5. Developing Efficient SQL Statements

5.3. Restructuring the SQL Statements

Often, rewriting an inefficient SQL statement is easier than modifying it. If you understand the purpose of a given statement, then you might be able to quickly and easily write a new statement that meets the requirement.

5.3.1. Compose Predicates Using AND and =

To improve SQL efficiency, use equijoins whenever possible. Statements that perform equijoins on untransformed column values are the easiest to tune.

5.3.2. Avoid Transformed Columns in the WHERE Clause

Use untransformed column values. For example, use:

```
WHERE a.order_no = b.order_no
```

rather than:

```
WHERE TO_NUMBER (SUBSTR(a.order_no, INSTR(b.order_no, '.') - 1))
= TO_NUMBER (SUBSTR(a.order_no, INSTR(b.order_no, '.') - 1))
```

5.3.2. Avoid Transformed Columns in the WHERE Clause

Do not use SQL functions in predicate clauses or **WHERE** clauses. Any expression using a column, such as a function having the column as its argument, causes the optimizer to ignore the possibility of using an index on that column, even a unique index, unless there is a function-based index defined that can be used.

Avoid mixed-mode expressions, and beware of implicit type conversions. When you want to use an index on the **VARCHAR2** column **charcol**, but the **WHERE** clause looks like this:

```
AND charcol = numexpr
```

5.3.2. Avoid Transformed Columns in the WHERE Clause

where numexpr is an expression of number type (for example, 1, USERENV('SESSIONID'), numcol, numcol+0,...), Oracle translates that expression into:

```
AND TO_NUMBER(charcol) = numexpr
```

Avoid the following kinds of complex expressions:

- col1 = NVL (:b1,col1)
- NVL (col1,-999) = ….
- TO_DATE(), TO_NUMBER(), and so on

5.3.2. Avoid Transformed Columns in the WHERE Clause

These expressions prevent the optimizer from assigning valid cardinality or selectivity estimates and can in turn affect the overall plan and the join method.

Add the predicate versus using **NVL**() technique.

For example:

```
SELECT employee_num, full_name Name, employee_id
  FROM mtl_employees_current_view
  WHERE (employee_num = NVL (:b1,employee_num)) AND (organization_id=:1)
  ORDER BY employee_num;
```

5.3.2. Avoid Transformed Columns in the WHERE Clause

Also:

```
SELECT employee_num, full_name Name, employee_id
  FROM mtl_employees_current_view
  WHERE (employee_num = :b1) AND (organization_id=:1)
  ORDER BY employee_num;
```

When you need to use SQL functions on filters or join predicates, do not use them on the columns on which you want to have an index; rather, use them on the opposite side of the predicate, as in the following statement:

```
TO_CHAR(numcol) = varcol
```

rather than

```
varcol = TO_CHAR(numcol)
```

Ejemplo

Supongamos que tienes dos tablas: "students" y "grades".

La tabla "students" contiene información sobre los estudiantes, incluido su número de estudiante ("student_id") almacenado como VARCHAR2.

La tabla "grades" contiene información sobre las calificaciones de los estudiantes, incluido su número de estudiante ("student_id") almacenado como NUMBER.

Sin conversión explícita:

```
SELECT s.first_name, s.last_name, g.grade
FROM students s
JOIN grades g ON s.student_id = g.student_id;
```

Con conversión explícita:

```
SELECT s.first_name, s.last_name, g.grade
FROM students s
JOIN grades g ON s.student_id = TO_CHAR(g.student_id);
```

5.3.3. Write Separate SQL Statements for Specific Tasks

SQL is not a procedural language. Using one piece of SQL to do many different things usually results in a less-than-optimal result for each task. If you want SQL to accomplish different things, then write various statements, rather than writing one statement to do different things depending on the parameters you give it.

It is always better to write separate SQL statements for different tasks, but if you must use one SQL statement, then you can make a very complex statement slightly less complex by using the **UNION ALL** operator.

