



Chapter 2

Building Blocks

The ability to use SQL as a reporting tool owes considerable credit to SQL*PLUS commands. It is these commands that allow you to interact with users at runtime, configure the reporting environment for display at the screen or at the printer, document reports with top and bottom titles, provide special column formatting instructions, compute summary statistics, structure the report output to produce different types of reports like frequency distributions, and create a variety of other effects.

Many frequently asked questions about ad hoc queries deal in some way with SQL*PLUS. Sometimes the solution is straightforward and involves only a single command; other times the solution is more complex and involves the coordination and interaction of several SQL*PLUS commands. This chapter deals with both types of questions.

Based only on the questions, there's little to distinguish the simple SQL*PLUS problems from the more complex ones. Producing one report from two queries turns out to be a simple problem, while sequencing page numbers across multiple reports that each begin on a new page turns out to be more complicated. Often, however, you can solve a complex problem once, save the result as an SQL*PLUS command file, give the file some meaningful name, and then reuse it whenever you want to achieve a certain effect.

It's this collection of single or multiple SQL*PLUS commands that provides query writers with building blocks useful for constructing queries. Do you need to configure an HP LaserJet printer, eliminate unwanted blank lines in a report, capture users' runtime responses for documentation, align pages for preprinted forms, or create a format for producing an ASCII output file? SQL*PLUS commands can handle all these problems and others.

Single Commands

Often a single SQL*PLUS command solves a query problem. One command in particular, the SET command, contains literally dozens of options that allow you to customize reports to achieve certain effects. Use SET to change page dimensions, determine the amount of space between report columns, force formfeeds in printed reports, eliminate trailing blanks from flat file output, pause between pages displayed at the screen, and solve other reporting problems. This section illustrates some of the many uses of the SET command.

The section also includes examples of the BREAK and COLUMN commands used in simple ways to double-space a report, reduce the display of redundant information, and hide columns from appearing in a report.

How do I concatenate a suffix to a substitution variable?

I've written a query that prompts the user for a data column value (e.g., 'F97'). I'd like to concatenate the string 'term.lis' to the user's response so that I can name the output file in a SPOOL command. I've tried '&&1'term.lis, but the single quotes get interpreted literally. How can I add a suffix to a substitution variable?

Approach

One of the SQL*PLUS system variables is CONCAT. This determines the character that ends a substitution variable when it's used in a longer string. By default, CONCAT is set to the period (.). To add a prefix or suffix to a substitution variable, use the CONCAT character as in the following command:

```
SPOOL &&yr.term.lis
```

The first period in the SPOOL command tells SQL*PLUS that the name of the substitution variable is 'yr' rather than 'yrterm.lis'.

See Also

1. Each of the shell programs in this book uses the CONCAT character to distinguish subdirectory path names. See, for example, Chapter 1 (Figure 1-8 on page 12).

How do I use a valid ampersand in a query?

I've written a query that includes an ampersand in one of the `WHERE` conditions. The ampersand is a legitimate part of the string comparison. Yet, when I run the query, I get prompted for a substitution variable. How can I designate the ampersand as valid?

Approach

Ampersands identify the start of an `SQL*PLUS` substitution variable. To use data column values like 'AT&T' or 'R&D' in the query, you'll need to alter the scanning for substitution variables. Several methods exist:

- Use the command `SET DEFINE OFF` to disable scanning for substitution variables. The older command `SET SCAN OFF` accomplishes the same thing.
- Change the character that identifies a substitution variable from an ampersand to something else. For example, '`SET DEFINE ~`' changes the substitution character from an ampersand to a tilde.
- Precede the ampersand in your query with the system escape character. In effect, this tells `SQL*PLUS` that the next character is valid. By default, the system `ESCAPE` feature is off. When `ESCAPE` is set on, the default character is the backslash (`\`). With `SET ESCAPE ON`, the following query would produce valid `SQL` matching `name_id` with 'B&N':

```
SELECT * FROM name WHERE name_id = 'B\&N';
```

See Also

1. The backslash `ESCAPE` character can also present problems. For example, HP printers use both ampersands and backslashes as part of the escape sequences that control print characteristics. In this case, you must tell `SQL*PLUS` that both the ampersand and the backslash are valid. See the discussion later in this chapter on page 66.

How do I reset substitution variables?

I've written a query that uses a substitution variable. The first time a user runs the program, she gets prompted for a value and the query completes correctly. But if she wishes to run the query again with another value for the substitution variable, the first query just reruns. What's going on here?

Approach

Substitution variables come in two flavors depending on how many ampersands precede them. An undefined substitution variable with one ampersand (e.g., &var1) generates a prompt each time it's encountered in the program. An undefined substitution variable with two ampersands (e.g., &&var2) only generates a prompt the first time it's encountered. This latter feature is convenient in queries that use the same substitution variable more than once.

Since you're always prompted for a value of an undefined single-ampersand substitution variable, it would not produce the problem described in the question above. However, double-ampersand substitution variables generate an implicit `DEFINE` that is in effect until you redefine or `UNDEFINE` the variable. This would create the query behavior described in the question.

Your query could `PROMPT` and then `ACCEPT` a value into a substitution variable; this redefines the variable. Or you could simply issue an `UNDEFINE` command (e.g., `UNDEFINE var2`) at the start or conclusion of the query. Either method allows you to run the query multiple times with new values each time.

See Also

1. Figure 2-9 on page 50 for an example of the use of `PROMPT` and `ACCEPT` commands.

How do I double-space a report?

Users sometimes complain that single-spaced reports are difficult to read if they're used for extended periods of time. They'd prefer double-spaced or even triple-spaced reports. How can I do this?

Approach

Use the `BREAK` command to skip lines. The following command breaks after each report row and skips one line before the next row displays, producing a double-spaced report:

```
BREAK ON ROW SKIP 1
```

A triple-spaced report would require `SKIP 2`.

See Also

1. You can skip to a new page using the `BREAK` command with the `SKIP PAGE` option. See Chapter 6 (Figure 6-12 on page 190) for an illustration of this technique.

How do I sort on a data column but not print it in the report?

A user requested a report that contains a data column with sensitive information. He doesn't need to see the actual values in this column, but the report does need to be sorted by the column so that he can identify who is in the top decile. How can I sort on a column but not print it?

Approach

This problem has an easy solution. Simply include the data column in the `ORDER BY` clause, but do not include it in the `SELECT` clause.

