CHAPTER 7

SQL*Plus and Server Manager

In This Chapter

What is SQL*Plus?

What is Server Manager?

Using SQL*Plus to enter and execute SQL statements

Using SQL*Plus to generate reports

Using SQL*Plus to write scripts

SQL*Plus and Server Manager are two popular Oracle utilities that DBAs use. Both tools have similar interfaces, and both tools can be used to perform pretty much the same function. This chapter focuses on the use of SQL*Plus because of its superior functionality, and also because Server Manager is on its way out. Server Manager used to have a few capabilities that SQL*Plus didn't have, notably the ability to start and stop a database. With the release of Oracle8i, Oracle has folded these functions into SQL*Plus and has further indicated that support for Server Manager will be dropped in a future release of Oracle.

What Is SQL*Plus?

SQL*Plus is an interactive tool that allows you to type and execute ad-hoc SQL statements and PL/SQL blocks. SQL*Plus also allows you to run scripts and generate simple reports. Beginning with the Oracle8i release, you can use SQL*Plus to start and stop a database or to recover a database. SQL*Plus is frequently used to query the data dictionary, to execute DDL commands, and sometimes just to query a table and see what's out there.

SQL*Plus has been a part of Oracle's software distribution almost from the beginning. Originally a command-line utility, SQL*Plus has been dressed up a bit for Windows platforms, where it is also available in a command-line–oriented GUI interface. SQL*Plus is also used by Oracle Enterprise Manager's SQLPlus Worksheet tool as the underlying engine for executing the commands that you enter. You would never know it to look at it, but underneath that slick-looking, Java-based GUI interface lies good old command-line SQL*Plus.

What Is Server Manager?

Server Manager is a command-line tool that allows you to perform administrative tasks on an Oracle database. These tasks include starting an instance, stopping an instance, and recovering a database. Oracle releases 7.3 and 8.0 used Server Manager to run a large number of administrative scripts that shipped with the Oracle software.

The Server Manager interface is similar to the SQL*Plus interface. At first glance, they both appear to work the same. However, Server Manager doesn't have the built-in line-editing capabilities of SQL*Plus, and Server Manager supports only a small subset of SQL*Plus's formatting functionality.

Since the list of Server Manager-specific functions is so small, and since the interface is already so similar to SQL*Plus, Oracle decided that it simply wasn't worth the trouble to maintain two sets of code. The Oracle8i release of SQL*Plus (release 8.1.5.0.0) now allows you to perform functions such as starting and stopping an instance that previously could only be accomplished using Server Manager. Server Manager is still around because many sites depend on it, but its days are numbered.

Comparing SQL*Plus and Server Manager

SQL*Plus and Server Manager differ in several ways. Table 7-1 lists these differences and highlights how the functionality of SQL*Plus has been enhanced to absorb Server Manager.

Table 7-1
SQL*Plus vs. Server Manager

Function	*SQL*Plus 8.1.x*	*SQL*Plus Prior to 8.1*	*Server Manager*
Start and stop a database	Yes	No	Yes
Recover a database, tablespace, or datafile	Yes	No	Yes
Issue `ARCHIVE LOG` commands	Yes	No	Yes
Run Oracle-supplied scripts such as `CATPROC` and `CATALOG`	Yes	No	Yes

Function	*SQL*Plus 8.1.x*	*SQL*Plus Prior to 8.1*	*Server Manager*
Execute ad-hoc queries	Yes	Yes	Yes
Format output (placing commas in numbers and so forth)	Yes	Yes	No
Format reports (column headings, page titles, and so forth)	Yes	Yes	No
Execute SQL*Plus scripts	Yes	Yes	No
Connect as SYSDBA or SYSOPER	Yes	No	Yes
Support SET commands	Yes	Yes	Very limited
Prompt a user for input	Yes	Yes	No
Use line-editing commands to edit SQL statements and PL/SQL blocks	Yes	Yes	No

You can see in Table 7-1 that SQL*Plus has the edge when it comes to formatting data and creating reports. SQL*Plus also has the edge in terms of user interaction. It can display messages, prompt a user for input, and implement a rudimentary set of line-editing commands that are useful for correcting mistakes. The only reason to use Server Manager, really, is to start, stop, and recover the database.

Converting Server Manager scripts to SQL*Plus

Many DBAs have written scripts for Server Manager. Since Server Manager is fading away, sooner or later, every Server Manager script will need to be converted to run under SQL*Plus. At first glance, this seems like a simple task, and it almost is. However, when converting from one to the other, you have to take into account the following differences in how the two products work:

- Server Manager allows comment lines to begin with the pound-sign (#) character; SQL*Plus does not.

- ✦ Server Manager allows blank lines within SQL statements; SQL*Plus does not.
- ✦ SQL*Plus uses the ampersand (&) character to mark substitution variables; Server Manager does not support substitution variables.
- ✦ SQL*Plus supports the use of the hyphen (-) character as a line continuation character; Server Manager does not.
- ✦ SQL*Plus requires that the `CREATE TYPE` and `CREATE LIBRARY` commands be terminated by a forward-slash (/) on a line by itself. Server Manager does not require this.

When you convert a Server Manager script to SQL*Plus, you have to work through each one of these issues.

Comment Lines

Server Manager allows you to mark a line as a comment by preceding it with a pound-sign (#) character. Here's an example:

```
# This is a comment in Server Manager
```

Nothing will happen when you type this comment line into Server Manager. Type the same line into SQL*Plus, however, and you will receive an Unknown Command error. To mark a line as a comment in SQL*Plus, precede it with either the `REM` command or by a double-hyphen (`--`). Consider this example:

```
REM This is a comment in SQL*Plus
-- This is also a comment in SQL*Plus
```

When converting a script from Server Manager to SQL*Plus, check your comments and change any that start with the pound sign (#) so that they start with `REM` or `--` instead.

Blank Lines

Both Server Manager and SQL*Plus allow you to enter SQL statements that span multiple lines. Server Manager, however, allows you to include blank lines in a SQL statement, while SQL*Plus does not. Here's an example:

```
SVRMGR> SELECT *
     2>
     3> FROM dual;
D
-
X
1 row selected.
```

Try to enter that same statement, including the blank line, into SQL*Plus, and here's what will happen:

```
SQL> SELECT *
```

```
  2
SQL> FROM dual;
SP2-0042: unknown command "FROM DUAL" - rest of line ignored.
```

What's happening here? SQL*Plus recognizes a blank line as the end of a statement, so the blank line after `SELECT *` causes SQL*Plus to terminate entry of the statement, leaving it in the buffer where you can edit it using line-editing commands. As far as SQL*Plus is concerned, the line `FROM dual;` is a new command. Because it terminates with a semicolon, SQL*Plus attempts to execute it. The `FROM dual;` command, of course, is not a valid statement, so the result is an error message.

