Chapter 5
Special Problems

Some aspects of SQL and SQL*PLUS generate more problems for query writers than others and elicit an "oh, no, not one of those queries." Often the problems occur because the SQL element gets used infrequently or it contains subtle aspects that require experience to learn.

This chapter discusses several of the prominent problems that most query writers face at one time or another. It includes:

- Hierarchical data relationships;
- NULL values;
- Subqueries;
- Set operations such as UNION, INTERSECT, and MINUS;
- Outer joins;
- LONG datatypes; and
- DISTINCT

One theme this chapter emphasizes is alternative methods of achieving the same effect. Sometimes you get stuck writing a query with one approach; in these situations, it's nice to have alternatives that accomplish the same thing. For example, the result from an outer join also can be duplicated with set operations. Choosing between the two approaches often depends on personal preference, but sometimes circumstances in the query or data tip the scale the other way.

It probably doesn't pay to read this chapter straight through. Use it whenever you encounter a specific problem; otherwise count your blessings.

Hierarchies

In the nonrelational database world, one common way to organize data is hierarchically in a parent-child relationship. The classic example is the manager-employee relationship, where any given employee may report to a manager and also may manage other employees. For querying data that contain hierarchical links, Oracle provides a clause in the SELECT command to specify a starting point in the hierarchy and to identify how the data are linked. The result is called a *hierarchical query*.

How do I sort a hierarchical query?

I've written a query that displays employees in a hierarchy based on their job reporting relationships, much like the classic hierarchical query supplied by Oracle in the scott/tiger database. But I can't figure out a way to order the report so that within a given level the employee names are alphabetical. How can I do this?

Approach

Figure 5-1 shows the scott/tiger example with an additional data column showing the default sort key. Note that this sort key is a string of employee identification numbers that traces a reporting arrangement from the top level (i.e., the president) down to a specific employee. The default sort is essentially an ORDER BY sort_key ASC.

Comments

1 The sort key used for the default ORDER BY contains a string of employee identification numbers that shows the job hierarchy. For example, the employee named Smith (7369) is managed by 7902, who is managed by 7566, who in turn reports to the president (7839). The sort key for this row is the top-down string of these four employee identification numbers.

Note that the employee names are not arranged in alphabetical order.

By explicitly including an ORDER BY in the query, you can sort the report in some other fashion than the default. In hierarchical queries you also have a pseudocolumn called LEVEL that displays the level of the hierarchy, with the top level being 1 for the president, level 2 including the three employees who report to the president, and so on.

```
SELECT
    LPAD('  ', 2*(LEVEL-1))||ename "org_chart",
    empno,
    mgr,
    job,
    get_sort_key(empno) "sortkey"
FROM
    emp
START WITH mgr IS NULL
CONNECT BY PRIOR empno = mgr;

org_chart     empno    mgr job       sortkey
------------ ------ ------ --------- ----------------
KING           7839        PRESIDENT 7839
  JONES        7566   7839 MANAGER   78397566
    SCOTT      7788   7566 ANALYST   783975667788
      ADAMS    7876   7788 CLERK     7839756677887876
    FORD       7902   7566 ANALYST   783975667902
      SMITH    7369   7902 CLERK     7839756679027369   ◀1
  BLAKE        7698   7839 MANAGER   78397698
    ALLEN      7499   7698 SALESMAN  783976987499
    WARD       7521   7698 SALESMAN  783976987521
    MARTIN     7654   7698 SALESMAN  783976987654
    TURNER     7844   7698 SALESMAN  783976987844
    JAMES      7900   7698 CLERK     783976987900
  CLARK        7782   7839 MANAGER   78397782
    MILLER     7934   7782 CLERK     783977827934
```

Figure 5-1: Hierarchical query with the default sort.

The problem with including a simple ORDER BY is that you lose the hierarchical nature of the report. For example, if you ORDER BY the employee's name (ename), you'll just get a straight alpha sort based on last name. This question poses a different problem — to keep the hierarchical nature of the report but to do an alpha sort within a LEVEL. The three people at LEVEL 2 (Jones, Blake, and Clark), for example, should appear in the order Blake, Clark, and Jones with the appropriate job hierarchy underneath each.

Figure 5-2 shows the desired report. It also includes the sort key used to accomplish the ORDER BY. Note that instead of a string of employee identification numbers, the sort key now includes employee names as well.

Comments

▶1 The sort key now contains names as well as employee identification numbers. Using an ORDER BY on the sort key retains the hierarchy but produces an alpha sort within a level (even though these rows may not appear next to each other in the report).

```
SELECT
     LPAD('  ', 2*(LEVEL-1))||ename "org_chart",
     empno,
     mgr,
     job,
     new_sort_key(ename, empno) "sortkey"  ❷
FROM
     emp
START WITh mgr IS NULL
CONNECT BY PRIOR empno = mgr
ORDER BY 5;

org_chart      empno    mgr job       sortkey
------------ ------ ------ --------- -----------------------------------
KING            7839        PRESIDENT KING7839
  BLAKE         7698   7839 MANAGER   KING7839BLAKE7698
    ALLEN       7499   7698 SALESMAN  KING7839BLAKE7698ALLEN7499
    JAMES       7900   7698 CLERK     KING7839BLAKE7698JAMES7900
    MARTIN      7654   7698 SALESMAN  KING7839BLAKE7698MARTIN7654
    TURNER      7844   7698 SALESMAN  KING7839BLAKE7698TURNER7844
    WARD        7521   7698 SALESMAN  KING7839BLAKE7698WARD7521
  CLARK         7782   7839 MANAGER   KING7839CLARK7782
    MILLER      7934   7782 CLERK     KING7839CLARK7782MILLER7934
  JONES         7566   7839 MANAGER   KING7839JONES7566
    FORD        7902   7566 ANALYST   KING7839JONES7566FORD7902
      SMITH     7369   7902 CLERK     KING7839JONES7566FORD7902SMITH7369  ❶
    SCOTT       7788   7566 ANALYST   KING7839JONES7566SCOTT7788
      ADAMS     7876   7788 CLERK     KING7839JONES7566SCOTT7788ADAMS7876
```

Figure 5-2: Hierarchical report including an alpha sort.

❷ A PL/SQL function called new_sort_key generated the sort keys used to do the ORDER BY.

For completeness, Figure 5-3 shows the PL/SQL function new_sort_key that produces the sort key used in Figure 5-2. It contains a loop that builds the sort key by tracing from a given employee up to the president through the manager relationships. Needless to say, you must have a compelling reason for wanting to sort a hierarchical report in some way other than the default.

