Oracle SQL*Plus Pocket Reference, 2nd Edition

By Jonathan Gennick

Publisher : O'Reilly

Pub Date : October 2002

ISBN : 0-596-00441-9

Pages : 120

- Table of Contents
- Index
- Reviews
- Reader Reviews
- Errata

Oracle SQL*PlusPocket Reference

1.1 Introduction

The *Oracle SQL*Plus Pocket Reference* is a quick-reference guide to SQL*Plus and to commonly used SQL query and data manipulation statements. The purpose of this book is to help you find the syntax of specific language elements. It is not a self-contained user guide; basic knowledge of SQL*Plus is assumed. For more information, see my book *Oracle SQL*Plus: The Definitive Guide* (O'Reilly) and *Mastering Oracle SQL* by Sanjay Mishra and Alan Beaulieu (O'Reilly).

1.1.1 Acknowledgments

Deborah Russell, Darl Kuhn, Ken Jacobs, and Alison Holloway all played a part in making this book a reality. For their assistance and support, I'm most grateful.

1.1.2 Conventions

UPPERCASE

Indicates SQL*Plus, SQL, or PL/SQL keywords

lowercase

Indicates user-defined items such as table names

Italic

Indicates filenames, emphasis, introduction of new terms, and parameter names

`Constant width`

Used for code examples

`Constant width bold`

Indicates user input in examples showing an interaction

`[ ]`

Used in syntax descriptions to denote optional elements

`{ }`

Used in syntax descriptions to denote a required choice

`|`

Used in syntax descriptions to separate choices

_

Used in syntax descriptions to indicate that the underlined option is the default

1.2 Interacting with SQL*Plus

This section covers essential information you need to know to interact with SQL*Plus. Here you will learn how to start SQL*Plus, enter commands, delimit strings, and name variables.

1.2.1 Starting SQL*Plus

Invoke SQL*Plus by issuing the *sqlplus* command from your operating-system command prompt. On Microsoft Windows systems, use either *sqlplus* or *sqlplusw* depending on whether you want SQL*Plus to run in a command-prompt window or in its own window; you can also select an icon from the Start menu. (Early releases of SQL*Plus on Windows used executable names such as *PLUS33* and *PLUS80W*.)

> Beware of passing your password as a command-line argument to SQL*Plus. Such passwords may be easily visible to other users on Linux and Unix systems.

1.2.1.1 Syntax for the sqlplus command

The syntax used to invoke SQL*Plus is as follows:

```
sqlplus [[-S[ILENT]] [-H[ELP]] [-V[ERSION]]
   [-R[ESTRICT] level] [-L[OGON]]
   [-M[ARKUP] "markup_options"]
   [ [username[/password][@connect]|/
       [AS {SYSDBA|SYSOPER}]]
     |/NOLOG]
   [@scriptfile [arg1 arg2 arg3...]]]
```

The -RESTRICT and -MARKUP parameters are new in Oracle8*i*. -HELP and -VERSION are new in Oracle9*i*. Here are the parameter descriptions:

-S[ILENT]

Tells SQL*Plus to run in silent mode. No startup message is displayed; no command prompt is displayed; no commands are echoed to the screen.

-H[ELP]

Causes SQL*Plus to display a short summary of this syntax. Prior to Oracle9*i*, use *sqlplus -* to get the help summary.

-V[ERSION]

Causes SQL*Plus to display version and copyright information. Prior to Oracle9*i*, use *sqlplus -?* to get version and copyright information.

-R[ESTRICT] level

Restricts what the user can do from SQL*Plus. The *level* must be one of the following:

1

Disables the EDIT, HOST, and ! commands

2

Disables the EDIT, HOST, !, SAVE, SPOOL, and STORE commands

3

Disables the EDIT, GET, HOST, !, SAVE, START, @, @@, SPOOL, and STORE commands

Level 3 also disables the reading of the *login.sql* file. The *glogin.sql* file is read, but restricted commands aren't executed.

-L[OGON] (new in Oracle9*i* Release 9.2)

Prevents SQL*Plus from reprompting for the username and password in the event that the first username and password passed is incorrect.

-M[ARKUP] markup_options

Allows you to specify the markup language to use when generating output. Except for HTML, all markup options are optional. The following are valid markup options. Default values are underlined or noted in the text.

HTML {ON | OFF}

Specifies the markup language to use and enables or disables the use of that markup language. You must specify a value for this option.

HEAD text

Specifies content for the <head> tag. The tag is written as <head>*text*</head>.

BODY text

Specifies attributes for the <body> tag. The tag is written as <body *text*>.

TABLE text

Specifies attributes for the <table> tag that formats query output. The tag is written as <table *text*>.

ENTMAP {ON | OFF}

Controls whether SQL*Plus uses HTML equivalents such as &lt; and &gt; for special characters such as "<" and ">".

SPOOL {ON | OFF}

Controls whether SQL*Plus writes <html>, <head>, and <body> tags to any spool file you create during your SQL*Plus session, while HTML is ON.

PRE[FORMAT] {ON | OFF}

Controls whether query output is enclosed within <pre> ... </pre> tags rather than within HTML tables.

On some operating systems, you need to enclose the entire string of markup options within double quotes. For example:

```
sqlplus -m "html on spool off"
```

Furthermore, any HEAD, BODY, and TABLE text should itself be quoted, and you must escape the quotes:

```
sqlplus -m "html on table \"width=50% align='left'\""
```

The backslash-quote (`\"`) syntax works on Windows XP, Linux, and Unix, and places a quote (`"`) within the string that is the value for the -M option.

username [*/password*][*@connect*]

Your database login information. *connect* is an Oracle Net connect identifier such as those defined in *$ORACLE_HOME/network/admin/tnsnames.ora*.

/

Connects you to a local database using operating-system authentication.

AS {SYSDBA | SYSOPER}

Connects you in an administrative role so that you can perform database administration tasks (e.g., starting and stopping a database instance). You may need to enclose the login within quotes:

```
sqlplus "sys/password as sysdba"
```

/NOLOG

Tells SQL*Plus to start without connecting to a database first.

scriptfile

The name of a SQL*Plus script file. SQL*Plus will start up and then execute the file. Beginning in Oracle9*i*, you may also specify the URL of a file. See At Sign (@) for an example of this.

arg1 agr2 arg3

Optional command-line arguments to pass to your script. Separate arguments by at least one space.

1.2.2 Entering Commands

How you enter commands in SQL*Plus depends on whether you are entering a command to SQL*Plus itself, or are entering a SQL statement or a PL/SQL block.

1.2.2.1 Entering SQL*Plus commands

Commands such as DESCRIBE, COLUMN, TTITLE, SET, and all the others listed in SQL*Plus Command Reference are commands to SQL*Plus itself. These must be entered on one line and are executed immediately after you enter them. For example:

```
SET ECHO ON
DESCRIBE employee
```

SQL*Plus commands may optionally be terminated by a semicolon. For example:

```
PROMPT This semicolon won't print.;
CONNECT system/manager;
```

You can change this behavior of SQL*Plus towards semi-colons by changing the SQLTERMINATOR setting.

Long SQL*Plus commands may be continued onto multiple physical lines. The SQL*Plus continuation character is a hyphen (`-`). Use it at the end of a physical line to continue a long SQL*Plus command to the next line. The following three lines, for example, are treated as one by SQL*Plus:

```
COLUMN employee_id -
FORMAT 099999 -
HEADING 'Emp ID'
```

The space in front of the continuation character is optional. Quoted strings may also be continued. For example:

```
SELECT 'Hello-
World!' FROM dual;
```

When you continue a quoted string, any spaces before the continuation character are included in the string. The line break also counts as one space.

1.2.2.2 Entering SQL statements

SQL statements may span multiple lines and must always be terminated. This may be done using either a semicolon (`;`) or a forward slash (`/`). For example:

```
SELECT user
FROM dual;

SELECT user
FROM dual
/
```

In both cases, the SQL statement is entered into a buffer known as the *SQL buffer* and is then executed. You may also terminate a SQL statement using either a blank line or a period, in which case the statement is stored in the buffer but not executed. For example:

```
SQL> SELECT user
  2  FROM dual
  3
SQL> SELECT user
```

```
  2  FROM dual
  3  .
```

Use the SET SQLTERMINATOR command to change the terminator from a semicolon to some other character. Use SET SQLBLANKLINES ON to allow blank lines within a SQL statement. To execute the statement currently in the buffer, enter a forward slash on a line by itself.

1.2.2.3 Entering PL /SQL blocks

PL/SQL blocks may span multiple lines and may contain blank lines. They must be terminated by either a forward slash or a period (`.`) on a line by itself. For example:

```
BEGIN
   DBMS_OUTPUT.PUT_LINE('Hello World!');
END;
/

BEGIN
   DBMS_OUTPUT.PUT_LINE('Hello World!');
END;
.
```

When a forward slash is used, the block is sent to the server and executed immediately. When a period is used, the block is stored only in the SQL buffer. Use the SET BLOCKTERMINATOR command to change the block terminator from a period to some other character.

