



Chapter 10

Interesting Queries

This chapter departs from the format used in earlier chapters. Rather than dealing with frequently asked questions and grouping those questions into common categories, this chapter examines several queries that pose interesting problems.

Some of the problems discussed in this chapter require only simple solutions (e.g., using `UNION ALL` instead of `UNION`), but these problems can still be subtle to identify and correct. In hindsight, many query problems appear simple, but when you're stuck, it's baffling.

Other problems discussed in this chapter require more complicated solutions. For example, one query is a brief journey into statistics. The query writer wishes to select two objects from a population of N objects when the order of the objects does not matter. In this case she wanted to list unique combinations of cities (e.g., Boston-Chicago) and report the distance between those two cities. The solution discussed in this chapter uses a cartesian product of data views defined by `FROM` clause subqueries. There are indeed times when a cartesian product proves useful.

You'll also again find an emphasis on performance in this chapter. `SQL`, `SQL*PLUS`, and `PL/SQL` are very flexible tools. While this flexibility provides definite advantages, it frequently produces several possible solutions to problems. Yet not all solutions, even if they're functionally equivalent, are equal. Queries that run faster and are more elegant are preferable to queries that run slower and are more cumbersome to decipher. This chapter discusses one situation in which the query writer got the correct results but used four correlated subqueries in the process. Performance suffered as the query ground through these subqueries. This chapter discusses two alternative approaches, one of which ran 15 times faster than the original query.

How can I write a query functionally equivalent to a NOT DISTINCT?

I'd like to get a list of column A and column B where the two columns are not unique, that is, where the same combination of column A and column B occurs more than once in a table. Functionally, this seems the opposite of the DISTINCT keyword, yet when I use NOT DISTINCT, I get an error message. How can I construct a query that will identify all non-unique combinations of two data columns?

Approach

Figure 10-1 shows three approaches to this problem.

```
Method 1
SELECT
    colA,
    colB,
    COUNT(*)
FROM
    tab1
GROUP BY
    colA,
    colB
HAVING
    COUNT(*) > 1;

Method 2
SELECT DISTINCT
    t1.colA,
    t1.colB
FROM
    tab1 t1
WHERE
    2 <=
    (SELECT COUNT(*)
     FROM tab1 t2
     WHERE t2.colA = t1.colA
     AND t2.colB = t1.colB);

Method 3
SELECT DISTINCT
    t1.colA,
    t1.colB
FROM
    tab1 t1
WHERE
    EXISTS
    (SELECT 'x'
     FROM tab1 t2
     WHERE t2.colA = t1.colA
     AND t2.colB = t1.colB
     AND t2.ROWID != t1.ROWID);
```

Figure 10-1: Three queries equivalent to NOT DISTINCT.

Comments

- ▶ Method 1 uses a single query with `GROUP BY` and `HAVING` clauses to identify the nonunique combinations of two columns.
- ▶ Method 2 uses a correlated subquery that identifies when the combination of the two columns occurs more than once.
- ▶ Method 3 also uses a correlated subquery but with an `EXISTS` that stops the subquery whenever the combination of the two columns in the main query occurs in the subquery and the `ROWIDS` differ. Methods 2 and 3 would only be appropriate when indexes existed on the two columns in question

Table 10-1 shows the results of performance tests with the three methods. The single query using `GROUP BY` and `HAVING` ran considerably faster than either of the two methods using subqueries. As expected, the subquery method that required only partial execution (Method 3) ran faster than the method that required complete execution.

Query	--Elapsed Time (sec)--		Mean Ratio	Statistical Difference?	
	Mean	StdDev			
Method 1	0.36	0.03	10	1.0	
Method 2	1.35	0.03	10	3.8	Yes
Method 3	1.19	0.05	10	3.3	Yes

Notes:

1. Optimizer: Rule
2. Hints: None
3. Indexes: Nonunique index on colA and colB
4. Number of rows: 1000
5. Statistical test: two-tailed t-test, 0.01 probability

Table 10-1: Performance for queries in Figure 10-1.