5.3.3. Write Separate SQL Statements for Specific Tasks

Optimization (determining the execution plan) takes place before the database knows what values will be substituted into the query. An execution plan cannot, therefore, depend on what those values are. For example:

```
SELECT info
FROM tables
WHERE ...
```

```
AND somecolumn BETWEEN DECODE(:loval, 'ALL', somecolumn, :loval)
AND DECODE(:hival, 'ALL', somecolumn, :hival);
```

5.3.3. Write Separate SQL Statements for Specific Tasks

Written as shown, the database cannot use an index on the **somecolumn** column, because the expression involving that column uses the same column on both sides of the **BETWEEN**.

This is not a problem if there is some other highly selective, indexable condition you can use to access the driving table. Often, however, this is not the case. Frequently, you might want to use an index on a condition like that shown but need to know the values of **:loval**, and so on, in advance. With this information, you can rule out the **ALL** case, which should not use the index.

5.3.3. Write Separate SQL Statements for Specific Tasks

If you want to use the index whenever real values are given for :loval and :hival (if you expect narrow ranges, even ranges where :loval often equals :hival), then you can rewrite the example in the following logically equivalent form:

```
SELECT /* change this half of UNION ALL if other half changes */ info
FROM tables
WHERE ...

AND somecolumn BETWEEN :loval AND :hival
AND (:hival != 'ALL' AND :loval != 'ALL')

UNION ALL
SELECT /* Change this half of UNION ALL if other half changes. */ info
FROM tables
WHERE ...

AND (:hival = 'ALL' OR :loval = 'ALL');
```

5.3.3. Write Separate SQL Statements for Specific Tasks

If you run **EXPLAIN PLAN** on the new query, then you seem to get both a desirable and an undesirable execution plan. However, the first condition the database evaluates for either half of the **UNION ALL** is the combined condition on whether **:hival** and **:loval** are **ALL**. The database evaluates this condition before actually getting any rows from the execution plan for that part of the query.

When the condition comes back false for one part of the **UNION ALL** query, that part is not evaluated further. Only the part of the execution plan that is optimum for the values provided is actually carried out. Because the final conditions on **:hival** and **:loval** are guaranteed to be mutually exclusive, only one half of the **UNION ALL** actually returns rows. (The **ALL** in **UNION ALL** is logically valid because of this exclusivity. It allows the plan to be carried out without an expensive sort to rule out duplicate rows for the two halves of the query.)

5.3.4. Use of EXISTS versus IN for Subqueries

In certain circumstances, it is better to use IN rather than EXISTS. In general, if the selective predicate is in the subquery, then use IN. If the selective predicate is in the parent query, then use EXISTS.

Sometimes, Oracle can rewrite a subquery when used with an IN clause to take advantage of selectivity specified in the subquery. This is most beneficial when the most selective filter appears in the subquery and there are indexes on the join columns. Conversely, using EXISTS is beneficial when the most selective filter is in the parent query. This allows the selective predicates in the parent query to be applied before filtering the rows against the EXISTS criteria.

5.3.4. Use of EXISTS versus IN for Subqueries

"**Example 1**: Using **IN** - Selective Filters in the Subquery" and "**Example 2**: Using **EXISTS** - Selective Predicate in the Parent" are two examples that demonstrate the benefits of **IN** and **EXISTS**. Both examples use the same schema with the following characteristics:

- There is a unique index on the **employees.employee_id** field.
- There is an index on the **orders.customer_id** field.
- There is an index on the **employees.department_id** field.
- The **employees** table has 27,000 rows.
- The **orders** table has 10,000 rows.
- The **OE** and **HR** schemas, which own these segments, were both analyzed with **COMPUTE**.

5.3.4. Use of EXISTS versus IN for Subqueries

Example 1: Using **IN** - Selective Filters in the Subquery

This example demonstrates how rewriting a query to use IN can improve performance. This query identifies all employees who have placed orders on behalf of customer 144.