1.2.3 Strings in SQL*Plus Commands

Many SQL*Plus-specific commands take string values as parameters. Simple strings containing no spaces or punc-tuation characters may be entered without quotes. Here's an example:

```
COLUMN employee_id HEADING emp_id
```

Generally, it's safer to use quoted strings. Either single or double quotes may be used. For example:

```
COLUMN employee_id HEADING 'Emp #'
COLUMN employee_id HEADING "Emp #"
```

To embed quotes in a string, either double them or use a different enclosing quote. The following two commands have equivalent results:

```
COLUMN employee_id HEADING '''Emp #'''
COLUMN employee_id HEADING "'Emp #'"
```

The single exception to these rules is the PROMPT command. All quotes used in a PROMPT command will appear in the output.

1.2.4 Specifying Filenames

Several SQL*Plus commands allow you to specify a file-name. In all cases, you may also include a path and/or an extension with the name. For example:

```
SPOOL my_report
SPOOL c:\temp\my_report
SPOOL create_synonyms.sql
```

Most file-related commands assume a default extension if you don't supply one. The default varies by command.

1.2.5 Naming Variables

SQL*Plus allows you to declare two types of variables: user variables and bind variables. The rules for naming each type are different.

User variable names may contain letters, digits, and underscores (_) in any order. They are case-insensitive and are limited to 30 characters in length.

Bind variable names must begin with a letter but after that may contain letters, digits, underscores, dollar signs (`$`), and number signs (`#`). They also are case-insensitive and are limited to 30 characters in length.

1.3 Selecting Data

The SELECT statement is the key to getting data out of an Oracle database. It's also very likely the most commonly executed SQL statement from SQL*Plus.

1.3.1 The SELECT Statement

The basic form of the SELECT statement looks like this:

```
SELECT column_list
FROM table_list
WHERE conditions
GROUP BY column_list
HAVING conditions
ORDER BY column_list;
```

The lists in this syntax are comma-delimited. The column list, for example, is a comma-delimited list of column names or expressions identifying the data you want the query to return.

1.3.1.1 Selecting columns from a table

To retrieve columns from a table, list the columns you want after the SELECT keyword, place the table name after the FROM keyword, and execute your statement. The following query returns a list of tables you own with the names of their assigned tablespaces:

```
SELECT table_name, tablespace_name
  FROM user_tables;
```

1.3.1.2 Ordering query results

You can use the ORDER BY clause to sort the results of a query. The following example sorts the results by table name:

```
SELECT table_name, tablespace_name
FROM user_tables
ORDER BY table_name;
```

The default is to sort in ascending order. You can specify descending order using the DESC keyword. For example:

```
ORDER BY table_name DESC;
```

While it's redundant, ASC may be used to specify ascending order. The following example sorts the table list first by tablespace name in descending order and then within that by table name in ascending order:

```
SELECT table_name, tablespace_name
FROM user_tables
ORDER BY tablespace_name DESC,
       table_name ASC;
```

If you want the sort to be case-insensitive, you can use Oracle's built-in UPPER function. For example:

```
SELECT table_name, tablespace_name
FROM user_tables
ORDER BY UPPER(table_name);
```

For symmetry, Oracle also has a built-in LOWER function. LOWER converts a string to lowercase; UPPER converts to uppercase.

1.3.1.3 Restricting query results

Use the WHERE clause to restrict the rows returned by a query to those that you need to see. The following example returns a list of any invalid objects that you own:

```
SELECT object_name, object_type
FROM user_objects
WHERE status = 'INVALID'
ORDER BY object_type, object_name;
```

The expression following the WHERE clause may be any valid Boolean expression. Oracle supports all the typical operators you'd expect: +, -, /, *, <, >, <>, <=, >=, AND, OR, NOT, ||, IS NULL, LIKE, BETWEEN, and IN. Parentheses are also supported and may be used to clarify the order of evaluation.

1.3.1.4 Using column aliases

If a SELECT statement includes columns that are expressions, Oracle generates a column name based on the expression. Take a look at the following SQL statement:

```
SELECT SUM(hours_logged)
FROM project_hours
WHERE project_id = 1001;
```

The name of the column returned by this query will be SUM(HOURS_LOGGED). That means that any COLUMN commands used to format the output need to look like this:

```
COLUMN SUM(HOURS_LOGGED) -
HEADING 'Total Hours'
```

As your expressions become more complicated, the Oracle-generated names become difficult to deal with. It's better to use a column alias to supply a more user-friendly name for the computed column. For example:

```
SELECT SUM(hours_logged) total_hours
FROM project_hours
WHERE project_id = 1001;
```

Now the column name is obvious. It's TOTAL_HOURS, and it won't change even if the expression changes.

1.3.2 Null Values

Null values are pernicious, especially in the WHERE clause of a query. With only a few exceptions, any expression containing a null value returns a null as the result. Because nulls are considered neither true nor false, this can have unexpected ramifications in how a WHERE clause is evaluated. Consider the following query that attempts to retrieve a list of NUMBER columns with a scale other than 2:

```
SELECT table_name, column_name
FROM user_tab_columns
WHERE data_type = 'NUMBER'
AND data_scale <> 2;
```

This query is an utter failure because it misses all the floating-point NUMBER columns that have no scale defined at all. Avoid this problem by explicitly considering nulls when you write your WHERE clause. Use either the IS NULL or the IS NOT NULL operator. For example:

```
SELECT table_name, column_name
FROM user_tab_columns
WHERE data_type = 'NUMBER'
AND (data_scale <> 2
OR data_scale IS NULL);
```

When sorting data, null values are treated as greater than all other values. When a standard ascending sort is done, null values sort to the bottom of the list. A descending sort causes null values to rise to the top. You can use the built-in NVL or COALESCE (Oracle9*i*) functions to modify this behavior.

1.3.2.1 Using the NVL function

If you wish to return results from a query that might be null or sort on results that might be null, you can use Oracle's built-in NVL function to replace null values with a selected non-null value. For example, the NUM_ROWS column in the USER_TABLES view has a value only for tables that have been analyzed. Here, the NVL function converts null values to zeros:

```
SELECT table_name, NVL(num_rows,0)
FROM user_tables
ORDER BY NVL(num_rows,0);
```

Be cautious about using NVL in a WHERE clause. Using NVL, or any other function, on an indexed column in a WHERE clause may prevent Oracle from using any index on that column.

1.3.2.2 Using the COALESCE function

New in Oracle9*i*, the COALESCE function extends the concept behind NVL. You can pass in any number of values as parameters, and COALESCE returns the first non-null value as its result. Consider this nested NVL invocation:

```
SELECT NVL(employee_nickname,
         NVL(employee_first_name,
            employee_last_name))
FROM employee;
```

The nesting of the calls to NVL makes it difficult to decipher the goal of this query. In Oracle9*i*, you can write the query much more simply using the COALESCE function:

```
SELECT COALESCE(employee_nickname,
             employee_first_name,
             employee_last_name)
```

```
FROM employee;
```

This query returns each employee's nickname. If there is no nickname for an employee, then the first name is returned instead. If there is no first name, the last name is returned. COALESCE makes this order of preference much more obvious than in the previous NVL solution.

If all parameters to COALESCE are null, the function returns a null result. You can avoid a null result by making your final parameter a constant rather than a column name.

1.3.3 CASE Expressions

CASE expressions represent an ANSI-standard mechanism for embedding IF...THEN...ELSE logic in a SQL statement. Prior to the introduction of CASE, you had to use the Oracle-specific DECODE function to implement conditional logic in a SQL statement. CASE expressions may be used in select lists, WHERE clauses, HAVING clauses, and anywhere else an expression is valid.

> CASE was introduced in Oracle8*i* but was enhanced in Oracle9*i* by the addition of searched CASE expressions. I've heard reports that in Release 8.1.7, CASE did not work from PL/SQL.

1.3.3.1 Simple CASE expressions

Simple CASE expressions are closest in concept to the DECODE function. The general form is:

```
CASE expression
WHEN expression_1 THEN return_expression_1
WHEN expression_2 THEN return_expression_2
...
ELSE return_expression
END
```

The ELSE portion of the CASE expression is optional. The following is an example of a simple CASE expression:

```
SELECT course_name,
      CASE period
         WHEN 1 THEN 'First'
         WHEN 2 THEN 'Second'
```

```
        WHEN 3 THEN 'Third'
        WHEN 4 THEN 'Fourth'
        WHEN 5 THEN 'Fifth'
        ELSE 'Unknown'
        END period_name
FROM course;
```

The period_name following the END keyword is a column alias. It's not necessary, but when using SQL*Plus, it's sometimes helpful to name columns that are the result of an expression.

The equivalent DECODE expression is as follows:

```
SELECT course_name,
      DECODE(period,1,'First',2,'Second',3,'Third',
                  4,'Fourth',5,'Fifth',
                  'Unknown') period_name
FROM course;
```

CASE is part of the SQL standard, whereas DECODE is not. Thus, the use of CASE is preferable when you have a choice.

1.3.3.2 Searched CASE expressions

A searched CASE expression allows you to do more than match on a single value. A searched CASE allows you to match on expressions:

```
SELECT course_name,
      CASE WHEN period >= 1
           AND period <= 3 THEN 'Morning'
          WHEN period >= 4
           AND period <= 6 THEN 'Afternoon'
          ELSE 'After School'
          END
FROM course;
```

Searched CASE expressions are versatile, and enable you to easily do things that were previously difficult or impossible to do.