Oracle does provide some relief from this problem through the use of the `SQLBLANKLINES` setting. This setting is a new feature added to SQL*Plus in the 8.1.5 release, specifically to make it easier to convert Server Manager scripts. The following command causes SQL*Plus to allow blank lines in SQL statements:

```
SET SQLBLANKLINES ON
```

Here's an example showing how the `SET SQLBLANKLINES ON` command allows the same command that failed earlier to run successfully:

```
SQL> SET SQLBLANKLINES ON
SQL> SELECT *
  2
  3  FROM dual;

D
-
X

1 row selected.
```

If you have a Server Manager script to convert, and you don't want to go through and remove blank lines from SQL statements, place a `SET SQLBLANKLINES ON` command at the beginning of the script.

If you use `SET SQLBLANKLINES ON` at the beginning of a script, you may want to add `SET SQLBLANKLINES OFF` to the end. That way, if you run several scripts interactively in one SQL*Plus session, you won't inadvertently cause subsequent scripts to fail.

Substitution Variables

To enable you to write generic scripts that you can run multiple times using different input values, SQL*Plus allows you to embed substitution variables in SQL statements. When SQL*Plus encounters one of these variables, it stops and prompts for a value before continuing. Consider this example:

```
SQL> SELECT animal_name
  2  FROM aquatic_animal
  3  WHERE id_no=&animal_id;
```

```
Enter value for animal_id: 100
old   3: WHERE id_no=&animal_id
new   3: WHERE id_no=100

ANIMAL_NAME
------------------------------
Flipper

1 row selected.
```

Notice how SQL*Plus prompted for an animal ID number when it encountered the `&animal_id` variable. Server Manager doesn't support this feature at all. At first that may not seem like a problem when converting away from Server Manager, but it actually is. Because Server Manager doesn't support substitution, you can place ampersands (`&`) in Server Manager scripts with impunity. The following statement will run just fine in Server Manager:

```
SELECT tank_no
FROM tank
WHERE tank_name = 'Blue & Red';
```

Try this same statement in SQL*Plus, however, and you suddenly find yourself being prompted for the value of red:

```
SQL> SELECT tank_no
  2  FROM tank
  3  WHERE tank_name = 'Blue & Red';
Enter value for red:
```

A relatively easy fix for this, if you don't want to hunt through your Server Manager script looking for ampersands, is to place the following commands at the beginning and end of your script:

```
SET DEFINE OFF
...
SET DEFINE ON
```

The command `SET DEFINE OFF` turns off the SQL*Plus substitution feature.

Continuation Characters

The use of the hyphen as a continuation character in SQL*Plus presents a subtle problem. Since Server Manager doesn't support continuation characters at all, you might think you don't need to worry about this. In fact, you do. The following SQL statement illustrates the problem:

```
SELECT 10 -
       3 FROM dual;
```

Server Manager will interpret this as a subtraction and will return a value of 7. SQL*Plus will interpret the hyphen as a continuation character, will combine the two lines into one, and will return an error, as shown here:

```
SQL> SELECT 10 -
>        3 FROM dual;
SELECT 10         3 FROM dual
                  *
ERROR at line 1:
ORA-00923: FROM keyword not found where expected
```

Unfortunately, no easy fix exists. It's a very subtle problem. You won't encounter it often, because it's rare for anyone to break up a subtraction by leaving the minus sign at the end of a line. Your only recourse here is to manually scan your Server Manager scripts, look for this problem, and fix it when you find it.

The CREATE TYPE and CREATE LIBRARY Commands

The last issue that you need to worry about is the one involving the `CREATE TYPE` and `CREATE LIBRARY` commands. Server Manager will allow these commands to be terminated by a semicolon, as in this example:

```
CREATE TYPE PERSON AS OBJECT (
    NAME VARCHAR2(30)
);
```

SQL*Plus, on the other hand, requires that these commands be terminated with a forward slash on a line by itself. For example:

```
CREATE TYPE PERSON AS OBJECT (
    NAME VARCHAR2(30)
);
/
```

The forward slash must be the first character on the line, and the semicolon at the end of the type definition is still required. There is no magic solution to this problem. If you convert a script containing `CREATE TYPE` or `CREATE LIBRARY` commands from Server Manager to SQL*Plus, you have to go in and add the forward-slash characters.

Using SQL*Plus to Enter and Execute SQL Statements

This section shows you how to start SQL*Plus and how to enter and execute SQL statements. You will see how you can edit those SQL statements, which can be quite helpful when you've mistyped something. Finally, you'll see how you can save a SQL statement to a file so that you can return to it later.

Starting SQL*Plus

How you start SQL*Plus depends on whether you are running the Windows platform, and, if you are running Windows, whether you want to run the GUI version. If you are running UNIX, or if you want to start SQL*Plus from a Windows command prompt, use the `sqlplus` command, as shown in this example:

```
C:\>sqlplus

SQL*Plus: Release 8.1.5.0.0 - Production on Wed Aug 4 18:02:53 1999

(c) Copyright 1999 Oracle Corporation.  All rights reserved.

Enter user-name:
```

If you are running Windows and you want to run the GUI version of SQL*Plus, click Start, point to Programs, point to Oracle – OraHome81, point to Application Development, and select SQL Plus. The SQL*Plus icon is shown in Figure 7-1.

Figure 7-1: The SQL*Plus icon

No matter how you start SQL*Plus, you will be prompted for a username and a password. The GUI version will also prompt you for a Net8 service name (the field title on the screen is actually Host String), so if you are connecting to a remote database, you may want to use that version.

Entering commands in SQL*Plus

How you enter a command in SQL*Plus depends partly on whether the command is a SQL statement, a PL/SQL block, or a command to SQL*Plus itself. For the most part, you just type commands. The differences are in how you terminate those commands.