See Also

1. CONNECT BY . . . START WITH is actually another clause in the SELECT command, seldom seen among the other more common clauses. Yet, for returning rows in hierarchical order, the CONNECT BY . . . START WITH clause is crucial. Query writers frequently use hierarchical data when they debug their queries with EXPLAIN PLAN. Figure 5-7 on page 142 shows another example of CONNECT BY . . . START WITH used with EXPLAIN PLAN.

```
CREATE OR REPLACE FUNCTION new_sort_key
  (in_name VARCHAR2, in_empno NUMBER)
RETURN VARCHAR2
AS
  empno_to_check NUMBER(8);
  top_level VARCHAR2(1) := 'N';
  sort_key VARCHAR2(100) := in_name||in_empno;
  CURSOR mgr_cursor IS
    SELECT e1.ename, e1.empno
    FROM emp e1
    WHERE e1.empno =
    (SELECT e2.mgr FROM emp e2 WHERE e2.empno = empno_to_check);
  mgr_rec mgr_cursor%ROWTYPE;

BEGIN
  empno_to_check := in_empno;
  WHILE top_level = 'N'
  LOOP
    OPEN mgr_cursor;
    FETCH mgr_cursor INTO mgr_rec;
    IF mgr_cursor%NOTFOUND THEN
      top_level := 'Y';
    ELSE
      sort_key := mgr_rec.ename||TO_CHAR(mgr_rec.empno)||sort_key;
      empno_to_check := mgr_rec.empno;
    END IF;
    CLOSE mgr_cursor;
  END LOOP;
  RETURN sort_key;
END;
/
```

Figure 5-3: PL/SQL function new_sort_key used in Figure 5-2.

How do I select only the leaves in hierarchical data?

One of my tables contains hierarchical data. I'd like to write a query that selects only the leaves of the hierarchy. How can I do this?

Approach

Hierarchical data can be arranged in an inverted tree structure. The top level is called the *root node*. Each node below the root is a *child*. If a node has children, it is also a *parent*. A node without children is a *leaf*.

Query writers commonly encounter hierarchical data when performing an EXPLAIN PLAN to describe the query execution plan. Figure 5-4 shows a simple query and its execution plan.

```
SELECT
     name_id "id", name_last "last",
     student_regseq "seq", student_sponsor "sponsor"
FROM
     student s1, name
WHERE
     name_seriesno = name_now(name_id)
     AND s1.student_id(+) = name_id
            AND
            (
            s1.ROWID IS NULL
            OR
            s1.student_regseq =
            (SELECT MAX(s2.student_regseq)
            FROM student s2
            WHERE s2.student_id = name_id)
            )
ORDER BY 2;

    id parent position query_plan
------ ------ -------- -------------------------------------
     1      0        1   SORT ORDER BY
     2      1        1     FILTER
     3      2        1       NESTED LOOPS OUTER
     4      3        1         TABLE ACCESS FULL NAME
     5      3  1     2         TABLE ACCESS BY ROWID STUDENT
     6      5        1           INDEX RANGE SCAN PK_STUDENT
     7      2        2       SORT AGGREGATE
     8      7        1         INDEX RANGE SCAN PK_STUDENT
```

Figure 5-4: Example of hierarchical data in an EXPLAIN PLAN.

Comments

1 The list of parents does not include nodes 4, 6, or 8. These are the leaves in the query.

Figure 5-5 displays the hierarchical data appearing in the EXPLAIN PLAN. Using this format, the leaves are immediately evident.

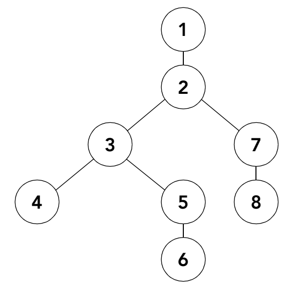

Figure 5-5: Alternative display of EXPLAIN PLAN hierarchy in Figure 5-4.

You can use several methods to find the leaves in hierarchical data. Figure 5-6 uses a correlated subquery and NOT EXISTS to return only those rows which do not serve as a parent.

```
SELECT
     id "id",
     parent_id "parent",
     position "position",
     operation||' '||options||' '||object_name "query_plan"
FROM
     plan_table p1
WHERE NOT EXISTS
     (SELECT 'x'
     FROM plan_table p2
     WHERE p2.parent_id = p1.id);

    id parent postion query_plan
------ ------ ------- -----------------------------------
     4      3       1 TABLE ACCESS FULL NAME
     6      5       1 INDEX RANGE SCAN PK_STUDENT
     8      7       1 INDEX RANGE SCAN PK_STUDENT
```

Figure 5-6: Leaves in the hierarchical data from EXPLAIN PLAN in Figure 5-4.

Comments

1 This correlated subquery retains only those rows where no other row includes the ID as a parent (i.e., the leaves).

See Also

1. Figure 5-7 on page 142 uses the MINUS operation to find leaves.

2. The PL/SQL for the name_now function used in Figure 5-4 appears in Chapter 1 (Figure 1-9 on page 13).

How can I do a bottom-up hierarchical query?

I find the typical EXPLAIN PLAN results to be somewhat confusing, because to interpret the graphic display you need to read left to right, bottom to top. For example, the query in Figure 5-4 and Figure 5-5 shows node 1 (the ORDER BY) as the root when it's actually the last step performed in executing the query. How can I do a bottom-up EXPLAIN PLAN that might more easily convey the order in which the query steps actually get performed?

Approach

A hierarchical query need not start with just a single root. Suppose we started with the leaves of the EXPLAIN PLAN data from Figure 5-4 on page 140 and follow the lineage of each leaf. Figure 5-7 shows the results.

```
SELECT
     id "id",
     parent_id "parent",
     position "pos",
     LPAD(' ',2*(LEVEL-1))||operation||' '||
            options||' '||object_name "query_plan"
FROM
     plan_table p1
CONNECT BY id = PRIOR parent_id AND id != 0
START WITH id in
     (SELECT id FROM plan_table
     MINUS
     SELECT parent_id FROM plan_table);


     id parent postion query_plan
------ ------ ------- -----------------------------------
     4      3       1 TABLE ACCESS FULL NAME
     3      2       1   NESTED LOOPS OUTER
     2      1       1     FILTER
     1      0       1       SORT ORDER BY
     6      5       1 INDEX RANGE SCAN PK_STUDENT
     5      3       2   TABLE ACCESS BY ROWID STUDENT
     3      2       1     NESTED LOOPS OUTER
     2      1       1       FILTER
     1      0       1         SORT ORDER BY
     8      7       1 INDEX RANGE SCAN PK_STUDENT
     7      2       2   SORT AGGREGATE
     2      1       1     FILTER
     1      0       1       SORT ORDER BY
```

Figure 5-7: Hierarchical query starting from leaves of EXPLAIN PLAN.

Comments

1 This subquery uses the MINUS operation to identify leaves in the hierarchical data. This structure produces the same results as

the correlated subquery shown in Figure 5-6 on page 141.
2 Note that the nodes 4, 6, and 8 appear at the top level in this query.

Using the bottom-up approach produces a straight lineage from child to parent to grandparent and so on, without the branching present when tracing down from the common ancestor. To understand the query using the bottom-up approach, you really need to graphically display the results. Figure 5-8 shows the three lineages combined so as to eliminate the redundancies present in Figure 5-7. It does allow the query to be interpreted by reading from left to right, top to bottom, but you'll have to be the judge whether it adds any value over the traditional EXPLAIN PLAN results.

