DBG Software

Real Performance…It's About Time

SQL Performance

Tips & Guidelines

Background

The purpose of this document is to serve as a guide for creating efficient SQL. This is a high level document and is not intended to contain all possible efficiencies related to SQL, but does highlight those that are most common.

The primary audience for this document includes database administrators and application developers who are responsible for using SQL in their code.

SQL Guidelines

1. Write SQL statements as simply as possible, unless a specific requirement is being addressed. A cleanly written SQL statement is the first step to tuning. When developing SQL statements, write them so others can understand what the SQL is trying to accomplish. The main table should be listed first in the FROM statement. In the WHERE statement the predicates from the main table should be listed first. Group join predicates by table with any local predicates listed first on the driving table so Oracle is encouraged to utilize an index on the main table. The local predicates should be listed last on the subordinate tables in the where clause to represent how the optimizer implements the joins.

2. Format SQL in an easy to understand manner, so it's easier to read the SQL. For example, in the SQL below notice the indentations and alignments. For instance, notice how the commas and AND's line up with each other. For example, notice the equal signs line up with each other and the SELECT/FROM/WHERE reserved words are on their own lines. Some SQL development tools (eg. TOAD) will format the SQL cleanly, but will differ in format than what's shown below which is OK.

(continued)


```
SELECT
       A.TREE_NAME
     , A.EFFDT
  FROM
       PSTREEDEFN A
     , PSTREENODE C
 WHERE
       A.TREE_NAME  = 'Z_ORG_TREE2'
   AND A.EFF_STATUS = 'A'
   AND A.SETID      = 'WORLD'
   AND A.EFFDT      = (SELECT
                             MAX(A_ED.EFFDT)
                        FROM
                             PSTREEDEFN A_ED
                       WHERE
                             A_ED.SETID      = 'WORLD'
                         AND A_ED.TREE_NAME  = 'Z_ORG_TREE2'
                         AND A_ED.EFFDT     <= SYSDATE)
   AND C.TREE_NAME  = A.TREE_NAME
   AND C.EFFDT      = A.EFFDT
   AND C.SETID      = A.SETID
```

3. Never code "SELECT *" in a program. Only include columns in SELECT statements required by the program. Otherwise, overhead occurs each time unnecessary columns are retrieved. However, each time a column is added to or deleted from a table that was used in the SQL, it must be updated.

4. When coding predicates in the WHERE clause, place the predicate eliminating the greatest number of rows first. For example, consider the following statement:

```
SELECT
       B.EMPNO
     , B.FIRSTNME
     , B.LASTNAME
  FROM
       DSN8310 A
     , EMP     B
 WHERE
       A.WORKDEPT = 'D21'
   AND B.SEX      = 'F'
   AND A.DEPTID = B.DEPTID
```

Suppose the WORKDEPT has ten distinct values and the SEX column only has 2 distinct values. Assuming an even skew, the predicate for WORKDEPT is coded first because it will eliminate more rows than the predicate for the SEX column.

Sequence WHERE clause predicates in order of most restrictive to least restrictive by table and predicate type. Try to place the predicate that filters out the most data first in the WHERE clause. The order of predicate selectivity comes into use when the Optimizer has a "tie" between predicate types and needs a tiebreaker. When this occurs, the Optimizer will always take the predicate coded first.

Also notice in the above example that every column reference in either the SELECT or the WHERE clause has a qualifier (either an "A" or "B"). Although syntactically not every column referenced needs a qualifier, doing so reduces the time it takes to parse the statement and is thus the most efficient way to present the SQL to the Optimizer.

5. Always include join predicates for all tables included in a join. When joining tables A and B, use join predicates A.KEY = B.KEY . Failure to do so will result in all rows from table A joined to all rows in table B (a Cartesian product).

6. Join on existing indexed columns if possible. These will generally flow from a primary key on a table to a foreign key on another table.

7. Ensure that join columns are of the same data type when possible, otherwise index usage may not occur.

8. When performing data conversion in the WHERE clause, be sure to perform the conversion on the host variable and not the table column.

    a. When conversion is necessary on a join predicate, only perform the conversion on the predicate of the table joining from. This allows the optimizer to utilize an index for the table joined.

9. Pre-test all SQL to reduce unit testing, ensure efficiency and guarantee correct syntax. Always check the results of your SQL to confirm they are as expected.

10. Return the minimum number of rows needed by the program.

11. If a table has multicolumn indexes and the first column of a given index is optional for your functionality, try to specify the first column in the WHERE clause of the query. This results in an index scan with at least one matching column. For further efficiency include other columns in the index in your selection as those columns are sequenced in the index.