See Also

1. Method 1 is nearly identical to one of the techniques used to find rows that multiply through joins when you’re debugging join problems. For examples that use this procedure, see Chapter 7 on page 226 and Chapter 7 on page 230. For another test of performance using this method, see Chapter 9 (Table 9-13 on page 280).
2. The `DISTINCT` keyword can sometimes pose special problems for query writers. For a discussion of these problems, see Chapter 5 on page 170.

How can I get a frequency distribution by a time interval?

A table that I use contains date and time information showing when users log on and log off our administrative database. I'd like to prepare a report that counts the number of logons by duration of connect time. The duration should be displayed in categories (e.g., ≤ 5 minutes, 5–10 minutes, etc.). How can I do this?

Approach

Figure 10-2 describes a table containing logon start and end times. The connect duration must be computed by subtracting the start time from the end time. This computation can then be recoded into categories such as 5 minutes or less, and the categories used in a standard frequency distribution query.

```

SQL> DESC logtime
Name                               Null?    Type
-----
LOGTIME_ID                          NOT NULL NUMBER(4)
LOGTIME_SEQNO                       NOT NULL NUMBER(4)
LOGTIME_LOGON                       DATE
LOGTIME_LOGOFF                      DATE

Primary key: LOGTIME_ID, LOGTIME_SEQNO
Index: on primary key

SQL> SELECT COUNT(*) "count" FROM logtime;

      count
-----
        995
    
```

Figure 10-2: LOGTIME table described.

This report can be handled easily with straight SQL, but not without creating a messy query that's difficult to understand, test, and debug. Figure 10-3 shows one approach that produces the desired report.

Comments

▶ Connect time is computed by first converting logtime_logoff and logtime_logon date and time into a numeric value. The 'sssss' that appears in the format converts the time into the number of seconds past midnight. Subtracting the converted logtime_logon value from the converted logtime_logoff value produces the connect time in seconds.

```

BREAK ON REPORT
COMPUTE SUM OF count ON REPORT
COLUMN online FORMAT A20
SELECT
  DECODE(
    DECODE(SIGN((TO_NUMBER(TO_CHAR(logtime_logoff,'YMMDDSSSS'))
      -
      TO_NUMBER(TO_CHAR(logtime_logon,'YMMDDSSSS')) - 300),
      -1, 0, 0, 0,
    DECODE(SIGN((TO_NUMBER(TO_CHAR(logtime_logoff,'YMMDDSSSS'))
      -
      TO_NUMBER(TO_CHAR(logtime_logon,'YMMDDSSSS')) - 600),
      -1, 1, 0, 1,
    2)),
    0, '0: <=5 min',
    1, '1: >5 and <=10 min',
    2, '2: >10 min') "online",
  COUNT(*) "count"
FROM logtime
GROUP BY
  DECODE(
    DECODE(SIGN((TO_NUMBER(TO_CHAR(logtime_logoff,'YMMDDSSSS'))
      -
      TO_NUMBER(TO_CHAR(logtime_logon,'YMMDDSSSS')) - 300),
      -1, 0, 0, 0,
    DECODE(SIGN((TO_NUMBER(TO_CHAR(logtime_logoff,'YMMDDSSSS'))
      -
      TO_NUMBER(TO_CHAR(logtime_logon,'YMMDDSSSS')) - 600),
      -1, 1, 0, 1,
    2)),
    0, '0: <=5 min',
    1, '1: >5 and <=10 min',
    2, '2: >10 min')
ORDER BY 1;

```

online	count
0: <=5 min	196
1: >5 and <=10 min	188
2: >10 min	611
sum	995

Figure 10-3: SQL query that produces a frequency distribution by a time interval.

- ▶ The `DECODE` and `SIGN` functions get used to create categories based on the connect time. In this case the connect time (in seconds) is compared with 600 (i.e., 10 minutes). If connect time minus 600 is negative or zero, the `DECODE` assigns the category 1 to the value.
- ▶ The outermost `DECODE` attaches descriptions to the time intervals. For example, the category 0 is given the label '0: <=5 min'. The leading zero in the label ensures that the categories sort in the proper order in the report. You could handle the category codes and the descriptions separately if this seemed preferable, but it would require an even messier query.