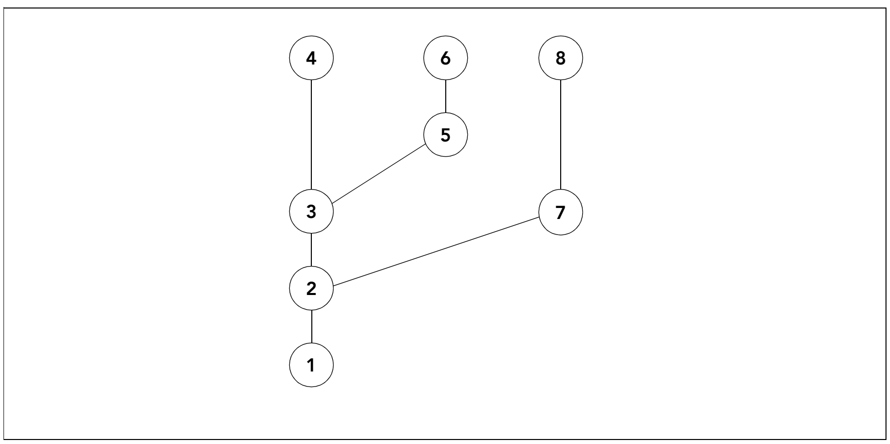

Figure 5-8: Alternate display of bottom-up hierarchical query in Figure 5-7.

See Also

1. The CONNECT BY . . . START WITH clause used in Figure 5-7 returns rows in hierarchical order. Another example of its use appears in Figure 5-1 on page 137, when displaying employee reporting relationships in a hierarchical fashion.

NULL Values

NULL values seem simple enough — a data value that is NULL doesn't exist; we just don't have any information about it. But rest assured, you'll pay lots of attention in your queries to the special problems that NULL values present. Whether they appear naturally in the data or get generated temporarily with outer joins, NULL values often demand their due.

How do I use != with NULL values?

I've written a query that's losing population items. I traced the problem to one condition in the WHERE clause that specifies that the state is not equal to NY. What could be simpler than this?

Approach

Using the != operator requires some care whenever the data column in question can contain NULL values. For example, WHERE state != 'NY' excludes all rows where the state is NY, but it also excludes all rows where the state is NULL. Since NULL is not a value, it cannot equal or not equal anything else.

If you want rows containing NULL values to remain in the report, you must modify the != condition slightly. Figure 5-9 shows one way to do this.

```
SELECT
     name_id "id",
     address_loc "state"
FROM address, name
WHERE
     name_seriesno = name_now(name_id)
     AND address_id = name_id
           AND address_type = 'PR'
           AND (address_loc != 'NY' OR address_loc IS NULL)
ORDER BY 1;
                                                          1

id    state
----- -----
@133  MA
@330  CO       1
@409
@925
```

Figure 5-9: Using != when NULL values exist.

Comments

1 If you want to exclude some rows based on the address state but also want to include rows containing NULL values for the state (as might occur with foreign addresses), then use an OR structure with IS NULL to explicitly include NULL values.

Another common but related error occurs when you try to use NULL with IN. For example, WHERE state IN ('NY', NULL) only includes states with the value NY. NULL values are again excluded, for the same reason that NULL values cannot equal or not equal anything.

See Also

1. You also could use the NVL or DECODE functions to solve this problem. For example, WHERE NVL(address_loc, 'MA') != 'NY' would work. Several examples of NVL and DECODE appear in this book. Check the index under NVL *function* or DECODE *function* for specific references.

2. The PL/SQL for the name_now function used in Figure 5-9 appears in Chapter 1 (Figure 1-9 on page 13).

How do I count NULL values?

Figure 5-10 shows a query I wrote to count addresses by state. I know that some of the rows have NULL values for the state, and I used an NVL function to display 'null' when this occurs. But the count indicates zero rows with NULL values. I know this is wrong. What's happening?

```
SELECT
    NVL(address_loc,'null') "state",
    COUNT(address_loc) "count"        ◀1
FROM address, name
WHERE
    name_seriesno = name_now(name_id)
    AND address_id = name_id
GROUP BY
    address_loc
ORDER BY 1;


state      count
----- ----------
CO             2
IL             1
MA             1
NY             2
null           0  ◀1
```

Figure 5-10: COUNT that ignores NULL values.

Comments

▶1 Counting the address_loc data column produced a count of zero even though some NULL values exist.

Approach

All group functions except one ignore NULL values. The exception is one flavor of the COUNT function. When used as COUNT(*), the count includes all rows meeting the query conditions regardless of whether they contain NULL values in the data column of particular interest. On the other hand, when used as COUNT(column_name) as done in Figure 5-10, the count ignores NULL values, hence the zero that appears in the report.

The solution to the problem is straightforward. You can either replace COUNT(address_loc) with COUNT(*) or with COUNT(expression) where the expression explicitly assigns a value to NULL. For example, COUNT(NVL(address_loc, 'null')) or COUNT(DECODE(address_loc, NULL, 'null')) would work.

See Also

1. For discussion of the same confusion that sometimes occurs between COUNT(*) and COUNT, see Chapter 3 on page 99.

2. Figure 5-10 uses the name_now function. The PL/SQL for this function appears in Chapter 1 (Figure 1-9 on page 13).

Subqueries

Subqueries abound in ad hoc queries. They are wonderful inventions with great flexibility. One form in particular appears frequently in queries. The correlated subquery acting as predicate to an EXISTS or NOT EXISTS finds considerable use in defining report populations when you need to check for the presence or absence of a characteristic but don't actually need to see the value in the report. This section includes a question about correlated subqueries.

Often the function of a subquery can be replaced conveniently with a PL/SQL function. This makes your query writing much cleaner; instead of a messy looking subquery, you simply employ a PL/SQL function conveniently named so as to convey intent. This section also discusses these alternatives to subqueries.

How do I get a maximum value for each population item?

To display the most recent student-related information for people attending our training classes, I need to retrieve only the last row pertaining to each student. In this case a sequential number is used as each new row of data is input for a student. What's the easiest way to retrieve the most recent information for each student?

Approach

The SQL structure normally used to solve this type of problem is the correlated subquery. For each item in the report population, the subquery searches other rows for that item and returns the row or rows meeting the subquery conditions. In this case, for each student in the report population, the subquery identifies the single row containing the maximum sequential number for that student (i.e., the row with the most recent data).

Figure 5-11 shows a simple example of two students, one registered at two separate times for training classes and the other at three separate times. Note that the column student_regseq identifies the most recent registration. The column student_sponsor identifies who paid for the training.