1.3.4 Table Joins (Oracle8i)

It's very common to combine data from two or more tables to return related information. Such a combination of two tables is referred to as a *join*.

You join two tables by listing them in the FROM clause, separated by commas. For example:

```
SELECT user_constraints.constraint_name,
      user_constraints.constraint_type,
      user_cons_columns.column_name
FROM user_constraints, user_cons_columns;
```

This query returns the *Cartesian product* 뿗 ll possible combinations of all rows from both tables. Conceptually, this is where all joins start. In practice, you almost always put some conditions in the WHERE clause so that only related rows are combined. The following, more useful, query returns a list of constraint names together with the columns involved in each constraint:

```
SELECT user_constraints.constraint_name,
      user_constraints.constraint_type,
      user_cons_columns.column_name
FROM user_constraints, user_cons_columns
WHERE user_constraints.constraint_name
     = user_cons_columns.constraint_name;
```

Because both tables contain columns with matching names, the column references must be qualified with the table name. You can see that this quickly gets cumbersome. The solution is to provide a shorter alias for each table and use that alias to qualify the column names. For example:

```
SELECT uc.constraint_name,
      uc.constraint_type,
      ucc.column_name
FROM user_constraints uc,
    user_cons_columns ucc
WHERE uc.constraint_name =
   ucc.constraint_name;
```

Here, the alias uc is used for the USER_CONSTRAINTS table, while UCC is used for USER_CONS_COLUMNS. The resulting query is much easier to read because you aren't overwhelmed with long table names.

1.3.4.1 Inner and outer joins

The joins that you've seen so far are inner joins. An *inner join* is one that returns data only when both tables have a row that matches the join conditions. For example, the following query returns only tables that have constraints defined on them:

```
SELECT ut.table_name, uc.constraint_name
FROM user_tables ut, user_constraints uc
WHERE ut.table_name = uc.table_name;
```

An *outer join* returns rows for one table, even when there are no matching rows in the other. You specify an outer join in Oracle by placing a plus sign (+) in parentheses following the column names from the optional table in your WHERE clause. For example:

```
SELECT ut.table_name, uc.constraint_name
FROM user_tables ut, user_constraints uc
WHERE ut.table_name = uc.table_name(+);
```

The (+) after uc.table_name makes the user_constraint table optional. The query returns all tables, and where there are no corresponding constraint records, Oracle supplies a null in the constraint name column.

1.3.5 Table Joins (Oracle9*i*)

Oracle9*i* introduces new table join syntax; this is the join syntax defined by the ANSI SQL/92 standard. Join conditions may now be written in the FROM clause, making it easier to follow the logic of a query. In addition, the new syntax supports full outer joins, something not possible using the old syntax. Unless you need to maintain compatibility with old releases of Oracle, I strongly recommend using the new syntax.

1.3.5.1 Sample tables

Example queries in this section on Oracle9*i* table joins are based on the following three tables:

```
SQL> SELECT *
```

```
  2  FROM course;

COURSE_NAME          PERIOD
--------------- ----------
Spanish I                1
Spanish 1                6
U.S. History             3
English II               4

SQL> SELECT *
  2  FROM enrollment;

COURSE_NAME          PERIOD STUDENT_NAME
--------------- ---------- ---------------
English II               4 Michael
Spanish I                1 Billy
Spanish I                6 Sky Lynn
Spanish I                1 Jeff
English II               4 Jenny

SQL> SELECT *
  2  FROM student;

STUDENT_NAME         GRADE
--------------- ----------
Michael                  6
Billy                    3
Sky Lynn                 1
Jeff                     1
Jenny                    8
```

Note that Sky Lynn's enrollment record is without a matching COURSE record. She is registered for Spanish I (letter "I"), while the course is Spanish 1 (digit "1"). This becomes significant when performing an outer join and when using the new USING clause.

1.3.5.2 Inner joins

Use the INNER JOIN keywords in the FROM clause to specify an inner join between the two tables. Use the ON clause to specify your join conditions. For example:

```
SELECT c.course_name, c.period, e.student_name
FROM course c INNER JOIN enrollment e
    ON c.course_name = e.course_name
       AND c.period = e.period;
```

Other clauses, such as WHERE and ORDER BY come after the FROM clause:

```
SELECT c.course_name, c.period, e.student_name
FROM course c INNER JOIN enrollment e
    ON c.course_name = e.course_name
       AND c.period = e.period
WHERE c.period < 9
ORDER BY c.period, c.course_name;
```

1.3.5.3 Join order

When joining more than two tables, use parentheses to control the join order. The following query joins COURSE and ENROLLMENT first, and then joins the table named STUDENT to the result:

```
SELECT c.course_name, c.period, s.student_name, s.grade
FROM (course c INNER JOIN enrollment e
       ON c.course_name=e.course_name
       AND c.period=e.period)
    INNER JOIN student s ON e.student_name=s.student_name;
```

If you omit parentheses, Oracle processes the joins from left to right. The example here uses parentheses to explicitly specify the default join order.

1.3.5.4 Left and right outer joins

To perform an outer join of the type traditionally supported by Oracle, use either LEFT OUTER JOIN or RIGHT OUTER JOIN. Left and right outer joins are similar; the difference lies in the ordering of the tables in the FROM clause. The following query uses the old syntax to return all rows from COURSE together with any matching rows from ENROLLMENT:

```
SELECT c.course_name, c.period, e.student_name
FROM course c, enrollment e
WHERE c.course_name = e.course_name(+)
  AND c.period = e.period(+);
```

In this query, ENROLLMENT is considered the optional table because (+) is appended to each of that table's columns. The following two queries accomplish the same join using the new, ANSI standard syntax:

```
SELECT c.course_name, c.period, e.student_name
FROM course c LEFT OUTER JOIN enrollment e
    ON c.course_name = e.course_name
       AND c.period = e.period;

SELECT c.course_name, c.period, e.student_name
FROM enrollment e RIGHT OUTER JOIN course c
    ON c.course_name = e.course_name
       AND c.period = e.period;
```

Note the difference between the two queries. The first query lists the COURSE table first, while the second query lists the ENROLLMENT table first. A LEFT OUTER JOIN makes the left table the required table. A RIGHT OUTER JOIN makes the right table the required table. In both queries, ENROLLMENT is the optional table, and COURSE is the required table.

1.3.5.5 Full outer joins

The full outer join represents a new capability in Oracle9*i*. A *full outer join* returns all rows from both tables. Where possible, rows from one table are matched with those from the other. In Oracle8*i*, you can simulate an outer join using a UNION query:

```
SELECT c.course_name, c.period, e.student_name
FROM course c, enrollment e
WHERE c.course_name = e.course_name(+)
  AND c.period = e.period(+)
UNION
SELECT e.course_name, e.period, e.student_name
FROM enrollment e
WHERE NOT EXISTS (
   SELECT *
   FROM course c2
   WHERE c2.course_name = e.course_name
     AND c2.period = e.period
   );
```

To execute this UNION query, Oracle needs to execute each SELECT statement separately and then combine the results. You potentially can end up with two full table scans for each table. In Oracle9*i*, you can use the FULL OUTER JOIN keyword to do the same thing:

```
SQL> SELECT c.course_name, c.period, e.student_name
2 FROM course c FULL OUTER JOIN enrollment e
3 ON c.course_name = e.course_name
4 AND c.period = e.period;

COURSE_NAME         PERIOD STUDENT_NAME
--------------- ---------- ---------------
English II              4 Michael
Spanish I               1 Billy
Spanish I               1 Jeff
English II              4 Jenny
Spanish 1               6
U.S. History            3
                          Sky Lynn
```

As you can see, this query is easier to understand than the UNION query. It is also one SELECT statement and should execute more efficiently than the UNION query shown previously.

1.3.5.6 Specifying join conditions

Oracle9*i* supports three ways to specify join conditions: the ON, NATURAL, and USING clauses. The most general approach is to use the ON clause, in which you can specify any type of join condition. For example:

```
SELECT c.course_name, c.period, e.student_name
FROM course c FULL OUTER JOIN enrollment e
ON c.course_name = e.course_name
AND c.period = e.period;
```

This particular query is an *equi-join*; it looks for cases in which the corresponding columns from the two tables contain the same values. Because the names of the join columns are identical in the two tables, this join is also considered a *natural join* . Oracle supports the following shorthand syntax for natural joins:

```
SELECT course_name, period, e.student_name
FROM course c NATURAL FULL OUTER JOIN enrollment e;
```

Note that table aliases are not used for the join columns in the select list. In a natural join, Oracle recognizes only one version of each join column.

> Natural joins are dangerous! Use them only for queries you type in interactively.

While convenient, you must exercise caution when using the NATURAL join syntax. Consider what will happen if you code your programs using the NATURAL syntax and then later add a column, UPDATE_TIMESTAMP, to each of your tables. This column is automatically included in all your joins, and your join queries will return erroneous results. I strongly recommend the USING clause over the NATURAL keyword.