Entering SQL Statements

To enter and execute a SQL statement, just type it and terminate it with a semicolon, as shown in this example:

```
SELECT * FROM dual;
```

You can also use a forward slash to terminate a SQL statement, as shown here:

```
SELECT * FROM dual
/
```

Don't mix the two methods in the same statement. Use either a semicolon or a forward slash, but not both. You can also terminate a SQL statement with a blank line. When you do that, the statement is not executed, it is simply held in the buffer of SQL*Plus, where you can edit it.

Entering PL/SQL Blocks

To enter and execute a PL/SQL block, you must terminate the block both with a semicolon and a forward slash, as shown in this example:

```
BEGIN
  NULL;
END;
/
```

Using both a semicolon and a forward slash together when PL/SQL is involved but not when SQL is involved doesn't represent an inconsistency in design. The semicolon is necessary because it is part of PL/SQL syntax. Because semicolons are used as statement terminators in PL/SQL, they can't be relied on to mark the end of a block — hence, the need for the forward slash.

You can enter a PL/SQL block without executing it, by terminating it with a period (.) instead of a forward slash. Consider this example:

```
BEGIN
  NULL;
END;
.
```

Using the period causes SQL*Plus to hold the block in the buffer for further editing.

Entering SQL*Plus Commands

In addition to accepting SQL statements and PL/SQL blocks, both of which are executed by the Oracle database software, SQL*Plus also recognizes a large number of commands. A good example is the `COLUMN` command, used to format output from a query:

```
COLUMN animal_name FORMAT A30
```

Commands like the `COLUMN` command have meaning only to SQL*Plus and are not passed on to the Oracle database. Because of that, these commands don't need any special terminator characters.

Sometimes, SQL*Plus commands can be quite long. This is especially true of the `TTITLE` and `BTITLE` commands that are used to define page headers and footers.

If you have a long command, SQL*Plus allows you to continue it across multiple lines by using the hyphen (-) as a continuation character. Consider this example:

```
TTITLE CENTER "IDG's Seapark" SKIP 1-
       LEFT "Animal Management" -
       RIGHT "Animal Feeding Report"
```

When you continue commands like this, you must have at least one space preceding the continuation character.

Editing your work in SQL*Plus

SQL*Plus contains a rudimentary set of editing commands. These commands allow you to edit SQL statements that you have entered into the buffer, and they allow you to save and restore SQL statements to and from files. The line-editing commands can be quite handy when you find that you've made a mistake typing a long SQL statement. If you don't like line editing, or if you have extensive changes to make, you can use the `EDIT` command to invoke an external editor such as vi (UNIX) or Notepad (Windows).

The Line-editing Commands

The line-editing commands that SQL*Plus implements are rudimentary but quite functional. They are useful for correcting small mistakes in SQL queries and PL/SQL blocks. The following list summarizes these commands:

- `L` — Lists the current SQL statement in the buffer.
- `L n` — Displays line n of the current SQL statement.
- `L n m` — Displays lines n–m of the current SQL statement.
- `C/x/y/` — Changes the first occurrence of x on a line into y. This command operates on the current line.
- `DEL` — Deletes the current line.
- `DEL n` — Deletes line n.
- `DEL n m` — Deletes lines n–m.
- `I` — Inserts a new line below the current line.
- `A text` — Appends text onto the end of the current line.
- `/` — Executes the statement in the buffer.

When you are editing a SQL statement, the concept of the current line is an important one to understand. Put simply, the *current line* is always the most recent line displayed by SQL*Plus. Unless you specify a line number, which you can do with the `L` and `DEL` commands, all line-editing commands operate on whichever line is current when you issue the commands. Read through the following example and commentary to get a feel for how the line-editing commands work:

A query is entered that has several mistakes. The result is an error message.

```
SQL> SELECT id_no, tank_num, animal_name
  2  Animal_parent
  3  FROM
  4  WHERE birth_date > SYSDATE;
WHERE birth_date > SYSDATE
*
ERROR at line 4:
ORA-00903: invalid table name


SQL>
```

The L command is used to list the first line so that a correction can be made.

```
SQL> L 1
  1* SELECT id_no, tank_num, animal_name
SQL>
```

The C command is used to change tank_num to tank_no.

```
SQL> C /tank_num/tank_no/
  1* SELECT id_no, tank_no, animal_name
SQL>
```

Line 2 is a mistake, and the DEL command is used to delete it.

```
SQL> DEL 2
SQL>
```

The buffer is relisted so that you can reorient yourself. Note that all the lines following the deleted line have been renumbered.

```
SQL> L
  1  SELECT id_no, tank_no, animal_name
  2  FROM
  3* WHERE birth_date > SYSDATE
SQL>
```

Line 2 is listed because it has the next and last error to be fixed.

```
SQL> L 2
  2* FROM
SQL>
```

The A command is used to append the table name onto the end of line 2.

```
SQL> A  aquatic_animal
  2* FROM aquatic_animal
SQL>
```

The command is listed one last time before it is executed.

```
SQL> L 1 3
  1  SELECT id_no, tank_no, animal_name
  2  FROM aquatic_animal
  3* WHERE birth_date > SYSDATE
SQL>
```

The forward slash is used to execute the command as it stands now.

```
SQL> /

no rows selected
```

Take time to experiment with the line-editing commands and become familiar with them. They may seem crude — they are crude — but they can still save you a lot of time.

The SAVE and GET Commands

SQL*Plus includes commands that allow you to save a statement from the buffer to a file and to load a statement from a file back into the buffer. The commands are SAVE and GET.

The SAVE command writes the current SQL statement (or PL/SQL block) from the buffer into a file. You have to supply a file name as an argument to SAVE, as shown in this example:

```
SQL> SELECT * FROM AQUATIC_ANIMAL
  2
SQL> SAVE c:\a\save_example.sql
Created file c:\a\save_example.sql
```

This SAVE command works if you are creating a new file. If you intend to overwrite an existing file, then you must use the REPLACE option, as shown here:

```
SAVE c:\a\save_example.sql REPLACE
```

The GET command does the reverse of SAVE. It loads a SQL statement from a file into the buffer where you can edit it. The following example shows an empty buffer and then demonstrates using the GET command to load a statement into that buffer:

```
SQL> L
SP2-0223: No lines in SQL buffer.
SQL> GET c:\a\save_example.sql
  1* SELECT * FROM AQUATIC_ANIMAL
```

Once you have loaded the SQL statement into the buffer, you can either execute it right away using the forward-slash command, or you can use the line-editing commands to change it first.

