



Chapter 9

Performance

This chapter deviates from previous chapters in that it does not introduce any new frequently asked questions. Instead, it revisits questions from other chapters where two or more approaches solved the problem. Each alternative method of accomplishing the same action has performance implications. Basically, does one method run faster than the others? This chapter examines this issue.

All performance tests that appear in this chapter were based on simulated data values using the following procedures:

- Data values for all columns were simulated using Pump-It-Up Lite from The Client Server Factory.
- Primary keys were created for all tables, ensuring indexes were available for the data columns comprising the keys.
- Each alternative method was run in a new SQL*PLUS session in which the SQL_TRACE facility was enabled.
- Each alternative method was run 11 times and results from the first trial discarded. All trials used the rule-based optimizer.
- After each trial, TKPROF was run with the INSERT option enabled and trace file statistics were stored in a database table.
- Combined fetch and execute elapsed times were used to evaluate query performance.
- Results from the 10 trials of each method were averaged. The mean and standard deviations from the trials were used to compute *t*-test statistics to determine if the average performance times differed significantly among the functionally equivalent queries.

Functionally Equivalent Alternatives

Interpretation of the performance tests depends in part on knowing what primary keys exist in each table. Table 9-1 lists these keys for reference.

Table	Key	Position
NAME	NAME_ID	1
	NAME_SERIESNO	2
ADDRESS	ADDRESS_ID	1
	ADDRESS_TYPE	2
	ADDRESS_SERIESNO	3
PERSON	PERSON_ID	1
STUDENT	STUDENT_ID	1
	STUDENT_REGSEQ	2
UNDERGRAD	UNDERGRAD_ID	1

Table 9-1: Primary keys in simulated data tables.

Which method of computing due dates is faster?

Chapter 3 presented two methods for determining due dates a specified number of business days after a closing date. One method, shown in Figure 3-6 on page 87, used a complex SQL `DECODE`; the second method, shown in Figure 3-8 on page 89, employed a PL/SQL function to return the due date. Both produce the same results. Which one is faster?

Approach

For ease of reference, Figure 9-1 shows the two functionally equivalent methods of determining due dates..

Table 9-2 compares the average elapsed time for 10 queries using each method of computing due dates. The SQL solution using the `DECODE` (i.e., Figure 3-6) is significantly faster than the PL/SQL function, by almost a 3:1 factor. Depending on your particular circumstances, you may want to trade the ease of interpretation provided by the PL/SQL function for the improved performance of the `DECODE`.

See Also

1. The discussion about due dates begins in Chapter 3 on page 86.
2. Chapter 3 (Figure 3-7 on page 88) shows the `due_date` function.

```

Figure 3-6
SELECT
  seminar_code "class",
  TO_CHAR(seminar_end_date, 'dd-MON-yyyy, Day') "enddate",
  TO_CHAR(
    seminar_end_date +
    TRUNC(14/5)*7 +
    DECODE(
      mod(14,5),
      4, DECODE(TO_CHAR(seminar_end_date, 'D'),1,4,2,4,7,5,6),
      3, DECODE(TO_CHAR(seminar_end_date, 'D'),1,3,2,3,3,3,7,4,5),
      2, DECODE(TO_CHAR(seminar_end_date, 'D'),5,4,6,4,7,3,2),
      1, DECODE(TO_CHAR(seminar_end_date, 'D'),6,3,7,2,1),
      0, DECODE(TO_CHAR(seminar_end_date, 'D'),1,-2,7,-1,0)
    ),
    'dd-MON-yyyy, Day') "duedate"
FROM seminars
ORDER BY
  TO_CHAR(seminar_end_date, 'D');

Figure 3-8
SELECT
  seminar_code "class",
  TO_CHAR(seminar_end_date, 'dd-MON-yyyy, Day') "enddate",
  TO_CHAR(du_date(seminar_end_date, 14), 'dd-MON-yyyy, Day') "duedate"
FROM seminars
ORDER BY
  TO_CHAR(seminar_end_date, 'D');

```

Figure 9-1: Two methods of computing due dates.

Query	--Elapsed Time (sec)--			Mean Ratio	Statistical Difference?
	Mean	StdDev	Trials		
Figure 3-6	6.03	0.94	10	1.0	
Figure 3-8	15.53	1.46	10	2.6	Yes

Notes:

1. Optimizer: Rule
2. Hints: None
3. Indexes: None
4. Number of rows: 1000
5. Statistical test: two-tailed t-test, 0.01 probability
6. Elapsed time = elapsed execute + elapsed fetch

Table 9-2: Query performance for two methods of computing due dates.

Which method of recoding numeric data is faster?

Chapter 3 presented two methods for recoding numeric sales data into a discount rate based on the level of total sales. One method, shown in Figure 3-10 on page 92, used the `DECODE` and `SIGN` functions in a fairly formidable looking expression. The second method, shown in Figure 3-12 on page 94, used a `PL/SQL` function that removed the complexity from the query. Which method is faster?

Approach

For ease of reference, Figure 9-2 shows the queries for each of the two methods used in Chapter 3 to recode total sales into a discount rate.

```
Figure 3-10
SELECT
  sales_total "sales",
  DECODE(SIGN(sales_total - 100), -1, 0.03,
        DECODE(SIGN(sales_total - 200), -1, 0.04,
              DECODE(SIGN(sales_total - 350), -1, 0.05, 0, 0.05,
                    0.06))) "discount1"
FROM
  xsales
ORDER BY 1;

Figure 3-12
SELECT
  sales_total "sales",
  discount(sales_total) "discount2"
FROM xsales
ORDER BY 1;
```

Figure 9-2: Two methods of recoding numeric data.