12. Although DISTINCT will remove duplicates from the result table, it also adds overhead by invoking a sort to perform the removal. Try using additional keys in joins to remove duplicates.

13. Limit the columns specified in ORDER BY. The more columns to be sorted, the less efficient the query.

14. The BETWEEN predicate is usually more readable than the combination of the less than or equal to predicate (<=) and the greater than or equal to predicate (>=).

15. Whenever feasible, use IN or >= and <= instead of LIKE in the WHERE clause. If only a few occurrences exist, using IN with the specific list is more efficient than using LIKE. For example:


```
IN ('VALUE1','VALUE2','VALUE3') instead of  LIKE 'VALUE%'
```

16. When coding LIKE operators in the WHERE clause and the first character of the value being searched is known, try not to put a wildcard (%) in the first character

```
LIKE 'VALUE%'                    instead of  LIKE '%VALUE%'
```

17. The UNION operator always results in a sort. When the UNION operator connects two SELECTs, both SELECTs are issued, the rows are sorted, and all duplicates are eliminated. To avoid duplicates, use the UNION operator.

    The UNION ALL operator, by contrast, does not invoke a sort. All rows from the first SELECT are appended to all rows from the second SELECT. Duplicate rows may exist. Use UNION ALL when duplicates are not a problem or when the SELECTs will not return duplicates.

18. When coding a subquery using negation logic, use NOT EXISTS instead of NOT IN to increase efficiency and also to ensure functional correctness. With NOT EXISTS, Oracle must verify only nonexistence. This can reduce processing time significantly. With the NOT IN predicate, Oracle must materialize the complete results set.

19. When using EXISTS to test for the existence of a particular row, specify a constant in the subquery SELECT list. The SELECT list is unimportant because the statement is checking for existence only and will not actually return columns. For example:

```
SELECT
       EMPNO
  FROM
       DSN8310.EMP
 WHERE
       EXISTS (SELECT
                       1
                 FROM
                       DSN8310.PROJ
                WHERE
                       RESPEMP = EMPNO)
```

20. When determining to use bind variables or literals in the WHERE clause, keep the following in mind:

    - Bind variables can significantly reduce time spent in Oracle parsing statements. Statements with high execution counts will benefit greatly from this type of savings. (Preferred in most cases)

    - Literals allow the Optimizer take advantage of advanced selectivity statistics allowing the optimizer to customize the access path based on the literal values. This is most helpful on columns that have relatively few, highly skewed values.

Next Steps: SQL Plan Overview

After the all of the above guidelines have been considered for your SQL, this section can be referenced as a next step in the SQL life cycle.

Ensure appropriate statistics exist (this does not apply to DB2/400 SQL) in the meta-data. Each table & index should have an appropriate level of statistics in order for the Optimizer to make sound SQL plan decisions. The statistics can either actually represent the current environment or they can be imported from a more stable (and reliable) environment, like production. This would ensure the same plan in non-production as production, but if the data volume is different, so will be the performance metrics.

Consider the efficiency of the SQL execution plan used to retrieve the data. Although we have another document that explores this in greater detail, there are three key concepts to keep in mind:

i. Join sequence

When the SQL contains more than one table, a "join sequence" will be created in the plan. Ensure the first table accessed (usually called the outer table) makes sense. The first table should always be the most restrictive in terms of rows returned; via the predicates used to access that table. Then, further ensure the sequence of the remaining tables is appropriate for the SQL in question.

ii. Join method

Connecting each join step in an SQL plan will be a "join method". There are several types of join methods (nested loop, merge join, hash join, hybrid join, etc); the types of join methods tend to change names between databases. Ensure the join method is appropriate for the SQL in question, which should be cross referenced by how the SQL is primarily being used (random access or sequential access).

iii. Access Method

"Access Method" refers to whether the Optimizer decided to scan a table or use an index to retrieve data for a given plan step. Sometimes, a full table scan can be a good thing (depending on the size of the table, the number of rows being returned and the SQL execution frequency). In general however, using indexes are preferred. The next issue would be "is the index being used optimal for the SQL in question?". Just because indexes are being used in the SQL plan does not mean the plan is optimal (this is important to highlight). If the index being used is optimal, you then have to determine how many columns are being "matched" on that index. This requires a command on what predicate operators will result in a match versus a scan of an index. There is a new column, as of 9iR2, on the PLAN_TABLE called SEARCH_COLUMNS that reveals the number of columns "used" on an index; but it does not distinguish between which columns were matched and which ones were scanned; thus, knowledge of how predicate operators affect this behavior is essential.