The EDIT Command

The SQL*Plus EDIT command allows you to invoke an external text editor to edit the statement in the buffer. On Windows platforms, the Notepad editor is used. Generally, you use these steps when using the EDIT command:

1. Type a SQL statement.
2. Invoke an editor by using the EDIT command.
3. Edit the statement in the editor.

4. Save the statement and exit the editor.
5. Use the forward slash (/) to execute the statement.

Figure 7-2 shows the results of entering a statement and then issuing the `EDIT` command on a Windows platform.

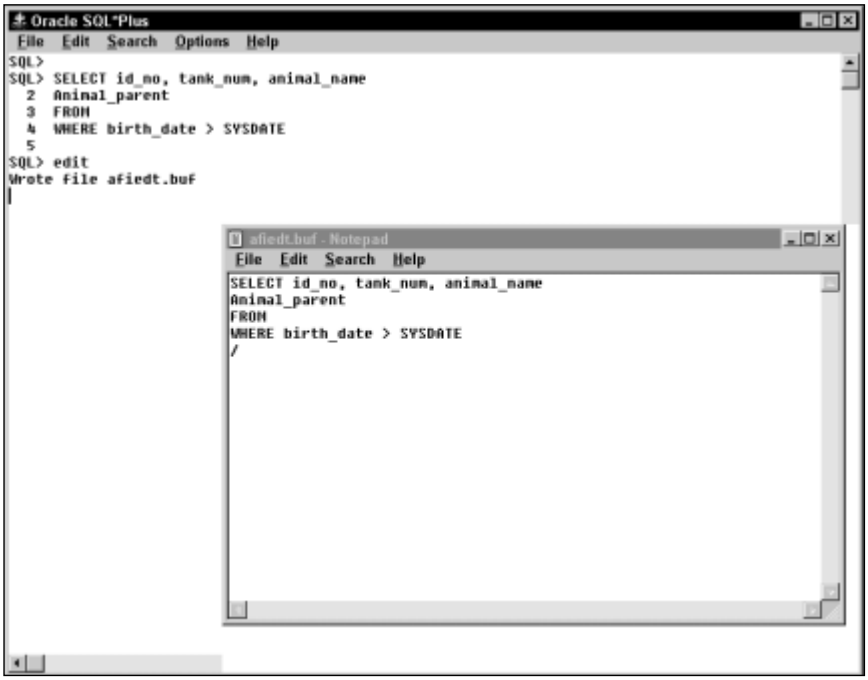


Figure 7-2: Using the SQL*Plus EDIT command

For large statements or those with major mistakes, it's often easier to edit them with an external editor than it is to use the line-editing commands. The advantage of using the internal editor is that it is always the same, and always available, regardless of the operating system being used.

Using SQL*Plus to Generate Reports

To its credit, SQL*Plus allows you to quickly create decent-looking ad-hoc reports. The SQL*Plus environment enables you to create column titles, page headers, and page footers, and even to prompt for variables. This section highlights the major steps to completing reports quickly.

Basic report commands

When writing a report with SQL*Plus, start by creating the query. Then you can add report features such as column headings, page titles, and so forth using SQL*Plus commands. This section shows you how to do the following basic reporting tasks:

- ✦ Modify the format and heading for a column
- ✦ Add a title to the report
- ✦ Write the formatted query results (that is, the report) to a file

The SQL*Plus `COLUMN`, `TTITLE`, `BTITLE`, and `SPOOL` commands are used for these tasks.

You can run all the report examples in this chapter if you install the sample schema from the CD-ROM. Appendix A contains instructions on how to install the sample schema into an Oracle8i database.

The COLUMN Command

The `COLUMN` command enables you to change the heading of a column, set the width of a column, and set the display format to use for a column. A simplified version of the syntax for the `COLUMN` command looks like this:

```
COLUMN columnname [HEADING headingtext]
       [FORMAT formattext] [WORD_WRAPPED|TRUNCATED]
```

Replace `columnname` with the actual column name from your query, and replace `headingtext` with your desired column heading. If the heading includes spaces, enclose it in double quotes (""). To make a heading that contains two lines, use the vertical bar (|) as a divider. For example, the following `COLUMN` command results in a heading with `Birth` and `Date` aligned vertically for the column named `BIRTH_DATE`.

```
COLUMN BIRTH_DATE HEADING "Birth|Date"
```

The resulting column heading looks like this:

```
Birth
Date
-----
```

The `FORMAT` clause, together with `formattext`, controls the display format used for the column. The `FORMAT` clause is most useful for numbers. For text columns, you can control only the display width. For numeric columns, you can control both the column width and the appearance of the numbers.

Table 7-2 shows examples of some format specifications. The column on the right shows a sample of some data formatted according to each specification.

Table 7-2
Formats in the SQL*Plus COLUMN Command

Datatype	*Sample Format*	*Sample Results*
Number	COLUMN x FORMAT 999,999.00	4,550.00
Number	COLUMN x FORMAT 000,000.00	004,550.00
Number	COLUMN x FORMAT $9999.99	$4550.00
Character	COLUMN y FORMAT A10	Supercalif
Character	COLUMN y FORMAT A20	Supercalifragilistic

For text columns, you can use `Axx`, where `xx` is a number to control the number of characters displayed in a column. Wrap text data within the column by adding the `WORD_WRAP` parameter to the `COLUMN` command. For example, the following `COLUMN` command allows the `MARKINGS_DESCRIPTION` column to wrap data:

```
COLUMN MARKINGS_DESCRIPTION FORMAT A10 WORD_WRAP
```

The resulting report will look like the one shown in this example:

```
SQL> COLUMN MARKINGS_DESCRIPTION FORMAT A10 WORD_WRAP
SQL> SELECT ANIMAL_NAME, MARKINGS_DESCRIPTION
  2  FROM AQUATIC_ANIMAL;

ANIMAL_NAME                    MARKINGS_D
------------------------------ ----------
Flipper                        Gray with
                               pink
                               tongue

Skipper                        Small scar
                               on right
                               fin
```

The `WORD_WRAPPED` setting is optional. If you don't specify it, then the lines in the column will be broken exactly at the column boundary, even if that means breaking the line in the middle of a word. If you don't want multiline columns, use the `TRUNCATE` option. This causes SQL*Plus to chop values that are longer than the column is wide, instead of wrapping them.