The following SQL statement uses **EXISTS**:

```
SELECT /* EXISTS example */
         e.employee_id, e.first_name, e.last_name, e.salary
  FROM employees e
 WHERE EXISTS (SELECT 1 FROM orders o                  /* Note 1 */
                  WHERE e.employee_id = o.sales_rep_id   /* Note 2 */
                    AND o.customer_id = 144);           /* Note 3 */
```

5.3.4. Use of EXISTS versus IN for Subqueries

The following plan output is the execution plan (from V$SQL_PLAN) for the preceding statement. The plan requires a full table scan of the employees table, returning many rows. Each of these rows is then filtered against the orders table (through an index).

```
ID   OPERATION             OPTIONS            OBJECT_NAME        OPT    COST
---- --------------------  ----------------   ------------------ ---    ----------
   0 SELECT STATEMENT                                            CHO
   1  FILTER
   2   TABLE ACCESS        FULL               EMPLOYEES          ANA    155
   3   TABLE ACCESS        BY INDEX ROWID     ORDERS             ANA      3
   4    INDEX              RANGE SCAN         ORD_CUSTOMER_IX    ANA      1
```

5.3.4. Use of EXISTS versus IN for Subqueries

Rewriting the statement using IN results in significantly fewer resources used.

The SQL statement using IN:

```
SELECT /* IN example */
       e.employee_id, e.first_name, e.last_name, e.salary
  FROM employees e
 WHERE e.employee_id IN (SELECT o.sales_rep_id        /* Note 4 */
                           FROM orders o
                          WHERE o.customer_id = 144);  /* Note 3 */
```

5.3.4. Use of EXISTS versus IN for Subqueries

The following plan output is the execution plan (from V$SQL_PLAN) for the preceding statement. The optimizer rewrites the subquery into a view, which is then joined through a unique index to the employees table. This results in a significantly better plan, because the view (that is, subquery) has a selective predicate, thus returning only a few employee_ids. These employee_ids are then used to access the employees table through the unique index.

```
ID   OPERATION              OPTIONS          OBJECT_NAME        OPT   COST
---- --------------------   ---------------  -----------------  ---   ----------
  0 SELECT STATEMENT                                             CHO
  1  NESTED LOOPS                                                        5
  2   VIEW                                                               3
  3    SORT                 UNIQUE                                       3
  4     TABLE ACCESS        FULL              ORDERS             ANA     1
  5   TABLE ACCESS          BY INDEX ROWID    EMPLOYEES          ANA     1
  6    INDEX                UNIQUE SCAN       EMP_EMP_ID_PK      ANA
```

5.3.4. Use of EXISTS versus IN for Subqueries

Example 2: Using EXISTS - Selective Predicate in the Parent

This example demonstrates how rewriting a query to use EXISTS can improve performance. This query identifies all employees from department 80 who are sales reps who have placed orders.

The following SQL statement uses IN:

```
SELECT /* IN example */
        e.employee_id, e.first_name, e.last_name, e.department_id, e.salary
  FROM employees e
 WHERE e.department_id = 80                                    /* Note 5 */
   AND e.job_id        = 'SA_REP'                              /* Note 6 */
   AND e.employee_id IN (SELECT o.sales_rep_id FROM orders o); /* Note 4 */
```

5.3.4. Use of EXISTS versus IN for Subqueries

The following plan output is the execution plan (from V$SQL_PLAN) for the preceding statement. The SQL statement was rewritten by the optimizer to use a view on the orders table, which requires sorting the data to return all unique employee_ids existing in the orders table. Because there is no predicate, many employee_ids are returned. The large list of resulting employee_ids are then used to access the employees table through the unique index.

```
 ID   OPERATION             OPTIONS          OBJECT_NAME       OPT   COST
----  --------------------  ---------------  ----------------  ---   ----------
  0 SELECT STATEMENT                                           CHO
  1  NESTED LOOPS                                                     125
  2   VIEW                                                            116
  3    SORT                 UNIQUE                                    116
  4     TABLE ACCESS        FULL              ORDERS           ANA     40
  5   TABLE ACCESS          BY INDEX ROWID   EMPLOYEES         ANA      1
  6    INDEX                UNIQUE SCAN      EMP_EMP_ID_PK     ANA
```