The USING clause represents another shortcut to performing an equi-join. The difference between USING and NATURAL is that with USING, you explicitly specify the join columns. Later changes to your tables won't alter the semantics of your queries. The following query uses USING to perform the same full outer join as in the previous two examples:

```
SQL> SELECT course_name, period, e.student_name
2 FROM course c FULL OUTER JOIN enrollment e
3 USING (course_name, period);

COURSE_NAME        PERIOD STUDENT_NAME
--------------- ---------- ---------------
English II              4 Michael
Spanish I               1 Billy
Spanish I               1 Jeff
English II              4 Jenny
Spanish 1               6
U.S. History            3
```

```
Spanish I              6 Sky Lynn
```

Compare the results of this query with the results shown previously in Section 1.3.5.5. Notice that Sky Lynn's course name shows up in these results. That's because when USING or NATURAL is used, Oracle recognizes only one version of each join column and draws the value from whichever table it can.

1.3.6 Summary Queries

The GROUP BY and HAVING clauses, together with Oracle's built-in aggregate functions, allow you to summarize the data returned by a query.

1.3.6.1 Using aggregate functions

Aggregate functions take data from multiple rows as input and return one summarized value. For example, the following query uses the COUNT function to return the number of tables you own:

```
SELECT COUNT(*)
FROM user_tables;
```

Oracle supports several different aggregate functions, all of which are listed in Table 1-1.

Table 1-1. Aggregate functions

Function	Description
AVG	Averages the values in each group.
COUNT	Counts the non-null values in each group. COUNT(*) is a special case and counts all rows.
MAX	Returns the maximum value in a group.
MIN	Returns the minimum value in a group.
STDDEV	Returns the standard deviation of all values in a group.
SUM	Returns the sum of all values in a group.
VARIANCE	Returns the variance (related to standard deviation) of all values in a group.

1.3.6.2 Using GROUP BY

In addition to summarizing the entire results of a query, you can summarize the data for each distinct value in a column. For example, the following query returns the number of columns in each table you own:

```
SELECT ut.table_name, COUNT(utc.column_name)
FROM user_tables ut, user_tab_columns utc
WHERE ut.table_name = utc.table_name
GROUP BY ut.table_name
ORDER BY ut.table_name;
```

The next query extends the previous query and displays the number of columns in each table to which you have access. This time the grouping results in one row for each distinct owner and table name combination:

```
SELECT at.owner, at.table_name, COUNT(atc.column_name)
FROM all_tables at, all_tab_columns atc
WHERE at.table_name = atc.table_name
GROUP BY at.owner, at.table_name
ORDER BY at.owner, at.table_name;
```

If you want the results of a GROUP BY query returned in any particular order, you must include an ORDER BY clause. However, an ORDER BY clause is not required. At times it may appear that Oracle automatically sorts GROUP BY queries. It does, but only to a point. If you want the results sorted, you must include an ORDER BY clause.

Columns in the select list of a GROUP BY query must be either listed in the GROUP BY clause or enclosed by one of the aggregate functions listed earlier in Table 1-1.

1.3.6.3 Restricting summarized results

You can use the HAVING clause to restrict the rows returned by a summary query to only the rows of interest. The HAVING clause functions just like the WHERE clause, except that the HAVING conditions are applied to the summarized results. For example, the following query returns a list of all tables for which you have not defined any indexes:

```
SELECT ut.table_name, COUNT(ui.index_name)
FROM user_tables ut, user_indexes ui
WHERE ut.table_name = ui.table_name(+)
```

```
GROUP BY ut.table_name
HAVING COUNT(ui.index_name) = 0;
```

This query works by first counting the number of indexes on each table and then eliminating those tables with nonzero counts.

Avoid placing conditions in the HAVING clause that do not test summarized values. Consider, for example, these two queries:

```
SELECT at.owner, at.table_name, COUNT(atc.column_name)
FROM all_tables at, all_tab_columns atc
WHERE at.table_name = atc.table_name
GROUP BY at.owner, at.table_name
HAVING at.owner <> 'SYS'
AND at.owner <> 'SYSTEM'
ORDER BY at.owner, at.table_name;

SELECT at.owner, at.table_name, COUNT(atc.column_name)
FROM all_tables at, all_tab_columns atc
WHERE at.table_name = atc.table_name
AND at.owner <> 'SYS'
AND at.owner <> 'SYSTEM'
GROUP BY at.owner, at.table_name
ORDER BY at.owner, at.table_name;
```

Both queries return the same result 뾞 count of columns in each table except for those tables owned by SYS or SYSTEM. The second query, however, executes more efficiently because tables owned by SYS and SYSTEM are eliminated by the WHERE clause before the data is summarized.

1.3.6.4 Using ALL and DISTINCT

The aggregate functions listed in Table 1-1 ignore null values. By default, they also exclude duplicate values. You can use the ALL and DISTINCT keywords to modify this behavior. For example:

```
SELECT COUNT (DISTINCT table_name)
FROM user_tab_columns;
```

```
SELECT COUNT (ALL table_name)
FROM user_tab_columns;
```

The first query uses the DISTINCT keyword to count the number of tables. The second query uses the ALL keyword to count the total number of columns defined for those tables.

1.3.7 Unions

SQL supports four union operators that allow you to take the results of two queries and combine them into one. These are listed in Table 1-2.

Table 1-2. SQL's union operators

Function	Description
UNION	Combines the results of two queries and then eliminates duplicate rows.
UNION ALL	Combines the results of two queries without eliminating duplicate rows.
MINUS	Takes the rows returned by one query and eliminates those that are also returned by another.
INTERSECT	Takes the results from two queries and returns only rows that appear in both.

The following example of a union query uses the MINUS operator to return a list of all tables for which you have not yet defined any indexes:

```
SELECT table_name
FROM user_tables
MINUS
SELECT DISTINCT table_name
FROM user_indexes
WHERE table_owner = USER
ORDER BY table_name;
```

The first query returns a list of all tables you own. The second query returns a list of all tables that are indexed. The MINUS union operation removes those indexed tables from the first list, leaving only the unindexed tables.

When two or more queries are unioned together, only one ORDER BY clause is allowed, and it must be at the end. Only the rows returned as the final result are sorted.

1.3.8 Partition Operations

When you're selecting data from partitioned tables, and you know that you want data only from a specific partition, you can explicitly specify the partition, or subpartition, to use. For example, specify a partition as follows:

```
SELECT *
FROM course PARTITION (2001_courses);
```

Specify a subpartition as follows:

```
SELECT *
FROM course SUBPARTITION (2001_qtr01_courses);
```

Be careful when using specific partition names in queries that you embed in programs, because those queries will fail if the specified partitions do not exist.

The pattern shown here for specifying partition and subpartition names can also be applied to INSERT, UPDATE, and DELETE statements.

1.4 Inserting Data

Use the INSERT statement to add new rows to a table. New in Oracle9*i* is the ability to perform direct path inserts and multitable inserts. For the examples in this section, I've added a column to the COURSE table shown previously in the section on Oracle9*i* table joins:

```
ALTER TABLE COURSE ADD (
   course_hours NUMBER DEFAULT 4);
```

1.4.1 Inserting One Row

To insert one row into a table, specify the list of columns for which you wish to insert a value and use the VALUES clause to specify values for the columns in your list:

```
INSERT INTO COURSE (course_name, period, course_hours)
VALUES ('French I', 5, DEFAULT);
```

The DEFAULT keyword is new in Oracle9*i* and is used in this query to explicitly request the default value for the COURSE_HOURS column. You can use the NULL keyword, available in all releases of Oracle, to explicitly insert a null value into a column.

You can omit the list of columns if you provide a value for each column in your table and if you provide those values in the same order in which the columns are listed when you DESCRIBE the table:

```
INSERT INTO COURSE
VALUES ('French I', 5, DEFAULT);
```

I don't recommend this shortcut unless you are just typing in a one-off query interactively. It's safer to specify the column names.

1.4.2 Inserting the Results of a Query

Use the INSERT...SELECT...FROM syntax to insert the results of a query into a table. For example, the following INSERT statement creates a new row in the COURSE table for any currently undefined courses for which students are registered:

```
INSERT INTO COURSE (course_name, period)
SELECT DISTINCT course_name, period
FROM enrollment e
WHERE NOT EXISTS (
   SELECT *
   FROM course c
   WHERE c.course_name = e.course_name
     AND c.period = e.period
   );
```

When using INSERT...SELECT...FROM to populate a table with a large amount of data, you may be able to improve performance by doing a direct path insert. Use the APPEND hint for this (see Section 1.10.4 later in this book). A direct path insert functions much like a direct path load:the database buffer cache is bypassed, and data is written directly to new extents in the datafiles.

1.4.3 Multitable Inserts (Oracle9*i*)

Multitable inserts are a new feature in Oracle9*i*; they allow you to insert the results of a SELECT query into several different tables at once. You write WHEN clauses into your INSERT statement in order to direct rows to the proper table.

The following INSERT statement uses Oracle9*i*'s multitable insert functionality to do the following:

- Insert courses for period 6 into the table named COURSE_6.
- Insert courses for period 3 into the table named COURSE_3.
- Insert all other courses into the table named COURSE_OTHER:

```
• INSERT ALL
• WHEN (period=6) THEN
•    INTO course_6 (course_name, period)
•         values(course_name, period)
• WHEN (period=3) THEN
•    INTO course_3 (course_name, period)
•         values(course_name, period)
• ELSE
•    INTO course_other (course_name, period)
•         values(course_name, period)
• SELECT course_name, period
  FROM course;
```

The ALL keyword causes Oracle to check each row returned by the SELECT query against each WHEN clause. If a row satisfies more than one WHEN clause, it's inserted into more than one table. Use the FIRST keyword to limit inserts to only the first matching WHEN clause.