Comments

1 Selecting the maximum student_regseq for each student returns the most recent training registration information.

```
id      seq  sponsor       fee  regdate
-----  ----  ----------  ------  ---------
@226      1  Mixed        1200  23-APR-97
@226      2  Self          200  11-JUL-97
@226      3  Self          400  02-DEC-97   ◀1
@505      1  Self          500  19-JUN-97
@505      2  Employer     1500  04-NOV-97
```

Figure 5-11: Sample training registration data.

Constructing a subquery to return the row with the maximum student_regseq for each student is straightforward. Figure 5-12 illustrates such a subquery.

```
SELECT
    s1.student_id "id",
    s1.student_regseq "seq",
    s1.student_sponsor "sponsor",
    s1.student_total_fee "fee"
FROM
    student s1, name
WHERE
    name_seriesno = name_now(name_id)
          AND name_id IN ('@226','@505')
    AND s1.student_id = name_id
          AND s1.student_regseq =
          (SELECT max(s2.student_regseq)
          FROM student s2                      ◀1
          WHERE s2.student_id = name_id)
ORDER BY 1;

id      seq  sponsor       fee
-----  ----  ----------  ------
@226      3  Self          400
@505      2  Employer     1500
```

Figure 5-12: Correlated subquery that retrieves most recent training data.

Comments

1▶ The subquery correlates with the main query through the student_id. It uses the group function MAX to retrieve the maximum student_regseq (i.e., the most recent training registration).

The interesting aspect of this question appears only when you begin adding WHERE conditions from the correlating table. Then you must be conscious of the symmetry between the main query and the subquery. For example, what would happen if we added a condition to the main query that student_sponsor = 'Self'? Figure 5-13 provides the results.

```
SELECT
     s1.student_id "id",
     s1.student_regseq "seq",
     s1.student_sponsor "sponsor",
     s1.student_total_fee "fee"
FROM
     student s1,
     name
WHERE
     name_seriesno = name_now(name_id)
            AND name_id IN ('@226','@505')
     AND s1.student_id = name_id
            AND s1.student_sponsor = 'Self'    ◀1
            AND s1.student_regseq =
            (SELECT MAX(s2.student_regseq)
            FROM student s2
            WHERE s2.student_id = name_id)
ORDER BY 1;


id      seq  sponsor       fee
-----  ----  ----------  ------
@226      3  Self           400  ◀1
```

Figure 5-13: Unbalanced subquery and main query (version 1).

Comments

▶1 Note that the main query contains the condition student_sponsor = 'Self' but that the subquery does not. The results of this query differ from those in Figure 5-12 (no row appears for @505). Think for a moment about what the query requests. The translation is: Consider only those rows where student_sponsor = 'Self' and from these display only those where the student_regseq is the maximum among all registrations for that student. In slightly different words: Retrieve the most recent training data for a student but display them only if the student paid for the course herself.

Consider the situation with @505 (see Figure 5-11 on page 149). This person does have one row meeting the main query condition that student_sponsor = 'Self.' But the student_regseq for this row is 1, which is not the most recent registration for the student. The subquery condition is not met, and no row for this student displays.

Whenever you add a WHERE condition from the correlating table, several situations can occur. These are:

- In a balanced structure, the main query and the subquery both contain the condition;
- In one unbalanced structure, the main query contains the condition and the subquery does not; or
- In a second unbalanced structure, the subquery contains the condition but the main query does not.

Each of the three structures produces a query whose meaning differs slightly from the others. The first unbalanced structure we've already discussed (see Figure 5-13). Here's what the other two structures yield:

- The balanced structure returns the most recent training data where the student paid for the courses. This report includes two rows: student_regseq = 3 for @226 and student_regseq = 1 for @505.
- The unbalanced structure with the condition in the subquery returns the same results as the balanced structure but does so in a less efficient manner because the main query does not filter the rows for consideration.

The take-home message here is that WHERE conditions in correlating tables should pose a caution flag for query writers. Often, balancing the main query and the subquery provides the desired result, but this may not always be the case. There may be situations where you deliberately choose to unbalance the main query and the subquery.

See Also

1. Correlated subqueries are wonderful when defining report populations. Used with EXISTS or NOT EXISTS they qualify a population much like an adjective qualifies a noun, identifying the presence or absence of some characteristic (e.g., whether any overdue balances exist). As such, they're good candidates for SQL segments that you can later reuse. See Chapter 7 (Figure 7-4 on page 223) for an example of a correlated subquery used to define a population.

2. The name_now function is used in Figure 5-12. Chapter 1 (Figure 1-9 on page 13) shows the PL/SQL for this function.

Are there alternatives to correlated subqueries?

How could I write the query in Figure 5-12 on page 149 without using a correlated subquery?

Approach

As we'll see shortly with outer joins, it's always helpful to have a variety of ways to achieve the same effect. When one approach proves difficult to implement or degrades system performance, you have a fallback.

Often you can easily replace a correlated subquery with a PL/SQL function. This approach has the added feature that it simplifies the SQL query and makes it easier to read and maintain. Figure 5-14 shows how this works.

```
SELECT
    student_id "id",
    student_regseq "seq",
    student_sponsor "sponsor",
    student_total_fee "fee"
FROM
    student,
    name
WHERE
    name_seriesno = name_now(name_id)
          AND name_id IN ('@226','@505')
    AND student_id = name_id
          AND student_regseq = max_regseq(name_id)   1
ORDER BY 1;


id      seq  sponsor        fee
-----  ----  ----------  ------
@226      3  Self           400
@505      2  Employer      1500
```

Figure 5-14: PL/SQL function that replaces a correlated subquery.

Comments

1 The PL/SQL function max_regseq returns the maximum student_regseq for each person in the report population. This single line replaces the correlated subquery shown in Figure 5-12 on page 149.

Figure 5-15 shows the PL/SQL function max_regseq.

See Also

1. Chapter 9 (Table 9-5 on page 261) shows the performance

```
CREATE OR REPLACE FUNCTION max_regseq
   (id VARCHAR2)
RETURN NUMBER
AS
   CURSOR student_cursor (p_id VARCHAR2) IS
     SELECT MAX(student_regseq)
     FROM student
     WHERE student_id = p_id;
   max_regseq NUMBER(2);


BEGIN
   OPEN student_cursor(id);
   FETCH student_cursor INTO max_regseq;

   IF student_cursor%NOTFOUND THEN
      max_regseq := NULL;
   END IF;

   CLOSE student_cursor;
   RETURN max_regseq;
END;
/
```

Figure 5-15: PL/SQL function max_regseq used in Figure 5-14.

implications of using a PL/SQL function or a correlated subquery. Again, you'll be faced with a trade-off between query simplicity and performance.

2. The PL/SQL for the name_now function used in Figure 5-14 appears in Chapter 1 (Figure 1-9 on page 13).

Set Operations

Oracle includes four set operations — UNION, UNION ALL, INTERSECT, and MINUS. Each operation allows you to combine two or more queries into what's called a *compound query*, but with different effects:

- UNION selects all rows in either query and eliminates duplicates;
- UNION ALL selects all rows in either query but does not eliminate duplicates;
- INTERSECT selects all distinct rows in both queries; and
- MINUS selects all rows in the first query that are not in the second query.

