



Chapter 4

Select Command

In one form or another the SQL `SELECT` command lies at the heart of every ad hoc query. It sometimes may seem lost, perhaps buried within an SQL*PLUS shell program or transfigured as a PL/SQL cursor, but it's always there. And if you don't get the `SELECT` right, your query and report are meaningless at best and dangerously misleading at worst.

The `SELECT` command is beguilingly simple, consisting of only six clauses in a natural language syntax:

- `SELECT` lists the items to appear in the report;
- `FROM` lists the tables needed;
- `WHERE` defines the report population and describes how to join the tables in the `FROM` clause;
- `GROUP BY` identifies criteria for grouping rows before performing group summaries such as counts, sums, and averages;
- `HAVING` limits the groups appearing in the final report; and
- `ORDER BY` sorts the query results.

In any given query, only the `SELECT` and `FROM` clauses are mandatory. You would be wrong, however, to assume that the `SELECT` command is simple. A single `SELECT` may consist of hundreds of lines of programming, with the most critical and most obtuse portions generally occurring in the `WHERE` clause that defines the report population and constructs the join relationships between tables. Reducing the complexity of `SELECT` commands with the appropriate use of SQL*PLUS and PL/SQL makes ad hoc queries much more accessible to nonprogrammers.

SELECT Clause

In the `SELECT` clause you list the data columns, expressions, pseudocolumns, literals, and other items that you want to appear in the report. Most questions about the `SELECT` clause concern expressions and how to build them properly to achieve certain effects. These have been discussed previously in Chapter 3 (Functions). This section examines only a single question concerning `ROWID`.

How do I display all data columns in a table and the ROWID?

I'd like to query all data columns in a table and also include the `ROWID`. When I do a "`SELECT *, ROWID FROM tablename`", I get an error message. I know I could explicitly name each data column in the `SELECT` clause, but is there a simpler method?

Approach

Make a minor modification in the query so that instead of `SELECT *` you use `SELECT tablename.*`. Figure 4-1 shows how this works.

```
SELECT name.*, ROWID FROM name;
```

id	last name	first name	middle name	name seqnum	ROWID
a505	Carrington	Thomas	J	1 000001F0.0000.0002	
a330	Vincent	Caroline		1 000001F0.0001.0002	
a133	O'Leary	Vincent		1 000001F0.0002.0002	
a330	Healey	Caroline	V	2 000001F0.0003.0002	
a226	Sacco	Danielle		1 000001F0.0004.0002	

Figure 4-1: Selecting `ROWID` and all data columns in a table.

Comments

► Using `NAME.*` selects all columns from the `NAME` table and allows you to also include literals, pseudocolumns like `ROWID`, or data columns from other tables listed in the `FROM` clause.

See Also

1. `ROWID` is another tool that serves query writers well. You can use `ROWID`, for example, to help optimize performance (see Figure 4-6 on page 113) or to debug queries (see Chapter 7, Figure 7-9 on page 229).

WHERE Clause

If the SQL SELECT command is the heart of each ad hoc query, then the WHERE clause is the heart of each SQL SELECT. Because all testing and debugging is predicated on a stable report population, it's here where you must consciously and carefully identify the report population. For example, once a population gets defined, the query should not lose population items — a process known as *falling through the joins*. Similarly, the query should not pick up population items — a process known as *multiplying through the joins*. Being rock solid about what constitutes the report population and constructing the joins so that they prevent population loss or gain and yet simultaneously return results in a reasonable time provide the major challenges in the WHERE clause. The WHERE clause also can significantly affect query performance and thus the cost of the query.

How do I create an address hierarchy?

A user requested a basic name and address report with a slight twist. She'd like a person's permanent address displayed if it exists; if not, she wants the business address displayed. How can I write a query to do this?

Approach

The user is essentially asking for addresses displayed in a hierarchy with permanent addresses as the highest priority and then business addresses as a second-level priority. In this case the hierarchy only includes two priorities, but the idea easily could be extended to three or more.

Figure 4-2 shows a few rows of data from an address table where the primary key is address_id, address_type, and address_seriesno. Assume that the status indicator (A = active and I = inactive) identifies a valid address. The data include each of the four situations that could occur in a two-level hierarchy. A single person could have

- One or more valid permanent addresses and no valid business address;
- One or more valid business addresses but no valid permanent address;
- One or more valid permanent addresses and one or more valid business addresses; or
- Neither a valid permanent or business address.

This question did not specify what action to take for people who have neither a permanent nor a business address. For now, let's assume that these people will not be included in the report. The next section extends the discussion to include situations where no valid data exist but the person must still appear in the report.

id	type	seq	status
@133	PR	1	A
@226	BU	1	A
@330	BU	1	A
@330	PR	1	I
@330	PR	2	A
@505	PR	1	I

Figure 4-2: Sample address data.

Comments

▶ The person with ID @330 has two permanent addresses, but only one is active. She also has one business address. This person represents one of the four situations that can occur in a two-level hierarchy — where the person has valid data for each priority.

Note that other people in the figure represent the other three situations. That is, @133 has a valid permanent address but no valid business address; @226 has the reverse — a valid business address but no valid permanent address; and @505 has neither a valid permanent address nor a valid business address.