The query in Figure 10-3 can be greatly simplified using PL/SQL functions. Figure 10-4 shows a query that uses two PL/SQL functions to produce the identical frequency distribution shown earlier. Note how easy it is to interpret the query, in part because the query itself is smaller, but also because the PL/SQL function names convey their purpose.

```
BREAK ON REPORT
COMPUTE SUM OF count ON REPORT
COLUMN online FORMAT A20
SELECT
  time_cat(online_duration(logtime_logon, logtime_logoff)) "online",
  COUNT(*) "count"
FROM
  logtime
GROUP BY
  time_cat(online_duration(logtime_logon, logtime_logoff))
ORDER BY 1;
```

Figure 10-4: Frequency distribution by time interval using two PL/SQL functions.

Comment

► This query uses two PL/SQL functions. One, named `online_duration`, uses the `logtime_logon` and `logtime_logoff` times to compute connect time. The second, named `time_cat`, uses the connect time to determine which time interval the value belongs in.

For completeness, Figure 10-5 shows the `online_duration` function and Figure 10-6 shows the `time_cat` function.

```
CREATE OR REPLACE FUNCTION online_duration
  (start_time IN DATE,
   end_time IN DATE)
RETURN NUMBER
AS
  duration_in_seconds NUMBER(8);
BEGIN
  duration_in_seconds :=
    TO_NUMBER(TO_CHAR(end_time, 'YYMMDDSSSS'))
    -
    TO_NUMBER(TO_CHAR(start_time, 'YYMMDDSSSS'));
  RETURN duration_in_seconds;
END;
/
```

Figure 10-5: PL/SQL function named `online_duration`.

If ease of interpretation, testing, and debugging are your primary concerns, then the query in Figure 10-4 that uses the two PL/SQL

```

CREATE OR REPLACE FUNCTION time_cat
(duration_in_seconds IN NUMBER)
RETURN VARCHAR2
AS
    time_category VARCHAR2(20);
    minute_duration NUMBER(8,2);
BEGIN
    minute_duration := duration_in_seconds/60;

    IF minute_duration <= 5.0 THEN
        time_category := '0: <=5 min';
    ELSIF minute_duration > 5.0
        AND minute_duration <= 10.0 THEN
        time_category := '1: <5 AND <= 10 min';
    ELSIF minute_duration > 10.0 THEN
        time_category := '2: >10 min';
    END IF;

    RETURN time_category;
END;
/

```

Figure 10-6: PL/SQL function named time_cat.

functions is definitely the better choice. However, as the following table makes clear, the query with the PL/SQL functions is also considerably slower than the messier but faster SQL query in Figure 10-3. Again, there's a trade-off to make between performance and elegance.

Query	--Elapsed Time (sec)--			Mean Ratio	Statistical Difference?
	Mean	StdDev	Trials		
Figure 10-3	1.93	0.13	10	1.0	
Figure 10-4	23.99	0.27	10	12.4	Yes

Notes:

1. Optimizer: Rule
2. Hints: None
3. Indexes: on primary key in LOGTIME table
4. Number of rows: 995
5. Statistical test: two-tailed t-test, 0.01 probability

Table 10-2: Performance for queries in Figure 10-3 and Figure 10-4.

See Also

1. For another example that uses DECODE and SIGN to recode numeric data, see Chapter 3 (Figure 3-10 on page 92).

How can I cross-tabulate dates by week and day of the week?

One of the tables that I use tracks overtime hours by date and department. I'd like to prepare a cross-tabulated report with days of the week as report columns and weeks as the rows. Totals by week and by day of the week and a grand total also should appear in the report. And finally, if one week has no overtime hours, that week should still appear in the report even though the cell values will all be zero. How can I prepare a report like this?

Approach

Let's discuss a partial solution first, for the well-behaved situation where at least one day in each week of the report contains overtime hours. Figure 10-7 describes a simple *OVERTIME* table, where the primary key is the department code and the date the overtime hours occurred.

```

SQL> DESC overtime
Name                               Null?    Type
-----
OVERTIME_UNIT_CODE                 NOT NULL NUMBER(4)
OVERTIME_DATE                       NOT NULL DATE
OVERTIME_HOURS                       NUMBER(2)

Primary key: overtime_unit_code, overtime_date
Index: none

SQL> SELECT count(*) "count" FROM overtime;

      count
-----
         24
    
```

Figure 10-7: Description of an *OVERTIME* table.