For numeric columns, you specify the format using a pattern of `9 0 , .` and `$` characters. The `9` and `0` characters control the number of digits to be displayed. Leading "0" characters cause the number to be displayed with leading zeros; otherwise, leading zeros are suppressed. A leading `$` can be used to print a dollar sign in front of dollar values. The `.` indicates the location of the decimal point, and the `,`s indicate how you want to use commas to separate groups of digits.

For date columns, you can't use SQL*Plus to define the display format for the date. Instead, you have to treat date columns as you would character columns, and you have to specify the date format in your SELECT statement by using Oracle's built-in TO_CHAR function. The TO_CHAR function has its own set of formatting characters. Table 7-3 shows the characters commonly used for dates.

See Appendix B, "SQL Built-in Function Reference," for a complete list.

Table 7-3
Date Conversion in the TO_CHAR(DATE) Function

Abbreviation	*Meaning*
DD	Day (01 through 31)
Day	Day of the week with initial letter capitalized (such as Saturday)
MM	Month (01 through 12)
Month	Month spelled out and initial letter capitalized
MON	First three letters of month in capital letters
YY	Year (00 through 99)
YYYY	Four-digit year (for example, 1999, 2000, and so on)
MI	Minute (00 through 59)
HH	Hour (01 through 12)
HH24	Hour (01 through 24)
SS	Second (00 through 59)

Watch out for the small but critical difference between MM (the month abbreviation) and MI (the minute abbreviation).

The following example demonstrates using the TO_CHAR function to format a date column in a report:

```
SQL> COLUMN animal_name FORMAT A11
SQL> COLUMN b_date FORMAT A11
SQL> SELECT animal_name,
  2         TO_CHAR(birth_date,'dd-Mon-yyyy') b_date
  3  FROM aquatic_animal;

ANIMAL_NAME BIRTH_DATE
----------- -----------
Flipper     01-Feb-1968
Skipper     01-Jan-1978
Bopper      11-Mar-1990
```

When you use the TO_CHAR function to format a date like this, be sure to provide a column alias in your SQL query. That's what gives the resulting column a name that you can work with. In this example, the alias is b_date. The column alias must match the name used with the COLUMN command.

The TTITLE and BTITLE Commands

SQL*Plus allows you to create page headers, or titles, for reports that you generate. You can also create page footers if you like. You use the TTITLE and BTITLE commands for these purposes. Title commands can get fairly long, and they contain a mixture of text strings and modifiers that control how and where those text strings print. Listing 7-1 demonstrates using the TTITLE command to place a multiline title at the top of a report:

Listing 7-1: Placing a title at the top of a report

```
SQL> SET LINESIZE 47
SQL> TTITLE CENTER "SeaPark" SKIP 1 -
>        LEFT "Animal Report" RIGHT "Page " -
>        FORMAT 999 SQL.PNO SKIP 3
SQL> COLUMN id_no FORMAT 999 HEADING "ID"
SQL> COLUMN animal_name FORMAT A30 HEADING "Name"
SQL> COLUMN birth_date FORMAT A11 HEADING "Birth Date" -
>           JUST RIGHT
SQL> SELECT id_no,
  2         animal_name,
  3         TO_CHAR(birth_date,'dd-Mon-yyyy') birth_date
  4  FROM aquatic_animal
  5  WHERE death_date IS NULL;

                    SeaPark
Animal Report                           Page    1


  ID Name                            Birth Date
---- ------------------------------ -----------
 100 Flipper                        01-Feb-1968
 105 Skipper                        01-Jan-1978
 112 Bopper                         11-Mar-1990
 151 Batty                          06-Jun-1996
 166 Shorty                         06-Jun-1996
 145 Squacky                        06-Jun-1996
 175 Paintuin                       14-May-1997
 199 Nosey                          05-Jan-1990
 202 Rascal                         01-Oct-1994

9 rows selected.

SQL>
```

Let's take a look at the TTITLE command used for this report. It is a rather long command that continues over three lines. The first line looks like this:

```
TTITLE CENTER "SeaPark" SKIP 1 -
```

The keyword CENTER causes subsequent text to be aligned to the center of the page. In this case, the subsequent text is "SeaPark", which did indeed print top and center. The SKIP 1 clause causes the title to advance one line. The second line of the TTITLE command is:

```
LEFT "Animal Report" RIGHT "Page " -
```

The keyword LEFT functions similarly to CENTER, except that it causes subsequent text to print aligned to the left of the page. After "Animal Report", the RIGHT keyword forces the page number to print flush right. The third line of the TTITLE command looks like this:

```
FORMAT 999 SQL.PNO SKIP 3
```

The FORMAT clause specifies a numeric format used for any numbers in the title that might follow this clause. In this case, the number being formatted is the page number. The SQL.PNO parameter is a special construct that SQL*Plus recognizes and replaces with the current page number. A final SKIP clause is used to advance three lines to allow some vertical space between the page title and the column headings.

The SKIP 3 clause results in only two blank lines. The report advances three lines, but the first advance simply gets it past the title line.

There are a couple of points worth examining about this report. First, the SET LINESIZE command sets the line width at 47. Why such an odd number? Because that's what it took to get the page number to print flush right with the right edge of the Birth Date column. The TTITLE command's CENTER and RIGHT clauses reference the linesize setting to determine exactly where the center and the right edge of a page are.

Also notice the use of the JUST RIGHT clause in the COLUMN command for the birth date data. This clause causes the column heading to print flush with the right edge of a character column, rather than the left, which looks better where dates are concerned.

The BTITLE command functions exactly like TTITLE. All the same clauses may be used. The only difference is that BTITLE defines a page footer that prints at the *bottom* of each page.

Titles and Dates

SQL*Plus makes it easy to get the page number into a report title; you just place SQL.PNO in the title where you want the page number to appear. Since it's so easy to get the page number, you might think that SQL*Plus would make it just as easy to get the date in the title. It doesn't.

Getting the current date into a report title requires that you use an arcane incantation of SQL*Plus commands and SQL statements. Essentially, you need to do the following:

1. Think up a name for a substitution variable that will hold the current date.
2. Place this substitution variable name in the TTITLE command where you want the date to appear. Do not use an ampersand here. SQL*Plus will replace the variable with its contents when it prints the page title.
3. Add a date column to your query. Use TO_CHAR to format the date the way you want it to appear.
4. Use the COLUMN command's NEW_VALUE clause to have SQL*Plus place the value of the date column into the substitution variable. Also use the NOPRINT clause so the date doesn't print as a column on the report.
5. Execute the query to produce the report. The date will appear in the report header.