There may be other times, however, when more elaborate measures must be taken to hide a column that must be included in the `SELECT` clause for other reasons. In these cases, use the `NOPRINT` option in the `COLUMN` command. Figure 2-1 demonstrates how this works.

```
COLUMN salary NOPRINT
SELECT
  name_id "id",
  name_last||', '||name_first||' '||name_middle "name",
  comp_salary "salary"
FROM
  compensation,
  name
WHERE
  name_seriesno = name_now(name_id)
  and comp_id = name_id
ORDER BY 3 DESC;

id      name
-----
@330    Healey, Caroline V
@226    Sacco, Danielle
@505    Carrington, Thomas J
@133    Berger, Vincent
```

Figure 2-1: Hiding a data column by suppressing its printing.

Comments

► The query sorts results by salary. However, the `NOPRINT` option in the `COLUMN` command suppresses the printing of salaries.

See Also

1. See Chapter 4 (Figure 4-11 on page 123) for another example where `NOPRINT` is used to suppress the printing of a column.
2. The PL/SQL for the `name_now` function used in Figure 2-1 appears in Chapter 1 (Figure 1-9 on page 13).

How do I eliminate blank lines from a report?

In reports where wrapping occurs in one of the columns, I get blank lines that I'd like to remove. Is this possible? Figure 2-2 shows the problem.

```

22-MAY-1996 05:28          TABLE: ADDRESS          Page: 1
Report: ADDRESS.dic        Owner: OR2             SQL: tabldict.sql
                          Table of Addresses

Column Name              Type          Width  Scale Nulls  Column comments
=====
ADDRESS_ID              VARCHAR2      5      NOT NULL  ID of person
ADDRESS_TYPE            VARCHAR2      2      NOT NULL  Code for address type
ADDRESS_SERIESNO        NUMBER        2      0 NOT NULL  Sequential number for
                          1
                          addresses of a specified
                          address_type
-----
ADDRESS_STREET          VARCHAR2      25     NULL      Street address
ADDRESS_CITY            VARCHAR2      20     NULL      City of address
ADDRESS_LOC              VARCHAR2      5      NULL      State, province, or other
                          location of address
-----
ADDRESS_COUNTRY         VARCHAR2      3      NULL      Country code of address
ADDRESS_DATESTAMP       DATE          7      NULL      Date of entry or last update
    
```

Figure 2-2: Blank lines occurring after wrapped text.

Comments

► The blank lines break the report into sections that detract from its appearance.

Approach

Two SQL*PLUS system variables cause the effect shown in Figure 2-2. Both variables refer to *record separators*, which are single lines that separate rows in a report. The character used to create a record separator is defined by the system variable RECSEPCHAR. By default, this character is a space, meaning that the record separator will appear as a blank line. The second system variable, RECSEP, controls when the record separator actually appears. By default, the setting is WRAPPED, meaning that a record separator only occurs after wrapped lines. Other options for RECSEP are EACH and OFF. EACH creates a record separator after every report row; OFF turns off the record separator feature.

You can eliminate blank lines after wrapped text simply by including in your query the command SET RECSEP OFF.

See Also

1. Figure 2-13 on page 58 shows an example where RECSEP was set off to eliminate blank lines after wrapped lines.

How do I use a valid single quote in a query?

I've written a query that includes a single quote in one of the `WHERE` conditions. The single quote is a legitimate part of the string comparison. Yet, when I run the query, I get an error message. How can I designate the single quote as a valid part of the string?

Approach

Single quotes in `SQL*PLUS` can get pretty confusing. The following SQL, for example, generates an error:

```
SELECT * FROM name WHERE name_Last = 'O'Leary';
```

Doubling the single quote produces valid SQL:

```
SELECT * FROM name WHERE name_Last = 'O''Leary';
```

The need to double-up legitimate single quotes can produce some strange looking queries. Figure 2-3 shows a section of the program that produces data dictionaries (see Appendix A on page 322 for a complete listing). The program retrieves table comments from `ALL_TAB_COMMENTS` for use in the report title. Note the use of the four successive single quotes used in two places. The net effect is to place one single quote at the start and one at the end of the table comment.

```
/* get several column values from all_tab_comments */
COLUMN tabletype NEW_VALUE xType NOPRINT
COLUMN table_comments NEW_VALUE xTable_desc NOPRINT
SELECT
  table_type "tabletype",
  DECODE(comments, NULL, 'No description available',
  '''||comments||''') "table_comments"
FROM
  ALL_TAB_COMMENTS
WHERE
  owner = UPPER('&&xOwner')
  AND table_name = UPPER('&&xTable');
```

Figure 2-3: Concatenating a single quote to a string.

Comments

► Concatenating a single quote to a string requires four single quotes in succession. The two inner quotes produce a string containing one single quote. The two outer quotes are then needed to concatenate the quote to another string (e.g., the comments

data column in Figure 2-3). To produce the string O'Henry would require `'O''''''''Henry'`.

See Also

1. Quotes and blank spaces also figure prominently in problems encountered when passing arguments to SQL programs. For a discussion, see Chapter 1 on page 28.

How do I suppress the printing of redundant information?

Some of my reports contain duplicate information. For example, in a name and address report, if someone has two addresses that appear in the report, then the name also appears twice (see Figure 2-4). Is there some way to eliminate the redundant information and make the report easier to read?

id	name	address type	city address	state/ prov	postal code
a133	Berger, Kimberley	PR	Ipswich	MA	01938
a505	Carrington, Thomas	PR	Crary Mills	NY	13310
a330	Healey, Caroline	BU	Westchester	CO	80312
a330	Healey, Caroline	PR	Glen Falls	NY	12750
a330	Healey, Caroline	PR	Westchester	CO	80310
a226	Sacco, Danielle	BU	Charleston	IL	60954

Figure 2-4: Name and address report with redundant information.

Comments

▶ In this report, Caroline Healey has three addresses — two permanent addresses (i.e., address_type is PR) and one business address. As a result of the multiple addresses, her ID number and name each appear three times. Furthermore, the PR address_type appears twice. Removing these redundancies would improve the legibility of the report and improve its understanding.

Approach

What constitutes redundant information depends on the ORDER BY clause. The report in Figure 2-4 is apparently sorted by name and then by address type. You can remove duplicate information using a BREAK command that essentially mirrors the action of ORDER BY. Figure 2-5 illustrates this function of BREAK.

Comments

- ▶ Redundant information appears in the name_id, name, and address_type columns. BREAK on each of these columns in the order they appear in the ORDER BY clause.
- ▶ Coordinating the BREAK and ORDER BY removes the duplicate information and makes the report cleaner.