The previous INSERT statement illustrated a *conditional* multitable insert. Use an *unconditional* multitable insert to perform each INTO clause you specify. For example:

```
INSERT ALL
   INTO courses_taken (course_name)
        values (course_name)
   INTO students_registered (student_name)
        values (student_name)
SELECT course_name, student_name
FROM enrollment;
```

Note that regardless of whether you perform an unconditional or a conditional multitable insert, the target tables for each INTO clause do not need to be the same.

1.5 Updating Data

Use the UPDATE statement to modify column values in existing table rows.

1.5.1 Simple Updates

A simple UPDATE statement takes on the following form:

```
UPDATE table_name
SET column_name = new_value,
    column_name = new_value,
    column_name = new_value,
    ...
WHERE selection_criteria;
```

For example, the following statement corrects a small problem with a course name; it changes the name to use an "I" (letter) instead of a "1" (digit):

```
UPDATE course
SET course_name = 'Spanish I'
WHERE course_name = 'Spanish 1';
```

Be careful with updates. If you omit the WHERE clause, your update will be applied to *all* rows in the table.

1.5.2 Noncorrelated Subqueries in the SET Clause

Rather than specify a new value in the SET clause for a column, you can specify a subquery that returns exactly one value (one column, one row). That value then becomes the new column value. For example:

```
UPDATE enrollment
SET period = (
      SELECT period
      FROM course
      WHERE course_name = 'English II'),
   course_name = (
      SELECT course_name
      FROM course
      WHERE course_name = 'English II'),
WHERE course_name = 'English II';
```

Setting the PERIOD and COURSE_NAME columns to their current values by way of a subquery doesn't make much sense. I did it only to show that you can use more than one subquery in an UPDATE statement.

1.5.3 Correlated Subqueries in the SET Clause

Subqueries in UPDATE statements are often more useful when they are correlated. A *correlated subquery* is one in which the row returned depends on the current row being updated. For example, the following UPDATE statement uses a correlated subquery to reset all periods in the enrollment table to values taken from the COURSE table:

```
UPDATE enrollment e
SET period = (
      SELECT MIN(period)
      FROM course c
      WHERE c.course_name = e.course_name);
```

Note the use of the table aliases *c* and *e* to qualify the column names in the WHERE clause. I used the MIN function in this case, because some courses (Spanish I, for example) are offered in more than one period.

When using correlated subqueries, you can specify multiple columns in your SET clause; just be sure to enclose them within parentheses:

```
UPDATE enrollment e
SET (course_name, period) = (
      SELECT course_name, period
      FROM course c
      WHERE c.course_name = e.course_name
        AND c.period = e.period);
```

This UPDATE statement, which serves only as a usage example, essentially uses a subquery to set COURSE_NAME and PERIOD to their current values.

1.6 Deleting Data

Use the DELETE statement to delete rows from a table.

1.6.1 Simple Deletes

A simple DELETE statement takes the following form:

```
DELETE FROM table_name
WHERE selection_criteria
```

All rows meeting the selection criteria will be deleted. For example, to delete the sixth-period Spanish I course, specify:

```
DELETE FROM course
WHERE course_name = 'Spanish I'
  AND period = 6;
```

Be careful with DELETE statements. If you omit the WHERE clause, you will delete all rows from your table.

1.6.2 Deleting All Rows (TRUNCATE)

You can delete all rows from a table by issuing a DELETE without a WHERE clause:

```
DELETE FROM course;
```

Deleting all rows like this exacts a price. The deletion of each row must be logged to the database redo log, and a copy of each row to be deleted must be written to a rollback segment (or to an undo tablespace) in case the transaction is rolled back. A more efficient mechanism for deleting all rows from a table is the TRUNCATE statement:

```
TRUNCATE TABLE course;
```

The TRUNCATE statement only requires one, short entry in the database redo log and generates no rollback or undo data. As a result, it's faster to truncate a table than it is to delete all rows using a

DELETE statement. Beware, however! You cannot roll back a TRUNCATE statement. Once you issue it, all the data in your table is gone for good.

By default, TRUNCATE deallocates all extents assigned to the table. If you wish, you can save the extents for later use:

```
TRUNCATE TABLE course REUSE STORAGE;
```

This resets the table's high-water mark so that it no longer contains any rows, but all existing extents remain allocated to the table. Retaining the extents can be helpful if you plan to reload the table.

1.6.3 Deleting Duplicate Rows

It's occasionally necessary to delete "duplicate" rows from a table. This need probably arises more often on test systems than on production systems. One approach to deleting duplicate rows is to arbitrarily delete all but the one with the lowest ROWID value:

```
DELETE FROM course
WHERE ROWID NOT IN (
   SELECT MIN(ROWID)
   FROM course
   GROUP BY course_name, period);
```

The GROUP BY clause in the subquery defines "duplicate" as two rows having the same course name and period. The subquery returns a list of ROWID values in which each value represents the minimum ROWID for a given combination of course name and period. Those rows are retained. The use of NOT IN results in all other rows being deleted.

1.7 Merging Data (Oracle9*i*)

A common data-processing problem is the need to take some data, decide whether it represents a new row in a table or an update to an existing row, and then issue an INSERT or UPDATE statement as appropriate. In the past, this has always been at least a two-step process, requiring two round-trips to the database. New in Oracle9*i* is the MERGE statement, which makes the process of inserting or updating easier and more efficient than before.

The general form of the MERGE statement is as follows:

```
MERGE INTO table
```

```
USING data_source
ON (condition)
WHEN MATCHED THEN update_clause
WHEN NOT MATCHED THEN insert_clause;
```

In this syntax, *data_source* can be a table, view, or query. The condition in the ON clause is what Oracle looks at to determine whether a row represents an insert or an update to the target table.

Here is an example of a MERGE statement:

```
MERGE INTO course c
USING (SELECT course_name, period,
      course_hours
   FROM course_updates) cu
ON (c.course_name = cu.course_name
    AND c.period = cu.period)
WHEN MATCHED THEN
   UPDATE
   SET c.course_hours = cu.course_hours
WHEN NOT MATCHED THEN
   INSERT (c.course_name, c.period,
          c.course_hours)
   VALUES (cu.course_name, cu.period,
          cu.course_hours);
```

When processing this statement, Oracle reads each row from the query in the USING clause and looks at the condition in the ON clause. If the condition in the ON clause evaluates to TRUE, the row is considered to be an update to the COURSE_HOURS column. Otherwise, the row is considered a new row and is inserted into the COURSE table.

The following example shows the effects of the MERGE statement on the data in the COURSE table:

```
SQL> SELECT * FROM course;

COURSE_NAME         PERIOD COURSE_HOURS
--------------- ---------- ------------
Spanish I                1
```

```
U.S. History           3
English II             4
French I               5          4

SQL> SELECT * FROM course_updates;

COURSE_NAME        PERIOD COURSE_HOURS
--------------- ---------- ------------
Spanish I               1           3
U.S. History            3           3
English II              4           3
French I                5           3
Spelling                6           2
Geography               2           3

SQL> MERGE INTO course c
  2  USING (SELECT course_name, period,
  3         course_hours
  4    FROM course_updates) cu
  5  ON (c.course_name = cu.course_name
  6     AND c.period = cu.period)
  7  WHEN MATCHED THEN
  8    UPDATE
  9    SET c.course_hours = cu.course_hours
 10  WHEN NOT MATCHED THEN
 11    INSERT (c.course_name, c.period,
 12            c.course_hours)
 13    VALUES (cu.course_name, cu.period,
 14            cu.course_hours);
6 rows merged.

SQL> SELECT * FROM course;

COURSE_NAME        PERIOD COURSE_HOURS
--------------- ---------- ------------
Spanish I               1           3
```

```
U.S. History           3          3
English II             4          3
French I               5          3
Spelling               6          2
Geography              2          3
```

Notice that the existing four rows in the COURSE table have had their COURSE_HOURS values updated, and that two new rows have been added, all as the result of one statement.

1.8 Transaction Management

Oracle implements several statements to help you manage transactions. By default, a transaction begins whenever you issue your first SQL statement. Once a transaction begins, you end it by doing one of the following:

- Issue a COMMIT.
- Issue a ROLLBACK.
- Issue a DDL statement.

DDL statements (the ALTER and CREATE statements, for example) are special in that they implicitly end any open transaction. Thus, when issuing a DDL statement, it's possible to both begin and end a transaction with the same statement.

1.8.1 SET TRANSACTION

Use SET TRANSACTION to explicitly begin a transaction, especially when you want to specify transaction attributes such as isolation level.

```
SET TRANSACTION [attribute [,attribute...]
   NAME 'transaction_name';

attribute :=
   {READ {ONLY | WRITE}
   | ISOLATION LEVEL {SERIALIZABLE | READ COMMITTED}
   | USE ROLLBACK SEGMENT segment_name }
```

The READ COMMITTED isolation level is Oracle's default. It allows you to see changes made by other transactions as soon as they have been committed. Isolation-level SERIALIZABLE is more strict. With SERIALIZABLE, you can't modify any data that has been modified by others (but not committed before your transaction started). SERIALIZABLE also gives a consistent view of the data. You won't see changes committed by other users after your transaction begins. The following statement gives you a serializable transaction:

```
SET TRANSACTION
   ISOLATION LEVEL SERIALIZABLE
   NAME 'Jonathan''s Transaction';
```

READ ONLY transactions allow you to issue queries but not to change any data. READ ONLY transactions also provide read consistency. You don't see changes committed by other users during a READ ONLY transaction.