The following code snippet provides an example of the technique just described:

```
TTITLE LEFT TODAYS_DATE CENTER 'The Animal Report' SKIP 2
COLUMN TODAYS_DATE NEW_VALUE TODAYS_DATE NOPRINT
SELECT TO_CHAR (SYSDATE, 'dd-Mon-yyyy') TODAYS_DATE,
       ID_NO,
       ANIMAL_NAME
FROM AQUATIC_ANIMAL
WHERE TANK_NO = 1;
```

When you execute this code, the results will look like this:

```
05-Aug-1999                    The Animal Report

    ID_NO ANIMAL_NAME
--------- ------------------------------
      100 Flipper
      105 Skipper
      112 Bopper

3 rows selected.
```

As you can see, the date shows up in the report header in the position indicated by the TODAYS_DATE substitution variable in the TTITLE command.

You can use this same technique to place other information into a title. Anything that you can query from your database, you can place into a substitution variable, which can then be used in a TTITLE or BTITLE command.

The SPOOL Command

The SQL*Plus SPOOL command is a useful command that writes the results of your query to a file. In fact, if you want to print a report generated with SQL*Plus, the only

way you can do it is to first spool it to a file. Later, you can copy the file to the printer to print the report. The syntax for the SPOOL command looks like this:

```
SPOOL filename
```

The filename argument may optionally include a path and an extension. If you don't specify an extension, SQL*Plus automatically adds one (usually .lis or .lst).

Once you issue the SPOOL command, SQL*Plus writes everything to the spool file that it writes to the display. It does this until you issue the following command to stop the spooling:

```
SPOOL OFF
```

Listing 7-2 demonstrates using the SPOOL command to send the output of a report to a file.

Listing 7-2: **Spooling a report to a file**

```
SET LINESIZE 47
TTITLE CENTER "SeaPark" SKIP 1 -
       LEFT "Animal Report" RIGHT "Page " -
       FORMAT 999 SQL.PNO SKIP 3
COLUMN id_no FORMAT 999 HEADING "ID"
COLUMN animal_name FORMAT A30 HEADING "Name"
COLUMN birth_date FORMAT A11 HEADING "Birth Date" -
       JUST RIGHT

SPOOL animal_report
SELECT id_no,
       animal_name,
       TO_CHAR(birth_date,'dd-Mon-yyyy') birth_date
FROM aquatic_animal
WHERE death_date IS NULL;
SPOOL OFF
```

Notice that the two SPOOL commands bracket the SELECT statement. There's no sense issuing the commands earlier, unless you also want your COLUMN commands to be written to the spool file.

Having written the report to the animal_report file, you can print the report by sending the file to a printer. On UNIX systems, you can easily do this using the lp command. On Windows systems, if you have a DOS printer mapped, you can copy the file to that printer. Otherwise, you can load the file into an editor such as Notepad or Microsoft Word, and print the file from there.

The BREAK and COMPUTE Commands

SQL*Plus provides a way to summarize and group sets of rows in a report. Using the BREAK and COMPUTE commands, you can make a report with details, breaks, and summaries.

The general syntax of the BREAK command follows:

```
BRE[AK] [ON report_element [action]
ON report_element [action]...]
```

The BREAK command is actually one of the more difficult SQL*Plus commands to master. Generally, you replace the *report_element* parameters with column names. Each report element in a BREAK command has an action associated with it. Whenever the value of one of the report elements changes, SQL*Plus executes the action that you specified for that element. Typical actions include suppressing repeating values, skipping one or more lines, or skipping to a new page.

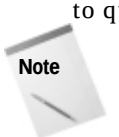
The following example shows one of the more common uses for the BREAK command, which is to suppress repeating values in a column. The query returns a list of animals and tank numbers that is sorted by tank number. Since all the animals for one tank are grouped together, the results will look better if the tank number is printed only once for each group. You use the BREAK command to accomplish this. Take a look at the example shown in Listing 7-3.

Listing 7-3: **Suppressing repeating values in a column**

```
SQL> BREAK ON tank_no NODUPLICATES SKIP 1
SQL> SELECT tank_no, id_no, animal_name
  2  FROM aquatic_animal
  3  ORDER BY tank_no, id_no;

  TANK_NO     ID_NO ANIMAL_NAME
--------- --------- ------------------------------
        1       100 Flipper
                105 Skipper
                112 Bopper

        2       145 Squacky
                151 Batty
                166 Shorty
                175 Paintuin

        3       199 Nosey
                202 Rascal
                240 Snoops


10 rows selected.
```

Notice that two break actions are listed for the tank_no column. The first action is NODUPLICATES, which causes the value of the tank_no column to print only when it changes. The result is that the tank number is printed only once per group.

The second action is SKIP 1, which tells SQL*Plus to skip a line. Whenever the value of the tank_no column changes, SQL*Plus executes both these actions. The result of SKIP 1 is that a blank line separates each group of animals, making it easy to quickly ascertain which animals are in which tank.



When you break on a column, you must also sort the report on the same column. In the previous example, the break column is also the first column listed in the SQL statement's ORDER BY clause. If your break columns and your ORDER BY columns don't correspond, your data won't be grouped in any rational fashion, and your report will look bad.

SQL*Plus can print summary information about the rows in the group. You use the COMPUTE command to do this. The COMPUTE command works in conjunction with the BREAK command, allowing you to print summary information on breaks. The general syntax of the COMPUTE command follows:

```
COMP[UTE] [function [LABEL] text OF
expression|column|alias  ON
expression | column | alias | REPORT | ROW]
```

The COMPUTE command is another difficult SQL*Plus command to master. Like the BREAK command, the COMPUTE command is most easily explained by using an example. Using the SEAPARK tables, you can create a report showing the number of animals each caretaker handles, listed by the caretaker name and then by the tank number. You can report a count of animals for each tank, each caretaker, and the entire report. To create this report, you will need three COMPUTE commands. The first causes SQL*Plus to report a count of records for the entire report, and it looks like this:

```
COMPUTE COUNT OF ID_NO ON REPORT
```

This command tells SQL*Plus to generate a count based on the ID_NO column. The ON REPORT clause generates that count for all the records in the report. You could actually count any non-null column. The ID_NO column is just a convenient choice because it is the table's primary key.