5.3.4. Use of EXISTS versus IN for Subqueries

The following SQL statement uses **EXISTS**:

```
SELECT /* EXISTS example */
        e.employee_id, e.first_name, e.last_name, e.salary
   FROM employees e
  WHERE e.department_id = 80                          /* Note 5 */
    AND e.job_id        = 'SA_REP'                    /* Note 6 */
    AND EXISTS (SELECT 1                              /* Note 1 */
                  FROM orders o
                 WHERE e.employee_id = o.sales_rep_id);  /* Note 2 */
```

5.3.4. Use of EXISTS versus IN for Subqueries

The following plan output is the execution plan (from **V$SQL_PLAN**) for the preceding statement. The cost of the plan is reduced by rewriting the SQL statement to use an **EXISTS**. This plan is more effective, because two indexes are used to satisfy the predicates in the parent query, thus returning only a few **employee_ids**. The **employee_ids** are then used to access the **orders** table through an index.

```
 ID OPERATION                 OPTIONS            OBJECT_NAME          OPT     COST
---- --------------------      ---------------    ---------------------- ---     ----------
   0 SELECT STATEMENT                                                  CHO
   1  FILTER
   2   TABLE ACCESS           BY INDEX ROWID     EMPLOYEES             ANA        98
   3    AND-EQUAL
   4     INDEX                RANGE SCAN         EMP_JOB_IX             ANA
   5     INDEX                RANGE SCAN         EMP_DEPARTMENT_IX     ANA
   6   INDEX                  RANGE SCAN         ORD_SALES_REP_IX       ANA         8
```

5.4. Controlling the Access Path and Join Order with Hints

You can influence the optimizer's choices by setting the optimizer approach and goal, and by gathering representative statistics for the query optimizer. Sometimes, the application designer, who has more information about a particular application's data than is available to the optimizer, can choose a more effective way to execute a SQL statement. You can use hints in SQL statements to instruct the optimizer about how the statement should be executed.

Hints, such as /*+**FULL** */ control access paths. For example:

```
SELECT /*+ FULL(e) */ e.last_name
 FROM employees e
WHERE e.job_id = 'CLERK';
```

5.4. Controlling the Access Path and Join Order with Hints

Join order can have a significant effect on performance. The main objective of SQL tuning is to avoid performing unnecessary work to access rows that do not affect the result. This leads to three general rules:

- Avoid a full-table scan if it is more efficient to get the required rows through an index.
- Avoid using an index that fetches 10,000 rows from the driving table if you could instead use another index that fetches 100 rows.
- Choose the join order so as to join fewer rows to tables later in the join order.

5.4. Controlling the Access Path and Join Order with Hints

The following example shows how to tune join order effectively:

```
 SELECT info
FROM taba a, tabb b, tabc c
WHERE a.acol BETWEEN 100 AND 200
```

```
 AND b.bcol BETWEEN 10000 AND 20000
AND c.ccol BETWEEN 10000 AND 20000
AND a.key1 = b.key1
AND a.key2 = c.key2;
```

5.4. Controlling the Access Path and Join Order with Hints

1-) Choose the driving table and the driving index (if any).

The first three conditions in the previous example are filter conditions applying to only a single table each. The last two conditions are join conditions.

Filter conditions dominate the choice of driving table and index. In general, the driving table is the one containing the filter condition that eliminates the highest percentage of the table. Thus, because the range of 100 to 200 is narrow compared with the range of **acol**, but the ranges of 10000 and 20000 are relatively large, **taba** is the driving table, all else being equal.

With nested loop joins, the joins all happen through the join indexes, the indexes on the primary or foreign keys used to connect that table to an earlier table in the join tree. Rarely do you use the indexes on the non-join conditions, except for the driving table. Thus, after **taba** is chosen as the driving table, use the indexes on **b.key1** and **c.key2** to drive into **tabb** and **tabc**, respectively.

5.4. Controlling the Access Path and Join Order with Hints

2-) Choose the best join order, driving to the best unused filters earliest.

The work of the following join can be reduced by first joining to the table with the best still-unused filter. Thus, if "**bcol BETWEEN** ..." is more restrictive (rejects a higher percentage of the rows seen) than "**ccol BETWEEN** ...", the last join can be made easier (with fewer rows) if **tabb** is joined before **tabc**.