The set operations are really wonderful tools to have available in your query writer's toolkit. Often you can duplicate the function of set operations with joins and subqueries, but sometimes the set operations are indispensable as alternatives. And some effects can only be achieved with set operations, as this section discusses.

Is there a join structure equivalent to UNION?

If I have two tables, call them A and B, how can I write a join between the tables so that the result set is equivalent to performing a UNION?

Approach

At present you can't. This requires something called a *full outer join* in which the outer join symbol appears on both sides of the join criteria (i.e., A(+) = B(+)). The full outer join was introduced in the ANSI/ISO SQL-92 standards but has not yet been implemented in most versions of SQL. If you attempt a full outer join in Oracle SQL, you presently receive error ORA-01468 (a predicate may reference only one outer-joined table).

There are times when set operations equivalent to outer joins are useful. Several examples appear in the next section on outer joins. Let's use a simple example to explore these equivalents.

Suppose we have two tables A and B. Table A contains three identification numbers — 101, 102, and 103. Table B contains two identification numbers — 103 and 104. Figure 5-16 shows several joins, their equivalent set operation, pseudo-SQL that you can use to implement the set operation in a query, and the query results.

```
Join                  Set equivalent     Pseudo-SQL          Results
=========================================================================
A.id = B.id           A INTERSECT B      q1:A = B            103

A.id(+) = B.id        (A INTERSECT B)    q1:A = B            103,104
                      UNION              UNION
                      (B MINUS A)        q2:B NOT EXISTS A

A.id = B.id(+)        (A INTERSECT B)    q1:A = B            101,102,103
                      UNION              UNION
                      (A MINUS B)        q2:A NOT EXISTS B

A.id(+) = B.id(+)     A UNION B          q1:A                101,102,103,104
(not available)                          UNION
                                         q2:B
```

Figure 5-16: Table joins and their set operation equivalents.

Comments

1 The outer join A.id(+) = B.id can be implemented with set operations by constructing two queries and UNIONing them. The first query contains a straight join between the two tables (i.e., A.id = B.id). The second query includes a main query against table B and a correlated subquery as a NOT EXISTS predicate (i.e., NOT EXISTS (SELECT 'x' FROM A WHERE A.id = B.id)).

See Also

1. Set operations can be quite useful when debugging queries, particularly when trying to find rows that unexpectedly fail join criteria. In such situations, you often need to identify the rows and examine the data to determine the exact problem. See Chapter 7 (Figure 7-10 on page 229) for an example of the MINUS operation used in this fashion.

Outer Joins

Outer joins confuse many query writers. The confusion starts with the (+) sign — when you're faced with the issue of where to put it. Let's see; does it go to the left or right of the equal sign? From this start, things only get worse when you throw in rules to follow, NULL values as complications, and limitations on the use of outer joins. No wonder so many frequently asked questions concern outer joins.

How do I prevent population loss when using outer joins?

I'm working with two tables, one a master and the second a detail. Some people who appear in the master table have no data in the detail table. I want the report to include everyone I select from the master table regardless of whether they have detail information. I know I should use an outer join, but it isn't working properly.

Figure 5-17 illustrates the problem for four people from my report population. For each of the four, I want to display all training registrations where the student paid for the class himself. The figure shows that I'm only getting one of the four people in the report even though I'm using an outer join. What's going on here?

```
id     last              seq  sponsor      fee
-----  ----------------  ---  ----------  -----
@133   O'Leary             1  Self         325   1
@330   Healey              1               740
@409   Alvarez
@925   Carrington          1  Employer     610

SELECT
    name_id "id",
    name_last "last",
    student_regseq "seq",
    student_sponsor "sponsor",
    student_total_fee "fee"
FROM
    student,
    name
WHERE name_seriesno = name_now(name_id)
          AND name_id IN ('@133','@925','@409','@330')
    AND student_id(+) = name_id   2
    AND student_sponsor = 'Self'
ORDER BY 1;

id     last              seq  sponsor      fee
-----  ----------------  ---  ----------  -----
@133   O'Leary             1  Self         325   2
```

Figure 5-17: Outer join that doesn't produce desired results.

Comments

1 Given the query criteria, only one person (@133) of the four has well-behaved data. This person has training registration data and paid for classes himself. Note that each of the other three people presents a problem for the query. One person (@409) has no training information at all; another (@925) has training data but the employer paid for the course; and the third (@330) has training data but the student_sponsor column is NULL.
2 The query includes an outer join between the ***STUDENT*** and ***NAME*** tables, but only the well-behaved data appear in the report.

Approach

For outer joins to work properly, you must follow two simple rules:

- Use the outer join on *all* conditions that join two tables. In this example the ***STUDENT*** and ***NAME*** tables are joined through only a single condition, and the outer join appears correctly in this condition. But if two or more join conditions occurred, then the outer join must appear in each of the conditions.

- Because outer joins produce NULL rows if necessary, any WHERE condition that compares a NULL data value to a literal also must include an outer join on the data column in question. You can ignore this rule if you explicitly deal with NULL values using a function such as NVL.

The query in Figure 5-17 falters by using student_sponsor = 'Self'. Rows created through the outer join student_id(+) = name_id are entirely NULL. Consequently, none of them contain the value 'Self' in student_sponsor. The remedy is straightforward; simply use an outer join so that student_sponsor(+) = 'Self'. You also can use an alternative structure to achieve the same effect; for example, NVL(student_sponsor, 'Self') = 'Self' would work.

If you want the NULL rows to display in the report in special ways, you may also need to use a function like NVL in the SELECT clause. The query in Figure 5-11 lists several data columns. But suppose instead you wanted to sum on a NUMBER column and wanted the NULL values created with outer joins to be considered zeroes. Then NVL(numeric_column, 0) would appear in the SUM.

The second rule listed above also applies when the data column containing NULL values appears in an expression. Figure 5-18 illustrates the use of an outer join in an expression that's compared with a literal.

```
SELECT
    name_id "id",
    name_last "last",
    student_regseq "seq",
    student_sponsor "sponsor",
    student_total_fee "fee",
    student_regdate "regdate"
FROM
    student,
    name
WHERE name_seriesno = name_now(name_id)
          AND name_id in ('@133','@925','@409','@330')
    AND student_id(+) = name_id
          AND TO_CHAR(student_regdate(+),'YYYY') = '1997'
ORDER BY 1;                                  1

id      last              seq  sponsor      fee   regdate
-----   ---------------   ---  ----------  -----  ---------
@133    O'Leary             1  Self          325  17-SEP-97
@330    Healey              1                740  03-FEB-97
@409    Alvarez
@925    Carrington          1  Employer      610  10-MAY-97
```

Figure 5-18: Outer join on data column in an expression.