SQL Approach

Figure 4-3 shows an SQL query that displays address information in a (PR:1, BU:2) hierarchy. Note the complex WHERE clause that uses OR and a NOT EXISTS correlated subquery to enforce the hierarchy. Also note that the query uses a PL/SQL function called max_address that returns the most recent address of a given type.

Comments

▶ The WHERE clause that selects the most recent active permanent address begins at this point. Both @133 and @330 meet these criteria.

▶ The WHERE clause that selects the most recent active business address begins here. Both @226 and @330 meet these criteria. Note that @330 meets both criteria at this point.

```

SELECT
  name_id AS id,
  address_type AS type,
  address_seriesno AS seq,
  address_status AS status
FROM
  address a1,
  name
WHERE
  name_seriesno = name_now(name_id)
  AND a1.address_id = name_id
  AND
  (
    1▶ (a1.address_type = 'PR'
        AND a1.address_status = 'A'
        AND a1.address_seriesno =
            max_address(name_id, a1.address_type))
    OR
    2▶ (a1.address_type = 'BU'
        AND a1.address_status = 'A'
        AND a1.address_seriesno =
            max_address(name_id, a1.address_type))
    3▶ AND NOT EXISTS
        (SELECT 'x' FROM address a2
         WHERE a2.address_id = name_id
         AND a2.address_type = 'PR'
         AND a2.address_status = 'A'))
  )
ORDER BY 1;

id   type seq status
-----
@133 PR   1  A
@226 BU   1  A
@330 PR   2  A
    
```

Figure 4-3: SQL query that returns an address hierarchy.

- ▶ The NOT EXISTS clause deals with people like @330 who have both an active permanent address and an active business address by excluding the active business address.
- ▶ The query correctly returns the three rows expected.

For completeness, the PL/SQL function that returns the most recent active address of a specified type is shown in Figure 4-4.

Comments

- ▶ The function gets passed an identification number and an address type. The cursor retrieves the most recent active address for that ID number and address type.

```
CREATE OR REPLACE FUNCTION max_address
  (id VARCHAR2, type VARCHAR2)
RETURN NUMBER
AS
max_seriesno NUMBER(2);

CURSOR main_cursor (p_id VARCHAR2, p_type VARCHAR2) IS
SELECT MAX(address_seriesno)
FROM address
WHERE address_id = p_id
AND address_type = p_type
AND address_status = 'A';

BEGIN
OPEN main_cursor(id, type);
FETCH main_cursor INTO max_seriesno;

IF main_cursor%NOTFOUND THEN
max_seriesno := NULL;
END IF;

CLOSE main_cursor;
RETURN max_seriesno;
END;
```

Figure 4-4: PL/SQL function that identifies the most recent address.

PL/SQL Approach

The SQL query shown in Figure 4-3 provided accurate results but required a fairly complex WHERE clause. Extending the hierarchy to three priority classifications would complicate the query even further. Constructing a PL/SQL function that returns the correct address in a hierarchy greatly simplifies the query. Figure 4-5 illustrates this simplification.

Comments

▶ A single PL/SQL function replaces the complex WHERE clause that produced the address hierarchy in Figure 4-3. Note that the function gets passed an ID number and the hierarchy (i.e., PR addresses as first choice and BU addresses otherwise). Note, too, that the function returns a ROWID that uniquely identifies a single row.

Figure 4-6 shows the PL/SQL function that enforces the address hierarchy. Note that even this is simpler than the SQL solution.

```

SELECT
  name_id AS id,
  address_type AS type,
  address_seriesno AS seq,
  address_status AS status
FROM
  address a1,
  name
WHERE
  name_seriesno = name_now(name_id)
  AND a1.address_id = name_id
  AND a1.ROWID =
    address_hierarchy(name_id, 'PR', 'BU')
ORDER BY 1;

```

id	type	seq	status
@133	PR	1	A
@226	BU	1	A
@330	PR	2	A

Figure 4-5: Query using a PL/SQL function to return an address hierarchy.

```

CREATE OR REPLACE FUNCTION address_hierarchy
  (id VARCHAR2, type1 VARCHAR2, type2 VARCHAR2)
RETURN ROWID
AS
  hierarchy_row ROWID;

CURSOR address_cursor
  (p_id VARCHAR2, p_type1 VARCHAR2, p_type2 VARCHAR2) IS
  SELECT ROWID
  FROM address
  WHERE address_id = p_id
    AND address_type IN (p_type1, p_type2)
    AND address_status = 'A'
    AND address_seriesno = max_address(p_id, address_type)
    ORDER BY DECODE(address_type, p_type1, 0, p_type2, 1);

BEGIN
  OPEN address_cursor(id, type1, type2);
  FETCH address_cursor INTO hierarchy_row;

  IF address_cursor%NOTFOUND THEN
    hierarchy_row := NULL;
  END IF;

  CLOSE address_cursor;
  RETURN hierarchy_row;
END;
/

```

Figure 4-6: PL/SQL function that retrieves an address hierarchy.