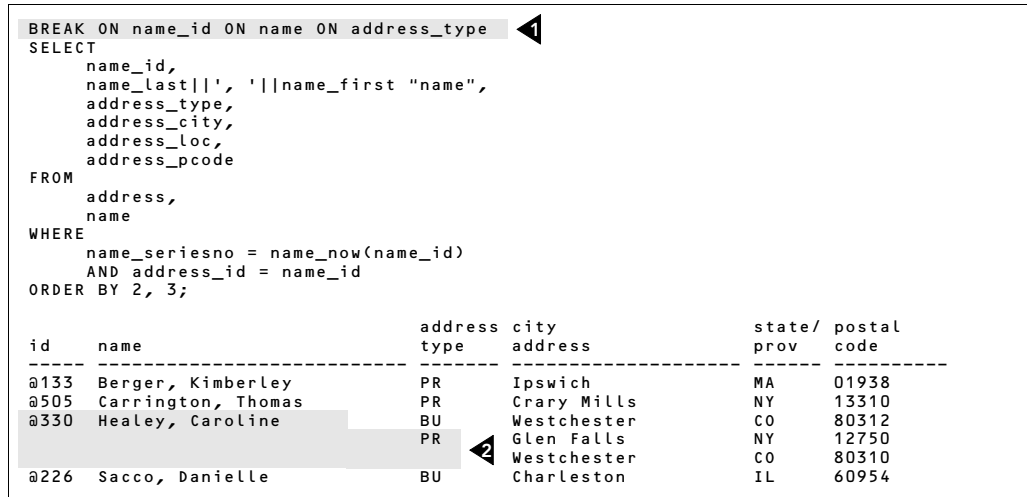


Figure 2-5: Redundant information removed with BREAK command.

See Also

1. Another example where the ORDER BY and BREAK commands coordinate to remove redundant information appears in Chapter 6 (Figure 6-14 on page 192).
2. The PL/SQL for the name_now function appears in Chapter 1 (Figure 1-9 on page 13).

How do I run two queries to produce one report?

For some reports I can't include all the summary information I'd like. For example, in Figure 2-6 I'd like to include an overall average sales. I can get this value easily enough in a second query, but the results always print on a new page. Is there some way I can combine the two reports?

Average sale by product		
product	xavg	count
MA1401	40	2
WK3224	10	1
*****		-----
Tot Num		3

Figure 2-6: Report missing overall average sale.

Approach

Under the default system settings, each report normally begins on a new page. However, you can disable this feature with the following command:

```
SET EMBEDDED ON
```

This allows a report to begin anywhere on a page. Thus you could run one query, then `SET EMBEDDED ON`, and then run a second query. The second report would begin immediately following the first query.

Figure 2-7 illustrates this technique.

Comments

▶ Use the `SET TRANSACTION` statement with the `READ ONLY` option to ensure that no changes committed by other users affect multiple queries. The `SET TRANSACTION` statement must be the first statement in your transaction. Ensure that this is the case by issuing a `COMMIT` or `ROLLBACK` before running the query. The `COMMIT` at the end of the two queries does not actually commit anything to the database; it merely ends the read-only transaction.

▶ With the `EMBEDDED` system variable on, two or more reports can be combined into what visually appears as one report. Note that `HEADING` was set off in the second report to suppress the display of header information, thus making it appear as if the second report is actually part of the first report.

```

SET TRANSACTION READ ONLY;
SET PAUSE OFF
SET ECHO OFF
SET FEEDBACK OFF

-- first query
BREAK ON product ON REPORT
COMPUTE SUM label 'Tot Num' OF count ON REPORT
SELECT
    sales_productid "product",
    avg(sales_total) "xavg",
    COUNT(*) "count"
FROM
    sales
GROUP BY
    sales_productid
ORDER BY 1;

-- second query
CLEAR BREAKS
CLEAR COMPUTES
SET HEADING OFF
SET EMBEDDED ON
COLUMN tavg FORMAT a8
SELECT
    'Tot Avg' "tavg",
    AVG(sales_total) "xavg"
FROM
    sales;
COMMIT

Average sale by product

product          xavg          count
-----
MA1401           40             2
WK3224           10             1
*****
Tot Num                    3
Tot Avg           30

```

Figure 2-7: Two queries combined into a single report.

See Also

1. The `SET TRANSACTION` command is useful whenever you have multiple coordinated queries. The `COMMIT` ends the read-only transaction. Chapter 8 (Figure 8-8 on page 241) shows another example of its use, where a bind variable gets defined in an anonymous PL/SQL block and then used in a SQL query.
2. For another `EMBEDDED` example, see Figure 2-18 on page 69.
3. Set `HEADING` off when you need total control over column headings. Block reports require this type of control. See Chapter 6 on page 186 for a discussion and illustration of this.

How do I force a formfeed in a report?

I have a preprinted form that I want to complete with results from a query. Yet, when I print the report on the form, the page doesn't eject correctly so nothing lines up properly. How can I force a formfeed after each row in the query gets printed? Figure 2-8 shows the query I'm using.

```
TTITLE 'Page: ' FORMAT 99 SQL.PNO;
SET HEADING OFF
COLUMN xrow FOLD_AFTER
COLUMN last FOLD_AFTER

SET PAGESIZE 7
BREAK ON ROW SKIP PAGE

SELECT
  NULL "xrow",
  NULL "xrow",
  LPAD(' ',5),
  name_id,
  NULL "xrow",
  LPAD(' ',8),
  name_last "Last",
  LPAD(' ',8),
  name_first||' '||name_middle
FROM
  name
WHERE
  name_seriesno = name_now(name_id)
ORDER BY 2;
```

Figure 2-8: Query without a physical page break.

Comments

► Ignore for now the actual SQL query that illustrates techniques for spooling results to a preprinted form. Chapter 6 discusses this in more detail. The misalignment problem occurs because the query relies on the `PAGESIZE` setting and the `SKIP PAGE` action in the `BREAK` command to align each new physical page correctly in the printer.

Approach

You need to distinguish between a page, as defined by `SQL*PLUS`, and the physical piece of paper in your printer. For `SQL*PLUS`, a page dimension is `PAGESIZE` lines high and `LINESIZE` characters wide. Each line in the top title, the bottom title, and the query results and any blank lines preceding the top title (i.e., `NEWPAGE`) count as lines in `PAGESIZE`.