The USE ROLLBACK SEGMENT clause allows you to assign a transaction to a specific rollback segment. This is especially useful for large transactions, because you can assign those large transactions to a correspondingly large rollback segment. For example:

```
SET TRANSACTION
   USE ROLLBACK SEGMENT large_batch;
```

1.8.2 SAVEPOINT

SAVEPOINT allows you to established a named point in a transaction to which you can roll back if necessary.

```
SAVEPOINT savepoint_name;
```

1.8.3 COMMIT

COMMIT ends a transaction and makes permanent any changes made during the transaction.

```
COMMIT [WORK]
   [COMMENT 'text'
   |FORCE 'text'[, system_change_number]];
```

WORK is a "noise-word" and is usually omitted. The COMMENT clause allows you to associate a comment with a distributed transaction. The comment will be visible from the DBA_2PC_PENDING view if the transaction is ever in doubt. The FORCE clause can manually commit an in-doubt, distributed transaction.

1.8.4 ROLLBACK

ROLLBACK is normally used to end a transaction and undo any changes made during the transaction. It can also undo changes back to a specified savepoint:

```
ROLLBACK [WORK]
   [TO [SAVEPOINT] savepoint_name
   |FORCE 'text'];
```

As in COMMIT, WORK is a noise-word and is rarely used. Use the FORCE clause to roll back an in-doubt, distributed transaction. Use TO *savepoint_name* to roll back to a specific savepoint.

1.9 Formatting Text Reports

SQL*Plus reports are columnar in nature. The program lets you define column headings and display formats for each column in a report. You may also define page headers and footers, page and line breaks, and summary calculations such as totals and subtotals.

1.9.1 Column Headings

Specify column headings using the HEADING clause of the COLUMN command:

```
COLUMN employee_name HEADING "Employee Name"
```

You can use either single or double quotes to enclose the heading text. The resulting heading looks like this:

```
Employee Name
-------------
```

To specify a multiline heading, use the vertical bar (|) character to specify the location of the line break. For example:

```
COLUMN employee_name HEADING "Employee|Name"
```

The resulting multiline heading looks like this:

```
Employee
Name
```

```
---------
```

Headings of text columns are aligned to the left. Headings of numeric columns are aligned to the right. Use the JUSTIFY clause to alter that behavior:

```
COLUMN employee_name HEADING "Employee|Name" -
   JUSTIFY RIGHT
COLUMN employee_name HEADING "Employee|Name" -
   JUSTIFY CENTER
```

Use SET HEADSEP to change the line-break character to something other than a vertical bar. Use SET UNDERLINE to change the underline character to something other than a hyphen.

1.9.2 Column Formats

You can specify display formats with the FORMAT clause of the COLUMN command. For numeric fields, format specifications can be quite detailed 뾠 ontrolling the length, the number of decimal places, and the punctuation used in the number. For text and date fields, you can control the column width and whether the column wraps. The later section Section 1.11 shows how to format different types of data.

1.9.3 Page Width and Length

Page width is controlled by the SET LINESIZE command. The default width is 80 characters. You can change it 뾰 o 60 characters, for example 뾵 ith this command:

```
SET LINESIZE 60
```

You can use the LINESIZE setting to center and right-justify page headers and page footers.

Page length is controlled by the SET PAGESIZE command. The default is to print 24 lines per page, and this includes the page header and page footer lines. The following command changes the page length to 50 lines:

```
SET PAGESIZE 50
```

When using SET MARKUP HTML ON to generate HTML output, PAGESIZE controls the number of HTML table rows that display before column headings are repeated; each row may display as one or more physical lines depending on how the browser window is sized.

Setting PAGESIZE to zero has a special meaning in SQL*Plus. A PAGESIZE of zero inhibits the display of page headers, page footers, and column headings.

1.9.4 Page Headers and Footers

Define page headers and footers using the TTITLE and BTITLE commands. TTITLE, for top title, defines the page header. BTITLE, for bottom title, defines the page footer. The syntax is identical for both.

1.9.4.1 Defining a title

The following example defines a multiline page header with the company name on the left and the page number on the right:

```
TTITLE LEFT "My Company" CENTER "Current" -
RIGHT "Page" FORMAT 999 SQL.PNO SKIP 1 -
CENTER "Employee Listing" SKIP 4
```

The resulting title looks like this:

```
My Company      Current       Page   1
          Employee Listing
```

The final SKIP clause provides three blank lines between the page title and the column headers. The same clauses work in the BTITLE command to define page footers.

1.9.4.2 Getting the date into a title

To get the current date into a page title, you must:

1. Get the date into a user variable.
2. Place the user variable into your BTITLE or TTITLE command.

You can use the following commands in a SQL*Plus script to get the current date into a user variable:

```
SET TERMOUT OFF
COLUMN curdate NEW_VALUE report_date
SELECT TO_CHAR(SYSDATE,'dd-Mon-yyyy') curdate
```

```
   FROM DUAL;
SET TERMOUT ON
```

After executing the commands shown here, the date will be in a user variable named REPORT_DATE. The following command places that value into a page footer:

```
BTITLE LEFT "Report Date: " report_date
```

This same technique can also retrieve other values from the database and place them in either a page header or page footer.

1.9.5 Page Breaks

By default, SQL*Plus prints one blank line between each page of output. That blank line, added to the PAGESIZE setting, must equal the physical size of the pages in your printer.

The SET PAGESIZE command controls the number of lines SQL*Plus prints on a page. SET NEWPAGE controls SQL*Plus' action when a page break occurs. You can change the number of blank lines between pages using a command such as this:

```
SET NEWPAGE 10
```

You can tell SQL*Plus to display one form-feed character between pages by setting NEWPAGE to zero. For example:

```
SET NEWPAGE 0
```

Newer releases of SQL*Plus also allow SET NEWPAGE NONE, which eliminates both blank lines and form-feed characters between pages.

1.9.6 Report Breaks

The BREAK and COMPUTE commands define breaks and summary calculations for a report. BREAK also allows you to inhibit the display of repetitive column values.

1.9.6.1 The BREAK command

To eliminate repetitive column values, use the BREAK command as shown in this example:

```
SQL> BREAK ON owner
SQL> SELECT owner, table_name
  2  FROM all_tables
  3  ORDER BY owner, table_name;

OWNER      TABLE_NAME
========== ===============
CTXSYS      DR$CLASS
           DR$DELETE
           DR$INDEX
DEMO        CUSTOMER
           DEPARTMENT
           EMPLOYEE
```

When you list a column in the BREAK command, SQL*Plus prints the value of the column only when it changes. It's very important that you remember to sort the query results on the same column.

You can also use the BREAK command to skip lines or skip to a new page whenever a value changes. For example:

```
BREAK ON owner SKIP 1
BREAK ON owner SKIP PAGE
```

The first command prints a blank line whenever the owner changes. The second results in a page break each time the owner changes.

Multiple breaks may be specified for a report, but that's always done using just one command. The following example causes a page break to occur whenever an owner changes and a blank line to be printed whenever the object type changes:

```
BREAK ON owner SKIP PAGE ON object_type SKIP 1
SELECT owner, object_type, object_name
  FROM dba_objects
ORDER BY owner, object_type, object_name;
```

Before performing the break actions for a column, SQL*Plus first performs the break actions for all inner columns. In this case, a change in the owner field results in one skipped line and *then* a page break.

1.9.6.2 The COMPUTE command

The COMPUTE command tells SQL*Plus to compute summary values for a group of records. COMPUTE is always used in tandem with BREAK. For example, to compute the number of tables owned by each user, you can do the following:

```
BREAK ON owner
COMPUTE COUNT OF table_name ON owner
SELECT owner, table_name
   FROM dba_tables
ORDER BY owner, table_name;
```

SQL*Plus counts the number of table names for each distinct owner value and displays the results whenever a break occurs in the owner field.

You can compute summaries on multiple columns at once using multiple COMPUTE commands. The following example counts the number of objects of each type and sums the extent sizes for each object:

```
COMPUTE SUM OF bytes ON segment_name
COMPUTE COUNT OF segment_name ON segment_type
BREAK ON segment_type ON segment_name
SELECT segment_name, segment_type, bytes
   FROM user_extents
ORDER BY segment_type, segment_name;
```

Notice that the display order 쀽 he order used in the select list 쁼 oes not need to match the sort order or the break order. Also note that multiple summaries are defined using multiple COMPUTE commands, but multiple breaks are defined using just one BREAK command.

1.10 Tuning SQL

SQL*Plus can help tune SQL statements. You can use SQL's EXPLAIN PLAN facility to get the execution plan for a statement into a table. You can then query that table using SQL*Plus to display that plan. If you don't like the plan Oracle is using, you can add optimizer hints to your SQL statement that specify how you want the statement to be executed.