Comments

- ▶ Queries that utilize ROWIDs generally optimize performance.
- ▶ The cursor selects active addresses of the two types passed as arguments in the function.
- ▶ Eligible address rows get sorted based on their type, with the highest-priority classification (i.e., type1) appearing first. In the execution section, the `FETCH` retrieves only the first row. In cases where someone had both a valid permanent address and a valid business address, the `ORDER BY` ensures that the permanent address appears first, gets `FETChEd`, and subsequently gets returned.

See Also

1. Chapter 9 (Table 9-4 on page 259) shows results of performance comparisons for the `SQL` and `PL/SQL` approaches to address hierarchies. These results again indicate a trade-off between query elegance and performance.

How do I create an address hierarchy (continued)?

The user would like to see addresses displayed in a permanent-then-business hierarchy, but if neither a permanent nor a business address exists, she still wants the person to appear in the report. How can I do this?

Approach

The problem has both SQL and PL/SQL solutions. The curious among you can revise the SQL in Figure 4-3 on page 111 to display the correct report. Note that in this query the person (i.e., ID = @505) without a valid permanent or a valid business address must appear in the report.

The PL/SQL solution only requires that an outer join be added to the query in Figure 4-5 on page 113. Figure 4-7 shows the results.

```

SELECT
  name_id AS id,
  address_type AS type,
  address_seriesno AS seq,
  address_status AS status
FROM
  address a1,
  NAME
WHERE
  name_seriesno = name_now(name_id)
  AND a1.address_id(+) = name_id
  AND a1.ROWID(+) =
      address_hierarchy(name_id, 'PR', 'BU')
ORDER BY 1;

```

id	type	seq	status
@133	PR	1	A
@226	BU	1	A
@330	PR	2	A
@505			

Figure 4-7: Preserving population using outer joins with an address hierarchy.

Comments

▶ The outer joins retain the person (@505) without a valid permanent or a valid business address. They also would retain anyone without any address information.

See Also

1. Outer joins can pose special problems for query writers. Beginning on page 156, Chapter 5 discusses several of these problems.

GROUP BY Clause

The `GROUP BY` and `HAVING` clauses tend to be the black sheep of the `SELECT` command clauses. They're seen less frequently and are subject to more misunderstanding than other clauses. But both play a vital role, `GROUP BY` to specify the criteria used to group rows in the result set so that summaries of those groups can be made and `HAVING` to limit which groups actually display in the report (i.e., a kind of `WHERE` clause but applied to groups).

How do I GROUP BY a date?

I'm trying to count the number of seminars by their starting date, but the count is always 1 even though some of the dates are clearly the same. What's going on here?

Approach

The problem occurs because of a misunderstanding about the `GROUP BY` criterion. Dates include information on hours, minutes, and seconds. Grouping by a date also uses this time information. Consequently, two seminars that start on the same day but at different times will be grouped separately and each will receive a count of 1 in a frequency distribution report. The trick is to format the date column. Figure 4-8 uses the `TRUNC` function to truncate dates to the nearest day with a time of midnight. Other techniques could accomplish the same thing.

Comments

- ▶ Using `TRUNC` without a format truncates to the nearest day. You could apply a format to truncate to another unit such as month, quarter, or year if you wanted counts by these more aggregate units.
- ▶ In this example, two seminars start on `15-JAN-1997`.

See Also

1. This problem is very similar to one discussed in Chapter 3 on page 80 where the query specified a date in a `WHERE` condition but returned the wrong results. All dates include a time component and require care when used without formatting functions like `TRUNC` or `TO_CHAR`.

```

BREAK ON REPORT
COMPUTE SUM OF count ON REPORT
SELECT
    TRUNC(seminar_start_date) AS sdate,
    COUNT(*) "count"
FROM
    seminars
GROUP BY
    TRUNC(seminar_start_date)
ORDER BY 1;

start
date      count
-----
04-JUN-96      1
16-JUN-96      1
24-JUN-96      1
15-JAN-97      2
10-FEB-97      1
31-DEC-99      1
-----
sum           7
    
```

Figure 4-8: Using TRUNC to GROUP BY dates.

When does a GROUP BY also ORDER BY?

I'd like to create a sorted view of data, but the `ORDER BY` clause cannot appear in a view. I've noticed that a `GROUP BY` seems to sort query results. But is this always the case? Is it possible to use a `GROUP BY` to accomplish the sort?

Approach

The `ORDER BY` clause cannot appear in a subquery. Thus commands that use subqueries, such as `CREATE VIEW` or `INSERT`, cannot include sorting criteria. A later section discusses sorted inserts (see Figure 4-16 on page 131). The approach used in that section also will work for the `CREATE VIEW` problem in this question. However, the question here has an interesting dimension aside from the immediate problem of creating a sorted view. Many times a `GROUP BY` will sort a report, but sometimes it will not. Let's explore the issue a little more deeply.