3-) You can use the ORDERED or STAR hint to force the join order.

5.4.1. Use Caution When Managing Views

Be careful when joining views, when performing outer joins to views, and when reusing an existing view for a new purpose.

Use Caution When Joining Complex Views

Joins to complex views are not recommended, particularly joins from one complex view to another. Often this results in the entire view being instantiated, and then the query is run against the view data.

5.4.1. Use Caution When Managing Views

For example, the following statement creates a view that lists employees and departments:

```
 CREATE OR REPLACE VIEW emp_dept
AS
SELECT d.department_id, d.department_name, d.location_id,
     e.employee_id, e.last_name, e.first_name, e.salary, e.job_id
FROM  departments d
     ,employees e
WHERE e.department_id (+) = d.department_id;
```

5.4.1. Use Caution When Managing Views

The following query finds employees in a specified state:

```
SELECT v.last_name, v.first_name, l.state_province
  FROM locations l, emp_dept v
 WHERE l.state_province = 'California'
  AND   v.location_id = l.location_id (+);
```

5.4.1. Use Caution When Managing Views

In the following plan table output, note that the **emp_dept** view is instantiated:

```
Operation                                    |  Name              |  Rows | Bytes|  Cost  | Pstart| Pstop |

| SELECT STATEMENT
|  FILTER
|   NESTED LOOPS OUTER
|    VIEW                                     EMP_DEPT
|     NESTED LOOPS OUTER
|      TABLE ACCESS FULL                      DEPARTMEN
|      TABLE ACCESS BY INDEX                  EMPLOYEES
|       INDEX RANGE SCAN                      EMP_DEPAR
|    TABLE ACCESS BY INDEX R                  LOCATIONS
|     INDEX UNIQUE SCAN                       LOC_ID_PK
```

5.4.1. Use Caution When Managing Views

Do Not Recycle Views

Beware of writing a view for one purpose and then using it for other purposes to which it might be ill-suited. Querying from a view requires all tables from the view to be accessed for the data to be returned. Before reusing a view, determine whether all tables in the view need to be accessed to return the data. If not, then do not use the view. Instead, use the base table(s), or if necessary, define a new view. The goal is to refer to the minimum number of tables and views necessary to return the required data.

Consider the following **example**:

```
SELECT department_name
FROM emp_dept
WHERE department_id = 10;
```

The entire view is first instantiated by performing a join of the **employees** *and* **departments** *tables and then aggregating the data. However, you can obtain* **department_name** *and* **department_id** *directly from the* **departments** *table. It is inefficient to obtain this information by querying the* **emp_dept** *view.*

5.4.1. Use Caution When Managing Views

Use Caution When Unnesting Subqueries

Subquery unnesting merges the body of the subquery into the body of the statement that contains it, allowing the optimizer to consider them together when evaluating access paths and joins.

Use Caution When Performing Outer Joins to Views

In the case of an outer join to a multi-table view, the query optimizer can drive from an outer join column, if an equality predicate is defined on it.

*An outer join **within** a view is problematic because the performance implications of the outer join are not visible.*

5.4.2. Store Intermediate Results

Intermediate, or staging, tables are quite common in relational database systems, because they temporarily store some intermediate results. In many applications they are useful, but Oracle requires additional resources to create them. Always consider whether the benefit they could bring is more than the cost to create them. Avoid staging tables when the information is not reused multiple times.

5.4.2. Store Intermediate Results

Some additional considerations:

- Storing intermediate results in staging tables could improve application performance. In general, whenever an intermediate result is usable by multiple following queries, it is worthwhile to store it in a staging table. The benefit of not retrieving data multiple times with a complex statement already at the second usage of the intermediate result is better than the cost to materialize it.
- Long and complex queries are hard to understand and optimize. Staging tables can break a complicated SQL statement into several smaller statements, and then store the result of each step.
- Consider using materialized views. These are precomputed tables comprising aggregated or joined data from fact and possibly dimension tables.

5.5. Restructuring the Indexes

Often, there is a beneficial impact on performance by restructuring indexes. This can involve the following:

- Remove nonselective indexes to speed the DML.
- Index performance-critical access paths.
- Consider reordering columns in existing concatenated indexes.
- Add columns to the index to improve selectivity.