Several methods exist to categorize dates by week. One convenient approach uses the TRUNC function with a DAY format that truncates dates to the starting date of a week. Starting days differ by NLS_TERRITORY; in the AMERICA territory, the start day for a week is Sunday. The query in Figure 10-8 uses this technique. It also relies on the DECODE function to create the report columns. As an alternative to the DECODE approach, you could use an identity table as discussed in Chapter 6.

The query in Figure 10-8 works fine for every week in which overtime hours exist. Suppose, however, that overtime hours are only inserted into the table when they occur so that no entry is made on days when overtime does not occur. In this case, if an entire week has no overtime

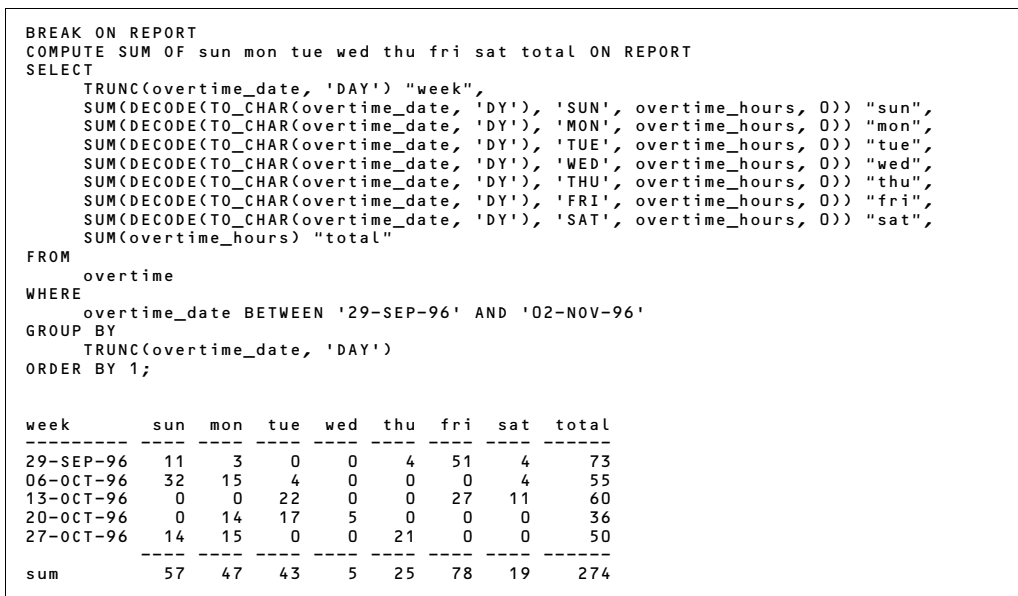


Figure 10-8: Overtime hours by week and day of the week.

hours, then that week will not appear in the report. To ensure that each week appears in the report, even if the overtime hours for that week are zero, you'll need to revise the query in Figure 10-8.

One revision employs a second table populated with each date covered by the report (i.e., for each date defined by the WHERE clause). Suppose you created and populated a table called *TABL1996* that contained each day in the year 1996. An outer join between this table and the *OVERTIME* table would ensure that each week appeared in the report even if no overtime hours existed for that week.

See Also

1. See Chapter 3 (Figure 3-5 on page 84) for another example that uses the TRUNC function to collapse days into weeks. This section also discusses how the starting day of weeks vary depending upon NLS_TERRITORY.
2. Chapter 6 (Figure 6-6 on page 182) shows an identity matrix table used to produce a cross-tabulated report.

How can I count items across two tables?

At present, I track sales at two stores in two separate tables. Frequently, I want to know the total number of sales and the total quantity sold. This means combining the data from the two tables. I tried creating a view using UNION, but this doesn't always seem to give the correct result. Figure 10-9 shows the SQL that I used to create the view.

```
CREATE OR REPLACE VIEW flower_sales
(flower_name,
flower_quantity)
AS
(SELECT f1_name, f1_quantity FROM f1
UNION
SELECT f2_name, f2_quantity FROM f2);
```

Figure 10-9: View for *FLOWER_SALES*.