1.10.1 Creating the Plan Table

Before you can use the EXPLAIN PLAN statement, you need to create a plan table to hold the results. Oracle provides a script named *utlxplan.sql* to create the plan table, and you'll find it in your *$ORACLE_HOME/rdbms/admin* directory. Execute the script as follows:

```
SQL> @c:\oracle\ora92\rdbms\admin\utlxplan
Table created.
```

The resulting table, PLAN_TABLE, looks like this:

```
 Name                Null?    Type
 ------------------- --------- -----------------
 STATEMENT_ID                  VARCHAR2(30)
 TIMESTAMP                     DATE
 REMARKS                       VARCHAR2(80)
 OPERATION                     VARCHAR2(30)
 OPTIONS                       VARCHAR2(255)
 OBJECT_NODE                   VARCHAR2(128)
 OBJECT_OWNER                  VARCHAR2(30)
 OBJECT_NAME                   VARCHAR2(30)
 OBJECT_INSTANCE               NUMBER(38)
 OBJECT_TYPE                   VARCHAR2(30)
 OPTIMIZER                     VARCHAR2(255)
 SEARCH_COLUMNS                NUMBER
 ID                            NUMBER(38)
 PARENT_ID                     NUMBER(38)
 POSITION                      NUMBER(38)
 COST                          NUMBER(38)
 CARDINALITY                   NUMBER(38)
 BYTES                         NUMBER(38)
 OTHER_TAG                     VARCHAR2(255)
 PARTITION_START               VARCHAR2(255)
 PARTITION_STOP                VARCHAR2(255)
 PARTITION_ID                  NUMBER(38)
 OTHER                         LONG
 DISTRIBUTION                  VARCHAR2(30)
 CPU_COST                      NUMBER(38)
 IO_COST                       NUMBER(38)
 TEMP_SPACE                    NUMBER(38)
 ACCESS_PREDICATES              VARCHAR2(4000)
 FILTER_PREDICATES              VARCHAR2(4000)
```

The columns in the plan table vary from one release of Oracle to the next. This version of the plan table is from Oracle9*i* Release 2 (9.2.0.1.0).

1.10.2 Explaining a Query

Use the EXPLAIN PLAN statement to get the execution plan for a SQL statement. Oracle places the execution plan into the plan table you've created.

1.10.2.1 EXPLAIN PLAN syntax

The syntax for EXPLAIN PLAN looks like this:

```
EXPLAIN PLAN
       [SET STATEMENT_ID = 'statement_id']
       [INTO table_name]
       FOR statement;
```

Here are the parameters:

statement_id
: Identifies the query you are explaining and is stored in the STATEMENT_ID field of the plan table records. This defaults to null.

table_name
: The name of the plan table; defaults to PLAN_TABLE.

statement
: The SELECT, INSERT, UPDATE, or DELETE statement to be explained.

1.10.2.2 EXPLAIN PLAN example

First, delete any existing plan table records with the statement ID you wish to use. For example:

```
DELETE FROM plan_table
WHERE statement_id = 'HOURS_BY_PROJECT';
```

Insert the EXPLAIN PLAN statement at the front of the SQL statement you are interested in explaining; then execute the resulting, longer statement. For example:

```
EXPLAIN PLAN
SET STATEMENT_ID = 'HOURS_BY_PROJECT'
FOR
SELECT employee_name, project_name,
      SUM(hours_logged)
FROM employee, project, project_hours
WHERE employee.employee_id
      = project_hours.employee_id
   AND project.project_id
      = project_hours.project_id
GROUP BY employee_name, project_name;
```

Your next step is to query the plan table for the results.

1.10.3 Querying the Plan Table

The typical way to look at an execution plan is to display it using a hierarchical query. Oracle breaks down query execution into a series of nested steps, each of which feeds data up to a parent step. The ultimate parent is the query itself, the output of which is returned to the application. One possible query to display execution plan output looks like this:

```
SELECT LPAD(' ', 2*(level-1)) ||
      operation || ' ' || options
      || ' ' || object_name || ' ' ||
      DECODE(id, 0, 'Cost = ' || position)
      "Query Plan"
FROM plan_table
START WITH id = 0 AND statement_id
            = 'HOURS_BY_PROJECT'
CONNECT BY prior id = parent_id
      AND statement_id = 'HOURS_BY_PROJECT';
```

The result of this query is a report showing the steps in the execution plan, with each child step indented underneath its parent, as shown in this example:

```
Query Plan
----------------------------------------
SELECT STATEMENT   Cost = 7
  SORT GROUP BY
    HASH JOIN
      HASH JOIN
        TABLE ACCESS FULL EMPLOYEE
        TABLE ACCESS FULL PROJECT_HOURS
      TABLE ACCESS FULL PROJECT
```

The cost for one execution plan can be interpreted only relative to another. A query with a cost of 14 requires twice the I/O and CPU resources of a query with a cost of 7.

> Statistics are required to compute a cost. Out-of-date statistics result in an inaccurate cost.

The innermost execution plan steps generally return a set of database ROWIDs. These ROWIDs then feed operations that may in turn feed other operations. Sometimes an operation requires two sets of ROWIDs. An example of this is the HASH JOIN operation in the preceding execution plan, which requires ROWIDs from full scans of all the rows in the EMPLOYEE and PROJECT_HOURS tables.

Many execution plan operations are understandable, or at least somewhat guessable, from their names alone, especially when you factor in your knowledge of the results you are requesting via your query. You can find full descriptions of execution plan operations in the *Oracle9i Database Performance Tuning Guide and Reference* manual, and also in the tuning chapter of *Oracle SQL: The Essential Reference* by David C. Kreines (O'Reilly).

1.10.4 Using Optimizer Hints

Rather than allow Oracle to have total control over how a query is executed, you can provide specific directions to the optimizer through the use of hints. A *hint*, in Oracle, is an optimizer directive that is embedded in a SQL statement in the form of a comment. For example, here is a query with an optimizer hint telling Oracle to do a full table scan:

```
SELECT /*+ FULL(employee) */
       employee_id,
      employee_name,
      employee_billing_rate
FROM employee
WHERE employee_name = 'Jenny Gennick';
```

The hint in this case is FULL(employee) telling Oracle to do a full table scan of the employee table. Oracle will honor this hint and perform a full table scan even if there happens to be an index on the employee name field.

Hints must be in a comment of the form `/*+...*/`, and that comment must follow the SQL keyword that begins the statement.

If you make a mistake specifying a hint, it will be treated as a comment; you won't get any error message. Whenever you use a hint, you should do an EXPLAIN PLAN before and after applying the hint to ensure that it is being interpreted and applied as you intend.

The following sections describe some commonly used optimizer hints. For a complete list of all hints, see Mark Gurry's *Oracle SQL Tuning Pocket Reference* (O'Reilly).

1.10.4.1 Optimizer goal hints

Optimizer goal hints allow you to influence the optimizer's overall goal when formulating an execution plan:

ALL_ROWS

Produces an execution plan that minimizes resource consumption.

FIRST_ROWS and FIRST_ROWS(num)

Produces an execution plan that gets to the first row, or to the first *num* rows, as fast as possible. FIRST_ROWS(*num*) is new in Oracle9*i*.

CHOOSE

Uses the cost-based optimizer if statistics are present for at least one of the tables referenced in a query.

RULE

Uses the rules-based optimizer.

1.10.4.2 Access method hints

Access method hints allow you to control the manner in which Oracle accesses data:

FULL(*table_name*)

Does a full table scan of the specified table.

INDEX(*table_name [index_name ...]*)

Accesses the specified table via an index scan. Optionally, you may specify a list of indexes from which to choose. You may also use INDEX_ASC and INDEX_DESC to explicitly specify an ascending or descending scan, respectively.

INDEX_JOIN(*table_name [index_name ...])*

Rather than access the table, joins two or more of the table's indexes to retrieve the data required by the query. The indexes must collectively contain all columns referenced by the query. This hint is new in Oracle9*i*.

NO_INDEX (*table_name [index_name ...])*

Disallows the use of specified indexes on a table. If no indexes are specified, none of the table's indexes may be used.

1.10.4.3 Join order hints

Join order hints allow you to exercise some control over the order in which tables are joined:

ORDERED

Joins tables left to right in the same order in which they are listed in the FROM clause.

LEADING(*table_name*)

Makes the specified table the first table in the join order. This hint is new in Oracle9*i*. The ORDERED hint overrides LEADING.

STAR

Uses a star query execution plan, if at all possible. This can work only if there are at least three tables being joined, and if the largest table has a concatenated index on columns that reference the two smaller tables. The two smaller tables are joined first, and then a nested-loop join retrieves the required rows from the largest table.

1.10.4.4 Join operation hints

Join operation hints allow you to control the manner in which two tables are joined:

USE_NL(*table_name*)

Joins two tables using an outer loop to read one table and an inner loop to read all corresponding rows from the other table. You specify the inner table in the hint. This is known as a *nested-loops join*; it generally begins to return results most quickly, but overall may take the most time to execute.

USE_MERGE(*table_name*)

Performs a join on a table by separately sorting that table's rows on the join key, also sorting the other rowset's rows on the same join key, and then reading through both resulting rowsets to merge the two rowsets.

USE_HASH(*table_name*)

Performs a hash join on the specified table. Creates a hash table in memory using the join key values from the smaller of the two rowsets (or tables) to be joined. It then scans the larger table, applying the same hash function to its join key columns. Hash joins may be used only for equi-joins (in which the = operator is used).