Table 9-3 compares the elapsed time for 10 queries using each of the two methods of recoding numeric data. The table shows that the SQL approach using the `DECODE` and `SIGN` functions is significantly faster than the method employing a `PL/SQL` function, by an 8:1 ratio. Depending on the circumstances of your query, you may wish to trade the ease of interpretation provided by the `PL/SQL` function for the improved performance provided by the `DECODE` expression.

See Also

1. The discussion about recoding numeric data and discount rates begins in Chapter 3 on page 95.
2. Chapter 3 (Figure 3-11 on page 93) shows the discount function.

Query	--Elapsed Time (sec)--			Mean Ratio	Statistical Difference?
	Mean	StdDev	Trials		
Figure 3-10	7.09	0.51	10	1.0	
Figure 3-12	58.58	1.84	10	8.3	Yes

Notes:

1. Optimizer: Rule
2. Hints: None
3. Indexes: None
4. Number of rows: 5000
5. Statistical test: two-tailed t-test, 0.01 probability
6. Elapsed time = elapsed execute + elapsed fetch

Table 9-3: Query performance for two methods of recoding numeric data.

Which method of returning an address hierarchy is faster?

Chapter 4 presented two methods for including an address hierarchy in queries, that is, for selecting one address if it exists but a second address if the first address does not exist. One method, shown in Figure 4-3 on page 111, used complex `WHERE` logic to retrieve the address hierarchy; the second method, shown in Figure 4-5 on page 113, used a `PL/SQL` function called `address_hierarchy` that makes the query much simpler and easier to understand. Which method performs better?

Approach

For ease of reference, Figure 9-3 shows the more complicated `SQL` method of producing an address hierarchy. Figure 9-4 shows the more elegant

```

Figure 4-3
SELECT
  count(distinct name_id) "count1",
  count(*) "count2"
FROM
  address a1,
  name
WHERE
  name_seriesno = name_now(name_id)
  AND a1.address_id = name_id
  AND
  (
    (a1.address_type = 'PR'
     AND a1.address_status = 'A'
     AND a1.address_seriesno =
       max_address(name_id, a1.address_type))
  OR
    (a1.address_type = 'BU'
     AND a1.address_status = 'A'
     AND a1.address_seriesno =
       max_address(name_id, a1.address_type)
     AND NOT EXISTS
       (SELECT 'x' FROM address a2
        WHERE a2.address_id = name_id
              AND a2.address_type = 'PR'
              AND a2.address_status = 'A'))
  );

```

Figure 9-3: `SQL` method of returning an address hierarchy.

query that uses a `PL/SQL` function.

Table 9-4 compares the performance of both approaches to address hierarchies based on elapsed times for running each query 10 times. Note that the query with the complex `WHERE` condition is significantly faster than the query that uses the `PL/SQL` function. Again, the trade-off is performance for ease of query construction and interpretation.

```

Figure 4-5
SELECT
    count(distinct name_id) "count1",
    count(*) "count2"
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');
    
```

Figure 9-4: PL/SQL method of returning an address hierarchy.

Query	--Elapsed Time (sec)--			Mean Ratio	Statistical Difference?
	Mean	StdDev	Trials		
Figure 4-3	87.80	2.23	10	1.0	
Figure 4-5	121.92	5.68	10	1.4	Yes

Notes:

1. Optimizer: Rule
2. Hints: None
3. Indexes: Primary keys on *NAME* and *ADDRESS* tables
4. Number of rows: 1200
5. Statistical test: two-tailed t-test, 0.01 probability
6. Elapsed time = elapsed execute + elapsed fetch

Table 9-4: Query performance for two methods to do an address hierarchy.

See Also

1. The discussion about address hierarchies begins in Chapter 4 on page 109.
2. Several PL/SQL functions appear in the two methods used to construct address hierarchies. Chapter 1 (Figure 1-9 on page 13) shows the name_now function; Chapter 4 (Figure 4-4 on page 112) shows the max_address function; and Chapter 4 (Figure 4-6 on page 113) shows address_hierarchy.

Is a correlated subquery or a PL/SQL function faster?

Chapter 5 discussed one query that used a correlated subquery to return the most recent training data on students (see Figure 5-12 on page 149) and a second equivalent query that used a PL/SQL function instead of the subquery (see Figure 5-14 on page 152). Which method is faster?

Approach

For ease of reference, Figure 9-5 shows each of the two methods for returning the most recent student training data.

Figure 5-12

```
SELECT
    COUNT(*) "count1",
    COUNT(DISTINCT name_id) "count2"
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);
```

Figure 5-14

```
SELECT
    COUNT(*) "count1",
    COUNT(DISTINCT name_id) "count2"
FROM
    student,
    name
WHERE
    name_seriesno = name_now(name_id)
    AND student_id = name_id
    AND student_regseq = max_regseq(name_id);
```

Figure 9-5: Correlated subquery and equivalent PL/SQL function.

Table 9-5 compares the elapsed times for 10 trials with each query. Note that the correlated subquery is significantly faster than the PL/SQL function. The mean elapsed time for the query using the PL/SQL function is double that for the query using the correlated subquery. Again, depending on your circumstances, you may prefer to trade query clarity and ease of interpretation for the greater speed available with the more complex correlated subquery.

Query	--Elapsed Time (sec)--			Mean Ratio	Statistical Difference?
	Mean	StdDev	Trials		
Figure 5-12	44.47	1.66	10	1.0	
Figure 5-14	99.43	1.94	10	2.2	Yes

Notes:

1. Optimizer: Rule
2. Hints: None
3. Indexes: Primary keys on *NAME* and *STUDENT* tables
4. Number of rows: 1200
5. Statistical test: two-tailed t-test, 0.01 probability
6. Elapsed time = elapsed execute + elapsed fetch

Table 9-5: Query performance for correlated subquery and PL/SQL function.