Figure 10-10 shows sample data that illustrate the problem. The counts and sums of Cacti are clearly wrong using the view.

Data	seqno	quantity
Dracaena	1	50
Cacti	1	20
Cacti	2	40
Cacti	3	25
Cacti	1	25
Cacti	2	75
Dracaena	1	35
Philodendron	1	10

```
Query
BREAK ON REPORT
COMPUTE SUM OF quantity count ON REPORT
COLUMN flower HEADING 'flower|group'
SELECT
    flower_name "flower",
    SUM(flower_quantity) "quantity",
    COUNT(*) "count"
FROM flower_sales
GROUP BY flower_name
ORDER BY 1;
```

Report	quantity	count
Cacti	160	4
Dracaena	85	2
Philodendron	10	1
sum	255	7

Figure 10-10: Query that gives the wrong result from *FLOWER_SALES*.

Approach

The problem is actually fairly simple. The view was created as a compound query that `UNION`d two data columns from each of the tables. In the `UNION` process an implicit `DISTINCT` eliminates duplicate rows. Note in Figure 10-10 that two rows have identical names and quantities (i.e., Cacti and 25, respectively). Only one of these rows gets retained with `UNION`. Thus the solution to the problem merely requires that the view be recreated with `UNION ALL` instead of `UNION`.

However, you also may want to consider replacing the view entirely. Oracle7 Release 7.2 allows the use of subqueries in the `FROM` clause, meaning you can achieve the same effect of the view by using a subquery. Figure 10-11 illustrates how this works.

Query		
BREAK ON REPORT		
COMPUTE SUM OF quantity count ON REPORT		
COLUMN flower heading 'flower group'		
SELECT		
flower_name "flower",		
sum(qty) "quantity",		
count(*) "count"		
FROM		
(SELECT f1_name AS flower_name, f1_quantity AS qty FROM f1		
UNION ALL		
SELECT f2_name, f2_quantity FROM f2)		
GROUP BY		
flower_name		
ORDER BY 1;		
Report		
flower		
group	quantity	count
-----	-----	-----
Cacti	185	5
Dracaena	85	2
Philodendron	10	1
-----	-----	-----
sum	280	8

Figure 10-11: Correct results using `UNION ALL` and a `FROM` subquery.

Comments

► The `FLOWER_SALES` view gets replaced in this query by a `FROM` clause subquery that uses `UNION ALL`.

See Also

1. See Chapter 8 (Figure 8-15 on page 250) for another example using a `FROM` clause subquery.

How do I select distinct combinations of two objects from N objects?

One of my tables contains information on cities and their locations. I want to use the location data to compute distance between any two cities. However, I only want each combination of cities to appear once in the report. For example, the report should include the distance between Boston and Chicago, but the distance between Chicago and Boston is redundant and should not appear. Is this possible?

Approach

There are $N!/r!(N-r)!$ distinct combinations of r objects selected from a population of N objects when the order of the items selected does not matter. In this case we wish to select two cities (i.e., $r = 2$) from N cities. When $r = 2$ the formula simplifies to $N(N - 1)/2$ distinct combinations. If, for example, there are three cities (i.e., $N = 3$), then there will be $3 \cdot 2 / 2 = 3$ distinct combinations of cities. Assume that the cities are Boston, Chicago, and Atlanta. The distinct combinations are Boston-Chicago, Boston-Atlanta, and Chicago-Atlanta.

With this bit of statistical review, it's possible to consider the query that will display only distinct combinations of cities. Figure 10-12 describes a simple table containing data on city locations; it also shows the data in the table. Note that there are four cities in the table. We expect $4 \cdot 3 / 2 = 6$ distinct combinations of cities in the report.

```

SQL> desc cityloc
Name                               Null?      Type
-----
CITY                                NOT NULL  VARCHAR2(3)
XPOS                                NOT NULL  NUMBER(2)
YPOS                                NOT NULL  NUMBER(2)

SQL> select * from cityloc;
city      xpos      ypos
-----
LA        0          0
CHI       3          0
ATL       3          4
BOS       1          6
    
```

Figure 10-12: Description of *CITYLOC* table.