Comments

1 The outer join in student_id(+) = name_id creates a NULL row for @409 who has no student data. To include this person in the report, you'll need an outer join on the student_regdate column that appears in the expression specifying the year is 1997. You also could use an alternative structure that dealt explicitly with NULL values (e.g., using the NVL or DECODE functions).

See Also

1. Query problems from outer joins and those from NULL values frequently appear together like a matched set. For questions on NULL values, see the discussion that begins on page 144.

2. Figure 5-17 and Figure 5-18 used the name_now function. The PL/SQL for this function appears in Chapter 1 (Figure 1-9 on page 13).

How can I do an outer join to a subquery?

With one of my queries I'm getting error message ORA-01799 (a column may not be outer-joined to a subquery). The situation is simple. I've got a population of people and all of them should appear in the report. If they have training registration data, I want to see their most recent information. Clearly, an outer join is required for people without training data. It's also clear that I need a subquery to return the most recent training data for people who have those data. Figure 5-19 shows the query I thought would work. Is there a solution to this problem?

```
id     last             seq  sponsor     fee   regdate
-----  ---------------  ---  ----------  ----- ---------
@133   O'Leary            1  Self          325  17-SEP-97
@226   Sacco              1  Mixed        1200  23-APR-97
@226   Sacco              2  Self          200  11-JUL-97
@226   Sacco              3  Self          400  02-DEC-97
@330   Healey             1                740  03-FEB-97
@409   Alvarez
@505   Carrington         1  Self          500  19-JUN-97
@505   Carrington         2  Employer     1500  04-NOV-97
@925   Appiah             1  Employer      610  10-MAY-96
```

```
SELECT
    name_id "id",
    name_last "last",
    s1.student_regseq "seq",
    s1.student_sponsor "sponsor",
    s1.student_total_fee "fee"
FROM
    student s1,
    name
WHERE name_seriesno = name_now(name_id)
    AND s1.student_id(+) = name_id
          AND s2.student_regseq(+) =
          (SELECT max(s2.student_regseq)
          FROM student s2
          WHERE s2.student_id = name_id)
ORDER BY 1;
```

Figure 5-19: Outer join to a subquery produces an error.

Comments

1. The query should return the highlighted rows.
2. The outer join student_id(+) = name_id retains people in the query even if they don't have student data. The second outer join, between student_regseq and a subquery, produces an error.

Approach 1

You'll have to remove the second outer join between student_regseq and the subquery yet still find a way to include the rows generated as a result of the first outer join. One way to accomplish this uses the fact that outer joins produce NULL rows. Figure 5-20 illustrates this approach.

```
SELECT
    name_id "id",
    name_last "last",
    s1.student_regseq "seq",
    s1.student_sponsor "sponsor",
    s1.student_total_fee "fee"
FROM
    student s1,
    name
WHERE name_seriesno = name_now(name_id)
    AND s1.student_id(+) = name_id
          and
          (
          s1.rowid IS NULL
          OR
          s1.student_regseq =
                (SELECT MAX(s2.student_regseq)
                FROM student s2
                WHERE s2.student_id = name_id)
          )
ORDER BY 1;


id    last            seq sponsor      fee
----- --------------- --- ---------- -----
@133  O'Leary           1 Self         325
@226  Sacco             3 Self         400
@330  Healey            1              740
@409  Alvarez
@505  Carrington        2 Employer    1500
@925  Appiah            1 Employer     610
```

Figure 5-20: Subquery structure that retains rows created with outer joins.

Comments

1 The only rows that meet the criteria 'WHERE s1.rowid IS NULL' are those created by the outer join. This query returns all rows created with the outer join or uses the subquery to return the most recent training registration data.

Approach 2

The UNION set operator provides another way to achieve the same effect as an outer join. One query includes rows from the report population that do not exist in the *STUDENT* table. A second query includes rows that do exist in the *STUDENT* table. In this second query, joins and subqueries can be used without recourse to outer joins. Doing a UNION of the two queries is equivalent to doing an outer join (see Figure 5-16 on page 155). Figure 5-21 illustrates how this works.

```
SELECT
     name_id "id",
     name_last "last",
     TO_NUMBER('') "seq",
     '' "sponsor",
     TO_NUMBER('') "fee"
FROM
     name
WHERE name_seriesno = name_now(name_id)
     AND NOT EXISTS
     (SELECT 'x'
     FROM student
     WHERE student_id = name_id)
UNION
SELECT
     name_id "id",
     name_last "last",
     s1.student_regseq "seq",
     s1.student_sponsor "sponsor",
     s1.student_total_fee "fee"
FROM
     student s1,
     name
WHERE name_seriesno = name_now(name_id)
     AND s1.student_id = name_id
            AND s1.student_regseq =
            (SELECT MAX(s2.student_regseq)
            FROM student s2
            WHERE s2.student_id = name_id)
ORDER BY 1;


id    last            seq sponsor     fee
----- --------------- --- ---------- -----
@133  O'Leary           1 Self        325
@226  Sacco             3 Self        400
@330  Healey            1 Mixed       740
@409  Alvarez
@505  Carrington        2 Employer   1500
@925  Appiah            1 Employer    610
```

Figure 5-21: UNION set operation equivalent to an outer join.

Comments

1 The first query selects everyone from the report population who does not have ***STUDENT*** data.

2 The UNION operation used with queries structured in this manner produces the equivalent of an outer join.

3 The second query selects everyone from the report population who does have ***STUDENT*** data. The subquery returns the most recent information.

Approach 3

You also can use a PL/SQL function that removes the messiness existing in the two previous approaches. Figure 5-22 shows a query that uses a PL/SQL function called max_regseq.

```
SELECT
    name_id "id",
    name_last "last",
    s1.student_regseq "seq",
    s1.student_sponsor "sponsor",
    s1.student_total_fee "fee",
    s1.student_regdate "regdate"
FROM
    student s1,
    name
WHERE name_seriesno = name_now(name_id)
    AND s1.student_id(+) = name_id
         AND s1.student_regseq(+) = max_regseq(name_id)
ORDER BY 1, 3;                                    1

id     last             seq  sponsor       fee  regdate
-----  ---------------  ---  ----------  -----  ---------
@133   O'Leary            1  Self          325  17-SEP-97
@226   Sacco              3  Self          400  02-DEC-97
@330   Healey             1  Mixed         740  03-FEB-97
@409   Alvarez
@505   Carrington         2  Employer     1500  04-NOV-97
@925   Appiah             1  Employer      610  10-MAY-96
```

Figure 5-22: PL/SQL function equivalent to outer join to a subquery.

Comments

1 This one line replaces the complex UNION or OR structures required in the previous SQL solutions. For each name_id, the function max_regseq returns the maximum student_regseq. If none exists, it returns NULL. The outer join s1.student_regseq(+) retains NULL rows for display. This simple structure is equivalent to an outer join to a subquery.

The PL/SQL that created the function max_regseq was shown earlier in Figure 5-15 on page 153.