Do not use indexes as a panacea. Application developers sometimes think that performance will improve if they create more indexes. If a single programmer creates an appropriate index, then this might indeed improve the application's performance. However, if 50 programmers each create an index, then application performance will probably be hampered.

5.6. Modifying or Disabling Triggers and Constraints

Using triggers consumes system resources. If you use too many triggers, then you can find that performance is adversely affected and you might need to modify or disable them.

5.7. Restructuring the Data

After restructuring the indexes and the statement, you can consider restructuring the data.

- Introduce derived values. Avoid GROUP BY in response-critical code.
- Review your data design. Change the design of your system if it can improve performance.
- Consider partitioning, if appropriate.

5.8. Maintaining Execution Plans Over Time

You can maintain the existing execution plan of SQL statements over time either using stored statistics or stored SQL execution plans. Storing optimizer statistics for tables will apply to all SQL statements that refer to those tables. Storing an execution plan (that is, plan stability) maintains the plan for a single SQL statement. If both statistics and a stored plan are available for a SQL statement, then the optimizer uses the stored plan.

5.9. Visiting Data as Few Times as Possible

Applications should try to access each row only once. This reduces network traffic and reduces database load. Consider doing the following:

- Combine Multiples Scans with CASE Statements
- Use DML with RETURNING Clause
- Modify All the Data Needed in One Statement

5.9.1. Combine Multiples Scans with CASE Statements

Often, it is necessary to calculate different aggregates on various sets of tables. Usually, this is done with multiple scans on the table, but it is easy to calculate all the aggregates with one single scan. Eliminating n-1 scans can greatly improve performance.

Combining multiple scans into one scan can be done by moving the **WHERE** condition of each scan into a **CASE** statement, which filters the data for the aggregation. For each aggregation, there could be another column that retrieves the data.

5.9.1. Combine Multiples Scans with CASE Statements

The following example asks for the count of all employees who earn less then 2000, between 2000 and 4000, and more than 4000 each month. This can be done with three separate queries:

```
 SELECT COUNT (*)
  FROM employees
  WHERE salary < 2000;

SELECT COUNT (*)
  FROM employees
  WHERE salary BETWEEN 2000 AND 4000;

SELECT COUNT (*)
  FROM employees
  WHERE salary>4000;
```

5.9.1. Combine Multiples Scans with CASE Statements

However, it is more efficient to run the entire query in a single statement. Each number is calculated as one column. The count uses a filter with the CASE statement to count only the rows where the condition is valid. For example:

```
SELECT COUNT (CASE WHEN salary < 2000
                  THEN 1 ELSE null END) count1,
      COUNT (CASE WHEN salary BETWEEN 2001 AND 4000
                  THEN 1 ELSE null END) count2,
      COUNT (CASE WHEN salary > 4000
                  THEN 1 ELSE null END) count3
 FROM employees;
```

5.9.2. Use DML with RETURNING Clause

When appropriate, use **INSERT**, **UPDATE**, or **DELETE**... **RETURNING** to select and modify data with a single call. This technique improves performance by reducing the number of calls to the database.

5.9.3. Modify All the Data Needed in One Statement

When possible, use array processing. This means that an array of bind variable values is passed to Oracle for repeated execution. This is appropriate for iterative processes in which multiple rows of a set are subject to the same operation.

5.9.3. Modify All the Data Needed in One Statement

For example:

```
 BEGIN
 FOR pos_rec IN (SELECT *
   FROM order_positions
   WHERE order_id = :id) LOOP
      DELETE FROM order_positions
      WHERE order_id = pos_rec.order_id AND
        order_position = pos_rec.order_position;
 END LOOP;
 DELETE FROM orders
 WHERE order_id = :id;
END;
```

Alternatively, you could define a cascading constraint on orders*. In the previous example, one* SELECT *and n* DELETE *are executed. When a user issues the* DELETE *on* orders DELETE FROM orders WHERE order_id *=* :id*, the database automatically deletes the positions with a single* DELETE *statement.*

Thank you