There is a fundamental difference between `GROUP BY` and `ORDER BY` that may be underappreciated. If you `ORDER BY col1 and col2` (i.e., `ORDER BY col1, col2`), the result set is sorted on `col1` and, in case of ties, then sorted on `col2`. However, if you group on the same two columns (i.e., `GROUP BY col1, col2`), the order of the columns in the `GROUP BY` clause can sometimes have absolutely no effect on the result set. You could reverse the column order (i.e., `GROUP BY col2, col1`) and get exactly the same result. Clearly, therefore, something other than a simple sorting occurs with `GROUP BY`. It's complicated by the presence of indexes.

Consider Figure 4-9 that retrieves an identification number and last name using an index where the leading edge of the index is last name. The report is not sorted by the `GROUP BY` criteria; that is, the sort order is not by `ID` and then last name.

Comments

- ▶ If the `GROUP BY` also sorted this query, then the order of the result set should be `name_id` and then `name_last`.
- ▶ Note that the `name_id` column is not sorted. Apparently in building the intermediate results, a sort on `name_id` was not necessary.
- ▶ Furthermore, note that not even the two rows for person @330 appear next to each other in the report. The result set is definitely grouped by the combined `name_id` and `name_last` columns, but not in a way analogous to the `ORDER BY` procedure.

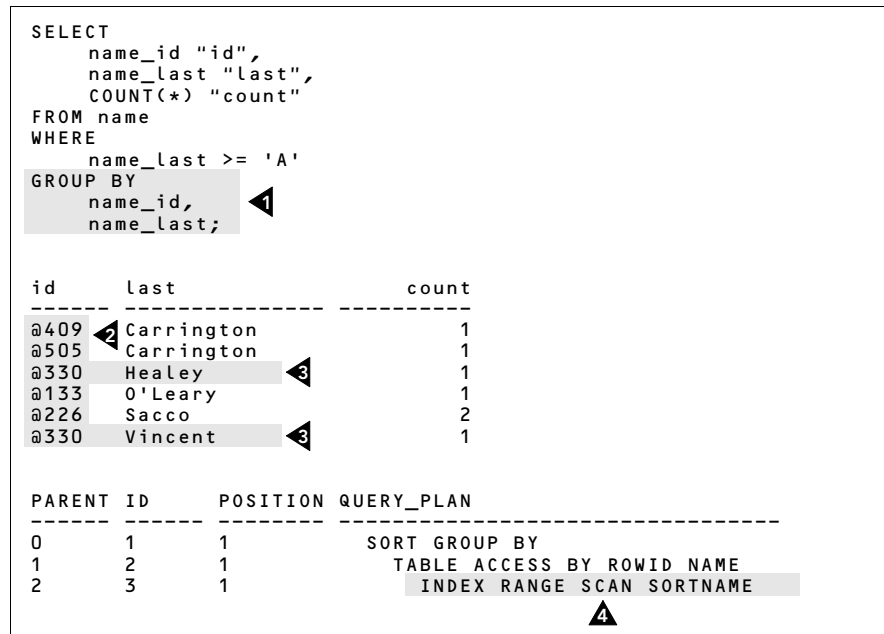


Figure 4-9: Example of GROUP BY that does not ORDER BY.

4 The EXPLAIN PLAN shows that the query used the sortname index to retrieve rows. The leading edge of this index is name_last — meaning rows got retrieved based on last name. Examine the report; it appears sorted on last name.

If you remove the complications provided by indexes by forcing a full table scan, for example, then the query results appear much different. Figure 4-10 shows exactly the same query without the WHERE clause condition that caused the Oracle optimizer in Figure 4-9 to utilize the sortname index. Note the changes in the results.

Comments

- 1 In this query the GROUP BY did sort the result set by name_id.
- 2 It also sorted by name_last.
- 3 Note that a full table scan of the name table occurred. The GROUP BY effectively sorted first by name_id and then by name_last when building its intermediate results. If you reverse the order of the two columns in the GROUP BY (i.e., GROUP BY

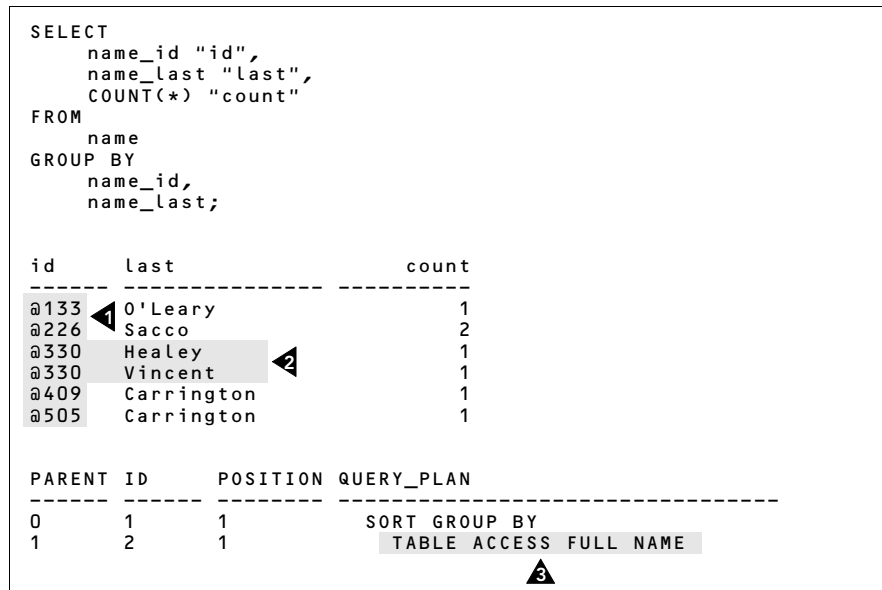


Figure 4-10: Full table scan and GROUP BY results.

name_last, name_id), the report is sorted first by last name and then by ID.