See Also

1. The discussion about correlated subqueries and their alternatives begins in Chapter 5 on page 148.
2. The PL/SQL for the name_now function appears in Chapter 1 (Figure 1-9 on page 13). Chapter 5 (Figure 5-15 on page 153) shows the PL/SQL for the max_regseq function.

Which method of doing an outer join to a subquery is fastest?

Oracle's version of SQL does not permit an outer join to a subquery. Chapter 5 discussed three procedures that provide this functionality. One method, shown as Figure 5-20 on page 160, used a correlated subquery and an OR operator; a second method, shown as Figure 5-21 on page 161, achieved the same effect using a UNION compound query; and the third method, shown as Figure 5-22 on page 162, hid the complexity in each of the other methods by using a PL/SQL function. Which of the three methods is fastest?

Approach

For ease of reference, Figure 9-6 shows two approaches that are equivalent to doing an outer join to a subquery — one using a correlated subquery and the second a PL/SQL function. The method that uses the compound query appears in Figure 9-7.

```

Figure 5-20
SELECT
    count(*) "count1",
    count(distinct name_id) "count2"
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)
      );

```

```

Figure 5-22
SELECT
    count(*) "count1",
    count(distinct name_id) "count2"
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);

```

Figure 9-6: Two methods equivalent to an outer join to a subquery.

```

Figure 5-21
SELECT
    count(*) "count1",
    count(distinct name_id) "count2"
FROM
    name
WHERE name_seriesno = name_now(name_id)
    AND NOT EXISTS
        (SELECT 'x'
         FROM student
         WHERE student_id = name_id)
UNION
SELECT
    count(*) "count1",
    count(distinct name_id) "count2"
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);
    
```

Figure 9-7: A third method equivalent to an outer join to a subquery.

Table 9-6 compares the average elapsed times for 10 trials with each of the three queries. Note that the complex SQL method (Figure 5-20) proved fastest by a significant margin. This query completed in approximately one-half the time of the others. The UNION compound query also was significantly faster than the PL/SQL function by about 13%.

Query	--Elapsed Time (sec)--		Trials	Mean Ratio	Statistical Difference?
	Mean	StdDev			
Figure 5-20	46.60	4.49	10	1.0	
Figure 5-21	88.44	2.27	10	1.9	Yes
Figure 5-22	99.56	4.09	10	2.1	Yes

Notes:

1. Optimizer: Rule
2. Hints: None
3. Indexes: Primary keys on *NAME* and *STUDENT* tables
4. Number of rows: 1200
5. Statistical test: two-tailed t-test, 0.01 probability
6. Elapsed time = elapsed execute + elapsed fetch

Table 9-6: Performance for methods equivalent to a subquery outer join.

See Also

1. See Chapter 5 on page 159 for the discussion of outer joins and

subqueries, and how you can provide equivalent functionality.

2. Subqueries often pose their own unique problems for query writers. For a discussion of these problems, see the section beginning in Chapter 5 on page 148

3. See Chapter 1 (Figure 1-9 on page 13) for the `name_now` function and Chapter 5 (Figure 5-15 on page 153) for `max_regseq`.

Which method equivalent to an outer join with OR or IN is fastest?

Outer joins with OR or IN produce errors in Oracle SQL. Chapter 5 discussed three methods that provide this functionality. One approach, shown in Figure 5-24 on page 164, used a single SQL query with a complex WHERE clause requiring several OR conditions. A second method, shown in Figure 5-25 on page 165, used a compound UNION query to achieve the same effect. The third method, shown in Figure 5-26 on page 166, produced the same results using a PL/SQL function. Which of the three approaches is fastest?

Approach

For convenient reference, Figure 9-8 through Figure 9-10 show the three methods discussed in Chapter 5 that are equivalent to performing an outer join with the OR logical operator or with the IN comparison operator. Figure 9-8 shows the first solution based on a single SQL query and a complex WHERE clause.

```

Figure 5-24
SELECT
    COUNT(*) "count1",
    COUNT(DISTINCT name_id) "count2"
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')))
    );

```

Figure 9-8: First method equivalent to an outer join with OR or IN.

Figure 9-9 shows the second solution, this one based on a compound UNION query.

```

Figure 5-25
SELECT
    COUNT(*) "count1",
    COUNT(DISTINCT name_id) "count2"
FROM
    student s1, name
WHERE
    name_seriesno = name_now(name_id)
    AND s1.student_id = name_id
    AND 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'))
UNION
SELECT
    COUNT(*) "count1",
    COUNT(DISTINCT name_id) "count2"
FROM
    name
WHERE
    name_seriesno = name_now(name_id)
    AND NOT EXISTS
        (SELECT 'x'
         FROM student
         WHERE student_id = name_id
         AND student_sponsor IN ('Self','Employer'));

```

Figure 9-9: Second method equivalent to an outer join with OR or IN.

Figure 9-10 shows the third solution based on a PL/SQL function.

```

Figure 5-26
SELECT
    COUNT(*) "count1",
    COUNT(DISTINCT name_id) "count2"
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');

```

Figure 9-10: Third method equivalent to an outer join with OR or IN.

Table 9-7 shows the results from 10 trials with each of the three queries. The single SQL query utilizing a complex WHERE clause is considerably faster than either of the other two approaches. Depending on your

query circumstances, you may wish to avoid the conceptually cleaner PL/SQL function and choose the performance gains available in the more complex SQL query.