The query in Figure 2-8 relies on the `SKIP PAGE` action in the `BREAK`

command to print each result row on a new page. When SQL*PLUS skips a page, it does so by skipping PAGESIZE (7) lines, which effectively moves to a new SQL*PLUS page. However, it may not be a new physical page unless PAGESIZE 7 adequately describes the preprinted form being used. Often you'll find yourself fiddling around to find the PAGESIZE setting that's consistent with the physical page.

It's safer to simply force a formfeed between pages. This is easily accomplished with the following command:

```
SET NEWPAGE 0
```

This produces a physical page break by sending a formfeed to the printer between each SQL*PLUS page, thus ensuring consistency between the top of a physical page and the top of an SQL*PLUS page.

See Also

1. If you'd like to know what's going on in the SELECT clause of the query in Figure 2-8, see the discussion on preprinted forms in Chapter 6 on page 190.
2. The PL/SQL for the name_now function appears in Chapter 1 (Figure 1-9 on page 13).

Coordinated Commands

Frequently the effect you wish to create in a report requires the use and coordination of several SQL*PLUS commands. Sometimes this is straightforward, as when coordinating the PROMPT and ACCEPT commands to provide runtime instructions to users and to capture their responses as user variables. However, at other times the coordination is more subtle, as when implicitly defining variables with the NEW_VALUE option in a COLUMN command.

This section discusses several common query problems, including ways to compress data columns so that they'll fit within the physical dimensions of a report, how to embed escape sequences in reports that will configure an HP LaserJet printer, how to use runtime responses from users to retrieve other database information, and how to creatively display NULL values in reports.

How do I customize runtime prompts for user variables?

Sometimes the default prompt that SQL*PLUS generates when I use substitution variables in queries is not particularly clear. I'd like to provide the user with more information about what kind of response is expected and valid. Is this possible?

Approach

Two SQL*PLUS commands make customizing runtime prompts straightforward. These commands are PROMPT and ACCEPT.

Appendix A (page 329) includes an SQL utility program that determines the report linesize needed to accommodate requested data columns. The program prompts the user for data column names that must be entered in a specific format or the SQL generated will not produce valid results. Figure 2-9 shows this portion of the program.

```

SET VERIFY OFF
SET TERMOUT ON

PROMPT Enter data columns using format: owner.table_name.column_name
PROMPT Use single quotes around entries and enclose within ().
PROMPT

ACCEPT ColumnName PROMPT 'Enter column names: '
SET TERMOUT OFF
    
```

Figure 2-9: Example of custom prompts for runtime user variables.

Comments

- ▶ By default, `SQL*PLUS` lists the portion of SQL containing substitution variables both before and after the substitution is made. Disable this setting with `SET VERIFY OFF`.
- ▶ The default prompt for a substitution variable is “Enter value for varname:”. Use the `PROMPT` command to add detail to the runtime request for user information. The line in Figure 2-9 that includes `PROMPT` without any text produces a blank line.
- ▶ When the user responds to the prompt, the value is accepted into a variable (called `ColumnName` in Figure 2-9). By default, `SQL*PLUS` makes the datatype of the user variable character, but you also can specify `DATE` or `NUMBER`. The maximum length of a character response is 240.

See Also

1. For another example where the `PROMPT` and `ACCEPT` commands get used to interact with a user, see Chapter 4 (Figure 4-12 on page 124). This figure illustrates how to sort query results based upon criteria set by the user at runtime.

How can I use runtime user responses to retrieve other information?

I'd like to construct new variables or retrieve additional report documentation based on the responses that users provide to runtime prompts. For example, if the query prompts for a table name and the user responds in lowercase, how can I use an uppercase table name in the report title?

Approach

There are many situations when user variables provide opportunities for further processing that would add value to the report. Examples include

- Cleaning the response for display, as when converting mixed-case responses to uppercase.
- Combining two or more responses into a single variable, as when concatenating an owner and table name into an owner.tablename or converting two numeric responses into a ratio.
- Verifying the accuracy of a response, as when the response should be a valid value in one of the database code tables.
- Retrieving additional information based on the response, as when accessing table comments based on the table name.

Chapter 1 discussed a useful SQL program that produces a data dictionary for a specified table. The listings for this program appear in Appendix A (page 322). The program contains two examples in which user variables are processed to provide additional information for the report.

Figure 2-10 shows one of the two examples from the data dictionary program. The technique implicitly defines a new variable called xReport that contains a report name constructed by concatenating the file extension '.dic' to the user-identified table name.

```

COLUMN reportname NEW_VALUE xReport NOPRINT
SELECT '&&xTable'||'.dic' "reportname" FROM DUAL;

```

Figure 2-10: Implicit define of a new variable.

Comments

- ▶ To implicitly define a new variable, you'll need to coordinate the COLUMN command and a simple query. The query processes the user variable, in this case concatenating a file extension to the

table name, and assigns this expression an alias. The alias is referenced in a COLUMN command containing the NEW_VALUE option to define a new user variable (called xReport).

The query in Figure 2-10 occurred against DUAL. This is convenient for simple processing of a user variable. However, you also may wish to access data in other tables or views. Figure 2-11 shows an example in which user responses are used to access ALL_TAB_COMMENTS.

```

/* define table type and table comments
from all_tab_comments */
COLUMN tabletype NEW_VALUE xType NOPRINT
COLUMN table_comments NEW_VALUE xTable_desc NOPRINT
SELECT
    table_type "tabletype",
    DECODE(comments, NULL, 'No description available',
    '||comments||') "table_comments"
FROM
    ALL_TAB_COMMENTS
WHERE
    owner = UPPER('&&xOwner')
    AND table_name = UPPER('&&xTable');

```

Figure 2-11: Implicit define of two user variables from data view.

Comments

- ▶ A COLUMN command with the NEW_VALUE option coordinates with the query to define two new user variables named xType and xTable_desc.
- ▶ Aliases used in the COLUMN command are identified in the SELECT query.
- ▶ The user was prompted for a table owner and a table name. The runtime responses to these prompts provide the information needed to access ALL_TAB_COMMENTS and retrieve the table type and comments that describe the table.

See Also

1. The DUAL table is owned by SYS. It contains only one data column and one row. You can use DUAL in many creative ways; for example, to create header lines in ASCII flat files for exporting to a word processor for mail merges, or to create escape sequences that control HP laser printers. See the index under DUAL table for specific examples.

How do I compress columns to fit the report linesize?

Frequently, the amount of information I want to include in a report exceeds the linesize my printer can handle. Are there any tricks to fitting data columns into limited report dimensions?