Figure 10-13 shows a query that returns only distinct combinations of cities. It is interesting in two respects. First, it uses a cartesian product of the *CITYLOC* table. But second, the cartesian product actually occurs between two versions of *CITYLOC* created by FROM clause subqueries.

This allows the addition of `ROWNUM` to each of the *CITYLOC* tables, which in turn makes it possible to construct a `WHERE` clause that ensures only distinct combinations get returned.

```

SELECT
  city1,
  city2
from
  (SELECT city AS city1, ROWNUM AS row1 FROM cityloc),
  (SELECT city AS city2, ROWNUM AS row2 FROM cityloc)
WHERE
  row2 < row1;

city1 city2
-----
CHI    LA
ATL    LA
BOS    LA
ATL    CHI
BOS    CHI
BOS    ATL

```

Figure 10-13: Query that displays only distinct combinations of two cities.

Comments

- ▶ Two versions of the *CITYLOC* table get used in the query, each created by a `FROM` clause subquery. This allows the addition of `ROWNUM` to each view of the *CITYLOC* data.
- ▶ Note that no join criterion exists; there is a cartesian product between the two *CITYLOC* views. Requiring that `ROWNUM` from the first view is less than `ROWNUM` from the second view ensures that only distinct combinations of cities will appear in the report. Note that the report does, as expected, include six distinct combinations of cities.

See Also

1. Cartesian products generally mean trouble, because in most cases they occur when a query writer forgets to join two tables. This can seriously degrade system performance if the tables in question are large. But there are other times, like the query in Figure 10-13, when you can use a cartesian product with good effect. For another example, see Chapter 3 (Figure 3-20 on page 104).

How do I find the maximum number of contacts any one customer has?

I'd like to write a query that returns the maximum number of contacts made with any of our customers. I can, of course, count contacts by customer and order the report in descending order to get the information. But all I really need is just the first line from this report. Is there some way I can prepare a report that shows only this one number?

Approach

Figure 10-14 describes a simple *CONTACT* table. It includes data columns that capture the customer identification number, the sequence number of the contact, the type of contact, and the date of the contact. The identification and sequence number columns comprise the primary key. The table contains approximately 4400 contacts with 990 customers.

```

SQL> DESC contact
Name                               Null?    Type
-----
CONTACT_ID                          NOT NULL NUMBER(4)
CONTACT_SEQNO                       NOT NULL NUMBER(3)
CONTACT_TYPE                         NOT NULL VARCHAR2(4)
CONTACT_DATE                         NOT NULL DATE

Primary key: CONTACT_ID, CONTACT_SEQNO
Index: on primary key

SQL> SELECT COUNT(*) "contacts",
2 COUNT(DISTINCT contact_id) "customers"
3 FROM contact;

  contacts  customers
-----
    4398      990

```

Figure 10-14: Customer *CONTACT* table described.

Finding the number of contacts for each customer is simple enough; you merely *GROUP BY* *contact_id* and count the contacts. The trick becomes reporting only the maximum count. Several approaches exist:

- Create a view that counts contacts by each *contact_id*, and then write the query to return the maximum number from this view;
- Use a *FROM* clause subquery to duplicate the effect of a view without the need to first create the view.
- Use a *PL/SQL* cursor and a bind variable to return the maximum contact count.

The following three figures illustrate each approach. Figure 10-15 shows a view used to return the maximum contact count.

```
View
CREATE VIEW contacts_by_id
  (id, num_contacts)
AS
  (SELECT contact_id, COUNT(*)
   FROM contact
   GROUP BY contact_id);
Query
SELECT MAX(num_contacts) FROM contacts_by_id;
```

Figure 10-15: Maximum contact count with a view.

Figure 10-16 provides the same result but uses a single query with a FROM clause subquery to replace the view.

```
SELECT
  MAX(num_contacts)
FROM
  (SELECT contact_id AS id, COUNT(*) AS num_contacts
   FROM contact
   GROUP BY contact_id);
```

Figure 10-16: Maximum contact count with FROM clause subquery.