Here's some advice on using GROUP BY to ORDER BY: be careful. If used on only a single data column, GROUP BY and ORDER BY do produce the same results. But if two or more data columns are involved, then the table indexes must be consistent with the ordering you hope to achieve, or you must structure the query to disable indexes and force a full table scan (or use the optimizer hint FULL).

See Also

1. With EXPLAIN PLAN you can better understand the logic applied by the Oracle optimizer as it interprets your query. EXPLAIN PLAN is particularly useful when a query takes an unacceptably long time to complete. Beginning on page 136, Chapter 5 shows examples of EXPLAIN PLAN and how you can display its results graphically.

ORDER BY Clause

With the `ORDER BY` clause you can sort query results by data columns and expressions that may or may not appear in the `SELECT` clause. It sounds pretty simple, yet the questions generated by `ORDER BY` provide some surprisingly interesting problems. Some deal with limitations imposed on queries because the `ORDER BY` may not appear in subqueries; others deal with novel ways to sort items.

How do I ORDER BY when the sort key is too long?

A data column in one of my tables contains lengthy descriptions of training courses. In some cases the length of the column was not sufficient to store the entire description, so I created another column that contains a sequence number. Then I store the course descriptions in a series of rows with different sequence numbers. The text column is `VARCHAR2(2000)`. But when I try to `ORDER BY` the sequence number, I get an error message `ORA-01467 (sort key too long)`. How can I avoid this error?

Approach

The sort key used in this query exceeds the length supported by Oracle. This normally results from too many columns or group functions in the `SELECT` clause. To solve the problem, Oracle recommends that the number of columns or group functions be reduced. This error also can be caused by other operations that sort, including `GROUP BY` and `DISTINCT`.

In this case, it's pretty difficult to take the recommended action because the query only contains one column. Two possibilities exist, neither one particularly attractive because they both require you to recreate the table using different datatype definitions.

First, you could reduce the length of data column, which is now `VARCHAR2(2000)` — choosing instead something like `VARCHAR2(1800)`. This might require a fishing expedition to determine a column length small enough so that it does not generate the error. Second, you could make the data column `LONG` and store the entire course description in one column. Then the `ORDER BY` is not necessary.

See Also

1. `LONG` datatypes can cause problems when you want to display them in reports. Chapter 5 on page 168 discusses these problems.

How do I get an EBCDIC sort?

For one of my queries I'd prefer to sort alpha strings before numbers — as occurs when the character set is EBCDIC. Is this possible?

Approach

Character sets use encoding schemes that assign each character a numeric value. Sorting is normally based on these numeric values. In the ASCII character set, for example, the numeric codes 48 to 57 represent the symbols 0 to 9, codes 65 to 90 represent uppercase letters A to Z, and codes 97 to 122 represent lowercase letters a to z. Reports sorted in ascending order using the 7-bit U.S. ASCII character set will sort numbers, uppercase letters, and lowercase letters in that order.

An EBCDIC character set uses numeric codes that differ from the ASCII character set. EBCDIC numbers use the codes 240 to 249; uppercase letters use codes 193 to 201, 209 to 217, and 226 to 233; lowercase letters use the codes 129 to 137, 145 to 153, and 162 to 169. Reports sorted in ascending order using an EBCDIC character set will sort lowercase letters, uppercase letters, and numbers in that order — the reverse of a sort using the ASCII character set.

Oracle SQL includes a function called `CONVERT` that lets you convert between character sets. The general form of the function is `CONVERT(CHAR, destination_set, [source_set])`. The `destination_set` is the character set to convert to; and the optional `source_set` is the character set to convert from, defaulting to the database character set.

Figure 4-11 illustrates how you can use the `CONVERT` function to sort according to an EBCDIC character set. In this case, the destination character set is `WE8EBCDIC500` (IBM West European EBCDIC Code Page 500). For more information on the character sets supported by Oracle, consult the *Oracle7 Server Reference* manual.

Comments

- ▶ An `ORDER BY` using an ASCII character set sorts numbers, uppercase letters, and lowercase letters in that order.
- ▶ Use the `NOPRINT` option in the `COLUMN` command to suppress the printing of the `CONVERT` expression. You could also achieve the same effect by removing the `CONVERT` expression from the `SELECT` clause and placing it in the `ORDER BY` clause.
- ▶ Here the IBM West European EBCDIC Code Page 500 character set was used in the conversion.

```
SELECT
  sorttest_code "char"
FROM sorttest
ORDER BY 1;

char
-----
419
A419
c419

COLUMN ebcdic NOPRINT
SELECT
  sorttest_code "char",
  CONVERT(sorttest_code, 'WE8EBCDIC500') "ebcdic"
FROM sorttest
ORDER BY 2;

char
-----
c419
A419
419
```

Figure 4-11: Using the CONVERT function to perform an EBCDIC sort.