Query	--Elapsed Time (sec)--		Trials	Mean Ratio	Statistical Difference?
	Mean	StdDev			
Figure 5-24	46.37	1.66	10	1.0	
Figure 5-25	91.49	2.17	10	2.0	Yes
Figure 5-26	83.19	4.35	10	1.8	Yes

Notes:

1. Optimizer: Rule
2. Hints: None
3. Indexes: Primary keys on *NAME* and *STUDENT* tables
4. Number of rows: 1200
5. Statistical test: two-tailed t-test, 0.01 probability
6. Elapsed time = elapsed execute + elapsed fetch

Table 9-7: Performance for queries in Figure 9-8 through Figure 9-10.

See Also

1. The discussion about outer joins with IN and OR operators begins in Chapter 5 on page 163.
2. Chapter 1 (Figure 1-9 on page 13) shows the PL/SQL function `name_now`; Chapter 5 (Figure 5-27 on page 167) shows the `sponsor_exist` function.

Which method of constructing a traditional cross-tabulation is faster?

Chapter 6 discussed two methods for constructing a traditional cross-tabulation. This type of report appears as a table, with counts of the population items in the table cells and the rows and columns each containing categorical data columns. One method to do cross-tabulations relied on the `DECODE` function (see Figure 6-4 on page 180); a second method (see Figure 6-6 on page 182) used an identity matrix. Which of these methods is faster?

Approach

For convenient reference, the two methods that produce traditional cross-tabulations are shown in Figure 9-11.

```
Figure 6-4
BREAK ON REPORT
COMPUTE SUM OF col1 col2 count ON REPORT
SELECT
    NVL(student_sponsor, 'Unknown') "sponsor",
    SUM(DECODE(person_gender, 'M', 1, 0)) "col1",
    SUM(DECODE(person_gender, 'F', 1, 0)) "col2",
    COUNT(*) "count"
FROM
    person, student
WHERE
    student_regseq = max_regseq(student_id)
    AND person_id = student_id
GROUP BY
    NVL(student_sponsor, 'Unknown')
ORDER BY 1;

Figure 6-6
BREAK ON REPORT
COMPUTE SUM OF col1 col2 count ON REPORT
SELECT
    NVL(student_sponsor, 'Unknown') "sponsor",
    SUM(sexiden_male) "col1",
    SUM(sexiden_female) "col2",
    COUNT(*) "count"
FROM
    sexiden, person, student
WHERE
    student_regseq = max_regseq(student_id)
    AND person_id = student_id
    AND sexiden_code = person_gender
GROUP BY
    NVL(student_sponsor, 'Unknown')
ORDER BY 1;
```

Figure 9-11: Two methods for constructing traditional cross-tabulations.

Table 9-8 compares the average elapsed times in 10 trials with each of the two methods that produce cross-tabulations. Note that there is no significant difference in the average time. Choosing between these two methods should be a question of personal preference and not dictated by performance considerations.

Query	--Elapsed Time (sec)--		Mean Ratio	Statistical Difference?	
	Mean	StdDev			
Figure 6-4	37.19	1.05	10	1.0	
Figure 6-6	37.62	1.55	10	1.0	No

Notes:

1. Optimizer: Rule
2. Hints: None
3. Indexes: Primary keys on *STUDENT* and *PERSON* tables
4. Number of rows: 1400
5. Statistical test: two-tailed t-test, 0.01 probability
6. Elapsed time = elapsed execute + elapsed fetch

Table 9-8: Performance for two methods of constructing cross-tabulations.

See Also

1. The discussion on traditional cross-tabulation reports begins in Chapter 6 on page 180.
2. See Chapter 5 (Figure 5-15 on page 153) for the PL/SQL function `max_regseq`.

Which method of reporting running totals is faster?

Chapter 6 discussed two methods of preparing a report that showed running totals of a quantitative data column. One method, shown in Figure 6-15 on page 194, used a nonequi self-join. The second method, shown in Figure 6-16 on page 195, used a PL/SQL function to produce the running totals. Both methods refer to the simple situation where each population item has only one value in the data column used to compute the running total. Which of the two methods is faster?

Approach

Figure 9-12 shows the two methods that produce running total reports.

```
Figure 6-15
BREAK ON report
COMPUTE SUM OF fee1 ON REPORT
SELECT
    s1.student_id "id",
    s1.student_total_fee "fee1",
    SUM(s2.student_total_fee) "fee2"
FROM
    student s2,
    student s1
WHERE
    s2.student_id <= s1.student_id
GROUP BY
    s1.student_id,
    s1.student_total_fee
ORDER BY 1;

Figure 6-16
BREAK ON REPORT
COMPUTE SUM OF fee1 ON REPORT
SELECT
    student_id "id",
    student_total_fee "fee1",
    running_fee(student_id) "fee2"
FROM
    student
ORDER BY 1;
```

Figure 9-12: Two methods for constructing running totals.

Table 9-9 compares the average elapsed times for 10 trials with each query. The method that used the PL/SQL function (i.e., the method in Figure 6-16) ran about 6% faster than the method that used the nonequi self-join. This difference was statistically significant, but both methods exact a performance penalty. Depending on your circumstances, you may wish to produce running totals using a statistical package.

Query	--Elapsed Time (sec)--			Mean Ratio	Statistical Difference?
	Mean	StdDev	Trials		
Figure 6-15	58.76	1.67	10	1.0	
Figure 6-16	55.04	2.29	10	0.9	Yes

Notes:

1. Optimizer: Rule
2. Hints: None
3. Indexes: Primary keys on *STUDENT* and *PERSON* tables
4. Number of rows: 325
5. Statistical test: two-tailed t-test, 0.01 probability
6. Elapsed time = elapsed execute + elapsed fetch

Table 9-9: Performance for two methods of constructing running totals.