Approach

A variation of Murphy's law holds true for ad hoc reports — the amount of information requested by the user expands to fill (and just exceed) the amount of space available on the printed page. When this happens, you'll need the query writer's equivalent of the shoe horn to slip that extra column into a report.

First, ask the big questions:

- Did the user request so much information that there's no hope for including it in the standard row-and-column report? If so, then you have basically two choices: (1) create a block report (see Chapter 6) in which data values appear in block format underneath each other instead of across the page in columns (e.g., an address on an envelope is blocked), or (2) use a procedural program to achieve total control over the placement of data values on the printed page.
- Is the report truly bound by the `LINESIZE`? Can you change printers, for example, to a laser printer with compressed print? Or can you import the report file into a word processor and change the font and type size so that the report linesize is increased?

Let's assume that the linesize is truly limited. You could, for example, be stuck using standard greenbar paper with a maximum of 132 characters. Or, in the case of the reports that appear in this book, the limits of legibility are reached at 85 characters given the physical dimension across the page and the font used for the reports. Let's use an 85 linesize limit and work through an example of compressing reports.

Suppose you've been asked to produce a name and address report with information on ID number; last, first, and middle names; and street, city, state, and zip code address. Also include the type of address and its sequence number if more than one address of a single type exists.

linesize.sql

Appendix A (page 328) includes a utility program called `linesize.sql` that lets you quickly compute the minimum linesize needed to accommodate

all the data columns a user has requested. If the value it returns is less than the available report linesize, then there's no problem. However, if the computed minimum linesize exceeds what you've got available, then it's time to get out that shoe horn.

When you run `linesize.sql`, you'll be prompted for the data columns to appear in the report. Enter the column names in the format that concatenates owner, table_name, and column_name separated by periods, as in `owner.table_name.column_name` (e.g., `or2.name.name_id`). The program assumes that the default value of the `COLSEP` system variable applies so that one space separates each of the columns in the report. The program then uses the following algorithm to compute the minimum line size needed to include all requested data columns in the report.

1. For data columns with `CHAR` or `VARCHAR2` datatypes, the default display width used by `SQL*PLUS` is the width of the column in the database. The utility program (`linesize.sql`) accesses `ALL_TAB_COLUMNS` and retrieves this information for the column.
2. For data columns with the `NUMBER` datatype, the default display width used by `SQL*PLUS` is the greater of the following two values: (a) the width of the column heading, which defaults to the column name, or (b) the value in the `NUMWIDTH` system variable, which defaults to 10. A column named `address_seriesno`, for example, with a `NUMBER(2)` datatype will display at 16 characters because the column heading (`address_seriesno` by default) contains 16 characters and this is larger than `NUMWIDTH`. The `linesize.sql` program assumes that you'll change these defaults to use the minimum column size that's still wide enough to display any valid data. It computes this minimum size as the precision of the data column plus one character for a sign (+ or -) and, if the number is real, one character for a decimal point. Thus a column with datatype `number(5,3)` would display in $5 + 1 + 1 = 7$ columns. Use the `COLUMN` command to change the format of `NUMBER` data columns accordingly.
3. For data columns with the `LONG` datatype, the default display width used by `SQL*PLUS` is the smaller of the two system variables `LONG` or `LONGCHUNKSIZE`. The utility program sets each of these system variables to 80 and uses that value in its computations.
4. For data columns with the `DATE` datatype, the default display width used by `SQL*PLUS` depends on the National Language Support (NLS) parameters in effect. The utility program accesses

NLS_SESSION_PARAMETERS and determines the length of NLS_DATE_FORMAT. It then uses this value in its computations. For example, if the format is DD-MON-YY, a display width of 9 is used.

Example

Running linesize.sql shows that we'll need a minimum of 116 characters to include all the data columns in the example discussed on page 54. With the report linesize limited to 85, we're left with the problem of compressing the report by 31 characters.

Several strategies exist for reformatting data columns so that they'll fit within a restricted space. These include

- Reformat all NUMBER data columns to the minimum display size computed with the algorithm in linesize.sql.
- Whenever possible create expressions that would shorten CHAR or VARCHAR2 data columns. For example, concatenate last, first, and middle names into a name expression.
- Use creative headings that do not exceed the default display width for character data columns. For example, suppose a data column person_gender is VARCHAR2(1). If you set the heading as "sex", then you'll need to reset the format as A3 to include the entire heading. Alternatively, you can make use of the HEADSEP system variable to create a heading of "s|e|x", which only requires an A1 format. In this case, each letter of the heading appears on a separate line, one under the other.
- Reformat DATE data columns so that they require less space. For example, if the default format is DD-MON-YYYY, you can conserve display width by changing the format to DD/MM/YY or MM/DD/YY.
- Use the WORD_WRAPPED option for CHAR, VARCHAR2, and LONG data columns and use a format smaller than the default value. For example, you could reformat a varchar2(40) data column as A25 with the WORD_WRAPPED option activated.
- If you object to wrapped report lines, then run a simple query to determine the maximum length of values in the data column for the report population. If address_city is VARCHAR2(30) but nobody in the report population has a city address that exceeds 20 characters, then reset the format as A20. Obviously, this is a one-time solution.

Figure 2-12 shows the final report from the machinations needed to cram 116 characters into 85 spaces.

```

Name and address report
Date: 23-MAY-1996      Time: 11:19
Report name: temp.lis
Page: 1
SQL name: temp.sql
    
```

id	name	type	seq	street	csz	date stamp
						dd/mm/yy
@133	Berger, Kimberley	PR	1	Bayview Rd	Ipswich, MA 01938	01/05/96
@505	Carrington, Thomas J	PR	1	RR4	Crary Mills, NY 13310	01/05/96
@330	Healey, Caroline V	BU	1	McArthur and Phillips	Westchester, CO 80312	14/05/96
		PR	1	15 West St	Glen Falls, NY 12750	14/04/96
		BU	2	14 Ridgewood Ln	Westchester, CO 80310	01/05/96
@226	Sacco, Danielle	BU	1	1428 First Ave	Charleston, IL 60954	21/05/96

Figure 2-12: Report that illustrates several techniques for conserving display space.

Comments

- The report concatenates name data columns into an expression to save space.
- Redundant information is suppressed. While this doesn't conserve display width, it does create more white space and make the report easier to read.
- Rather than use an A4 format and the heading "type", the report uses the HEADSEP system variable to limit the format to A2.
- The datatype for address_seriesno is NUMBER(2). As an integer, the minimum display width is 2 + 1 = 3. The report sets a format and heading consistent with this minimum display width.
- Street address is WORD_WRAPPED to conserve display space. RECSEP is set off to eliminate blank lines after wrapped lines.
- The report concatenates city, state, and zip code data columns into an expression.
- The default date format was DD-MON-YY. The report uses a different format that conserves some space.