▶ An ORDER BY using an EBCDIC character set sorts lowercase letters, uppercase letters, and numbers in that order. This reverses the sort order obtained with the ASCII character set.

See Also

1. CONVERT and some of the other conversion functions like CHARTOROWID and ROWIDTOCHAR don't find much use in ad hoc queries. But occasionally they prove useful. For examples where some of these uncommon conversion functions can be helpful, check the index under *conversion functions*.

How can I sort query results based on a user's runtime preference?

Because users run one report sorted in different ways, I presently maintain several versions of a single query that differ only in the `ORDER BY` clause. Can I maintain just a single query and have users specify the sorting criteria at runtime?

Approach

If a query includes substitution variables in the `ORDER BY` clause, users can specify different sort criteria each time they run the query. Only the way they specify the criteria presents an issue. Two easy options exist. First, allow users to choose among aliases assigned to columns or expressions in the `SELECT` clause. Second, allow users to choose the positional locations of columns or expressions in the `SELECT` clause.

Figure 4-12 shows a query that allows users to sort by an identification number or a name. The `PROMPT` command provides instructions on how to use either aliases or positional notations to specify the sort order.

```

SET VERIFY OFF
SET ECHO OFF
COLUMN name FORMAT A25
PROMPT To sort by:
PROMPT ID:   enter 1 or id.
PROMPT Name: enter 2 or sortname.
PROMPT
ACCEPT sort_selection PROMPT 'Enter selection: '
SELECT
    name_id AS id,
    name_last||', '||name_first||' '||name_middle AS sortname
FROM
    name
ORDER BY &sort_selection;

To sort by:
ID:   enter 1 or id.
Name: enter 2 or sortname.

Enter selection: sortname

ID      SORTNAME
-----
@505   Carrington, Thomas J
@133   O'Leary, Vincent
@226   Sacco, Danielle
@330   Vincent, Caroline

```

Figure 4-12: Specifying runtime sort criteria.

Comments

- ▶ The query uses a series of `PROMPT` and `ACCEPT` commands to provide instructions to the user. In this case, for example, the user can sort the report by name if he enters an alias (`sortname`) or the number 2 that locates the name expression in the `SELECT` clause.
- ▶ Both the identification number and the name expression are assigned aliases by using `AS`. This allows you to use the alias in an `ORDER BY` clause.
- ▶ In this case the user entered the `sortname` alias at runtime and the report got sorted by name.

See Also

1. Chapter 2 (Figure 2-9 on page 50) shows another use of `PROMPT` and `ACCEPT` commands when interacting with query users at runtime.

How do I sort the UNION of two queries?

When I use UNION to create a compound query, the result rows are sorted automatically. How can I change this sort order?

Approach

UNION works by combining the result sets from two queries and eliminating any duplicates. In the process, the two result sets get sorted by the data columns requested in the queries — column 1 first, then column 2, and so on. Data columns in the queries UNIONED need not be the same, but they must be of the same datatype. For example, if column 1 in the first query is numeric, then column 1 in each UNIONED query also must be numeric.

Sorting UNIONED queries is fairly straightforward. Simply use a positional locator or alias in an ORDER BY clause in the last query. Figure 4-13 illustrates how this works.

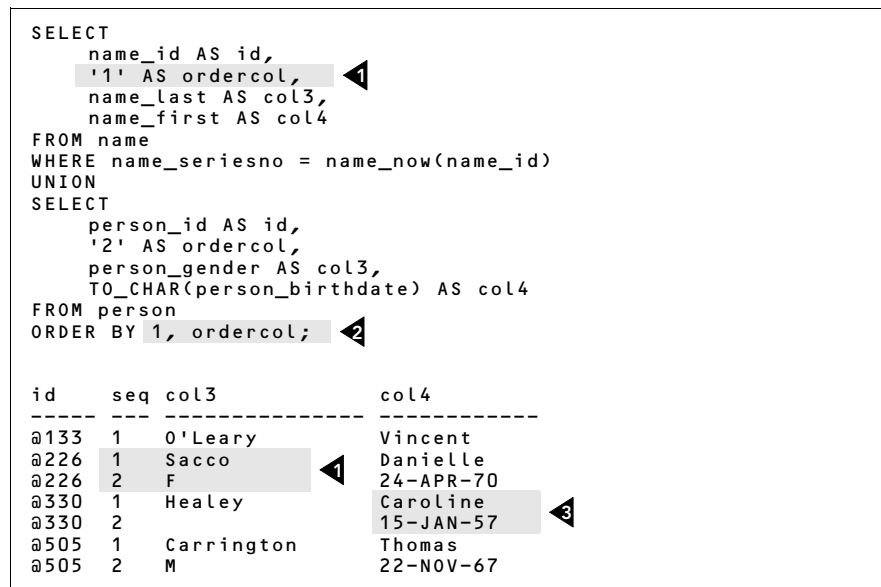


Figure 4-13: Sorting a compound UNION query.