See Also

1. The discussion about running total reports begins in Chapter 6 on page 194.
2. The PL/SQL for the `running_fee` function appears in Chapter 6 (Figure 6-17 on page 196).

Which method of preparing percentage distributions is faster?

Chapter 6 discussed two methods for constructing percentage distribution reports (i.e., reports that count population items by a categorical data column such as gender and show an associated percentage). Both methods require that the denominator in the percentage computations be determined separately from the main query. One method, shown in Figure 6-20 on page 199, used a view to determine the denominator; the second method, shown in Figure 6-22 on page 201, used PL/SQL to save the denominator value in a bind variable. Which method is faster?

Approach

For convenient reference, Figure 9-13 shows a view used to prepare percentage distribution reports. Figure 9-14 shows the PL/SQL approach.

```
Figure 6-20
BREAK ON REPORT
COMPUTE SUM OF count pct ON REPORT
SELECT
    person_gender "sex",
    COUNT(*) "count",
    ROUND(100*COUNT(*)/totname_total,0) "pct"
FROM
    person, name, totname
WHERE
    name_seriesno = name_now(name_id)
    AND person_id = name_id
GROUP BY
    person_gender,
    totname_total
ORDER BY 1;
```

Figure 9-13: View used to construct percentage distributions.

Table 9-10 shows the average elapsed times for 10 trials using each query method to produce percentage distributions. There is no statistically significant difference in the averages. Allow personal preference to guide your selection of a query method.

See Also

1. The discussion on percentage distribution reports begins in Chapter 6 on page 199.
2. See Chapter 6 (Figure 6-21 on page 200) for the *TOTNAME* view.
3. See Chapter 1 (Figure 1-9 on page 13) for the *name_now* function.

```

Figure 6-22
VARIABLE nametot NUMBER;
BEGIN
  SELECT COUNT(*) INTO :nametot
  FROM name
  WHERE name_seriesno = name_now(name_id);
END;
/
BREAK ON REPORT
COMPUTE SUM OF count pct ON REPORT
SELECT
  person_gender "sex",
  COUNT(*) "count",
  ROUND(100*COUNT(*)/:nametot,0) "pct"
FROM
  person, name
WHERE
  name_seriesno = name_now(name_id)
  AND person_id = name_id
GROUP BY
  person_gender
ORDER BY 1;
    
```

Figure 9-14: Bind variable used for percentage distributions.

Query	--Elapsed Time (sec)--		Mean Ratio	Statistical Difference?
	Mean	StdDev		
Figure 6-20	85.61	2.29	10	1.0
Figure 6-22	86.36	3.66	10	1.0 No

Notes:

1. Optimizer: Rule
2. Hints: None
3. Indexes: Primary keys on *NAME* and *PERSON* tables
4. Number of rows: 1200
5. Statistical test: two-tailed t-test, 0.01 probability
6. Elapsed time = elapsed execute + elapsed fetch

Table 9-10: Performance for two methods of doing percentage distributions.

Which method of preparing a top *N* report is faster?

Chapter 6 discussed two methods for reporting the top *N* values based on the rank order of a data column. One method, shown in Figure 6-25 on page 204, relied only on a nonequi self-join to select the correct rows. The second method, shown in Figure 6-26 on page 205, determined a cutoff value with a PL/SQL cursor, saved this value in a bind variable, and then used the bind variable in a query. Which method is faster?

Approach

Figure 9-15 shows the two methods discussed in Chapter 6 that produce a top *N* report.

```
Figure 6-25
SELECT
  s1.student_id "id",
  s1.student_total_fee "fee"
FROM
  student s1
WHERE
  3 >=
  (SELECT COUNT(DISTINCT s2.student_total_fee)
   FROM
     student s2
   WHERE s2.student_total_fee >= s1.student_total_fee)
ORDER BY 2 DESC;

Figure 6-26
SET TRANSACTION READ ONLY
VARIABLE cutoff NUMBER
DECLARE
  i NUMBER;
  CURSOR main_cursor IS
    SELECT student_total_fee
    FROM student
    ORDER BY 1 DESC;
BEGIN
  OPEN main_cursor;
  FOR i IN 1 .. 3
  LOOP
    FETCH main_cursor INTO :cutoff;
  END LOOP;
  CLOSE main_cursor;
END;
/
SELECT
  student_id "id",
  student_total_fee "fee"
FROM
  student
WHERE
  student_total_fee >= :cutoff
ORDER BY 2 DESC;
```

Figure 9-15: Two methods for constructing top *N* reports.

Table 9-11 compares the average elapsed times for 10 trials with each method for producing top *N* reports. Note that the PL/SQL approach is vastly superior to the SQL subquery. Stay away from nonequi self-joins!

Query	--Elapsed Time (sec)--			Mean Ratio	Statistical Difference?
	Mean	StdDev	Trials		
Figure 6-25	225.32	6.55	10	1.0	
Figure 6-26	0.52	0.03	10	0.0	Yes

Notes:

1. Optimizer: Rule
2. Hints: None
3. Indexes: Primary keys on *STUDENT* table
4. Number of rows: 1400
5. Statistical test: two-tailed t-test, 0.01 probability
6. Elapsed time = elapsed execute + elapsed fetch

Table 9-11: Performance for two methods of preparing top *N* reports.

See Also

1. The discussion about top *N* reports begins in Chapter 6 on page 202.

Which method of finding foreign key violations is fastest?

Chapter 7 discussed four approaches to the general problem of identifying rows in one table that do not have corresponding rows in a second table. This problem arises frequently when debugging SQL programs where population items fall through a join. The four methods appeared in Figure 7-9 on page 229. Which method is fastest?

