price ON product\_type\_id

SET ECHO OFF

SET VERIFY OFF

SET PAGESIZE 50

SET LINESIZE 70

CLEAR COLUMNS

COLUMN price HEADING Price FORMAT $999.99

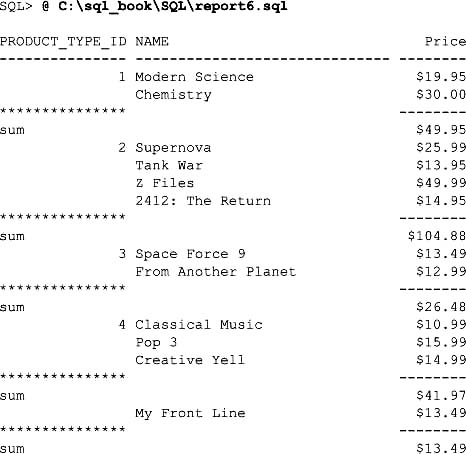
SELECT product\_type\_id, name, price

FROM products

ORDER BY product\_type\_id;

CLEAR COLUMNS

The following example shows a run of report6.sql:



Notice that whenever a new value for product\_type\_id is encountered, SQL\*Plus breaks up the output and computes a sum for the price columns for the rows with the same product\_type\_id. The product\_type\_id value is shown only once for rows with the same product\_type\_id. For example, "Modern Science" and "Chemistry" are both books and have a product\_type\_id of 1, and 1 is shown once for "Modern Science." The sum of the prices for these two books is $49.95. The other sections of the report contain the sum of the prices for products with different product\_type\_id values.

**GETTING HELP FROM SQL\*PLUS**

You can get help from SQL\*Plus using the HELP command. The following example runs HELP:

SQL> **HELP**

HELP

----

Accesses this command line help system. Enter HELP INDEX or ? INDEX

for a list of topics. In iSQL\*Plus, click the Help button to display

iSQL\*Plus online help.

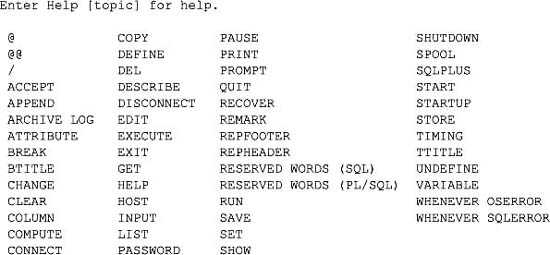
You can view SQL\*Plus resources at http://otn.oracle.com/tech/sql\_plus/

and the Oracle Database Library at http://otn.oracle.com/documentation/

HELP|? [topic]

The next example runs HELP INDEX:

SQL> **HELP INDEX**



The following example runs HELP EDIT:

SQL> **HELP EDIT**

EDIT

----

Invokes an operating system text editor on the contents of the

specified file or on the contents of the SQL buffer. The buffer

has no command history list and does not record SQL\*Plus commands.

ED[IT] [file\_name[.ext]]

Not available in iSQL\*Plus

**AUTOMATICALLY GENERATING SQL STATEMENTS**

In this section, I’ll briefly show you a technique of writing SQL statements that produce other SQL statements. This capability is very useful and can save you a lot of typing when writing SQL statements that are similar. One simple example is an SQL statement that produces DROP TABLE statements, which remove tables from a database. The following query produces a series of DROP TABLE statements that drop the tables from the store schema:

**SELECT ’DROP TABLE ' || table\_name || ';'**

**FROM user\_tables;**

’DROPTABLE'||TABLE\_NAME||';'

------------------------------------------

DROP TABLE COUPONS;

DROP TABLE CUSTOMERS;

DROP TABLE EMPLOYEES;

DROP TABLE PRODUCTS;

DROP TABLE PRODUCT\_TYPES;

DROP TABLE PROMOTIONS;

DROP TABLE PURCHASES;

DROP TABLE PURCHASES\_TIMESTAMP\_WITH\_TZ;

DROP TABLE PURCHASES\_WITH\_LOCAL\_TZ;

DROP TABLE PURCHASES\_WITH\_TIMESTAMP;

DROP TABLE SALARY\_GRADES;

Image

**NOTE**  
user\_tables *contains the details of the tables in the user’s schema. The* table\_name *column contains names of the tables*.

You can spool the generated SQL statements to a file and run them later.

**DISCONNECTING FROM THE DATABASE AND EXITING SQL\*PLUS**

You can disconnect from the database and keep SQL\*Plus running by entering DISCONNECT (SQL\*Plus also automatically performs a COMMIT for you). 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 reconnect to a database by entering CONNECT.

To end SQL\*Plus, you enter EXIT (SQL\*Plus also automatically performs a COMMIT for you).

**SUMMARY**

In this chapter, you learned how to do the following:

Image View the structure of a table

Image Edit an SQL statement

Image Save, retrieve, and run files containing SQL and SQL\*Plus commands

Image Format the results returned by SQL\*Plus

Image Set the page and line size for SQL\*Plus output

Image Use variables in SQL\*Plus

Image Create simple reports

Image Get help from SQL\*Plus

Image Write SQL statements that generate other SQL statements

Image Disconnect from the database and exit SQL\*Plus