See Also

1. Chapter 9 (Table 9-6 on page 263) shows results from performance comparisons of the three approaches to this outer join problem.

2. Several figures in this section use the name_now function. The PL/SQL for this function appears in Chapter 1 (Figure 1-9 on page 13).

How can I do an outer join with an IN or OR?

One of my queries that worked with Version 6 of Oracle now returns the error message ORA-01719 (outer join operator not allowed in operand of OR or IN). Figure 5-23 shows the query. Is there another way of writing the query that's equivalent to performing an outer join with IN?

```
id     last              seq  sponsor       fee
-----  ----------------  ---  ----------  -----
@133   O'Leary             1  Self          325
@330   Healey              1                740
@409   Alvarez
@925   Appiah              1  Employer      610


SELECT
    name_id "id",
    name_last "last",
    s1.student_regseq "seq",
    s1.student_sponsor "sponsor",
    s1.student_total_fee "fee"
FROM
    student s1,
    name
WHERE name_seriesno = name_now(name_id)
          AND name_id IN ('@133','@330','@409','@925')
    AND s1.student_id(+) = name_id
          AND s1.student_sponsor(+) IN ('Self','Employer')
ORDER BY 1;
```

Figure 5-23: Query using an outer join with IN produces an error.

Comments

1 An outer join with IN or OR now produces error ORA-01719.

Approach 1

Figure 5-23 includes data for the report population. Note there are three different categories that the data fall into:

- No ***STUDENT*** data exist (@409);
- ***STUDENT*** data meeting the criteria exist (@133 and @925); and
- ***STUDENT*** data exist, but they do not meet the query criteria (@330).

The query must accommodate all three types of data. Figure 5-24 shows one alternative that uses an outer join, a fairly complex OR condition, and DECODEs in the SELECT. It's not a pretty sight, but it does produce the desired results.

```
SELECT
     name_id "id",
     name_last "last",
     DECODE(s1.student_sponsor,
            'Self', s1.student_regseq,
            'Employer', s1.student_regseq, NULL) "seq",
     DECODE(s1.student_sponsor,
            'Self', s1.student_sponsor,
            'Employer', s1.student_sponsor, NULL) "sponsor",
     DECODE(s1.student_sponsor,
            'Self', s1.student_total_fee,
            'Employer', s1.student_total_fee, NULL) "fee"
FROM
     student s1,
     name
WHERE name_seriesno = name_now(name_id)
     AND s1.student_id(+) = name_id
            AND
            (
            s1.ROWID IS NULL
            OR
            s1.student_sponsor IN ('Self','Employer')
                   AND s1.student_regseq =
                   (SELECT MAX(s2.student_regseq)
                   FROM student s2
                   WHERE s2.student_id = name_id
                   AND s2.student_sponsor IN ('Self','Employer'))
            OR
            s1.ROWID =
            (SELECT MAX(s2.ROWID)
            FROM student s2
            WHERE s2.student_id = name_id
                   AND NOT EXISTS
                   (SELECT 'x'
                   FROM student s3
                   WHERE s3.student_id = name_id
                   AND s3.student_sponsor IN
                          ('Self','Employer')))
            )
ORDER BY 1;

id     last             seq  sponsor      fee
-----  ---------------  ---  ----------  -----
@133   O'Leary            1  Self          325
@330   Healey
@409   Alvarez
@925   Appiah             1  Employer      610
```

Figure 5-24: SQL query displaying results expected from an outer join and IN.

Comments

1 The DECODEs on the ***STUDENT*** data columns only display data if student_sponsor is one of the two values (i.e., 'Self' or 'Employer') specified in the WHERE conditions. Why this is necessary is explained below.

2 This condition retains people like @409 who have no

STUDENT data.

3 This condition returns the well-behaved data situations (i.e., @133 and @925) where student_sponsor is either 'Self' or 'Employer'.

4 This condition returns one row for each person who has ***STUDENT*** data but no rows where student_sponsor is 'Self' or 'Employer' (i.e., @330). The DECODEs in the SELECT clause display NULL values for these rows when they appear in the report.

Approach 2

By contrast to the solution shown in Figure 5-24, constructing the UNION equivalent to an outer join provides a more appealing approach to the problem. Figure 5-25 illustrates this approach.

```
SELECT
    name_id "id", name_last "last", s1.student_regseq "seq",
    s1.student_sponsor "sponsor", s1.student_total_fee "fee"
FROM
    student s1, name
WHERE
    name_seriesno = name_now(name_id)
    AND s1.student_id = name_id
        AND s1.student_sponsor IN ('Self','Employer')          ◄1
        AND s1.student_regseq =
        (SELECT MAX(s2.student_regseq)
        FROM student s2
        where s2.student_id = name_id
        and s2.student_sponsor in ('Self','Employer'))
UNION
SELECT
    name_id "id", name_last "last", TO_NUMBER('') "seq",
    '' "sponsor", TO_NUMBER('') "fee"
FROM
    name
WHERE
    name_seriesno = name_now(name_id)
    AND NOT EXISTS
    (SELECT 'x'
    FROM student                                          ◄2
    WHERE student_id = name_id
    AND student_sponsor IN ('Self','Employer'));

id     last              seq  sponsor      fee
-----  ---------------   ---  ----------  -----
@133   O'Leary             1  Self         325
@330   Healey
@409   Alvarez
@925   Appiah              1  Employer     610
```

Figure 5-25: SQL query using UNION equivalent to an outer join with IN.

Comments

1 The first query selects all rows with the well-behaved data where student data exist with student_sponsor of 'Self' or 'Employer' (i.e., @133 and @925).

2 The second query creates rows for members of the report population who do not have a student_sponsor of 'Self' or 'Employer' (i.e., @330 and @409). When UNIONed, the two queries produce the desired report.

Approach 3

By contrast to both the previous solutions, a PL/SQL function provides the most appealing solution to the problem. Figure 5-26 illustrates this approach.

```
SELECT
    name_id "id",
    name_last "last",
    s1.student_regseq "seq",
    s1.student_sponsor "sponsor",
    s1.student_total_fee "fee"
FROM
    student s1,
    name
WHERE name_seriesno = name_now(name_id)
    AND s1.student_id(+) = name_id
         and s1.ROWID(+) =
              sponsor_exist(name_id, 'Self', 'Employer')
ORDER BY 1;                                  1


id     last              seq  sponsor      fee
-----  ----------------  ---  ----------  -----
@133   O'Leary             1  Self          325
@330   Healey
@409   Alvarez
@925   Appiah              1  Employer      610
```

Figure 5-26: PL/SQL function equivalent to an outer join used with IN.

Comments

1 The PL/SQL function sponsor_exist returns ROWIDs for each person where student data exist with student_sponsor of 'Self' or 'Employer'. Whenever no such data exist, the function returns a NULL. The outer join s1.ROWID(+) retains these rows for display in the report.