1.10.4.5 Query transformation hints

A single query can sometimes be expressed in more than one way. Oracle will sometimes rewrite your queries to enable the use of a more efficient execution plan. Query transformation hints allow you to control how, or even whether, such rewriting takes place:

NO_MERGE

Use in a view's SELECT statement to prohibit merging of that view into an outer statement.

REWRITE(*[view...*])

Forces the use of materialized views where possible, regardless of cost. If you specify one or more materialized view names, only those views are considered for use. Use NOREWRITE to prevent the use of materialized views.

USE_CONCAT

Turns a query with OR conditions into two or more queries unioned together with a UNION ALL. This is known as *OR-expansion*. Use the NO_EXPAND hint to prevent OR-expansion.

1.10.4.6 Miscellaneous hints

Here are two useful miscellaneous hints:

APPEND

Don't attempt to reuse any free space that may be available in any extents currently allocated to the table. Instead, add rows above the current high-water mark. Applies only to INSERT statements and has the same results as doing a direct path load using SQL*Loader. Use NOAPPEND to explicitly request that existing free space be reused. APPEND is the default behavior for parallel inserts; otherwise, NOAPPEND is the default behavior.

ORDERED_PREDICATES

Causes predicates to be evaluated in the order in which you specify them in a WHERE clause. Unlike the other hints, you place this hint into the WHERE clause following the WHERE keyword.

.11 SQL*Plus Format Elements

The COLUMN, ACCEPT, SET NUMBER, TTITLE, BTITLE, REPHEADER, and REPFOOTER commands allow you to control data formats using what is called a format specification. A *format specification* is a string of characters that tells SQL*Plus exactly how to format a number, date, or text string when it is displayed.

1.11.1 Formatting Numbers

Table 1-3 shows the format elements that may be used when formatting numeric output.

Table 1-3. Numeric format elements

Format element	Function
9	Represents a digit in the output.
0	Marks the spot at which you want to begin displaying leading zeros.
$	Includes a leading dollar sign in the output.
,	Places a comma in the output.
.	Marks the location of the decimal point.
B	Forces zero values to be displayed as blanks.
C	Marks the place where you want the ISO currency indicator to appear. For U.S. dollars, this is USD.
D	Marks the location of the decimal point.
DATE	Causes SQL*Plus to assume that the number represents a Julian date and to display it in MM/DD/YY format.
EEEE	Causes SQL*Plus to use scientific notation to display a value. You must use exactly four Es, and they must appear at the right end of the format string.
G	Places a group separator (usually a comma) in the output.
L	Marks the place where you want the local currency indicator to appear. For U.S. dollars, this is the dollar sign character.
MI	Adds a trailing negative sign to a number and may be used only at the end of a format

	string.
PR	Causes negative values to be displayed within angle brackets. For example, -123.99 is displayed as <123.99>.
RN	Allows you to display a number using Roman numerals. An uppercase RN yields uppercase Roman numerals, while a lowercase r n yields lowercase Roman numerals. Numbers displayed as Roman numerals must be integers and must be between 1 and 3,999, inclusive.
S	Adds a + or - sign[1] to the number and may be used at either the beginning or end of a format string.
V	Displays scaled values. The number of digits to the right of the V indicates how many places to the right the decimal point is shifted before the number is displayed.

[1] SQL*Plus always allows for a sign somewhere when you display a number. The default is for the sign to be positioned to the left of the number and to be displayed only when the number is negative. Positive numbers have a blank space in the leftmost position.

Table 1-4 contains several examples illustrating the use of the various format elements.

Table 1-4. Numeric format examples

Value	Format	Result
`123`	`9999`	`123`
`1234.01`	`9,999.99`	`1,234.01`
`23456`	`$999,999.99`	`$23,456.00`
`1`	`0999`	`0001`
`1`	`99099`	`001`
`-1000.01`	`9,999.99mi`	`1,000.01-`
`1001`	`S9,999`	`+1,001`
`-1001`	`9,999PR`	`<1,001>`
`1001`	`9,999PR`	`1,001`

1.11.2 Formatting Character Strings

Character strings are formatted using only one element. That element is "A", and it is followed by a number specifying the column width in terms of characters. For example:

```
SQL> COLUMN a FORMAT A40

SQL> SELECT 'An apple a day keeps the doctor away.' A
  2    FROM dual;

A
----------------------------------------
An apple a day keeps the doctor away.
```

By default, longer text values are wrapped within the column. You can use the WORD_WRAPPED, WRAPPED, and TRUNCATED parameters of the COLUMN command to control whether and how wrapping occurs. For example:

```
SQL> COLUMN a FORMAT A18 WORD_WRAPPED

SQL> SELECT 'An apple a day keeps the doctor away.' A
  2    FROM dual;

A
------------------
An apple a day
keeps the doctor
away.
```

When text columns wrap to multiple lines, SQL*Plus prints a blank line called a *record separator* following the record. Use SET RECSEP OFF to prevent that behavior.

When used with the ACCEPT command, a character format defines the maximum number of characters SQL*Plus will accept from the user.

1.11.3 Formatting Dates

The date format elements in Table 1-5 may be used with Oracle's built-in TO_CHAR function to convert date values to character strings. For example:

```
SQL> SELECT TO_CHAR(SYSDATE,
  2 'dd-Mon-yyyy hh:mi:ss PM')
  3   FROM dual;

TO_CHAR(SYSDATE,'DD-MON
-----------------------
13-Dec-2001 09:13:59 PM
```

When used with the ACCEPT command, a date format string requires the user to enter a date in the format specified.

Table 1-5. Date format elements

Format element	Function
- / , . ; :	Punctuation to be included in the output.
'text'	Quoted text to be reproduced in the output.
AD or A.D. BC or B.C.	AD, A.D., BC, or B.C. indicator included with the date.
AM or A.M. PM or P.M.	AM, A.M., PM, or P.M. printed, whichever applies given the time in question.
CC	The century number?0 for years 1900 through 1999. See SCC.
D	The number of the day of the week 뾦 ne through seven.
DAY	The full name of the day.
DD	The day of the month.
DDD	The day of the year.
DY	The abbreviated name of the day.
E	The abbreviated era name. Valid only for Japanese, Imperial, ROC Official, and Thai Buddha calendars.
EE	The full era name. See E.
FF	The fractional seconds. Valid only for TIMESTAMP types.
FM	Suppresses extra blanks and zeros in the character string representation of a date. For

	example, use `'FMMonth DD'` to get `'July 4'` rather than `'July 4'`.
HH	The hour of the day on a 12-hour clock.
HH12	The hour of the day on a 12-hour clock.
HH24	The hour of the day on a 24-hour clock.
I	The last digit of the year number.
IW	The ISO week number, which can be 1-53. See IYYY.
IY	The last two digits of the ISO year number.
IYY	The last three digits of the ISO year number.
IYYY	The four-digit ISO year. The ISO year begins on Jan 1 only when Jan 1 falls on a Monday; it begins on the previous Monday when Jan 1 falls on a Tuesday through Thursday; it begins on the subsequent Monday when Jan 1 falls on a Friday through Sunday.
J	The Julian day. Day one is equivalent to Jan 1, 4712 BC.
MI	The minute.
MM	The month number.
MON	The three-letter month abbreviation.
MONTH	The month name, fully spelled out.
Q	The quarter of the year. Quarter one is Jan-Mar, quarter two is Apr-Jun, and so forth.
RM	The month number in Roman numerals.
RR	The two-digit year.
RRRR	The four-digit year.
SCC	Same as CC, but negative for BC dates.
SP	A suffix that converts a number to its spelled format (e.g., ONE, FOUR). See TH.
SPTH	A suffix that converts a number to its spelled and ordinal form (e.g., FIRST, FOURTH). See TH.
SS	The second.
SSSSS	The number of seconds since midnight.
SYEAR	The year spelled out in words with a leading negative sign when the year is BC. See YEAR.
TH	A suffix, which can appear at the end of any format element, resulting in a number (e.g., *DD*th) and which results in the ordinal version of the number (e.g., 1st, 4th).
TZD	The abbreviated time zone name (e.g., EST, PST, etc.).

TZH	The hour portion of the time zone displacement from UTC (Coordinated Universal Time) (e.g., -05 for U.S. EST).
TZM	The minute portion of the time zone displacement (usually zero).
TZR	The time zone region (e.g., "US/Eastern").
WW	The week of the year.
W	The week of the month. Week one starts on the first of the month; week two starts on the eighth of the month; and so forth.
X	The local radix character (e.g., the period [.] in American English).
Y,YYY	The four-digit year with a comma after the first digit.
YEAR	The year spelled out in words.
YYYY	The four-digit year.
SYYYY	The four-digit year with a leading negative sign when the year is BC.
YYY	The last three digits of the year number.
YY	The last two digits of the year number.
Y	The last digit of the year number.

When you use a date format element that displays a text value, such as the name of a month, the case used for the format element drives the case used in the output. Table 1-6 shows examples of formatting dates.

Table 1-6. Date format examples

Format	Result
dd-mon-yyyy	13-dec-2002
dd-Mon-yyyy	13-Dec-2002
DD-MON-YYYY	13-DEC-2002
Month dd, yyyy	December 13, 2002
mm/dd/yyyy	12/13/2002
Day	Friday