The SQL query that produced this report appears in Figure 2-13.

Comments

- The BREAK command suppresses the printing of redundant information on name, ID, and address type. Setting RECSEP off prevents blank lines from being inserted after wrapped lines.

```

BREAK ON name_id ON name ON address_type
SET RECSEP OFF
COLUMN name_id HEADING 'id'
COLUMN name FORMAT a20
COLUMN address_type HEADING 't|y|p|e'
COLUMN address_seriesno HEADING 'seq' FORMAT 99
COLUMN address_street HEADING 'street' FORMAT a18 WORD_WRAPPED
COLUMN csz FORMAT a22
COLUMN xdate HEADING 'date|stamp|dd/mm/yy' FORMAT a8
SELECT
    name_id,
    name_last||', '||name_first||' '||name_middle "name",
    address_type,
    address_seriesno,
    address_street,
    address_city||', '||address_loc||' '||address_pcode "csz",
    TO_CHAR(address_datestamp, 'dd/mm/yy') "xdate"
FROM
    address,
    name
WHERE
    name_seriesno = name_now(name_id)
    AND address_id = name_id
ORDER BY 2, 3;

```

Figure 2-13: Query that produced the report in Figure 2-12.

- The HEADSEP character helps retain meaningful headers whenever the width of the header would exceed the format used on the data column. Here, a “type” header would require A4 for the data column, but using the HEADSEP character allows you to use the default A2 instead.
- A format of 99 and a heading “seq” displays the integer values for address_seriesno in a column 3 characters wide (1 is for a sign). To control the display width for numeric data, you need to consider both the heading width and the data column format. If you don’t specify either a format or heading, address_seriesno displays in 16 characters (the length of the data column name). If you specify only a heading and the length of the heading is less than NUMWIDTH, then address_seriesno displays in 10 characters (the default NUMWIDTH). If you specify only a format, address_seriesno again displays in 16 characters because it defaults to the larger of NUMWIDTH or the length of the data column name. In short, specify both a heading and a format for numeric data columns.
- Wrapping character data at words retains meaning yet allows you to use a smaller format than the default.
- Concatenating character data columns usually reduces the

display space needed.

▶ The `TO_CHAR` function reformats the date column `address_datestamp`.

Creating formats and headers for data columns can be a real pain, especially when you need to squeeze too much information into too small a physical space. Make sure you save these `COLUMN` commands in the `SQL*PLUS` command file you use to create default headers for queries. Having created the column settings once, there's no reason to relive the experience.

See Also

1. Chapter 6 on page 186 discusses block reports.
2. `LONG` datatypes create special problems when you wish to display them in reports. See Chapter 5 on page 168 for a discussion.
3. The `PL/SQL` for the `name_now` function used in Figure 2-13 appears in Chapter 1 (Figure 1-9 on page 13).
4. By creating a command file with column headings you use frequently, you improve consistency between reports. The command file can be started from the shell of your queries. For a discussion, see Chapter 1 on page 20.

How can I display a value instead of a NULL in a report?

One of my users objects to the blank spaces in reports when the query returns NULL values. How can I display an 'unknown' instead?

Approach

Several methods provide control over the display of NULL values. The methods differ on how they affect the display globally. To change the display of NULL values at the most global level, for all data columns, change the default NULL system setting to include the text of your preference:

```
SET NULL 'unknown'
```

If you only want to change the display of NULL values in a single data column, specify the text to display by including the NULL option in a COLUMN command (e.g., COLUMN name_middle NULL 'not available').

And if you only want to change the display of NULL values for specific values in a report, use the DECODE or NVL functions to establish the criteria and display text.

The Figure 2-14 illustrates the three methods of controlling the display of NULL values.

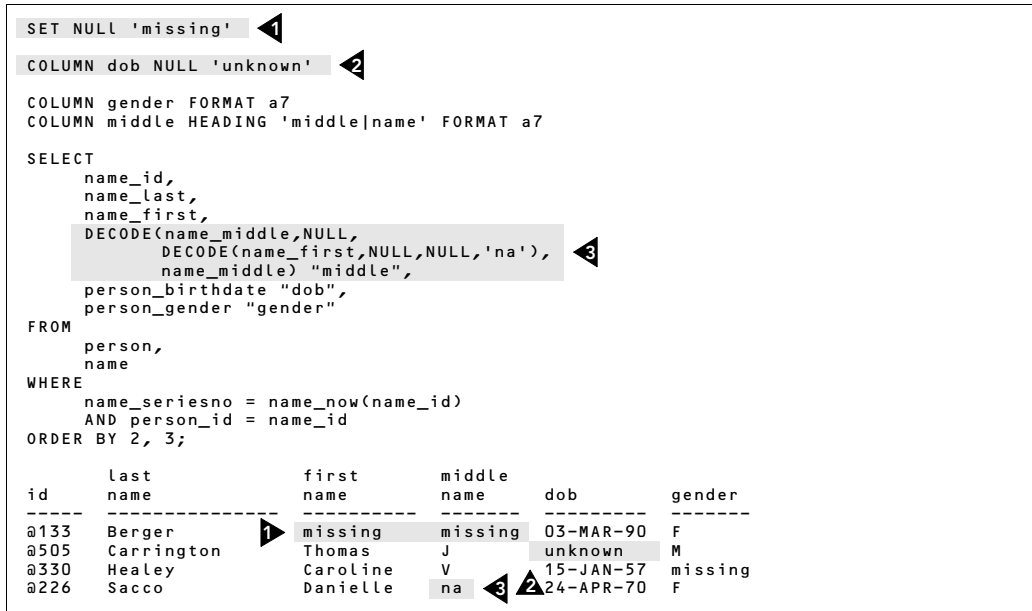


Figure 2-14: Three methods for displaying NULL values.

Comments

- ▶ **NULL** values appear by default as blank spaces in reports. You can change this default by setting the **NULL** system variable as some character string of your choice (e.g., 'missing' in this example). This change affects all data columns unless you specifically override the command.
- ▶ You can override the global treatment of **NULL** values in the **COLUMN** command. In Figure 2-14 if the date of birth is **NULL**, the report overrides the global setting and displays 'unknown'.
- ▶ You also can override global or column **NULL** settings by evaluating individual data values with a **DECODE** function. Based on the **DECODE** criteria, you can display **NULL** values in a variety of ways. In Figure 2-14 the **DECODE** checks the middle name. If middle name is **NULL**, the **DECODE** then checks the first name. If a first name exists, then a 'na' displays on the report for the **NULL** middle name; otherwise, the middle name is left as **NULL** and the global or column setting takes over.