Finally, Figure 10-17 uses a PL/SQL cursor and a bind variable to produce the same effect as the other two approaches.

```
PL/SQL
SET TRANSACTION READ ONLY;
VARIABLE max_contacts NUMBER
DECLARE
  CURSOR main_cursor IS
    SELECT COUNT(*)
    FROM contact
    GROUP BY contact_id
    ORDER BY 1 DESC;
BEGIN
  OPEN main_cursor;
  FETCH main_cursor INTO :max_contacts;
  CLOSE main_cursor;
END;
/
Query
SELECT :max_contacts FROM DUAL;
```

Figure 10-17: Maximum contact count with a PL/SQL cursor and bind variable.

Table 10-3 presents performance results based on 10 trials with each query. Note that none of the three methods is significantly faster than the others. In this case, personal preference can best guide your selection of a query strategy.

Query	--Elapsed Time (sec)--		Trials	Mean Ratio	Statistical Difference?
	Mean	StdDev			
Figure 10-15	1.39	0.19	10	1.0	
Figure 10-16	1.42	0.27	10	1.0	No
Figure 10-17	1.62	0.27	10	1.1	No

Notes:

1. Optimizer: Rule
2. Hints: None
3. Indexes: on primary key in *CONTACT* table
4. Number of rows: 4400
5. Statistical test: two-tailed t-test, 0.01 probability

Table 10-3: Performance for queries in Figure 10-15 through Figure 10-17.

See Also

1. FROM clause subqueries give query writers added flexibility when approaching some query problems. For other examples of this type of subquery, see Figure 10-11 on page 301 in this chapter and also Chapter 8 (Figure 8-15 on page 250).

How do I improve the performance of a query?

I've got a simple problem, but the query I wrote takes forever to complete. Since the query will be run frequently, I'd like to make it faster.

I'm using a table called *LOC* (for location) that contains data on the branch location of employees, several characteristics of their employment, and the date when any change became effective. Employees can only be assigned to one location at a time. However, they may work in several locations during their employment with the company, including the same location more than once. Whenever any of the characteristics of employment change, a new row is added to the table. Figure 10-18 describes the table columns and provides basic information about the table.

```

SQL> DESC Loc
Name                               Null?    Type
-----
LOC_STAFF_ID                       NOT NULL NUMBER(4)
LOC_LOCATION                       NOT NULL VARCHAR2(15)
LOC_EFF_DATE                       NOT NULL DATE
other employment columns

Primary key: LOC_STAFF_ID, LOC_EFF_DATE
Index: on primary key

SQL> SELECT COUNT(*) "rows",
2 COUNT(DISTINCT loc_staff_id) "employees"
3 FROM loc;

      rows  employees
-----
      9944       1000
    
```

Figure 10-18: Employee location table (*LOC*) described.

I need to determine the date when the current location became effective for each employee. Here's the approach I took. The current location is *loc_location* at *MAX(loc_eff_date)*; that is, it's the most recent location. I then found the *MIN(loc_eff_date)* for the current location. But in case the person had worked at the current location more than once, I also had to add logic that no other location could have a *loc_eff_date > MIN(loc_eff_date)* for the current location. Figure 10-19 shows data for one employee and identifies the correct row that the query should return.

Comments

► The current location for this employee is Newark. She worked previously at Newark, but this was followed by a transfer to Nashville. Her present assignment began on 16-May-1997.

id	location	eff_date
8	Newark	25-JUL-95
8	Nashville	16-JUN-96
8	Newark	16-MAY-97
8	Newark	04-SEP-97

Figure 10-19: Sample data from the employee location table.

Figure 10-20 shows the SQL query that I wrote based on this approach. The query gives the correct results but it takes a very long time to complete. I'd like to revise the query so it runs faster. How can I do that?

```

SELECT
  l1.loc_staff_id,
  l1.loc_location,
  l1.loc_eff_date
FROM loc l1
WHERE l1.loc_eff_date =
  (SELECT MIN(l2.loc_eff_date)
   FROM loc l2
   WHERE l2.loc_staff_id = l1.loc_staff_id
   AND l2.loc_location =
     (SELECT l3.loc_location
      FROM loc l3
      WHERE l3.loc_staff_id = l1.loc_staff_id
      AND l3.loc_eff_date =
        (SELECT MAX(l4.loc_eff_date)
         FROM loc l4
         WHERE l4.loc_staff_id = l1.loc_staff_id)))
   AND NOT EXISTS
   (SELECT 'x'
    FROM loc l5
    WHERE l5.loc_staff_id = l1.loc_staff_id
    AND l5.loc_eff_date > l2.loc_eff_date
    AND l5.loc_location != l2.loc_location));
    
```

Figure 10-20: Query approach 1.