The next COMPUTE command looks like this:

```
COMPUTE COUNT OF ID_NO ON C_NAME
```

This COMPUTE command is similar to the previous one, except that it computes a count of records for each distinct value of C_NAME, or for each caretaker. For the results of this command to make sense, you must sort the report on the same value.

The last of the three COMPUTE commands is this:

```
COMPUTE COUNT OF ID_NO ON TANK_NO
```

This final command tells SQL*Plus to generate a count of records (animals) in each tank. When you are counting records for specific columns, as in this report, you must have the report sorted correctly. It's also important that you break on the same columns that you name in your COMPUTE commands. This report is computing totals for each caretaker and tank combination, and for each caretaker. The BREAK command used looks like this:

```
BREAK ON REPORT ON C_NAME SKIP 2 ON TANK_NO SKIP 1
```

The COMPUTE command tells SQL*Plus to generate summary information for a column, but it's the BREAK command that allows SQL*Plus to print that information. Notice that this BREAK command lists all the columns — REPORT, C_NAME, and TANK_NO — that were mentioned in the three COMPUTE statements.

The final requirement is to sort the query results to match the break order. The ORDER BY clause used in the query is:

```
ORDER BY T.CHIEF_CARETAKER_NAME, T.TANK_NO, A.ID_NO;
```

This ORDER BY clause sorts the report first on the caretaker's name, then within that on the tank number, and within that on the animal ID number. The leading columns in the ORDER BY clause match the column order in the BREAK command. That's important. If that order doesn't match, you'll get a mixed-up report. It's okay to have other columns in the ORDER BY clause, beyond those listed in the BREAK command, but they must come at the end.

The complete SQL*Plus script to generate this report looks like this:

```
COLUMN C_NAME FORMAT A15
BREAK ON REPORT ON C_NAME SKIP 2 ON TANK_NO SKIP 1
COMPUTE COUNT OF ID_NO ON REPORT
COMPUTE COUNT OF ID_NO ON C_NAME
COMPUTE COUNT OF ID_NO ON TANK_NO
SELECT T.CHIEF_CARETAKER_NAME C_NAME,
       T.TANK_NO,
       A.ID_NO,
       A.ANIMAL_NAME
FROM TANK T, AQUATIC_ANIMAL A
WHERE T.TANK_NO = A.TANK_NO
ORDER BY T.CHIEF_CARETAKER_NAME, T.TANK_NO, A.ID_NO;
```

Listing 7-4 presents the results from executing this script.

Listing 7-4: **Animal counts by tank and caretaker**

```
C_NAME            TANK_NO     ID_NO ANIMAL_NAME
------------------------------------------------
Harold Kamalii          1       100 Flipper
                                105 Skipper
                                112 Bopper
                  *********     -----
                  count             3

***************               -----
count                             3


Jan Neeleson            3       199 Nosey
                                202 Rascal
                                240 Snoops
                  *********     -----
                  count             3

***************               -----
count                             3


Joseph Kalama           2       145 Squacky
                                151 Batty
                                166 Shorty
                                175 Paintuin
                  *********     -----
                  count             4

***************               -----
count                             4


                                -----
count                            10
10 rows selected.
```

The results of the query give you a count for each tank, as well as for each caretaker. You can tell which count is which by the position of the label `count`, which SQL*Plus automatically places in the column corresponding to the `ON` clause of the `COMPUTE` command. Thus, the count label for the animals per caretaker is under the `C_NAME` column. Together, the `COMPUTE` and `BREAK` commands allow you to produce some sophisticated reports using SQL*Plus.

Substitution variables

Substitution variables help you write flexible queries that you can easily reuse. For example, you can write a script to produce a report and have that script prompt you for selection criteria each time that you run it.

Defining and Using a Substitution Variable

The DEFINE command allows you to define a SQL*Plus substitution variable. You can then use that variable anywhere in a SQL query. Typically, you would use it to supply a value in the query's WHERE clause.

The syntax for defining a variable looks like this:

```
DEF[INE] [variable = text]
```

You actually don't need to define a variable before using it. The only real reasons to use the DEFINE command are to define a constant or to define a variable with an initial value.

You can place a substitution variable in a query by preceding it with an ampersand (&). When you place a substitution variable in a SQL statement, SQL*Plus prompts you to supply a value for that variable when it executes the statement. Take a look at the following statement, which uses a substitution variable named STARTS_WITH:

```
SELECT ID_NO, ANIMAL_NAME
FROM AQUATIC_ANIMAL
WHERE ANIMAL_NAME LIKE '&STARTS_WITH%';
```

The ampersand is a sign to SQL*Plus that the word that follows is the name of a substitution variable — in this case, STARTS_WITH. When you execute the query, SQL*Plus asks you to supply a value for this variable. SQL*Plus then displays the before and after versions of the line involved. Finally, it executes the query and returns the results. The following example illustrates how this process works:

```
SQL> SELECT ID_NO, ANIMAL_NAME
  2  FROM AQUATIC_ANIMAL
  3  WHERE ANIMAL_NAME LIKE '&STARTS_WITH%';
Enter value for starts_with: B
old   3: WHERE ANIMAL_NAME LIKE '&STARTS_WITH%'
new   3: WHERE ANIMAL_NAME LIKE 'B%'

    ID_NO ANIMAL_NAME
--------- ------------------------------
      112 Bopper
      151 Batty

2 rows selected.
```

In this case, the user supplies a value of B for the variable. This causes the query to become a query for the names of all animals whose names begin with B. There are two of those, named Bopper and Batty.

Using substitution variables like this doesn't make much sense when you are executing queries interactively. They make a lot of sense when you encapsulate a query into a script file that you can execute repeatedly, because they allow you to supply different inputs each time. It's a lot faster and easier to answer a prompt than it is to load a query, edit it to change the selection criteria, execute it, edit it again, and so forth.

Using Double-Ampersand Variables

When you preface a substitution variable with a single ampersand, as shown in the previous example, SQL*Plus will prompt for a value each time it encounters the variable. This can be a bit of a nuisance if you use a variable in two different places in the same SQL statement. Not only will you get prompted twice, but you must also be sure to supply the same value each time, or risk having your query return the wrong results.