See Also

1. **NULL** values frequently create problems for query writers. Beginning on page 144, Chapter 5 discusses several of these problems.
2. The **PL/SQL** for the `name_now` function used in Figure 2-14 appears in Chapter 1 (Figure 1-9 on page 13).

How can I easily format query output as an ASCII flat file?

I've written a query that produces a comma-delimited ASCII flat file. To format the query so that there aren't any titles, headers, or other extraneous lines, however, I need to set a bewildering array of system variables. Is there some easy way to do this?

Approach

No single system variable creates an SQL*PLUS environment suitable for producing a flat file that contains only the query results. However, one command does help considerably:

```
SET PAGESIZE 0
```

Setting PAGESIZE at 0 suppresses headers, top and bottom titles, and any blank lines that normally print before the top title. It even disables pausing. However, it does not affect feedback lines, timing statistics, or the echoing of SQL commands. You'll still have to disable each of these.

Figure 2-15 illustrates the use of PAGESIZE 0.

```
SET HEADING ON
SET NEWPAGE 4
SET PAUSE ON
TTITLE 'top title' SKIP 2;
BTITLE 'bottom title' SKIP 2;

SET ECHO OFF
SET FEEDBACK OFF
SET TIMING OFF

SET PAGESIZE 0
SET LINESIZE 40
SET TRIMSPOOL ON

SELECT
    name_id||','||
    name_last||','||
    name_first||','||
    name_middle||','x'
FROM name
WHERE name_seriesno = name_now(name_id);

@505,Carrington,Thomas,J,x
@133,Berger,Kimberley,,x
@330,Healey,Caroline,V,x
@226,Sacco,Danielle,,x
```

Figure 2-15: System variable setting suitable for producing an ASCII flat file.

Comments

▶ PAGESIZE 0 suppresses each of these system settings and title

commands.

▶ You'll still need to disable the echoing of SQL commands, the display of timing statistics, and any feedback information on the number of rows returned by the query.

▶ When producing an ASCII flat file, set the `LINESIZE` large enough so that it exceeds the maximum characters in a single output row. Setting `TRIMSPOOL` on trims trailing blanks from each output row.

See Also

1. See Chapter 6 on page 212 for a discussion of how to produce an ASCII flat file suitable for use with mail merges in word processors.
2. The PL/SQL for the `name_now` function appears in Chapter 1 (Figure 1-9 on page 13).

How do I format a report for output to an HP LaserJet printer?

Many of my reports get printed on a Hewlett-Packard (HP) LaserJet. I almost always print the reports in landscape orientation with compressed pitch and a right margin setting so that I can get at least 160 characters on each line. Is there any way that I can control these characteristics from the output report without having to manually set them on the printer or first import the report into a word processor and format it there? In other words, I'd like to simply print the file and have it properly formatted without requiring any other steps.

Approach

HP LaserJet printers use escape sequences to control many printer functions, including such things as page orientation, typeface families, margins, and font characteristics such as pitch and weight. By embedding the escape sequences in the report itself, you can control the printer without manual intervention.

You'll need two kinds of escape sequences embedded in your report — one to turn on the printer characteristics and a second to turn them off again so that the next person using the printer doesn't get surprised when her report prints all crazy. There are different methods you can use to embed the escape sequences. One convenient method embeds them in titles, using `TTITLE` to turn on the printer characteristics and `BTITLE` to turn them off. This toggles the escape sequences on and off for each page of the report.

Figure 2-16 illustrates this method of controlling an HP LaserJet printer.

Comments

- ▶ The `TTITLE` and `BTITLE` each include a user-defined variable; `xLandcomp` is the escape sequence that places an HP LaserJet IIIp in landscape orientation with compressed pitch (16.67 characters per inch), while `xReset` reactivates the default settings. These two variables get defined in the program `landcomp.sql` (see Figure 2-17). The precise escape sequence will depend on which printer you use.
- ▶ If you examine the report output file, it will contain escape sequences in the top and bottom titles. These sequences configure the printer; they will not appear on the printout. Note that in Figure 2-16 the `^` character denotes the escape character (ASCII

```

SET NEWPAGE 0
START &&object.\landcomp

TTITLE xLandcomp SKIP 1 'Page: ' FORMAT 99 SQL.PNO SKIP 2;
BTITLE LEFT xReset;

SELECT * FROM name;

■E■&l10■&k2s
Page: 1

id      last      first      middle      name
name      name      name      name      seqnum
-----
@505 Carrington Thomas J 1
@330 Vincent Caroline 1
@133 Berger 1
■E

■E■&l10■&k2s
Page: 2

id      last      first      middle      name
name      name      name      name      seqnum
-----
@330 Healey Caroline V 2
@226 Sacco Danielle 1
■E
    
```

Figure 2-16: Using escape sequences to an configure HP LaserJet printer.

character code 027). Other escape sequences determine other print characteristics. For example, the question indicated a preference for printing 160 characters on a line. You can set the right margin in an HP LaserJet printer with an escape sequence so that there are 160 characters per line.

The program landcomp.sql actually defines the escape sequences that appear in top and bottom titles. Figure 2-17 shows how this is done.

Comments

▶ HP LaserJet printer escape sequences sometimes contain ampersands. Yet SQL*PLUS uses ampersands (&) to identify the start of a substitution variable. When you need to use an ampersand as a valid character in a string, you must alert SQL*PLUS to that fact by preceding the ampersand with the system ESCAPE character. Here ESCAPE is set to the # character.

```

SET ESCAPE #
COLUMN landcomp NEW_VALUE xLandcomp NOPRINT
SELECT
  CHR(27)||'E' ||
  CHR(27)||'#&l10' ||
  CHR(27)||'#&k2S' "Landcomp"
FROM DUAL;

COLUMN reset NEW_VALUE xReset NOPRINT
SELECT CHR(27)||'E' "reset" FROM DUAL;

SET ESCAPE OFF

```

Figure 2-17: Defining escape sequences for HP LaserJet printers.

- ▶ This series of escape characters sets an HP LaserJet IIIp to landscape orientation with compressed pitch. Note that in two of the sequences a # character precedes an ampersand to signify that the ampersand is valid and is not the start of a substitution variable. The defines of xLandcomp and xReset occur implicitly using the same technique shown earlier in Figure 2-10 on page 52, by coordinating the NEW_VALUE option in the COLUMN command with the query aliases.