Comments

- ▶ A literal (i.e., '1' or '2') was used to help sort rows so that name information appeared before gender and birthdate for each

person. Without using the literals, for example, the gender/birthdate row for @226 would appear before the name row — because the ‘F’ gender sorts before the ‘S’ in ‘Sacco.’

▶ Note that the report is sorted by column with position 1 in the `SELECT` clause and then sorted by the column with alias `ordercol`.

▶ Columns from two `UNIONED` queries need not contain the same data columns, but they must use the same datatype. In this case a `DATE` column (`person_birthdate`) was converted to a character expression using the `TO_CHAR` function. The datatype of the expression then matches the datatype for the last name. Both appear in the fourth column in the query.

See Also

1. The fact that `UNION` compound queries do an implicit sort can be exploited when you’d like to sort but are prevented from doing so because the `ORDER BY` clause cannot be used (e.g., in subqueries). This side-effect of the `UNION` operation gets used in Chapter 6 (Figure 6-24 on page 203) to produce a top *N* report when you want to display only those *N* rows with the highest values for a data column.

How do I sort by sequence instead of numeric order?

Sections in our training manuals use a sequential numbering scheme. For example, 1.10 refers to item 10 in section 1. I want to sort the section numbers so that they appear in sequential order (i.e., 1, 1.1, 1.2, . . . , 1.9, 1.10, . . .). But when I do an `ORDER BY`, the sorting occurs in numeric order (i.e., 1, 1.1, 1.10, 1.11, . . . , 1.19, 1.2, . . .). How can I get a sequential sort? The data column is called `section_item` and is `VARCHAR2(5)`.

Approach

The trick here is to realize that the section numbering scheme actually contains two pieces of information. The integer portion of the value indicates the section number; the decimal portion of the value indicates the item number within a section. If the table had been created with two `NUMBER` data columns, one for the section number and a second for the item number, then the sorting would be simple — just `ORDER BY section_number, item_number`.

But since we're dealing with a single `VARCHAR2` column instead of two `NUMBER` columns, the task becomes one of creating two numeric expressions that can be used to do the sorting. Figure 4-14 illustrates one possibility.

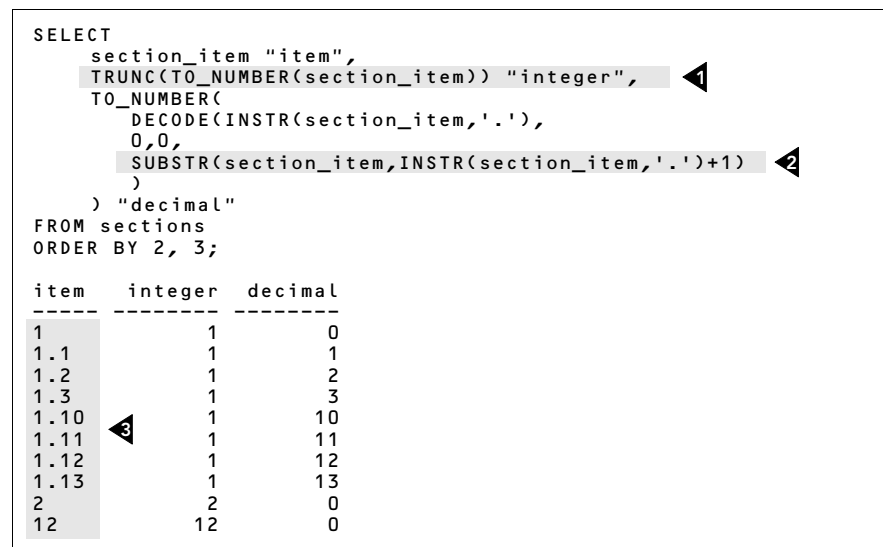


Figure 4-14: Sorting a sequential rather than numeric numbering scheme.

Comments

- ▶ The integer portion of the section numbering scheme is easily obtained by converting `section_item` to a number and then truncating that number to zero decimal places.
- ▶ The decimal portion of the numbering scheme requires a bit more work. The heart of the solution shown in Figure 4-14 relies on the `INSTR` function to search for a decimal point. If none exists, then a 0 is returned. If a decimal does exist, then the string of digits to the right of the decimal is returned. Converting the character string to a number completes the expression.
- ▶ Note that sorting on the integer expression and then the decimal expression produces the desired sorting order.

See Also

1. Whenever one data column contains two or more conceptually separate elements, potential query problems arise. The problem in this section is analogous to questions that occur when misusing dates because they contain both a date and time component. See, for example, discussions on page 116 in this chapter and on page 80 in Chapter 3.
2. For another example where the `INSTR` function gets used in a query, see Chapter 6 (Figure 6-31 on page 210).

How can I do a sorted INSERT?

I'd like to copy data from one table into a temporary table, but to do the copy after first sorting the data. The `INSERT` command does not permit an `ORDER BY` in a `SELECT` subquery. Is there any way to get around this problem?