Figure 5-27 shows the PL/SQL function itself.

```
CREATE OR REPLACE FUNCTION sponsor_exist
   (id VARCHAR2, sponsor1 VARCHAR2, sponsor2 VARCHAR2)
RETURN ROWID
AS
   student_rowid ROWID;
   CURSOR main_cursor
   (p_id VARCHAR2, p_sponsor1 VARCHAR2, p_sponsor2 VARCHAR2) IS
     SELECT ROWID
     FROM student
     WHERE student_id = p_id
     AND student_sponsor IN (p_sponsor1, p_sponsor2)
     ORDER BY student_regseq desc;

BEGIN
   OPEN main_cursor(id, sponsor1, sponsor2);
   FETCH main_cursor INTO student_rowid;

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

   CLOSE main_cursor;
   RETURN student_rowid;
END;
/
```

Figure 5-27: PL/SQL function sponsor_exist used in Figure 5-26.

See Also

1. The three approaches to this problem exact different performance costs. Chapter 9 (Table 9-7 on page 267) shows results from performance comparisons among the three approaches. Once again, your choice of an approach will require compromises.

2. Chapter 1 (Figure 1-9 on page 13) shows the PL/SQL for the name_now function used in several figures in this section.

LONG Datatypes

Query writers frequently look for information that they or others can use to make decisions and take action. This gives the query writer a printed-page orientation to the database, because the printed page is usually the format in which database information gets presented. LONG datatypes, which in Oracle7 can contain up to 2 gigabytes of information, don't fit well with a printed report that might only be 80 or 130 characters wide. Thus LONG datatypes pose special query problems.

How do I format a LONG column for display?

I created a table to store the SQL queries I write. The data column containing the SQL has a LONG datatype. When I use the COLUMN command to format the display width of this column in reports, it doesn't seem to have any effect. For example, COLUMN query_sql FORMAT A100 only displays the first 80 characters and then stops. What's happening here?

Approach

Three factors interact to affect the display of LONG data columns in reports. Two of these factors are the SQL*PLUS system variables LONG and LONGCHUNKSIZE. The third factor is the FORMAT assigned in the COLUMN command, if any.

The LONG system variable sets the maximum display width for columns with LONG datatypes. By default, this value is 80, meaning that only 80 characters appear in the report unless you change the system setting. The maximum setting in Oracle7 is 2 gigabytes. You'll want to SET LONG big enough to display the largest value that will appear in the report.

The LONGCHUNKSIZE system variable sets the character size of increments used to retrieve LONG values. By default, the system setting is 80. You can increase this value up to a system-dependent maximum determined by MAXDATA.

Without a COLUMN command, the default display width of LONG data columns is the smaller of the two system variables LONG and LONGCHUNKSIZE. If you do use a COLUMN command, the display width is the smaller of LONGCHUNKSIZE and the format provided in the COLUMN command. Therefore, if LONGCHUNKSIZE is 80 and you use a format of A100 in a COLUMN command, the display width will be 80 characters.

Figure 5-28 shows the use of LONG, LONGCHUNKSIZE, and FORMAT to display a LONG data column in a report.

```
SET LONG 1000
SET LONGCHUNKSIZE 100
COLUMN sql FORMAT a60 WORD_WRAPPED
SELECT
     query_code "id",
     query_create_date "create",
     query_sql "sql"
FROM
     queries
WHERE
     query_code = 102;

    id create    sql
------ --------- ------------------------------------------------------------
   102 09-JUL-96 select name_id id, name_last last, s1.student_regseq seq,
                 s1.student_sponsor sponsor, s1.student_total_fee fee from
                 student s1, name where name_seriesno = name_now(name_id) and
                 name_id in ('@133','@330','@409','@925') and
                 s1.student_id(+) = name_id and s1.rowid(+) =
                 sponsor_exist(name_id, 'Self','Employer') order by 1;
```

Figure 5-28: Formatting a LONG data column for display.

Comments

1. By setting the system variable LONG to 1000, you can display up to 1000 characters in a LONG column.
2. Retrieval of LONG data columns will occur in increments of 100 characters. The default is 80.
3. The COLUMN command uses a FORMAT of A60 for the LONG column. Because this value is smaller than the LONGCHUNKSIZE setting, the display width in the report will be 60.

See Also

1. For a related discussion of LONG and LONGCHUNKSIZE, see the section in Chapter 2 on page 54 that describes how to compress report columns when the physical line size of the report is limited.

DISTINCT Keyword

The DISTINCT keyword is a curious beast. It finds considerable use when testing and debugging queries to ensure the integrity of the report population (see Chapter 7). But it also can act as a crutch to a sloppy query, allowing you to display the desired results without writing a tight query. Try to avoid using DISTINCT as you construct queries.

How do I display only those rows where one data column is distinct?

I need to select a complete row whenever one of the columns is distinct. SELECT DISTINCT does not produce the desired result. For example, SELECT DISTINCT colA, colB, colC returns all rows when the combination of colA, colB, and colC is distinct. I want the query to return colA, colB, and colC but only when colA is distinct. Is there a way to do this?

Approach

Let's deal with a specific example using the training registration data shown in previous figures. Suppose we want to identify all students who have only registered once for training classes. That is, student_id must be distinct. Further, suppose that the report should display the three columns student_id, student_regseq, and student_sponsor.

Figure 5-29 shows one approach to the problem. Note that it uses a subquery to return a count grouped by the column that must be distinct. The main query retains only those rows where the count is 1 (i.e., the column is distinct).

Comments

1 The data include three rows for people who each registered only once for training classes.

2 The DISTINCT feature is captured here by requiring the count returned by the subquery to be 1. The GROUP BY in the subquery groups rows by the column that must be distinct.

Other SQL structures produce the same effect. For example, you could use a correlated subquery where the correlating column is the one that must be distinct. In general, the solution is WHERE 1 = (SELECT COUNT(*) FROM SameTable t2 WHERE t2.ColDistinct = t1.ColDistinct). In the example in Figure 5-29, this becomes WHERE 1 = (SELECT COUNT(*) FROM student s2 WHERE s2.student_id = s1.student_id).

```
id      seq  sponsor
-----  ----  ----------
@133      1  Self
@226      1  Mixed
@226      2  Self
@226      3  Self
@330      1  Mixed
@505      1  Self
@505      2  Employer
@925      1  Employer


SELECT
     s1.student_id "id",
     s1.student_regseq "seq",
     s1.student_sponsor "sponsor"
FROM
     student s1
WHERE
     (s1.student_id, 1) IN
     (SELECT s2.student_id, COUNT(*)
     FROM student s2
     GROUP BY s2.student_id);


id      seq  sponsor
-----  ----  ----------
@133      1  Self
@330      1  Mixed
@925      1  Employer
```

Figure 5-29: Query returning rows for distinct values in one column.

See Also

1. When testing a query to determine if join problems exist, DISTINCT plays an important role. Beginning on page 218, Chapter 7 discusses the special uses of DISTINCT when finding and correcting join errors.