Approach

One of the four approaches used an outer join and returned rows where ROWID is NULL; the second approach used a correlated subquery as predicate to NOT EXISTS; the third approach used a subquery and NOT IN; and the fourth approach used a compound query with the MINUS operation.

For purposes of tuning, specific queries are needed. Figure 9-16 through Figure 9-19 each illustrate one of the four methods. Figure 9-16 uses an outer join to find rows in the *NAME* table that do not have corresponding rows in the *STUDENT* table.

```
SELECT
  COUNT(*) "count1",
  COUNT(DISTINCT name_id) "count2"
FROM
  student,
  name
WHERE
  name_seriesno = name_now(name_id)
  AND student_id(+) = name_id
  AND student.ROWID IS NULL;
```

Figure 9-16: Outer join approach.

Figure 9-17 uses a correlated subquery to identify the same rows.

```
SELECT
  COUNT(*) "count1",
  COUNT(DISTINCT name_id) "count2"
FROM
  name
WHERE
  name_seriesno = name_now(name_id)
  AND NOT EXISTS
  (SELECT 'x'
   FROM student
   WHERE student_id = name_id);
```

Figure 9-17: Correlated subquery approach.

Figure 9-18 provides the same results using a subquery and NOT IN.

```
SELECT
  COUNT(*) "count1",
  COUNT(DISTINCT name_id) "count2"
FROM
  name
WHERE
  name_seriesno = name_now(name_id)
  AND name_id NOT IN
  (SELECT student_id FROM student);
```

Figure 9-18: Subquery and NOT IN approach.

Figure 9-19 uses a compound query with the MINUS operation to produce the same report.

```
SELECT
  COUNT(*) "count1",
  COUNT(DISTINCT name_id) "count2"
FROM
  name
WHERE
  name_id IN
  (SELECT name_id
   FROM name
   WHERE name_seriesno = name_now(name_id)
   MINUS
   SELECT student_id
   FROM student);
```

Figure 9-19: Compound query with MINUS approach.

Table 9-12 compares average elapsed times for 10 trials using each of the four queries. The NOT IN method required significantly longer processing time than any of the other approaches. None of the other three methods differed significantly among themselves. Choosing among the three can be based on personal preference or other features of a particular query that may tilt the decision in one direction.

See Also

1. See Chapter 7 on page 228 for the discussion of alternative methods to find rows in one table without corresponding rows in a second table.
2. The PL/SQL for the name_now function appears in Chapter 1 (Figure 1-9 on page 13).

Query	--Elapsed Time (sec)--		Mean Trials	Mean Ratio	Statistical Difference?
	Mean	StdDev			
Figure 9-16	43.08	1.67	10	1.0	
Figure 9-17	42.59	1.22	10	1.0	No
Figure 9-18	104.13	2.01	10	2.4	Yes
Figure 9-19	42.73	1.69	10	1.0	No

Notes:

1. Optimizer: Rule
2. Hints: None
3. Indexes: Primary keys on *NAME* and *STUDENT* tables
4. Number of rows: 1200
5. Statistical test: two-tailed t-test, 0.01 probability
6. Elapsed time = elapsed execute + elapsed fetch

Table 9-12: Performance for queries in Figure 9-16 through Figure 9-19.

Which method of finding rows that multiply through a join is fastest?

Chapter 7 discussed three methods to find duplicate keys in a table. These methods also can be used to debug queries by finding rows that multiply through a join, that is, by finding rows in one table that have several corresponding rows in a second table. The three approaches appeared in Figure 7-11 on page 230. Which of the methods is fastest?

Approach

One of the three methods used a `HAVING` clause and `COUNT(*)` to identify rows in one table represented more than once in a second table. The second method used a correlated subquery; the third method used a self-join and `WHERE` criteria that included a `ROWID` mismatch.

To test the performance of the three methods, specific queries are required. Figure 9-20 through Figure 9-22 illustrate each approach to find rows in the *NAME* table that occur multiple times in the *STUDENT* table.

Figure 9-20 uses a `HAVING` clause as the key component.

```
SELECT
    name_id "id",
    COUNT(*) "count"
FROM
    student,
    name
WHERE
    name_seriesno = name_now(name_id)
    AND student_id = name_id
GROUP BY
    name_id
HAVING
    COUNT(*) > 1;
```

Figure 9-20: Approach that uses a `HAVING` clause.

Figure 9-21 relies on a correlated subquery to find the correct rows.

```
SELECT
    name_id
FROM
    name
WHERE
    name_seriesno = name_now(name_id)
    AND 1 <
    (SELECT COUNT(*)
     FROM student
     WHERE student_id = name_id);
```

Figure 9-21: Approach that uses a correlated subquery.

Figure 9-22 uses a third approach to the problem and employs a subquery in which a self-join of the *STUDENT* table occurs. The ROWIDs could only be different if more than *STUDENT* row existed for that person.

```

SELECT
  name_id
FROM
  name
WHERE
  name_seriesno = name_now(name_id)
  AND EXISTS
    (SELECT 'x'
     FROM
       student s2,
       student s1
     WHERE
       s1.student_id = name_id
       AND s2.student_id = name_id
       AND s2.ROWID != s1.ROWID);
    
```

Figure 9-22: Approach that uses a self-join.

Table 9-13 compares the average elapsed times for 10 trials with each of the three queries. All three methods completed in about the same average time (i.e., there was no significant difference in the average times). The method you choose to identify rows in one table with multiple corresponding rows in a second table can be dictated by personal preference or by the circumstances of a specific query which favors one approach over the others..