Approach

There are work-arounds to this problem using `SQL` and more direct and logically appealing solutions in `PL/SQL`. Let's examine both. For purposes of illustration, suppose you have a `NAME` table ordered as shown in Figure 4-15, and you want to insert the rows into a new table ordered by last, first, and middle names.

```
SQL> SELECT * FROM name;
```

id	last	first	middle	seq
a505	Carrington	Thomas	J	1
a330	Vincent	Caroline		1
a133	O'Leary	Vincent		1
a330	Healey	Caroline	V	2
a226	Sacco	Danielle		1

Figure 4-15: Unsorted data from a `NAME` table.

Comments

► Note that the original data are not in alphabetical order. The task becomes one sorting the data by last, first, and middle names before doing an `INSERT` to a new table.

SQL Approach

The trick using `SQL` requires a `SELECT` subquery in the `INSERT` command that retrieves rows in the correct order without using the `ORDER BY` clause. There are a couple of approaches you can take. You could, for example, create an index on the data columns you want sorted and then force the use of that index by structuring the query appropriately and providing hints to the Oracle optimizer.

In our example, we'd want to create a compound index on `name_last`, `name_first`, and `name_middle`. Suppose the index is called `sortname_ind`. Figure 4-16 shows how you can use this index and the hint `INDEX_ASC` to retrieve and then insert data in the desired sort order.

```

INSERT INTO zname
  SELECT /*+ INDEX_ASC(name sortname_ind) */
    name_id, name_last, name_first, name_middle,
    name_seriesno
  FROM name
  WHERE name_last >= 'A';

SQL> SELECT * FROM zname;

```

id	last	first	middle	seq
@505	Carrington	Thomas	J	1
@330	Healey	Caroline	V	2
@133	O'Leary	Vincent		1
@226	Sacco	Danielle		1
@330	Vincent	Caroline		1

Figure 4-16: Using an index and optimizer hints to do a sorted INSERT.

Comments

- ▶ The INDEX_ASC hint causes the Oracle optimizer to use an index scan (with the index sortname_ind) on the NAME table. The scan occurs in ascending order of the indexed values. That is, you have implicitly ordered row retrieval.
- ▶ Include the leading edge of the index in the WHERE clause to ensure that the index gets used.
- ▶ Note that the inserted rows now appear in alphabetical order.

Using a second approach (see Figure 4-17), you can get the same effect with the implicit sort that occurs in UNION compound queries.

```

INSERT INTO zname
  (zname_last, zname_first, zname_middle, zname_id,
  zname_seriesno)
  SELECT name_last, name_first, name_middle, name_id,
  name_seriesno
  FROM name
  UNION
  SELECT 'a', 'b', 'c', 'd', 1
  FROM DUAL
  WHERE 1 = 2;

```

id	last	first	middle	seq
@505	Carrington	Thomas	J	1
@330	Healey	Caroline	V	2
@133	O'Leary	Vincent		1
@226	Sacco	Danielle		1
@330	Vincent	Caroline		1

Figure 4-17: Using the UNION set operation to do a sorted INSERT.

Comments

- ▶ In the first `SELECT` of the `UNION` query, put the data columns in the order in which you want them sorted. Here sorting will occur by last, first, and middle names.
- ▶ The second `SELECT` in the `UNION` query is a dummy. Note that the `WHERE` conditions never equate to `TRUE`. This `SELECT` only gets included so you can use `UNION` to force a sort.

PL/SQL Approach

PL/SQL solutions to the ordered insert problem require no such shenanigans as special indexes, optimizer hints, and appropriately structured subqueries. You simply declare a cursor that contains the `ORDER BY` statement and then loop through the cursor to do the inserts. Figure 4-18 illustrates one possible PL/SQL procedure to do this. When you `EXECUTE` the `insert_sorted_names` procedure, it does the same sorted insert as shown in Figure 4-16.

```
CREATE OR REPLACE PROCEDURE insert_sorted_names
IS
  CURSOR sorted_name_cursor IS
    SELECT *
    FROM name
    ORDER BY name_last, name_first, name_middle;
    sorted_name_rec sorted_name_cursor%ROWTYPE;

BEGIN
  FOR sorted_name_rec IN sorted_name_cursor
  LOOP
    INSERT INTO sname VALUES
      (sorted_name_rec.name_id,
       sorted_name_rec.name_last,
       sorted_name_rec.name_first,
       sorted_name_rec.name_middle,
       sorted_name_rec.name_seriesno);
  END LOOP;
END;
/
```

Figure 4-18: PL/SQL procedure to do a sorted `INSERT`.

Comments

- ▶ The cursor `sorted_name_cursor` contains an `ORDER BY` that performs the desired sort.
- ▶ Using a cursor `FOR` loop allows you to `INSERT` one cursor record at a time into the new table. Because the cursor records are sorted, the `INSERT` also occurs in the correct sequence.

See Also

1. For examples that use the implicit sorting of the UNION operation to work around restrictions on ORDER BY in subqueries, see Chapter 6 (Figure 6-24 on page 203) and Chapter 8 (Figure 8-4 on page 236). Both examples illustrate the use of a dummy query against the DUAL table in the UNION operation.