Comments

- 1 For each employee, choose the minimum effective date . . .
- 2 For the location . . .
- 3 That is the most recent . . .
- 4 And where no other location has a later effective date. This ensures that you're not picking up an earlier employment at the current location that was followed by employment in other branch locations.

Approach

Tuning a query is still more artform than standard practice, despite the common advice about optimizer goals, access paths, hints, driving tables, inadvertent disabling of indexes, and other dangers to avoid or improvements to seek. Often the most dramatic performance gains occur when you rephrase your definitions and recast your query logic. The query in Figure 10-20 uses four correlated subqueries. Let's see if we can't make it less complicated.

The query in Figure 10-20 defines the current location as the location with the most recent effective date (i.e., it is `loc_location` at `MAX(loc_eff_date)`). This is certainly accurate. But the current location is also the location at the `MIN(loc_eff_date)` where no other location has a later effective date. Using this definition simplifies the query, as shown in Figure 10-21. The revised query reduces the number of correlated subqueries from four to two. Performance should improve (and does, as we'll see shortly).

```

SELECT
  l1.loc_staff_id,
  l1.loc_location,
  l1.loc_eff_date
FROM loc l1
WHERE l1.loc_eff_date =
  (SELECT MIN(l2.loc_eff_date)
   FROM loc l2
   WHERE l2.loc_staff_id = l1.loc_staff_id
   AND NOT EXISTS
     (SELECT 'x'
      FROM loc l3
      WHERE l3.loc_staff_id = l1.loc_staff_id
      AND l3.loc_eff_date > l2.loc_eff_date
      AND l3.loc_location != l2.loc_location));

```

Figure 10-21: Query approach 2.

Comments

- ▶ For each employee, choose the minimum effective date . . .
- ▶ Where no other location has a later effective date.

The query in Figure 10-21 still uses two correlated subqueries, one of which merely returns a minimum effective date. You could accomplish the same thing with a `MIN` group function. Figure 10-22 shows how the `MIN` group function simplifies the query even more.

```

SELECT
  loc_staff_id,
  loc_location,
  MIN(loc_eff_date)
FROM
  loc l1
WHERE
  NOT EXISTS
  (SELECT 'x'
   FROM loc l2
   WHERE l2.loc_staff_id = l1.loc_staff_id
        AND l2.loc_eff_date > l1.loc_eff_date
        AND l2.loc_location != l1.loc_location)
GROUP BY
  loc_staff_id,
  loc_location;
    
```

Figure 10-22: Query approach 3.

Comments

- ▶ For each employee, choose the minimum effective date . . .
- ▶ Where no other location has a later effective date.

The performance tests in Table 10-4 show that reducing the number of correlated subqueries from four to two improved performance significantly. But the dramatic improvement occurred when using a group function to reduce the number of correlated subqueries to one. Compared with the original query, the final query ran nearly 15 times faster. If the query will be run against an enormous database or if it will be run repeatedly, even a 1500 percent improvement may not be sufficient. But it does demonstrate the potential for improved performance when you rethink your definitions and query strategy.

Query	---Elapsed Time (sec)--		Mean Ratio	Statistical Difference?
	Mean	StdDev		
Figure 10-20	648.66	42.20	10	1.00
Figure 10-21	400.05	15.52	10	0.62
Figure 10-22	43.70	1.30	10	0.07

Notes:

1. Optimizer: Rule
2. Hints: None
3. Indexes: Primary key on LOC table
4. Number of rows: 9944
5. Statistical test: two-tailed t-test, 0.01 probability

Table 10-4: Performance for the queries in Figure 10-20 to Figure 10-22.

See Also

1. Correlated subqueries are marvelous query tools, but don't go overboard using them. To replace correlated subqueries with `PL/SQL` functions, see the discussion in Chapter 5 on page 152 — but also see the performance comparisons in Chapter 9 (Table 9-5 on page 261).