There's a solution to this problem, and that is to use a double ampersand (&&) when you are referencing the same variable twice. The double ampersand tells SQL*Plus not to prompt twice for the same variable. If the variable has already been defined, then SQL*Plus reuses the previous value. Here's an example showing the STARTS_WITH variable being used twice in the same query:

```
SQL> SELECT ANIMAL_NAME, ' Begins with a '||'&&STARTS_WITH'
  2  FROM AQUATIC_ANIMAL
  3  WHERE ANIMAL_NAME LIKE '&&STARTS_WITH%';
Enter value for starts_with:
old   1: SELECT ANIMAL_NAME, ' Begins with a '||'&&STARTS_WITH'
new   1: SELECT ANIMAL_NAME, ' Begins with a '||'B'
old   3: WHERE ANIMAL_NAME LIKE '&&STARTS_WITH%'
new   3: WHERE ANIMAL_NAME LIKE 'B%'

ANIMAL_NAME                   'BEGINSWITHA'||'
----------------------------- ----------------
Bopper                         Begins with a B
Batty                          Begins with a B

2 rows selected.
```

As you can see from the example, SQL*Plus prompts for the STARTS_WITH variable once but uses it twice.

Using SQL*Plus to Write Scripts

When you start to use SQL*Plus to generate reports, you'll quickly find that it often takes several commands executed in succession to get the results that you want, as you've already seen in some of the examples in this chapter. Naturally, if you have a complex report that you want to run occasionally, you don't want to have to type all the commands each time. The good news is that you don't need to. You can place all the commands for a report into a text file and tell SQL*Plus to execute the commands in that file. Such a file is referred to as a script file. *Script files* allow you to store complex reports, or other series of commands, so that you can easily execute them later. Script files also allow you to submit SQL*Plus commands as background jobs.

Using the @ command

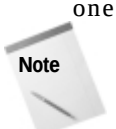
The SQL*Plus @ command invokes a script file. Suppose you have a text file named `list_tables.sql` with the following commands in it:

```
COLUMN index_name FORMAT A15 WORD_WRAPPED HEADING "Index"
COLUMN column_name FORMAT A30 HEADING "Column"
SELECT index_name, column_name
FROM user_ind_columns
WHERE table_name=UPPER('&table_name')
ORDER BY index_name, column_position;
```

This script prompts you for a table name and then displays a list of indexes for that table. You can execute this script file from SQL*Plus using the @ command, as shown in this example:

```
SQL> @list_indexes
Enter value for table_name: aquatic_animal
old   3: WHERE table_name=UPPER('&table_name')
new   3: WHERE table_name=UPPER('aquatic_animal')

Index           Column
--------------- ------------------------------
AQ_ANIMAL_PK    ID_NO

1 row selected.
```

Using substitution variables in scripts like this is a powerful technique. You can automate any number of repetitive tasks by writings scripts such as the one shown here.



SQL*Plus also supports an @@ command. Use @@ in a script when you want to execute another script that is stored in the same directory as the first.

Executing a script from the command line

If you're writing batch jobs or shell scripts, you can invoke SQL*Plus and run a SQL*Plus script all with one command. You can do this by passing your database username and password and the script file name as arguments to the *sqlplus* command. The general format to use looks like this:

```
sqlplus username/password[@service] @script_file
```

Replace *username* and *password* with your database username and password. If you are connecting to a remote database, replace *service* with the Net8 service name of that database. Be sure to separate the service name from the password with an ampersand, and don't leave any spaces. After the username and password, you can place the @ command to run the script file. This time, you do need a space before the @ character. This is how SQL*Plus tells the difference between a service name and a script file name.

When you invoke a script from the operating system prompt like this, you probably don't want to be prompted for any values. That's especially true if you are planning to run the script as a background job. You can still use substitution variables in the script, but you may want to use the special variables &1, &2, and so forth. These correspond to arguments that you can pass on the command line. Here is a different version of the list_indexes script, which accepts the table name as a command-line argument:

```
COLUMN index_name FORMAT A15 WORD_WRAPPED HEADING "Index"
COLUMN column_name FORMAT A30 HEADING "Column"
SELECT index_name, column_name
FROM user_ind_columns
WHERE table_name=UPPER('&1')
ORDER BY index_name, column_position;
EXIT
```

Notice the use of &1 to hold the place of the table name. In just a moment, you will see how that value is passed into the script. First, though, take a look at the last line. An EXIT command is added so that SQL*Plus exits immediately after the script finishes. Listing 7-5 demonstrates invoking this version of the list_indexes script from the operating system command line.

Listing 7-5: **Passing parameters to a script**

```
E:\bible_scripts>sqlplus seapark/seapark@bible_db @list_indexes aquatic_animal

SQL*Plus: Release 8.1.5.0.0 - Production on Thu Aug 5 14:51:26 1999

(c) Copyright 1999 Oracle Corporation.  All rights reserved.
```

```
Connected to:
Oracle8i Personal Edition Release 8.1.5.0.0 - Production
With the Partitioning and Java options
PL/SQL Release 8.1.5.0.0 - Production

old   3: WHERE table_name=UPPER('&1')
new   3: WHERE table_name=UPPER('aquatic_animal')

Index           Column
--------------- ------------------------------
AQ_ANIMAL_PK    ID_NO

Disconnected from Oracle8i Personal Edition Release 8.1.5.0.0 - Production
With the Partitioning and Java options
PL/SQL Release 8.1.5.0.0 - Production

E:\Jonathan\Oracle_Bible\ch7>
```

You can see that SQL*Plus picks up the word following `@list_indexes` and places it in the `&1` substitution variable. This is then substituted into the query, which returns the list of indexes defined on the table. No user interaction is required, other than the initial command to invoke the script.

Summary

In this chapter, you learned:

- SQL*Plus allows you to enter and execute ad-hoc SQL statements and |PL/SQL blocks.
- Server Manager is phasing out, so you should convert your Server Manager scripts to run under SQL*Plus.
- SQL*Plus has a number of line-editing commands, and you can use the `EDIT` command to invoke an external text editor.
- Use the `COLUMN` command to format column data and to provide column headings.
- Use the `TTITLE` and `BTITLE` commands to define page headers and footers.
- Use `BREAK` and `COMPUTE` to generate summary data.
- Use the `@` command to execute commands from a file.

✦ ✦ ✦