Query	--Elapsed Time (sec)--		Mean	Statistical
	Mean	StdDev	Ratio	Difference?
Figure 9-20	42.22	1.05	10	1.0
Figure 9-21	41.76	1.20	10	1.0 No
Figure 9-22	43.46	1.92	10	1.1 No

Notes:

1. Optimizer: Rule
2. Hints: None
3. Indexes: Primary keys on *NAME* and *STUDENT* tables
4. Number of rows: 1200
5. Statistical test: two-tailed t-test, 0.01 probability
6. Elapsed time = elapsed execute + elapsed fetch

Table 9-13: Performance for queries in Figure 9-20 through Figure 9-22.

See Also

1. The discussion on finding rows that multiply through joins begins in Chapter 7 on page 230.
2. See Chapter 1 (Figure 1-9 on page 13) for the PL/SQL function called name_now.

Which method of sampling the *N*th row from a query is faster?

Chapter 8 discussed two methods of sampling the *N*th row from a query. One method, shown in Figure 8-2 on page 234, used standard SQL with `GROUP BY` and `HAVING` clauses defining a `ROWNUM` to return. The second method, shown in Figure 8-3 on page 235, used a PL/SQL cursor to loop to the designated row, saved the `ROWID` as a bind variable, and then used this bind variable in a simple SQL query. Which method is faster?

Approach

For convenient reference, Figure 9-23 shows the two methods used in Chapter 8 to sample the *N*th row from a query.

```
Figure 8-2
SELECT
    name_id "id",
    ROWNUM "rownum"
FROM
    name
WHERE
    name_seriesno = name_now(name_id)
GROUP BY
    name_id,
    rownum
HAVING
    ROWNUM = 590;

Figure 8-3
SET TRANSACTION READ ONLY
VARIABLE rown VARCHAR2(20)
DECLARE
    i NUMBER;
    CURSOR main_cursor IS
        SELECT ROWIDTOCHAR(ROWID)
        FROM name
        WHERE name_seriesno = name_now(name_id);
BEGIN
    OPEN main_cursor;
    FOR i IN 1 .. 590
    LOOP
        FETCH main_cursor INTO :rown;
    END LOOP;
    CLOSE main_cursor;
END;
/
SELECT
    name_id "id"
FROM
    name
WHERE
    name.rowid = CHARTOROWID(:rown);
```

Figure 9-23: Two methods for sampling the *N*th row from a query.

Table 9-14 compares the average elapsed times for 10 trials with each of the two queries. The PL/SQL approach was about 20% faster in this performance test. If performance is a significant issue for your query, choose the PL/SQL method of sampling the *N*th row from a query.

Query	--Elapsed Time (sec)--		Trials	Mean Ratio	Statistical Difference?
	Mean	StdDev			
Figure 8-2	42.68	2.17	10	1.0	
Figure 8-3	33.33	1.54	10	0.8	Yes

Notes:

1. Optimizer: Rule
2. Hints: None
3. Indexes: Primary key on *NAME* table
4. Number of rows: 1200
5. Statistical test: two-tailed t-test, 0.01 probability
6. Elapsed time = elapsed execute + elapsed fetch

Table 9-14: Performance for two methods of sampling the *N*th row.

See Also

1. See Chapter 8 on page 234 for the discussion of sampling the *N*th row of data in a query.
2. The PL/SQL for the name_now function appears in Chapter 1 (Figure 1-9 on page 13).

Which method of systematically sampling is faster?

Chapter 8 discussed two methods of systematically sampling rows from a query. One method, shown in Figure 8-5 on page 238, used standard SQL with a MOD function in a HAVING clause to perform the actual sampling. The second method, shown in Figure 8-6 on page 239, used a PL/SQL cursor and an FOR loop to accomplish the same task; it also used the DBMS_OUTPUT package to construct the report. Which method is faster?

Approach

For convenient reference, the two methods used in Chapter 8 to systematically sample are shown in Figure 9-24.

```

Figure 8-5
SELECT
    name_id "id",
    ROWNUM "rownum"
FROM
    name
WHERE
    name_seriesno = name_now(name_id)
GROUP BY
    name_id,
    ROWNUM
HAVING
    MOD(ROWNUM,15) = 0;

Figure 8-6
SET SERVEROUTPUT ON
DECLARE
    CURSOR main_cursor IS
        SELECT name_id, ROWNUM
        FROM name
        WHERE name_seriesno = name_now(name_id);
BEGIN
    DBMS_OUTPUT.PUT_LINE('id      last          rownum');
    DBMS_OUTPUT.PUT_LINE('-----');
    FOR name_rec IN main_cursor
    LOOP
        IF MOD(name_rec.ROWNUM, 15) = 0 THEN
            DBMS_OUTPUT.PUT(RPAD(name_rec.name_id,6));
            DBMS_OUTPUT.PUT_LINE(LPAD(TO_CHAR(name_rec.ROWNUM,
                '999'),10));
        END IF;
    END LOOP;
END;
/

```

Figure 9-24: Two methods for systematically sampling every Nth row.

Table 9-15 compares the average elapsed times for 10 trials with each query. Note that the standard SQL approach is significantly faster than

the PL/SQL approach.

Query	--Elapsed Time (sec)--		Trials	Mean Ratio	Statistical Difference?
	Mean	StdDev			
Figure 8-5	41.92	2.02	10	1.0	
Figure 8-6	62.79	1.68	10	1.5	Yes

Notes:

1. Optimizer: Rule
2. Hints: None
3. Indexes: Primary key on *NAME* table
4. Number of rows: 1200
5. Statistical test: two-tailed t-test, 0.01 probability
6. Elapsed time = elapsed execute + elapsed fetch

Table 9-15: Performance for two methods of systematically sampling.

See Also

1. The discussion about systematic samples begins in Chapter 8 on page 238.
2. See Chapter 1 (Figure 1-9 on page 13) for the PL/SQL function `name_now`.