The procedures shown in Figure 2-16 and Figure 2-17 are useful if you typically print on one HP LaserJet. If you have several different HP LaserJet models, however, and they use different escape sequences for the same function, then you may find it more convenient to create an Oracle table with data columns for the printer name (e.g., LJ3p), escape sequence name (e.g., reset), and escape sequence (e.g., CHR(27)||'E'). Then, instead of landcomp.sql, you would create an exactly analogous query against your Oracle printer table instead of DUAL. The query would contain substitution variables that prompt for printer name and escape sequence name. When it's run, the query defines the appropriate escape sequences needed in the top and bottom titles of your report to configure the printer.

See Also

1. The character used to prefix substitution variables can be changed with the SET DEFINE command. By doing so, you could use valid ampersands without needing to precede them with the system ESCAPE character.

2. You can use the `DUAL` table in many creative ways. See Chapter 6 (Figure 6-32 on page 213) for an example where `DUAL` produces a header line in an ASCII flat file. Also see Chapter 8 (Figure 8-4 on page 236) where `DUAL` gets used in a `UNION` compound query to force a specific sort order when you cannot use `ORDER BY` (e.g., in subqueries).

How do I sequence pages numbers across multiple reports?

I have several queries that run nightly in batch from a single command file. Each query includes a page number in the top title. The page number resets to 1 for each new query even though I consider the queries all part of the same report. I'd prefer that the page numbering continue sequentially across queries. Is there some way I can do this?

Approach

When a query produces output, SQL*PLUS tracks the current page number (SQL.PNO) and line number (SQL.LNO). Under default system settings, the start of each new query resets the current page number and line number. Each display line in the report increments the current line number by 1. If more output remains after the current line number reaches the PAGESIZE setting, then the current page number gets incremented by 1, the current line number gets reset, and the process begins again.

Sequencing page numbers across reports requires the coordination of several SQL*PLUS commands. Figure 2-18 illustrates how this works.

Comments

- ▶ To increment page numbers across reports, SET EMBEDDED ON after the first query. Also ensure that the BTITLE command is active. Note in this example that the PAGESIZE was set at a small value (9) only for demonstration. Setting NEWPAGE 0 forces a formfeed between report pages.
- ▶ Setting EMBEDDED ON disables the reset of SQL.PNO and SQL.LNO that normally occurs between queries. This creates the incremented page numbers across multiple reports. However, it also means that the second and subsequent reports would start on the line immediately following the ending of the previous report. Sometimes this is desirable, but in this situation it is not. Each query should start on a new page (i.e., SQL.LNO should reset); it's only the page numbers that should continue to increment. Here's where the BTITLE command comes into play. With EMBEDDED ON, the BTITLE forces each new query to begin on a new page. The two commands coordinate to produce the desired effect.
- ▶ Note that pages are numbered continuously.
- ▶ With BTITLE on and PAGESIZE set at 9, each page has 9 lines.
- ▶ In this query, where the bottom title is only one line long, line 9 on each page contains the title.

```

TTITLE 'Page: ' FORMAT 999 SQL.PN0;
BTITLE ' ';
SET PAGESIZE 9
SET NEWPAGE 0
SELECT * FROM name;
SET EMBEDDED ON
SELECT address_id, address_city FROM address;

```

Page: 1				
id	last name	first name	middle name	name seqnum
@505	Carrington	Thomas	J	1
@330	Vincent	Caroline		1
@133	Berger	Kimberley		1
@330	Healey	Caroline	V	2

Page: 2				
id	last name	first name	middle name	name seqnum
@226	Sacco	Danielle		1

Page: 3	
id	city address
@505	Crary Mills
@330	Glen Falls
@133	Ipswich
@B&N	Cambridge

Page: 4	
id	city address
@330	Westchester
@330	Westchester
@226	Charleston

Figure 2-18: Multiple queries with sequential page numbers.

See Also

1. See page 46 in this chapter for another example where EMBEDDED gets used to produce a single report from two queries.
2. See page 48 for discussion on forcing formfeeds with NEWPAGE.

How can I put the value of a bind variable into a report title?

I've written a query that declares and then assigns a value to a bind variable that gets referenced in an anonymous PL/SQL block. I'd like to document my report by including the value of the bind variable in the report title. Is this possible?

Approach

Unlike a substitution variable, you cannot use a bind variable directly in the top and bottom title commands. You'll need to first implicitly define a user variable that contains the value of the bind variable. This can be done using the technique demonstrated earlier in Figure 2-10 on page 52.

The query in Figure 2-19 reports name information for a specified ID number. Two bind variables get declared, one for the value of the ID number to use in the report and a second that references a PL/SQL cursor that actually retrieves data for the report. Note that the ID number appears in the top title of the report, placed there after first defining a user variable called 'xvar' that contains the value of the ID number used in the report.

```

VARIABLE e REFCURSOR
VARIABLE id VARCHAR2(5)
EXECUTE :id := '@330'
DECLARE
    TYPE name_cursor IS REF CURSOR RETURN name%ROWTYPE;
    lc name_cursor;
BEGIN
    lc := :e;
    OPEN lc FOR SELECT * FROM name WHERE name_id = :id;
END;
/

SET TERMOUT OFF
COLUMN var NEW_VALUE xvar NOPRINT
SELECT :id "var" FROM DUAL;

SET TERMOUT ON
TTITLE LEFT 'ID is: ' xvar;
PRINT e

ID is: @330
      last      first      middle      name
id   name      name      name      seqnum
-----
@330 Vincent    Caroline   V          1
@330 Healey    Caroline

```

Figure 2-19: Using a bind variable in a report title.

Comments

- ▶ The `SQL*PLUS VARIABLE` command declares a bind variable named 'id'. The `EXECUTE` command assigns a value to the variable.
- ▶ The bind variable gets used in a query against `DUAL`. The `COLUMN` command defines the user variable 'xvar' as the value in the bind variable.
- ▶ Including 'xvar' in a `TTITLE` command places the value of the bind variable into the report title.
- ▶ Note that the `SQL*PLUS PRINT` command actually displays the report results retrieved by the `PL/SQL` cursor (named 'e').

See Also

1. Bind variables frequently get used in ad hoc queries that require more than one pass at the data. For example, you can use bind variables effectively when computing a median, percentage distributions, a top *N* report, or when sampling a row from a table. See the index under *bind variables* for examples that demonstrate the variety of their uses.