Which method of determining the median is faster?

Chapter 8 discussed two methods for finding the median value in a distribution. One approach, shown in Figure 8-11 on page 245, used a SQL query with a complex HAVING clause. The second approach, shown in Figure 8-12 on page 247, used a PL/SQL cursor to determine the total number of values in the distribution, saved this value in a bind variable, and finally used the bind variable in a SQL query. Which method is faster?

Approach

For convenient reference, Figure 9-25 shows the straight SQL method to find a median. Figure 9-26 shows the PL/SQL approach.

```

Figure 8-11
BREAK ON REPORT
COMPUTE AVG LABEL 'Median' OF fee ON REPORT
SELECT
  ROWNUM "rownum",
  student_total_fee "fee"
FROM
  (SELECT student_total_fee FROM student
   WHERE student_total_fee IS NOT NULL
   UNION
   SELECT 1 FROM DUAL WHERE 1=2)
GROUP BY
  student_total_fee,
  ROWNUM
HAVING
  ROWNUM >=
  (SELECT DECODE(MOD(total_freq,2),
    1,TRUNC(total_freq/2 + 1),
    0,TRUNC(total_freq/2))
   FROM
    (SELECT COUNT(*) AS total_freq FROM student
     WHERE student_total_fee IS NOT NULL))
 AND ROWNUM <=
  (SELECT DECODE(MOD(total_freq,2),
    1,TRUNC(total_freq/2 + 1),
    0,TRUNC(total_freq/2) + 1)
   FROM
    (SELECT COUNT(*) AS total_freq FROM student
     WHERE student_total_fee IS NOT NULL));

```

Figure 9-25: One method that determines the median of a distribution.

Table 9-16 shows no statistically significant performance differences between the two approaches..

See Also

1. The discussion on medians begins in Chapter 8 on page 244.

```

Figure 8-12
SET TRANSACTION READ ONLY
VARIABLE total_freq NUMBER
DECLARE
    CURSOR c1 IS
        (SELECT COUNT(*) FROM student
         WHERE student_total_fee IS NOT NULL);
BEGIN
    OPEN c1;
    FETCH c1 INTO :total_freq;
    CLOSE c1;
END;
/
BREAK ON REPORT
COMPUTE AVG LABEL 'Median' OF fee ON REPORT
SELECT
    ROWNUM "rownum",
    student_total_fee "fee"
FROM
    (SELECT student_total_fee FROM student
     WHERE student_total_fee IS NOT NULL
     UNION
     SELECT 1 FROM DUAL WHERE 1=2)
GROUP BY
    student_total_fee,
    ROWNUM
HAVING
    ROWNUM >=
        (SELECT DECODE(MOD(:total_freq,2),
            1,TRUNC(:total_freq/2 + 1),
            0,TRUNC(:total_freq/2))
     FROM DUAL)
    AND ROWNUM <=
        (SELECT DECODE(MOD(:total_freq,2),
            1,TRUNC(:total_freq/2 + 1),
            0,TRUNC(:total_freq/2) + 1)
     FROM DUAL);
    
```

Figure 9-26: One method that determines the median of a distribution.

Query	--Elapsed Time (sec)--		Mean Ratio	Statistical Difference?
	Mean	StdDev		
Figure 8-11	1.30	0.32	10	1.0
Figure 8-12	1.08	0.28	10	0.8 No

Notes:

1. Optimizer: Rule
2. Hints: None
3. Indexes: Primary key on *STUDENT* table
4. Number of rows: 1000
5. Statistical test: two-tailed t-test, 0.01 probability
6. Elapsed time = elapsed execute + elapsed fetch

Table 9-16: Performance for two methods of finding the median.

Which method of determining the mode is faster?

Chapter 8 discussed two methods for determining the modal value in a distribution. One approach, shown in Figure 8-14 on page 249, used `GROUP BY` and `HAVING` clauses in the main query. The second approach, shown in Figure 8-15 on page 250, used a subquery in the `FROM` clause and a second subquery in the `WHERE` clause to achieve the same effect. Which of the two methods is faster?

Approach

For convenient reference, Figure 9-27 shows the two methods to find the modal value of a distribution.

```
Figure 8-14
SELECT
    undergrad_age "age",
    COUNT(*) "count"
FROM
    undergrad
GROUP BY
    undergrad_age
HAVING COUNT(*) =
    (SELECT MAX(COUNT(*))
     FROM undergrad
     GROUP BY undergrad_age);

Figure 8-15
SELECT
    undergrad_age "age",
    freq "count"
FROM
    (SELECT undergrad_age, COUNT(*) AS freq
     FROM undergrad
     GROUP BY undergrad_age)
WHERE
    freq =
    (SELECT MAX(freq1)
     FROM
        (SELECT COUNT(*) AS freq1
         FROM undergrad
         GROUP BY undergrad_age));
```

Figure 9-27: Two methods that determine the mode of a distribution.

Table 9-17 shows that there were no performance differences between the two approaches used to determine the mode.

See Also

1. The discussion about finding the mode of a distribution begins in Chapter 8 on page 249.

Query	--Elapsed Time (sec)--			Mean Ratio	Statistical Difference?
	Mean	StdDev	Trials		
Figure 8-14	0.39	0.03	10	1.0	
Figure 8-15	0.39	0.04	10	1.0	No

Notes:

1. Optimizer: Rule
2. Hints: None
3. Indexes: Primary key on *UNDERGRAD* table
4. Number of rows: 1000
5. Statistical test: two-tailed t-test, 0.01 probability
6. Elapsed time = elapsed execute + elapsed fetch

Table 9-17: Performance for two methods of finding the mode.

